1 Scripting Framework

1.1 Introduction

A %PRODUCTNAME macro is a short program used to automate a number of steps. The Scripting Framework is a new feature in %PRODUCTNAME %VERSION. It allows users to write and run macros for %PRODUCTNAME in a number of programming and scripting languages including:

- BeanShell (http://www.beanshell.org/)
- JavaScript (http://www.mozilla.org/rhino/)
- Java (http://www.java.com)
- %PRODUCTNAME Basic *[CHAPTER:BasicAndDialogs]*

The framework is designed so that developers can add support for new languages.

In this chapter, the terms macro and script are interchangeable.

1.1.1 Structure of this chapter

This chapter is organized into the following sections:

- Section *[CHAPTER:ScriptingFramework..UsingtheScriptingFramework]* describes the user interface features of the Scripting Framework
  - Describes how to run a macro using the Run Macro dialog.
  - Describes how to use the Organizer dialogs to create, edit and manage macros.
- Section *[CHAPTER:ScriptingFramework.WritingMacros]* provides a guide on how to get started with writing Scripting Framework macros
  - Describes how to write a simple HelloWorld macro.
  - Describes how Scripting Framework macros interact with %PRODUCTNAME and the %PRODUCTNAME API.
  - Describes how to create a dialog from a Scripting Framework macro.
  - Describes how to compile and deploy a Java macro.

- Section *[CHAPTER:ScriptingFramework.HowItWorks]* describes how the pluggable architecture of the Scripting Framework allows support for new scripting languages to be added easily.
- Section *[CHAPTER:ScriptingFramework.WritingaLanguageScriptProviderUNOcomponentusingtheJavahelper classes]* describes how to use the Scripting Framework Java helper classes to add support for a new scripting language
  - Describes how to use the ScriptProvider abstract base class.
  - Describes how to add editor and management support.
  - Describes how to build and register a ScriptProvider.
- Section *[CHAPTER:ScriptingFramework.WritingaLanguageScriptProviderUNOComponentfromscratch]* describes how to write a LanguageScriptProvider UNO component.

1.1.2 Who should read this chapter

If you are interested in automating %PRODUCTNAME using BeanShell, JavaScript, Java or % PRODUCTNAME Basic then you should read sections *[CHAPTER:ScriptingFramework.UsingtheScriptingFramework]* and *[CHAPTER:ScriptingFramework:WritingMacros].*

If you are interested in adding support to run and write macros in a languagewith a Java based interpreter then you should read section *[CHAPTER:ScriptingFramework.WritingaLanguageScriptProviderUNOcomponentusingtheJavahelperclasses].*

If you are interested in adding support for a scripting language from scratch then you should read section *[CHAPTER:ScriptingFramework.WritingaLanguageScriptProviderUNOComponentfromscratch].*

1.2 Using the Scripting Framework

This section describes how to run and organize macros using the **Tools-Macros** submenu:

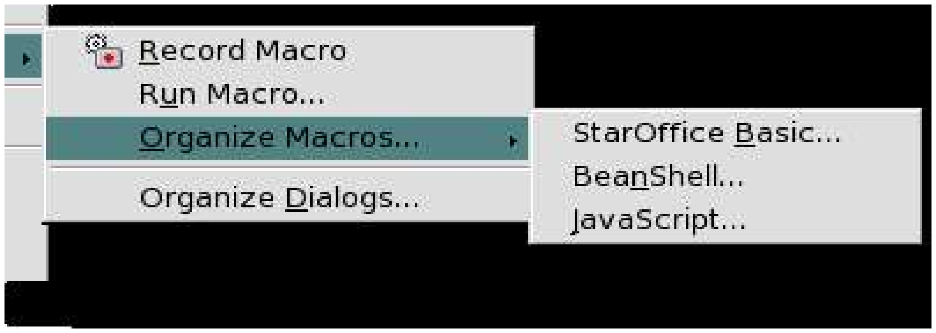
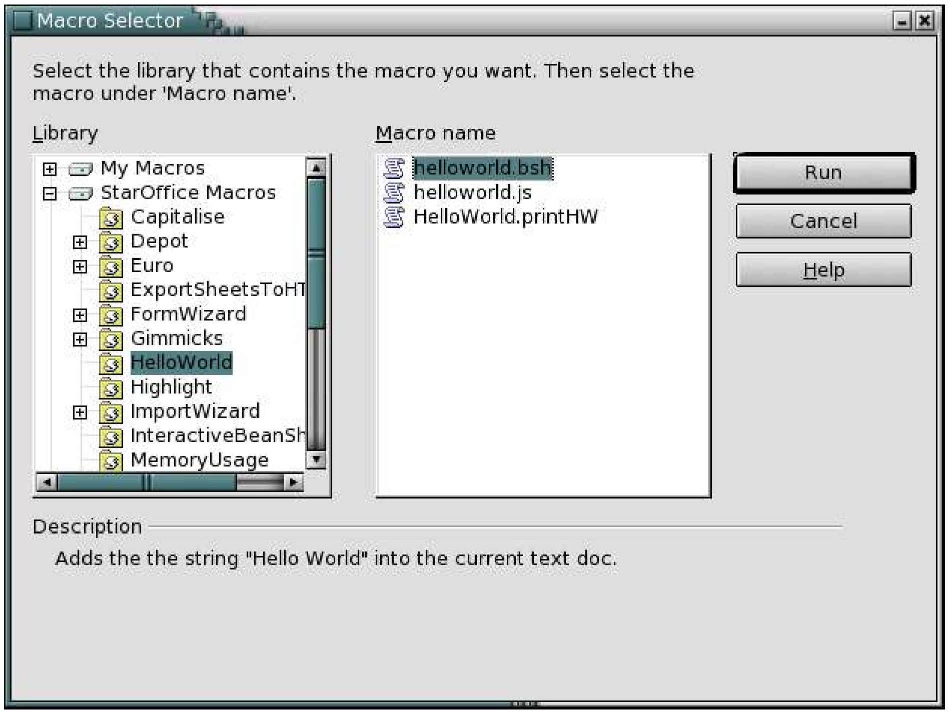


Illustration 1.1Tools-Macros submenu

1.2.1 Running macros

To run a macro use the menu item **Tools – Macros - Run Macro...** This will open the Macro Selector dialog:



Select the library that contains the macro you want. Then select the macro under 'Macro name'.

Adds the the string "Hello World" into the current text doc.

Illustration 1.2Macro Selector dialog

The Library listbox contains a tree representation of all macro libraries. At the top level, there are three entries:

- My Macros (macros belonging to the current user)
- %PRODUCTNAME Macros (macros available to all users of the installation)
- DocumentName Macros (macros contained in the currently active document)

Each of these entries can be expanded to show any macro libraries they contain. When a library has been selected, the macros contained in that library are displayed in the Macro name listbox. When a macro is selected its description, if one exists, is displayed at the bottom of the dialog. Selecting a macro and clicking Run will close the dialog and run the macro. Clicking Cancel will close the dialog without running a macro.

Macros can also be run directly from the Macro Organizer *[CHAPTER:ScriptingFramework.UsingtheScriptingFramework.EditingCreatingandManagingMacros]* and from some of the macro editors.

1.2.1 Editing, Creating and Managing Macros

The Scripting Framework provides support for editing, creating and managing macros via the **Tools – Macro – Organize Macros...** menu. From there you can open a macro management dialog for BeanShell, JavaScript or %PRODUCTNAME Basic macros.

The Organizer dialogs for BeanShell and JavaScript

The Organizer dialogs for BeanShell and JavaScript dialogs work in the same way. The dialog allows you to run macros and edit macros, and create, delete and rename macros and macro libraries.

