# Scripting Mozilla Applications with XPCOM and XUL

Bachelor thesis for course
1076 IT-Spezialisierung E-Commerce: Vertiefungskurs VI WS 2008/09

Ao. Univ.-Prof.
Mag. Dr. Rony G. Flatscher

Institute for Management Information Systems

Andreas Mitschek (0251215)

*Supervising tutor:* Mag. Dr. Rony G. Flatscher

## *Abstract*

This bachelor thesis aims to explain the working principles of Mozilla's extensive component and interface architecture, XPCOM. Nutshell code examples serve as a mean to explain the underlying mechanisms and show the practical use of the different programming and scripting languages. Finally, XPCOM services are accessed and manipulated from within different language environments. The programming language Java and its several bridges to other technologies play a crucial role throughout this thesis.

# Table of Contents

1 Motivation and Interest ... 5
2 The Mozilla Suite ... 6
2.1 The story of Mozilla ... 6
2.2 XPCOM – the backbone of Mozilla applications ... 9
2.2.1 The Advantages of XPCOM ... 9
2.2.2 XPCOM component architecture ... 10
2.2.3 Interfaces ... 11
2.2.4 Criticism ... 13
2.3 A comparison between Java and Mozilla ... 14
2.3.1 Introduction to JAVA ... 14
2.3.2 Application Interfaces comparison ... 16
2.3.3 Conclusion ... 18
2.4 XPCOM-related technologies ... 20
2.4.1 XPIDL ... 20
2.4.2 XUL ... 21
2.4.3 XBL ... 23
2.4.4 XPConnect ... 25
2.4.5 XPT ... 27
3 Building powerful XPCOM applications ... 28
3.1 Bean Scripting Framework ... 28
3.1.1 Working with BSF ... 30
3.1.2 Working with Beans ... 34
3.2 Open Object Rexx ... 36
3.2.1 Introduction to Open Object Rexx ... 36
3.2.2 Working with ooRexx ... 37
3.2.3 BSF4ooRexx ... 39
3.2.4 Working with BSF4ooRexx ... 40
3.3 JavaXPCOM and XULRunner ... 44

3.3.1 Working with JavaXPCOM ..... 45
3.3.2 Implementing JavaXPCOM with ooRexx ..... 49
3.4 XULRunner Applications ..... 53
3.4.1 The Javascript Application Content ..... 53
3.4.2 The XULRunner Packaging Process ..... 57
3.4.3 Extending the Application with JavaXPCOM ..... 59
4 Conclusion ..... 67
5 Literature ..... 68
6 List of Figures ..... 71
7 Project Management ..... 73
8 Appendix ..... 74
8.1 XUL examples ..... 74
8.2 XBL examples ..... 76
8.3 BSF examples ..... 78
8.4 JavaXPCOM examples ..... 84
8.5 XULRunner Javascript examples ..... 86
8.6 XULRunner ooRexx examples ..... 89

# 1 Motivation and Interest

Since the success of Web 2.0 in recent years internet users around the globe feel the growing need to customize and personalize their web experience. The open source project Mozilla was among the first software companies to recognize this desire and along with the paradigm of Cross Platform Component Modeling (XPCOM) made it possible for the average programmer to include themselves into the development process. By allowing everyone to inspect the source code of their applications and setting new interface standards it became much easier to integrate new components into complex applications.

The impossible task of creating software that meets everyone's needs was replaced by the idea of involvement and customization. By developing and selecting individual modules and components the customer can modify applications like the Mozilla browser Firefox or the E-Mail client Thunderbird to his personal needs.

This thesis will evolve around the idea of the cross platform component modeling – (XPCOM), which made it possible for the average "home developer" to include new ideas and services into their favorite applications, using complex interfaces provided by the Mozilla development center. The primary goal of this work will be the development, documentation and testing of "nutshell" examples, small applications, to illustrate the available features of the Mozilla architecture.

# 2 The Mozilla Suite

This chapter introduces the open source project Mozilla. On one hand Mozilla is a well known software suite, popular for providing a diversity of free licensed applications. On the other hand it is a development platform, which uses unique technologies for its functionality. It is not a separate programming language, but a collection of open source software components, which can be assembled and manipulated to meet your goals.

The technologies, which are unique to Mozilla are first and foremost the *Cross-Platform Component Object Model (XPCOM)* and the graphical rendering engine *Gecko*, both are open source and will be explained in more detail in this chapter.

## 2.1 The story of Mozilla

Mozilla’s history reaches back to the early years of the internet and the “browser war” between the Netscape Navigator and the Microsoft Internet Explorer in the 1990s. Netscape was at that time a key player in the browser sector, but Microsoft managed to drastically increase its market share due to several smart, but harshly criticized measures. The most successful and legitimate step was the inclusion of the Internet Explorer as the standard web browser for Microsoft’s platform Windows, at that time and still to this day the most used operating system in the world.

After the fall of the Navigator, Mozilla was created in 1998 by the Netscape Communications Corp. to coordinate the community's open source development of the Netscape source code. Four years later, in 2002, Mozilla 1.0 was presented to the world and made public under the GNU General Public License (GPL) [InNe09].

Since then, the popularity and use of Mozilla applications soared, and due to the large community of developers, testers and users products like Firefox and Thunderbird are threatening the market leadership of Microsoft’s counterparts, the Internet Explorer and Outlook.