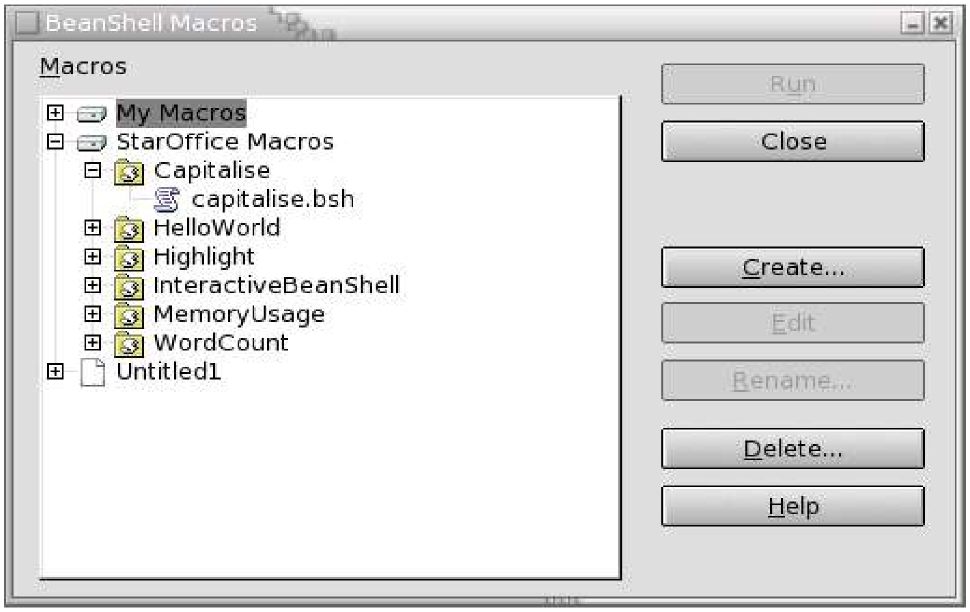


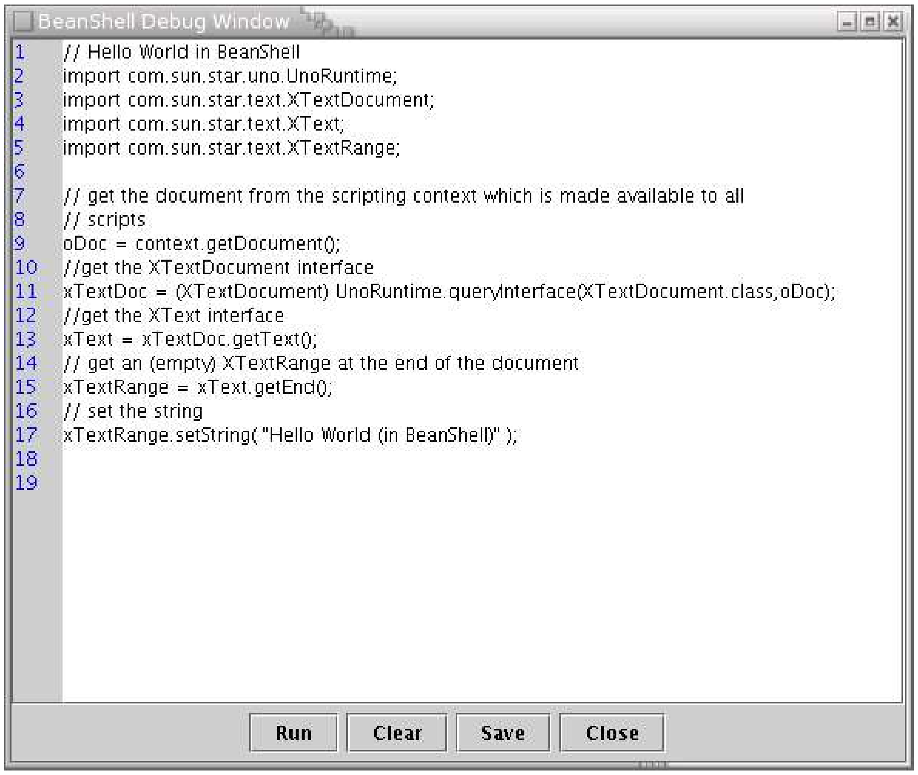
Illustration 1.3BeanShell Organizer

The dialog displays the complete hierarchy of macro libraries and macros that are available for the language. The buttons in the dialog are enabled or disabled depending on which item is selected, so for example, if a read only library is selected the Create button will be disabled. The buttons in the dialog are used as follows:

- Run
  Closes the dialog and runs the selected macro.
- Create
  Pops up a dialog prompting the user for a name for the new library (if a top-level entry is selected) or macro (if a library is selected). The dialog will suggest a name which the user can change if they wish. When the OK button is pressed the new library or macro should appear in the Organizer.
- Edit
  Opens an Editor window for the selected macro.
- Rename
  Opens a dialog prompting the user for a new name for the selected library or macro. By default the dialog will contain the current name, which the user can then change. If the user presses OK the library or macro is renamed in the Organizer.
- Delete
  Deletes the currently selected entry.

BeanShell Editor

Clicking the Edit button in the BeanShell Organizer will open the BeanShell Editor:



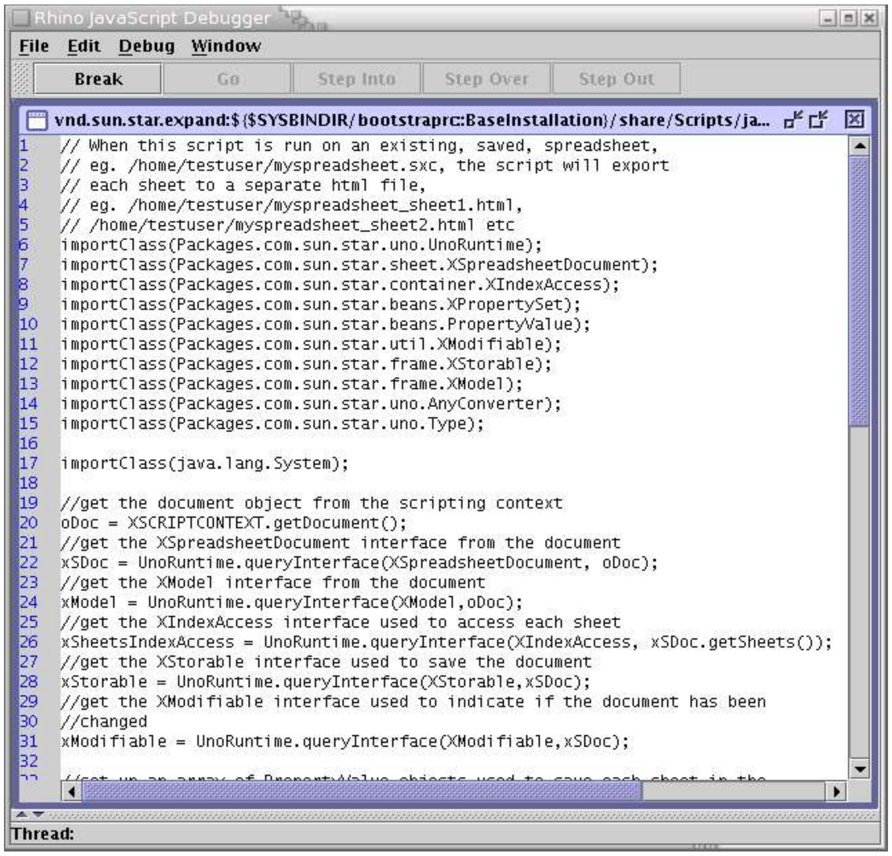
```
// Hello World in BeanShell
import com.sun.star.uno.UnoRuntime;
import com.sun.star.text.XTextDocument;
import com.sun.star.text.XText;
import com.sun.star.text.XTextRange;

// get the document from the scripting context which is made available to all
// scripts
oDoc = context.getDocument();
//get the XTextDocument interface
xTextDoc = (XTextDocument) UnoRuntime.queryInterface(XTextDocument.class,oDoc);
//get the XText interface
xText = xTextDoc.getText();
// get an (empty) XTextRange at the end of the document
xTextRange = xText.getEnd();
// set the string
xTextRange.setString( "Hello World (in BeanShell)" );
```

Illustration 1.4BeanShell Editor

The macro source is listed in the main window with line numbers in the left-hand sidebar. The editor supports simple editing functions (Ctrl-X to cut, Ctrl-C to copy, Ctrl-V to paste, double click to select a word, triple click to select a line). The Run button will execute the source code as displayed in the Editor window.

JavaScript Editor

Clicking the Edit button in the JavaScript Organizer will open the Rhino JavaScript Debugger:


Rhino JavaScript Debugger
File Edit Debug Window
Break
Go
Step Into
Step Over
Step Out
vnd.sun.star.expand:${$SYSBINDIR/bootstraprc:BaseInstallation}/share/Scripts/ja...
Thread:

```
// When this script is run on an existing, saved, spreadsheet,
// eg. /home/testuser/myspreadsheet.sxc, the script will export
// each sheet to a separate html file,
// eg. /home/testuser/myspreadsheet_sheet1.html,
// /home/testuser/myspreadsheet_sheet2.html etc
importClass(Packages.com.sun.star.uno.UnoRuntime);
importClass(Packages.com.sun.star.sheet.XSpreadsheetDocument);
importClass(Packages.com.sun.star.container.XIndexAccess);
importClass(Packages.com.sun.star.beans.XPropertySet);
importClass(Packages.com.sun.star.beans.PropertyValue);
importClass(Packages.com.sun.star.util.XModifiable);
importClass(Packages.com.sun.star.frame.XStorable);
importClass(Packages.com.sun.star.frame.XModel);
importClass(Packages.com.sun.star.uno.AnyConverter);
importClass(Packages.com.sun.star.uno.Type);

importClass(java.lang.System);

//get the document object from the scripting context
oDoc = XSCRIPTCONTEXT.getDocument();
//get the XSpreadsheetDocument interface from the document
xSDoc = UnoRuntime.queryInterface(XSpreadsheetDocument, oDoc);
//get the XModel interface from the document
xModel = UnoRuntime.queryInterface(XModel,oDoc);
//get the XIndexAccess interface used to access each sheet
xSheetsIndexAccess = UnoRuntime.queryInterface(XIndexAccess, xSDoc.getSheets());
//get the XStorable interface used to save the document
xStorable = UnoRuntime.queryInterface(XStorable,xSDoc);
//get the XModifiable interface used to indicate if the document has been
//changed
xModifiable = UnoRuntime.queryInterface(XModifiable,xSDoc);
```

Illustration 1.5JavaScript Debugger

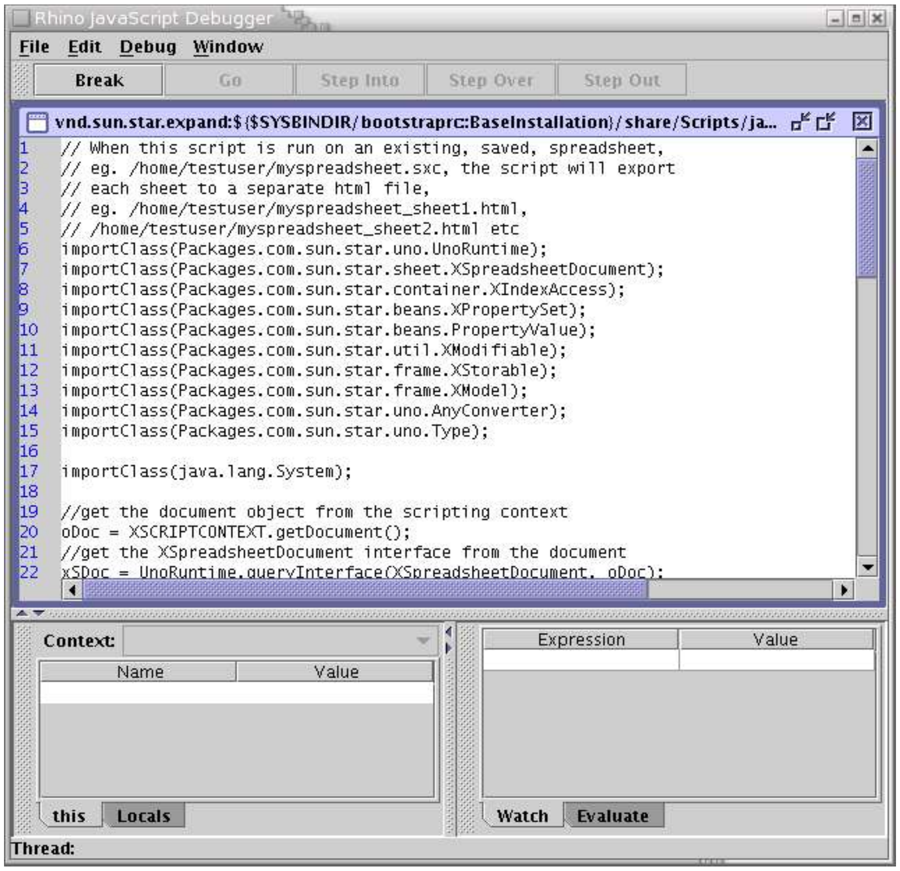
The source of the JavaScript macro is displayed in the main window. The line numbers are shown in the left-hand sidebar. Clicking in the sidebar will set and remove breakpoints in the macro. There is currently a bug in the debugger which is not clearing the symbol in the sidebar when breakpoints are removed.

The contents of the text window can be saved by selecting the **File – Save** menu item. The macro can be run by selecting the **File – Run** menu item. This activates the four buttons above the main text window:

- Break
  Sets a breakpoint at the line where the cursor is.
- Go
  This will run the macro, stopping at the next breakpoint (if one is set).
- Step Into
  This will run a single line of code, stepping into functions if they exist and then stop.

- Step Over
  This will run a single line of code, without stepping into functions and then stop.
- Step Out
  This will continue the execution of the macro until it exits the current function.

There are two other panes in the debugger which are hidden by default. These allow the developer to view the stack and watch variables:


Rhino JavaScript Debugger
File Edit Debug Window
Break Go Step Into Step Over Step Out
vnd.sun.star.expand:$($SYSBINDIR/bootstraprc:BaseInstallation)/share/Scripts/ja...
Context:
Name Value
this Locals
Expression Value
Watch Evaluate
Thread:

```
// When this script is run on an existing, saved, spreadsheet,
// eg. /home/testuser/myspreadsheet.sxc, the script will export
// each sheet to a separate html file,
// eg. /home/testuser/myspreadsheet_sheet1.html,
// /home/testuser/myspreadsheet_sheet2.html etc
importClass(Packages.com.sun.star.uno.UnoRuntime);
importClass(Packages.com.sun.star.sheet.XSpreadsheetDocument);
importClass(Packages.com.sun.star.container.XIndexAccess);
importClass(Packages.com.sun.star.beans.XPropertySet);
importClass(Packages.com.sun.star.beans.PropertyValue);
importClass(Packages.com.sun.star.util.XModifiable);
importClass(Packages.com.sun.star.frame.XStorable);
importClass(Packages.com.sun.star.frame.XModel);
importClass(Packages.com.sun.star.uno.AnyConverter);
importClass(Packages.com.sun.star.uno.Type);

importClass(java.lang.System);

//get the document object from the scripting context
oDoc = XSCRIPTCONTEXT.getDocument();
//get the XSpreadsheetDocument interface from the document
xSDoc = UnoRuntime.queryInterface(XSpreadsheetDocument, oDoc);
```

Illustration 1.6JavaScript Debugger with Stack and Watch tabs displayed

For more information on the Rhino JavaScript Debugger see:
http://www.mozilla.org/rhino/debugger.html

Basic and Dialogs

The %PRODUCTNAME Basic and Dialog Organizers are described in the *[CHAPTER:BasicAnd-Dialogs]* chapter.

Macro Recording

Macro Recording is only supported for %PRODUCTNAME Basic and is accessible via the **Tools-Macro-Record Macro** menu item.

1.3 Writing Macros

1.3.1 The HelloWorld macro

When the user creates a new macro in BeanShell or JavaScript, the default content of the macro is the HelloWorld. Here is what the code looks like for BeanShell:

```
import com.sun.star.uno.UnoRuntime;

import com.sun.star.text.XTextDocument;
import com.sun.star.text.XText;
import com.sun.star.text.XTextRange;

oDoc = context.getDocument();
xTextDoc = (XTextDocument) UnoRuntime.queryInterface(XTextDocument.class,oDoc);
xText = xTextDoc.getText();
xTextRange = xText.getEnd();
xTextRange.setString( "Hello World (in BeanShell)" );

// BeanShell OpenOffice.org scripts should always return 0
return 0;
```

Here is the same code in JavaScript:

```
importClass(Packages.com.sun.star.uno.UnoRuntime);
importClass(Packages.com.sun.star.text.XTextDocument);
importClass(Packages.com.sun.star.text.XText);
importClass(Packages.com.sun.star.text.XTextRange);

oDoc = XSCRIPTCONTEXT.getDocument();
xTextDoc = UnoRuntime.queryInterface(XTextDocument,oDoc);
xText = xTextDoc.getText();
xTextRange = xText.getEnd();
xTextRange.setString( "Hello World (in JavaScript)" );
```

Here is the code for HelloWorld in Java:

```
import com.sun.star.uno.UnoRuntime;
import com.sun.star.frame.XModel;
import com.sun.star.text.XTextDocument;
import com.sun.star.text.XTextRange;
import com.sun.star.text.XText;
import com.sun.star.script.provider.XScriptContext;


public class HelloWorld {
    public static void printHW(XScriptContext xScriptContext)
    {
        Xmodel xDocModel = xScriptContext.getDocument()

        // getting the text document object
        XTextDocument xtextdocument = (XTextDocument) UnoRuntime.queryInterface(
            XTextDocument.class, xDocModel);

        XText xText = xtextdocument.getText();
        XTextRange xTextRange = xText.getEnd();
        xTextRange.setString( "Hello World (in Java)" );
    }
}
```

The table below outlines some of the features of macro development in the different languages:

Language	***Interpreted***	***Typeless***	***Editor***	***Debugger***
BeanShell	Yes	Yes	Yes	No
JavaScript	Yes	Yes	Yes	Yes
Java	No	No	No	No

Instructions on compiling and deploying Java macros can be found later in this chapter.

1.3.2 Using the %PRODUCTNAME API from macros

All BeanShell, JavaScript and Java macros are supplied with a variable of type `[IDL:com.sun.star.script.XScriptContext]` which can be used to access the %PRODUCT-NAME API. This type has three methods:

- `[IDL:com.sun.star.frame.XModel] getDocument( )`
  Returns the XModel interface of the document for which the macro was invoked (see *[CHAPTER:OfficeDevelopment.UsingtheComponentFramework]*)
- `[IDL:com.sun.star.frame.XDesktop] getDesktop( )`
  Returns the XDesktop interface for the application which can be used to access open document, and load documents (see *[CHAPTER.OfficeDevelopement.UsingTheDesktop]*)
- `[IDL:com.sun.star.uno.XComponentContext] getComponentContext( )`
  Returns the XComponentContext interface which is used to create instances of services (see *[CHAPTER:ProfUno.ServiceManagerAndComponentContext]*)

Depending on the language the macro accesses the XScriptContext type in different ways:

- **BeanShell**: Using the global variable XSCRIPTCONTEXT

```
oDoc = XSCRIPTCONTEXT.getDocument();
```

- **JavaScript**: Using the global variable XSCRIPTCONTEXT

```
oDoc = XSCRIPTCONTEXT.getDocument();
```

- **Java**: The first parameter passed to the macro method is always of type XScriptContext

```
Xmodel xDocModel = xScriptContext.getDocument();
```

1.3.3 Handling arguments passed to macros

In certain cases arguments may be passed to macros, for example, when a macro is assigned to a button in a document. In this case the arguments are passed to the macro as follows:

- **BeanShell**: In the global Object[] variable ARGUMENTS

```
event = (ActionEvent) ARGUMENTS[0];
```

- **JavaScript**: In the global Object[] variable ARGUMENTS

```
event = ARGUMENTS[0];
```

- **Java**: The arguments are passed as an Object[] in the second parameter to the macro method

```
public void handleButtonPress(
    XScriptContext xScriptContext, Object[] args)
```

Each of the arguments in the Object[] are of the UNO type Any. For more information on how the Any type is used in Java see *[CHAPTER.ProfUNO.JavaLanguageBinding.TypeMappings]*.

The ButtonPressHandler macros in the Highlight library of a %PRODUCTNAME installation show how a macro can handle arguments.

1.3.4 Creating dialogs from macros

Dialogs which have been built in the Dialog Editor can be loaded by macros using the `[IDL:com.sun.star.awt.XDialogProvider]` API. The XDialogProvider interface has one method createDialog() which takes a string as a parameter. This string is the URL to the dialog. The URL is formed as follows:

vnd.sun.star.script:DIALOGREF?location=[application|document]

where DIALOGREF is the name of the dialog that you want to create, and location is either application or document depending on where the dialog is stored.

For example if you wanted to load dialog called MyDialog, which is in a Dialog Library called MyDialogLibrary in the %PRODUCTNAME dialogs area of your installation then the URL would be:

```
vnd.sun.star.script:MyDialogLibrary.MyDialog?location=application
```

If you wanted to load a dialog called MyDocumentDialog which in a library called MyDocumentLibrary which is located in a document then the URL would be:

```
vnd.sun.star.script:MyDocumentLibrary.MyDocumentDialog?location=document
```

The following code shows how to create a dialog from a Java macro:

```
public XDialog getDialog(XScriptContext context)
{
    XDialog theDialog;

    // We must pass the XModel of the current document when creating a DialogProvider object
    Object[] args = new Object[1];
    args[0] = context.getDocument();

    Object obj;
    try {
```

```
        obj = xmcf.createInstanceWithArgumentsAndContext(
            "com.sun.star.awt.DialogProvider", args, context.getComponentContext());
    }
    catch (com.sun.star.uno.Exception e) {
        System.err.println("Error getting DialogProvider object");
        return null;
    }

    XDialogProvider xDialogProvider = (XDialogProvider)
        UnoRuntime.queryInterface(XDialogProvider.class, obj);

    // Got DialogProvider, now get dialog
    try {
        theDialog = xDialogProvider.createDialog(
            "vnd.sun.star.script:MyDialogLibrary.MyDialog?location=application");
    }
    catch (java.lang.Exception e) {
        System.err.println("Got exception on first creating dialog: " + e.getMessage());
    }
    return theDialog;
}
```

1.3.5 Compiling and Deploying Java macros

Because Java is a compiled language it is not possible to execute Java source code as a macro directly from within %PRODUCTNAME. The code must first be compiled and then deployed within a %PRODUCTNAME installation or document. The following steps show how to create a Java macro using the HelloWorld example code:

- Create a HelloWorld directory for your macro
- Create a HelloWorld.java file using the HelloWorld source code
- Compile the HelloWorld.java file. The following jar files from the *program/classes* directory of a %PRODUCTNAME installation must be in the classpath: ridl.jar, unoil.jar, sandbox.jar, jurt.jar
- Create a HelloWorld.jar file containing the HelloWorld.class file
- Create a parcel-descriptor.xml file for your macro

```
<?xml version="1.0" encoding="UTF-8"?>

<parcel language="Java" xmlns:parcel="scripting.dtd">
    <script language="Java">
        <locale lang="en">
            <displayname value="HelloWorld"/>
            <description>
                Prints "Hello World".
            </description>
        </locale>
        <functionname value="HelloWorld.printHW"/>
        <languagedepprops>
            <prop name="classpath" value="HelloWorld.jar"/>
        </languagedepprops>
    </script>
</parcel>
```

The parcel-descriptor.xml file is used by the Scripting Framework to find macros. The function-name element indicates the name of the Java method which should be executed as a macro. The classpath element can be used to indicate any jar or class files which are used by the macro. If the classpath element is not included, then the directory in which the parcel-desciptor.xml file is

found and any jar files in that directory will be used as the classpath. All of the jar files in the `program/classes directory` are automatically placed in the classpath.

- Copy the HelloWorld directory into the *`share/Scripts/java`* directory of a %PRODUCT-NAME installation or into the *`user/Scripts/java`* directory of a user installation. If you want to deploy the macro to a document you need to place it in a *`Scripts/java`* directory within the document zip file.
- If %PRODUCTNAME is running, you will need to restart it in order for the macro to appear in the Macro Selector dialog.

The parcel-descriptor.xml file is also used to detect BeanShell and JavaScript macros. It is created automatically when creating macros using the Organizer dialogs for BeanShell and JavaScript.

1.4 How the Scripting Framework works

The goals of the ScriptingFramework are to provide pluggable support for new scripting languages and allow macros written in supported languages to be:

- Executed
- Displayed
- Organized
- Assigned to %PRODUCTNAME events, key combinations, menu and toolbar items

This is achieved by enabling new language support to be added by deploying an UNO component that satisfies the service definition specified by `[IDL:com.sun.star.script.provider.LanguageScriptProvider]`. The ScriptingFramework detects supported languages by discovering the available components that satisfy the service specification and obey the naming convention "`com.sun.star.script.provider.ScriptProviderFor[Language]`"

%PRODUCTNAME comes with a number of reference LanguageScriptProviders installed by default.

LanguageScriptProviders	
Language	***Service name***
`Java`	com.sun.star.script.provider.ScriptProviderForJava
`JavaScript`	com.sun.star.script.provider.ScriptProviderForJavaScript
`BeanShell`	com.sun.star.script.provider.ScriptProviderForBeanShell
`Basic`	com.sun.star.script.provider.ScriptProviderForBasic

For more details on naming conventions, interfaces and implementation of a LanguageScriptProvider please see .Writing a LanguageScriptProvider UNO component > and < Chapter. ScriptingFramework helper.

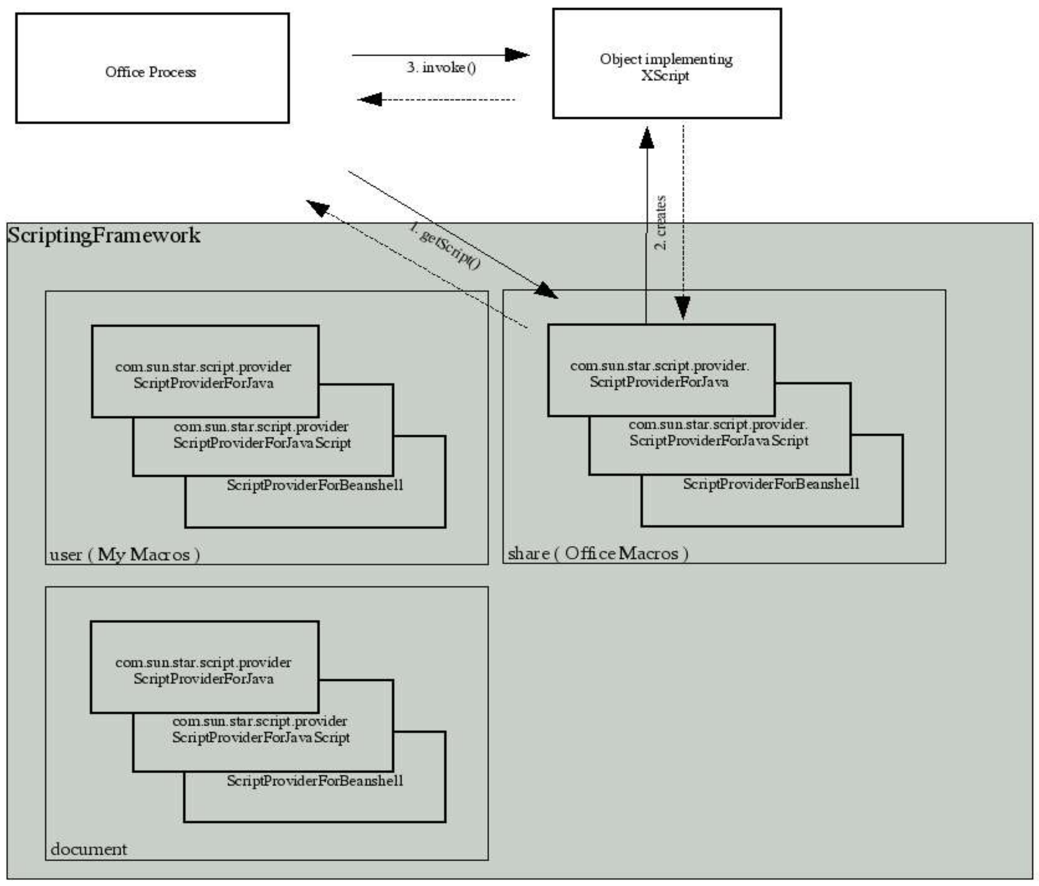

Office Process
3. invoke()
Object implementing XScript
1. getScript()
2. creates
ScriptingFramework
com.sun.star.script.provider ScriptProviderForJava
com.sun.star.script.provider ScriptProviderForJavaScript
ScriptProviderForBeanshell
user (My Macros)
com.sun.star.script.provider. ScriptProviderForJava
com.sun.star.script.provider. ScriptProviderForJavaScript
ScriptProviderForBeanshell
share (Office Macros)
com.sun.star.script.provider ScriptProviderForJava
com.sun.star.script.provider ScriptProviderForJavaScript
ScriptProviderForBeanshell
document

Illustration 1.7: LanguageScriptProvider

The illustration 1.2 above shows the simplified interaction between the Office Process and the ScriptingFramework when invoking a macro. Macros are identified by a URI < add ref to section on URI > and are represented by objects implementing the `[IDL:com.sun.star.script.provider.XScript]` interface. When the `getScript()` method is called the ScriptingFramework uses the URI to determine the correct LangaugeScriptProvider to call `getScript()` on. The LanguageScriptProvider translates a URI into a object that implements `Xscript`. Office can then invoke the macro by calling invoke on that object.

1.5 Writing a LanguageScriptProvider UNO component using the Java helper classes

The Scripting Framework provides a set of Java Helper classes which make it easier to add support for scripting languages for which a Java interpreter exists. This set of classes will handle all of the UNO plumbing required to implement a LanguageScriptProvider, leaving the developer to focus on writing the code to execute their scripting language macros. The steps to add a new LanguageScriptProvider using Java are:

1. Create a new ScriptProviderForYourLanguage by inheriting from the abstract ScriptProvider Java base class
2. Implement the `[IDL:com.sun.star.script.provider.XScript]` UNO interface with code to run your scripting language interpreter from Java
3. Optionally, add support for editing your scripting language macros by implementing the ScriptEditor Java interface
4. Build and register your ScriptProvider

1.5.1 The ScriptProvider abstract base class

The ScriptProvider class is an abstract Java class with three abstract methods:

```
// this method is used to get a script for a script URI
public abstract XScript getScript( String scriptURI )
    throws com.sun.star.uno.RuntimeException,
           com.sun.star.script.provider.ScriptFrameworkErrorException;


// This method is used to determine whether the ScriptProvider has a ScriptEditor
public abstract boolean hasScriptEditor();

// This method is used to get the ScriptEditor for this ScriptProvider
public abstract ScriptEditor getScriptEditor();
```

The most important method is the getScript() method which must be implemented in order for % PRODUCTNAME to execute macros in your scripting language. Fortunately this is made easy by a set of helper methods in the ScriptProvider class, and by a set of helper classes which implement the BrowseNode API described in *[CHAPTER:ScriptingFramework.WritingaLanguageScriptProviderUNOcomponentfromscratch].*

Here is an example of a ScriptProvider implementation for a new ScriptProviderForYourLanguage:

```
import com.sun.star.uno.XComponentContext;
import com.sun.star.lang.XMultiServiceFactory;
import com.sun.star.lang.XSingleServiceFactory;
import com.sun.star.registry.XRegistryKey;
import com.sun.star.comp.loader.FactoryHelper;
import com.sun.star.lang.XServiceInfo;
import com.sun.star.lang.XInitialization;

import com.sun.star.script.provider.XScriptContext;
import com.sun.star.script.provider.XScript;
import com.sun.star.script.framework.provider.ScriptProvider;
import com.sun.star.script.framework.provider.ScriptEditor;
```

```
import com.sun.star.script.framework.container.ScriptMetaData;

public class ScriptProviderForYourLanguage
{
    public static class _ScriptProviderForYourLanguage extends ScriptProvider
    {
        public _ScriptProviderForYourLanguage(XComponentContext ctx)
        {
            super (ctx, "YourLanguage");
        }

        public XScript getScript(String scriptURI)
            throws com.sun.star.uno.RuntimeException,
                   com.sun.star.script.provider.ScriptFrameworkErrorException
        {
            YourLanguageScript script = null;

            try
            {
                ScriptMetaData scriptMetaData = getScriptData(scriptURI);
                XScriptContext xScriptContext = getScriptingContext();
                script = new YourLanguageScript(xScriptContext, scriptMetaData);
            }
            catch (com.sun.star.uno.Exception e)
            {
                System.err.println("Failed to get script: " + scriptURI);
            }
            return script;
        }

        public boolean hasScriptEditor()
        {
            return true;
        }

        public ScriptEditor getScriptEditor()
        {
            return new ScriptEditorForYourLanguage();
        }
    }

    // code to register and create a service factory for ScriptProviderForYourLanguage
    // this code is the standard code for registering classes which implement UNO services
    public static XSingleServiceFactory __getServiceFactory( String implName,
            XMultiServiceFactory multiFactory,
            XRegistryKey regKey )
    {
        XSingleServiceFactory xSingleServiceFactory = null;

        if ( implName.equals( ScriptProviderForYourLanguage._ScriptProviderForYourLanguage.class.getName
() ) )
        {
            xSingleServiceFactory = FactoryHelper.getServiceFactory(
                ScriptProviderForYourLanguage._ScriptProviderForYourLanguage.class,
                "com.sun.star.script.provider.ScriptProviderForYourLanguage",
                multiFactory,
                regKey );
        }

        return xSingleServiceFactory;
    }

    public static boolean __writeRegistryServiceInfo( XRegistryKey regKey )
    {
        String impl =
            "ScriptProviderForYourLanguage$_ScriptProviderForYourLanguage";

        String service1 =
            "com.sun.star.script.provider.ScriptProvider";

        String service2 =
            "com.sun.star.script.provider.LanguageScriptProvider";

        String service3 =
            "com.sun.star.script.provider.ScriptProviderForYourLanguage";

        FactoryHelper.writeRegistryServiceInfo(impl, service1, regKey);
        FactoryHelper.writeRegistryServiceInfo(impl, service2, regKey);
        FactoryHelper.writeRegistryServiceInfo(impl, service3, regKey);

        return true;
    }
}
```

The getScriptData() and getScriptingContext() methods, make the implementation of the getScript () method easy.

The __getServiceFactory() and __writeRegistryServiceInfo() methods are standard %PRODUCT-NAME methods for registering UNO components. The only thing you need to change in them is the name of your ScriptProvider.

1.5.2 Implementing the XScript interface

The next step is to provide the YourLanguageScript implementation which will execute the macro code. The following example shows the code for the YourLanguageScript class:

```
import com.sun.star.uno.Type;
import com.sun.star.uno.Any;
import com.sun.star.lang.IllegalArgumentException;
import com.sun.star.lang.WrappedTargetException;
import com.sun.star.reflection.InvocationTargetException;
import com.sun.star.script.CannotConvertException;

import com.sun.star.script.provider.XScriptContext;
import com.sun.star.script.provider.XScript;
import com.sun.star.script.provider.ScriptFrameworkErrorException;
import com.sun.star.script.provider.ScriptFrameworkErrorType;

import com.sun.star.script.framework.provider.ClassLoaderFactory;
import com.sun.star.script.framework.container.ScriptMetaData;

public class YourLanguageScript implements XScript
{
    private XScriptContext xScriptContext;
    private ScriptMetaData scriptMetaData;

    public YourLanguageScript(XScriptContext xsc, ScriptMetaData smd)
    {
        this.xScriptContext = xsc;
        this.scriptMetaData = smd;
    }

    public Object invoke( Object[] aParams,
                          short[][] aOutParamIndex,
                          Object[][] aOutParam )
        throws com.sun.star.script.provider.ScriptFrameworkErrorException,
               com.sun.star.reflection.InvocationTargetException
    {
        // Initialise the out paramters - not used at the moment
        aOutParamIndex[0] = new short[0];
        aOutParam[0] = new Object[0];

        // Use the following code to set up a ClassLoader if you need one
        ClassLoader cl = null;
        try {
            cl = ClassLoaderFactory.getURLClassLoader( scriptMetaData );
        }
        catch ( java.lang.Exception e )
        {
            // Framework error
            throw new ScriptFrameworkErrorException(
                e.getMessage(), null,
                scriptMetaData.getLanguageName(), scriptMetaData.getLanguage(),
                ScriptFrameworkErrorType.UNKNOWN );
        }

        // Load the source of your script using the scriptMetaData object
        scriptMetaData.loadSource();
        String source = scriptMetaData.getSource();
        Any result = null;

        // This is where you add the code to execute your script
        // You should pass the xScriptContext variable to the script
        // so that it can access the application API

        result = yourlanguageinterpreter.run( source );
```

```
        // The invoke method should return a com.sun.star.uno.Any object
        // containing the result of the script. This can be created using
        // the com.sun.star.uno.AnyConverter helper class
        if (result == null)
        {
            return new Any(new Type(), null);
        }
        return result;
    }
}
```

If the interpreter for YourLanguage supports Java class loading, then the ClassLoaderFactory helper class can be used to load any class or jar files associated with a macro. The ClassLoaderFactory uses the parcel-descriptor.xml file (see *[CHAPTER:ScriptingFramework.WritingMacros.CompilingandDeployingJavaMacros]*) to discover what class and jar files need to be loaded by the script.

The ScriptMetaData class will load the source code of the macro which can then be passed to the YourLanguage interpreter.

1.5.3 Implementing the ScriptEditor interface

If you want to add support for editing scripts you need to implement the ScriptEditor interface:

```
package com.sun.star.script.framework.provider;

import com.sun.star.script.provider.XScriptContext;
import com.sun.star.script.framework.container.ScriptMetaData;

public interface ScriptEditor
{
    public Object execute() throws Exception;
    public void indicateErrorLine( int lineNum );
    public void edit(XScriptContext context, ScriptMetaData entry);
    public String getTemplate();
    public String getExtension();
}
```

The edit() method is called when a user presses the Edit button in the Macro Organizer. The ScriptEditor implementation can use the ScriptMetaData object to obtain the source code for the macro and display it.

The getTemplate() method should return a template of a macro in your language, for example the code to write HelloWorld into a document. The getExtension() method should return the filename extension used for macros written in your language. These methods are called when the Create button is pressed in the Macro Organizer.

The execute() and indicateErrorLine() methods are not called by the Macro Organizer and so they do not have to do anything. They are used by the implementation of the ScriptProviderForBeanShell to execute the source code that is displayed in the ScriptEditorForBeanShell, and to open the ScriptEditorForBeanShell at the line for which an error has occurred. The developer may wish to do the same when writing their ScriptProviderForYourLanguage and ScriptEditorForYourLanguage.

The following code shows an example ScriptEditorForYourLanguage.java file:

```
import com.sun.star.script.framework.provider.ScriptEditor;
import com.sun.star.script.provider.XScriptContext;
import com.sun.star.script.framework.container.ScriptMetaData;
```

```
import javax.swing.*;

public class ScriptEditorForYourLanguage implements ScriptEditor
{
    public Object execute() throws Exception
    {
        return null;
    }

    public void indicateErrorLine( int lineNum )
    {
        return;
    }

    public void edit(XScriptContext context, ScriptMetaData entry)
    {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

        JTextArea ta = new JTextArea();
        entry.loadSource();
        ta.setText(entry.getSource());

        frame.getContentPane().add(ta);
        frame.setSize(400, 400);
        frame.show();
    }

    public String getTemplate()
    {
        return "the code for a YourLanguage script";
    }

    public String getExtension()
    {
        return "yl";
    }
}
```

1.5.4 Building and registering your ScriptProvider

In order to compile these classes you need to include the UNO and Scripting Framework jar files in your classpath. You can find these in the program/classes directory of your %PRODUCT-NAME installation. The jar files that you need to include are: ridl.jar, sandbox.jar, unoil.jar, jurt.jar and ScriptFramework.jar.

To compile ScriptProviderForYourLanuage:

1. Compile the ScriptProviderForYourLanguage.java, ScriptEditorForYourLanguage.java and YourLanguageScript.java files
2. Create a jar file for ScriptProviderForYourLanguage with the following in the manifest file. (Use the -m switch to the jar command to add the manifest data)

```
Built-By: Yours Truly
RegistrationClassName: ScriptProviderForYourLanguage
```

3. Register the ScriptProviderForYourLanguage jar file using the *[CHAPTER:PackageMangaer]*

Now you should see an entry for YourLanguage in the **Tools-Macros-Organize Macros...** menu.

1.6 Writing a LanguageScriptProvider UNO Component from scratch

To provide support for a new scripting language a new LanguageScriptProvider for that language needs to be created. The new LanguageScriptProvider, an UNO component, must be written in a language from which there is an existing UNO bridge. Details about UNO bridges can be found at [Chapter *Advanced UNO.Language Bindings.UNO C++ bridges*].

The LanguageScriptProvider is an UNO component that provides the environment to execute a macro for a specific language. For example when %PRODUCTNAME encounters a Scripting Framework URI (see [Chapter *ScriptingFramework.ScriptingFramework URI Specification*]) the ScriptingFramework finds the appropriate LanguageScriptProvider to execute the script. A LanguageScriptProvider has the following responsibilities:

- It must support the `[IDL:com.sun.star.script.provider.LanguageScriptProvider]` service.
- It is responsible for creating the environment necessary to run a script.
- It is responsible for implementing the `[IDL:com.sun.star.script.browse.BrowseNode]` service to allow macros to be organized and browsed.
- Given a script URI it is responsible for returning a command like object that implements the `[IDL:com.sun.star.script.provider.XScript]` interface that can execute the macro indicated by the URI.
- The name of the any LanguageScriptProvider service must be of the form "`com.sun.star.script.provider.ScriptProviderFor[Language]`", where Language is the language name as it appears in a script URI.

The name of the LanguageScriptProvider MUST be as above otherwise it will not operate correctly.

1.6.1 Scripting Framework URI Specification

`vnd.sun.star.script:MACROREF?language=Language&location=[user|share|document]`

where:

- MACROREF is a name that identifies the macro and the naming convention for MACROREF identifiers is LanguageScriptProvider specific. It allows the LanguageScriptProvider to associate MACROREF with a macro. In the case of the LanguageScriptProviders for the Java based languages supported by %PRODUCTNAME e.g. (Java, JavaS-

cript & Beanshell) the convention is Library.functionname where Library is the subdirectory under the language specific directory and functionname the functionname from the parcel-descriptor.xml in the "Library" directory. See [Chapter *ScriptingFramework.Writing a LanguageScriptProvider UNO component from scratch.Scripting Framework URI Specification*]

- Language specifies the LanguageScriptProvider needed to execute the macro as described.

Example 1 – URI for a JavaScript macro Library1.myMacro.js located in the share directory of a %PRODUCTNAME installation.

```
vnd.sun.star.script:Library1.myMacro.js?language=JavaScript&location=share
```

In general macros contained in UNO packages have the format

```
vnd.sun.star.script:MACROREF?language=TheLanguage&location=
[user:uno_package/packageName|share:uno_package/packageName]
```

Example 2 - URI for a JavaScript macro Library1.myMacro.js located in an uno package called myUnoPkg.uno.pkg located in share directory of a %PRODUCTNAME installation.

```
vnd.sun.star.script:Library1.myMacro.js?language=JavaScript&location=share:uno_
package/myUnoPkg.uno.pkg
```

Note: In the case of the %PRODUCTNAME Basic language, no distinction is made internally between macros deployed in UNO packages on those not deployed in UNO packages. Therefore in the case of a %PRODUCTNAME Basic macro located in an UNO package the location attribute in the URI contains just “user” or “share”.

1.6.2 Storage of Scripts

A LanguageScriptProvider is responsible for knowing about how its own macros are stored: where, what format and what kind of directory structure is used. The Scripting Framework attempts to standardize how to store and discover macros by defining:

- A default directory structure.

  Macros can only be stored under a directory with the language name (as it appears in the script URI) in lowercase under a directory called Scripts which is located in either user or share directories of a %PRODUCTNAME installation or a %PRODUCTNAME document.

Example for a LanguageScriptProvider for “JavaScript” the macro library Highlight is located in

<OfficePath>/share/Scripts/javascript/Highlight

- A generic mechanism for enabling discovery of macros in macro libraries and associating meta-data with scripts located in this libraries. See parcel-descriptor.xml in [Chapter

ScriptingFramework.Writing Macros.Compiling and Deploying Java macros]. Example the parcel-descriptor for the JavaScript Highlight macro library is located in

<OfficePath>/share/Scripts/javascript/Highlight/parcel-descriptor.xml

1.6.3 Implementation

A LanguageScriptProvider implementation must follow the service definition `[IDL:com.sun.star.script.provider.LanguageScriptProvider]`

Since a LanguageScriptProvider is an UNO component, it must additionally contain the component operations needed by a UNO service manager. These operations are certain static methods in Java or export functions in C++. It also has to implement the core interfaces used to enable communication with UNO and the application environment. For more information on the component operations and core interfaces, please see *[CHAPTER:Components.Architecture]* and *[CHAPTER:Components.CoreInterfaces].*

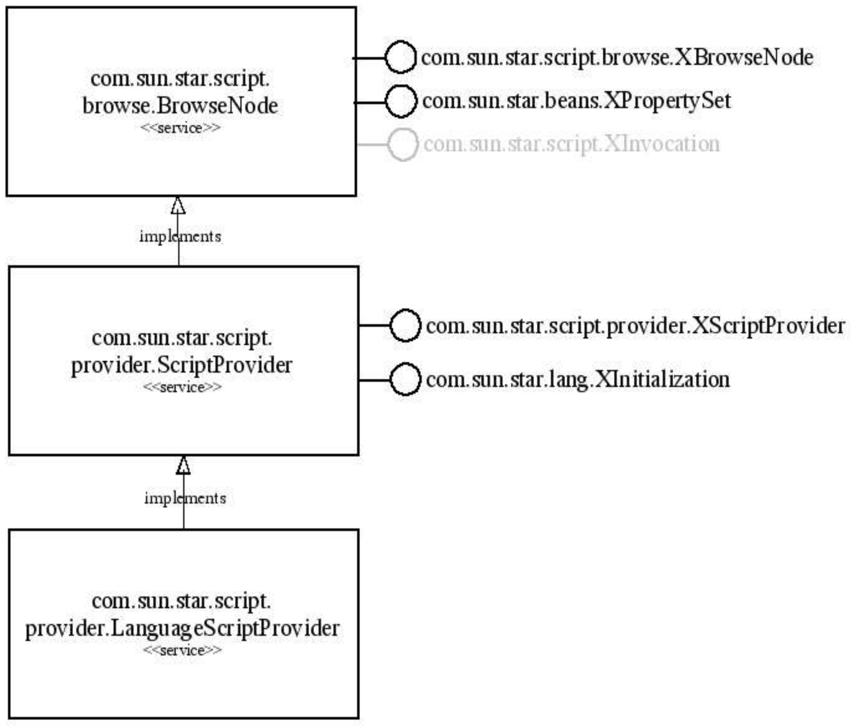

com.sun.star.script.
browse.BrowseNode
<<service>>
com.sun.star.script.browse.XBrowseNode
com.sun.star.beans.XPropertySet
com.sun.star.script.XInvocation
implements
com.sun.star.script.
provider.ScriptProvider
<<service>>
com.sun.star.script.provider.XScriptProvider
com.sun.star.lang.XInitialization
implements
com.sun.star.script.
provider.LanguageScriptProvider
<<service>>

Illustration 1.8: LanguageScriptProvider

The interface `XInitialization` supports method:

```
void initialize(  [in] sequence<any> aArguments )
```

The LanguageScriptProvider is responsible for organizing and execution of macros written in a specific language for a certain location. The possible locations for macros are within a document or either the user or share directories in a %PRODUCTNAME installation. The LanguageScript-Provider is initialized for given location context which is passed as the first argument to the `initialize()` method. The location context is a `string` with the following possible values

Location context	
“user”	`string. Denotes the user directory in a %PRODUCTNAME installation`
“share”	`string. Denotes the share directory in a %PRODUCTNAME installation`
url	string. Where for user or share directory the url has scheme *vnd.sun.star.expand* example: user directory "*vnd.sun.star.expand:${$SYSBINDIR/bootstraprc/::UserInstallation}/user*" share directory "*vnd.sun.star.expand:${$SYSBINDIR/bootstraprc/::BaseInstallation}/share*"
	Where for a currently open document the url has scheme *vnd.sun.star.tdoc* example: *vnd.sun.star.tdoc:/1* <reference UCB tdoc info in Dev guide >

The `[IDL:com.sun.star.script.browse.XBrowseNode]` interface supported by a LanguageScriptProvider service is the initial point of contact for the %PRODUCTNAME application. In order for the %PRODUCNAME to process and display macros it needs to be able to list those macros. Additionally the MacroOrganizer dialogs use the `[IDL:com.sun.star.script.browse.BrowseNode]` service to create/delete new macros and macro libraries.

The interface `[IDL::com.sun.star.script.browse.XBrowseNode]` supports the following methods:

```
string getName()
sequence < ::com::sun::star::script::browse::XBrowseNode > getChildNodes()
boolean hasChildNodes()
short getType()
```

The method `getName()` returns the name of the node.

For the root node of a LanguageScriptProvider the name returned from `getName()` is expected be the language name as it would appear in a script URI e.g. Basic

The method `getChildNodes()`method should return the nodes which represent the next level in the hierarchy of macros and macro Libraries the LanguageScriptProvider is managing.

The method `getType()` returns the type of the node.

Nodes can be of three types represented by the constants `[IDL:com.sun.star.script.browse.BrowseNodeTypes]`

Constants of [IDL:com.sun.star.script.browse.BrowseNodeTypes]	
Value	***Description***
`[IDLS:com.sun.star.script.browse.BrowseNodeTypes:SCRIPT]`	Indicates that the node is a script.
`com.sun.star.script.browse.BrowseNodeTypes:CONTAINER`	Indicates that the node is a container of other nodes e.,g. Library
`com.sun.star.script.browse.BrowseNodeTypes:ROOT`	Reserved for use by the ScriptingFramework.

The objects implementing XBrowseNodes can must also implement `[com.sun.star.beans.XPropertySet]`.

Properties of object implementing [IDL:com.sun.star.browse.BrowseNode]	
`Uri`	`string`. Found on script nodes only, is the script URI for the macro associated with this node.
`Description`	`string`. Found on script nodes only, is a description of the macro associated with this node.
`Creatable`	`boolean.` True if the implementation can create a new container or macro as a child of this node.
`Creatable`	`boolean.` True if the implementation can delete this node.
`Editable`	`boolean.` True if the implementation is able to open the macro in an editor.
`Renamable`	`boolean.` True if the implementation can rename this node.

Note: A node that has the `Creatable`, `Deletable`, `Editable` or `Renamable` properties set to `true` is expected to implement the `[IDL:com.sun.star.script.XInvocation]` interface.

The interface `[IDL::com.sun.star.script.XInvocation]` supports the following methods:

```
com::sun::star::beans::XIntrospectionAccess getIntrospection();
any invoke( [in] string aFunctionName,
            [in] sequence<any> aParams,
            [out] sequence<short> aOutParamIndex,
            [out] sequence<any> aOutParam )

void setValue( [in] string aPropertyName,
               [in] any aValue )

any getValue( [in] string aPropertyName )

boolean hasMethod( [in] string aName )

boolean hasProperty( [in] string aName )
```

The `invoke()` function is passed as an argument the property keys of the `[IDL:com.sun.star.script.browse.BrowseNode]` service

Elements of aParam sequence in invoke call	
aFunctionName	***aParams***
`Editable`	`None required.`
`Creatable`	`aParam[0] should contain the name of the new child node to be created.`
`Deletable`	`None required.`
`Renamable`	`aParam[0] should contain the new name for the node.`
`Uri`	`None required.`
`Description`	`None required.`

Access to a macro if provided for by the `[IDL:com.sun.star.script.provider.XScriptProvider]`interface which supports the following method:

```
::com::sun::star::script::provider::XScript getScript( [in] string sScriptURI )
```

The `getScript()` method is passed a script URI `sScriptURI` and the LanguageScriptProvider implementation needs to parse this URI so it can interpret the details and validate them. As the LanguageScriptProvider is responsible for exporting and generating the URI associated with a macro it is also responsible for performing the reverse translation for a give n URI and returning an object implementing `[IDL:com.sun.star.script.provider.XScript]` interface which will allow the macro to be invoked.

`[IDL:com.sun.star.script.provider.XScript]` which supports the following methods:

```
any invoke( [in] sequence<any> aParams,
            [out] sequence<short> aOutParamIndex,
            [out] sequence<any> aOutParam )
```

In addition to the parameters that may be passed to an object implementing `[IDL:com.sun.star.script.provider.XScript]`it is up to the that object to decide what extra information to pass to a running macro. It makes sense to pass information to the macro which makes the macro writer's job easier. It is recommended that information such as a reference to the document (context), a reference to the service manager (available from the component context passed into the LanguageScriptProvider component's constructor by UNO) and a reference to the desktop (available from UNO using this service manager).

All of the Java based reference LanguagesScriptProvider provided with %PRODUCTNAME make this information available to the running macro in the form of an object implementing the interface `[IDL:com.sun.script.provider.XScriptContext]`. This provides accessor methods to get the current document, the desktop and the component context. Depending on the contraints of the language this information is passed to the macros in different ways, for example in Beanshell and JavaScript this is available as an environment variable and in the case of Java it is passed as the first argument to the macro.

1.6.4 Integration with Package Manager

The Package Manager is a mechanism for deploying components, configuration data and macro libraries. It provides a convenient mechanism for macro developers to distribute their macros. Its functionality is available via a command- line tool or from the **Tools - Package Manager** menu.

For more detail about the Package Manager, it's API and deployment options please refer to <ChapterX-section>. It is recommended that readers are familiar with this material before reading this section. The scripting framework supports deployment of macros in UNO packages. Currently only UNO package bundles for the media type `”application/vnd.sun.star.framework-script”` are supported. Macros deployed in UNO package bundles of this media type must use the ScriptingFramework storage scheme and parcel-descriptor.xml as described in < Chapter/section > to function correctly. An implementation of the `[IDL:com.sun.star.deployment.PackageRegistryBackend]` service is provided which supports deployment of macro libraries of media type `”application/vnd.sun.star.framework-script”` with the Package Manager.

Note: %PRODUCTNAME Basic macros are handled via a separate media type `”application/vnd.sun.star.basic-script”` and hence handled by a different mechanism.

Overview of how ScriptingFramework integrates with the Package Manager API

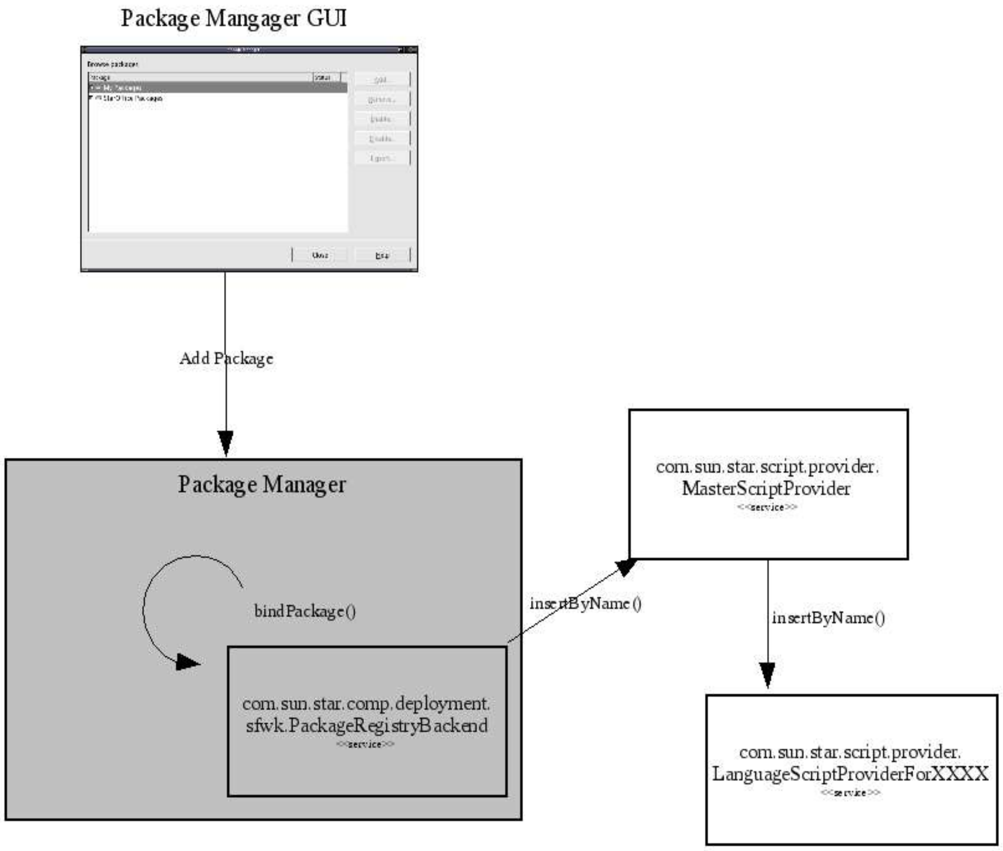

Package Mangager GUI
Add Package
Package Manager
bindPackage()
com.sun.star.comp.deployment.
sfwk.PackageRegistryBackend
<<service>>
insertByName()
com.sun.star.script.provider.
MasterScriptProvider
<<service>>
insertByName()
com.sun.star.script.provider.
LanguageScriptProviderForXXXX
<<service>>

Illustration 1.9: Registration of macro library using Package Mangager

Registration

Macro libraries contained in UNO script packages are registered to the user or share installation deployment context via the unopkg command-line tool or from **Tools – Package Manager** the Package Manager subsystem informs the LanguageScriptProvider (for the appropriate installation deployment context) by calling its `insertByName()` method. The LanguageScriptProvider persists the registration of the macro library in order to be aware of registered libraries when %PRODUCTNAME is restarted at a future time.

Deregistration

Deregistration of a macro library contained in an UNO script package is similar to the registration process described above, the Package Manager subsystem informs the LanguageScriptProvider that has been initialised with the appropriate installation deployment context that a macro library has been removed by calling its `removeByName()` method. The LanguageScriptProvider removes the macro library from its persisted store of registered macro libraries.

Implementation of LanguageScriptProvider with support for Package Manager

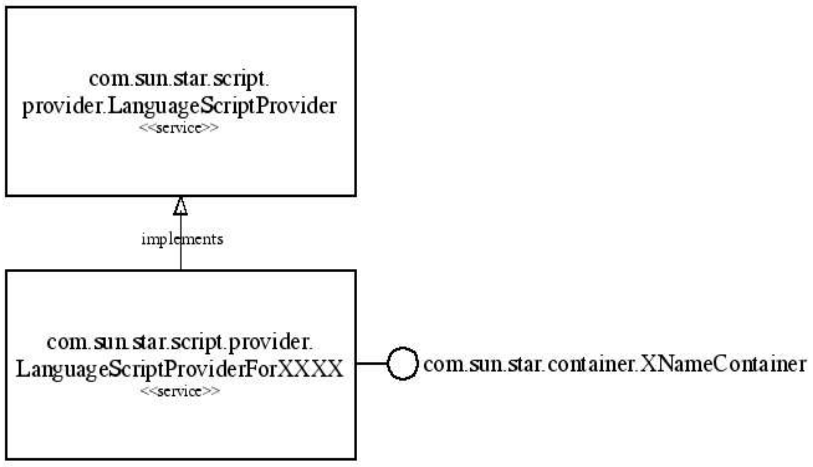

com.sun.star.script.
provider.LanguageScriptProvider
<<service>>
implements
com.sun.star.script.provider.
LanguageScriptProviderForXXXX
<<service>>
com.sun.star.container.XNameContainer

Illustration 1.10: LanguageScriptProvider

In order for the LanguageScriptProvider to handle macro libraries contained in UNO packages with media type `"application/vnd.sun.star.framework-script"` it's `initialize()` method must be able to accept a special location context that indicates to the LanguageScriptProvider that it is dealing with UNO packages.

Location context	
`“user:uno_packages”`	`string. Denotes the user installation deployment context.`
`“share:uno_packages”`	`string. Denotes the share installation deployment context.`

On initialization the LanguageScriptProvider needs to determine what macro libraries are already deployed by examining its persistent store.

Tip: LanguageScriptProviders created by implementing the abstract Java helper class `com.sun.star.script.framework.provider.ScriptProvider` do not need to concern themselves with storing details of registered macro libraries in UNO packages. This support is provided automatically. An XML file called unopkg-desc.xml contains the details of deployed UNO script packages . This file located in either *<OfficePath>/user/Scripts* or *<OfficePath>/share/Scripts* depending on the installation deployment context. The DTD for unopkg-desc.xml follows

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD for unopkg-desc for OpenOffice.org Scripting Framework Project -->

<!ELEMENT package EMPTY>
<!ELEMENT language (package+)>
<!ELEMENT unopackages (language+)>
<!ATTLIST language
        value CDATA #REQUIRED
>
<!ATTLIST package
        value CDATA #REQUIRED
>
```

An example of a sample an uno-desc.xml file is shown below.

```
<unopackages xmlns:unopackages="unopackages.dtd">
  <language value="BeanShell">
    <package value="vnd.sun.star.pkg://vnd.sun.star.expand:$UNO_USER_PACKAGES_CACHE%
2Funo_packages%2Flatest.uno.pkg/WordCount" />
  </language>
  <language value="JavaScript">
    <package value="vnd.sun.star.pkg://vnd.sun.star.expand:$UNO_USER_PACKAGES_CACHE%
2Funo_packages%2Flatest.uno.pkg/ExportSheetsToHTML" />
    <package value="vnd.sun.star.pkg://vnd.sun.star.expand:$UNO_USER_PACKAGES_CACHE%
2Funo_packages%2Flatest.uno.pkg/JSUtils" />
  </language>
</unopackages
```

A LanguageScriptProvider that does not use the Java abstract helper class `com.sun.star.script.framework.provider.ScriptProvider` will need to persist the UNO packages deployed for the supported language themselves.

The LanguageScriptProvider additionally needs to support the `[IDL:com.sun.star.beans.XNameContainer]` interface which supports the following methods.

```
void insertByName( [in] string aName,
                   [in] any aElement )
void removeByName( [in] string Name )
```

On registration of an UNO package bundle for scripts the LanguageScriptProvider's `insertByName()` method is called with `aName` containing the URI to a macro library contained in an UNO package and `aElement` contains an object implementing `[IDL:com.sun.star.deployment.XPackage]` Note: the URI contains the full path to the macro library contained in the UNO package including the path to the UNO package itself. For more information please see <Chapter on Package Manager API>

On deregistration of an UNO package bundle for scripts the LanguageScriptProvider's `removeByName()` method is called with `aName` containing the URL to a macro library to be de-registered.

`[IDL:com.sun.star.container.XNameContainer]` interface itself inherits from `[IDL:com.sun.star.container.XNameAccess]` which supports the following method

```
boolean hasByName( [in] string aName )
```

To determine whether the macro library in an UNO package is already registered the LanguageScriptProvider's `hasByName()` is called with `aName` containing the URL to the script library. The other methods of the interfaces inherited by `[IDL:com.sun.star.container.XNameContainer]` are omitted for brevity and because they are not used in the interaction between the Package Manager and the LanguageScriptProvider. A Developer however still must implement these methods.

Implementation of the BrowseNode service

The LanguageScriptProvider created for an installation deployment context needs to expose the macro and macro libraries that it is managing. How this is achieved is up to the developer. A LanguageScriptProviders created by extending the Java abstract helper class `com.sun.star.script.framework.provider.ScriptProvider` creates nodes for each UNO package that contain macro libraries for the supported language . Each UNO package node contains the macro library nodes for the supported language and those nodes in turn contain macro nodes.

An alternative implementation could merge the macro libraries into the existing tree for macro libraries and not distinguish whether the macros are located in an UNO package or not. This is loosely the approach taken for %PRODUCTNAME Basic.

Example of creating a Package containing a macro library suitable for deploying with Package Manager.

The following example shows how to create an UNO package from the Beanshell macro library Capitalise. This macro library is located in the *<OfficeDir>/share/beanshell/Capitalise* directory of a %PRODUCTNAME installation . The UNO package created will be deployable using the Package Manager dialog or the unopkg command-line tool.

First create a scratch directory for example called *temp*. Copy the macro library directory and its contents into *temp*. In *temp* create a sub-directory called META-INF and within this directory create a file called manifest.xml.

```
<Dir> Temp
|
|-<Dir> Capitalise
|   |
|   |--parcel-desc.xml
|   |--capitalise.bsh
|
|-<Dir> META-INF
    |
    |--manifest.xml
```

The contents of the manifest.xml file for the Capitalise macro library are as follows

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE manifest:manifest PUBLIC "-//OpenOffice.org//DTD Manifest 1.0//EN" "Manifest.dtd">
<manifest:manifest xmlns:manifest="http://openoffice.org/2001/manifest">
 <manifest:file-entry manifest:media-type="application/vnd.sun.star.framework-script" manifest:full-
path="Capitalise/"/>
```

```
</manifest:manifest>
```

Next create a zip file containing the contents (but not including) the *temp* directory. The name of the file should have the extension ".uno.pkg" e.g. Capitalise.uno.pkg.

Deploying a macro library contained in an UNO Package.

To deploy the UNO package created you need to use the **Tools - Package Manager** dialog or the unopkg command-line tool. For full details on Package Manager dialog and the unopkg command-line tool please see [Chapter.X.Y.Z]. Once the package has been deployed successfully the macro will be available for assignment or execution.