## The success of Mozilla

Due to the success of its most famous product, the open source web browser Firefox, the Mozilla Suite quickly gained popularity and market share. In 2004, for the first time in nearly a decade, the Mozilla browser nudged the Internet Explorer below the 90 percent mark.

The most important factor in Mozilla's success was that the underlying architecture was completely open source. With the help of an avid and ambitioned development community bug-fixing and the coding of new add-ons brought a sense of customizability that no other browser could offer at the time. The Mozilla foundation correctly detected the customer's need for personalization and individualized software.

As of November 2008, Firefox officially climbed the 20 percent mark and there seems to be no end to this development. Although the browser giant Internet Explorer is still the clear market leader (69.77%), Firefox (20.78%) and Apple's Safari (7.13%) are steadily increasing their market shares over the years [MaSh09].

**Fig. 1: Firefox closes the market share gap to the Internet Explorer (11/2008) [MaSh09]**

Mozilla's CEO John Lilly about the rising success of the web browser Firefox: "Reaching 20 percent worldwide market share is a significant milestone for Firefox and Mozilla. It's a huge achievement by the global Mozilla community, one that just a few years ago most would have considered impossible. The open web is more vibrant than ever, and the thousands of Mozilla contributors around the world have played a major role in making it that way [BrMa09]."

**Fig. 2: In some countries, Firefox is already the most popular browser (11/08) [MaSh09]**

According to the latest statistics from late 2008, provided by Net Applications (see figure 2), Firefox has already established itself as the number one web browser in some European and Asian countries, in three different countries the Mozilla browser even surpassed the 50 percent mark. Indonesian (57.44%), Macedonian (53.13%) and Slovenian (51.99%) internet users are opting to use Firefox as their standard web browser [MaSh09].

## 2.2 XPCOM – the backbone of Mozilla applications

The *Cross Platform Component Object Model* (XPCOM) "is a framework which allows developers to break up monolithic software projects into smaller modularized pieces. The pieces, known as *components*, are then assembled back together at runtime [OvXP09]." Its structure is related to existing Component Object Models like CORBA and COM. Before we dive into the code parts it would be best to get to know the XPCOM-related technologies.

### 2.2.1 The Advantages of XPCOM

The goal of XPCOM and all other Component Object Models is setting up a standardized framework of components that ease the development of new software modules. Every programmer can then pick the pieces he needs to change existing code or write new applications. XPCOM also provides several methods to load, manipulate and maintain these components. It enables developers to write modular cross-platform code, existing components can then be easily replaced or upgraded without breaking the entire application. No one has to reinvent the wheel, which has the following advantages: [MoXP09]

- Reuse: Modularized code can be reused for other purposes than the original developer intended.
- Updates: A developer can easily replace or update certain parts without recompiling the entire application.
- Performance: Some modules may not be used all the time. They can be "lazy loaded" to improve the overall performance.
- Maintenance: Finding and maintaining certain areas of the code is much easier when the application is divided into components.

**Gecko**

XPCOM is in a lot of ways similar to Microsoft COM, except that XPCOM is used mainly at the application level. XPCOM is part of the *Gecko* layout engine, so its services are actually Gecko services, therefore those two terms are often used as synonyms.

Gecko is an open source embeddable web browser, but it is also a rendering engine for documents and a toolkit to create new web browsers. It is used in many internet applications and is at the moment the most popular and successful open source web browser [OvXP09].

### 2.2.2 XPCOM component architecture

Applications like Firefox and Thunderbird are modularized clients of XPCOM components. Almost any kind of services you would need to write new applications are already defined in XPCOM components, e.g.:

- Browser-Navigation
- Window Management
- Cookie Management
- Bookmarks
- Address books
- Security
- Searching
- Rendering
- Operating System

For example, accessing the cookie manager can be as easy as the following lines of JavaScript code:

```
// get the cookie manager component in JavaScript

var cookieManager =
Components.classes["@mozilla.org/cookiemanager;1"].getService();


cookieManager =
cookieManager.QueryInterface(Components.interfaces.nsICookieManager);
```

And now you can work with this XPCOM component, be it in JavaScript, C++, Java or any other programming language that offers a bridge to XPCOM. You will find more on the practical aspects behind these lines of code in the following chapters. First, let us take a look at all the involved technologies surrounding XPCOM.

If you are new to Mozilla and want to find out more about the available components you should probably take a look at the XPCOMViewer at the Mozilla Development Center mozdev.org [MoXV08]. This tool enables you to browse through all supported XPCOM interface and serves as an API information tool. In the spirit of Mozilla development, it is written as an open source extension.

### 2.2.3 Interfaces

Now that we know what the advantages of a component-based architecture are, we can talk about linking those components. After we have determined which functionalities should be grouped together, it is the *interfaces* that manage the communication between components. Several interfaces are usually bundled to a single service component.

Like in every other programming language calling certain interfaces provides your application with more functionality, but there is a major difference between XPCOM and object-oriented programming languages. A programming language looks for the necessary components and interfaces while compiling the source code, XPCOM does not look for those functionalities until the start of the runtime. In this respect it is much closer to the working principles of a scripting language.

Understanding how exactly components and interfaces work together is a very complex task and would be too much for this work, but there are some issues that we should know of. In XPCOM all interfaces derive from the “root interface” called *nsISupports* and inherite certain methods from it. In general, *nsISupports* has a monitoring role and controls administrative tasks like:

- *Component lifetime*
- *Interface querying*

Since every XPCOM component can use a variety of interfaces it is very important to know how they are connected. For this reason XPCOM components use *reference counting* to determine how often an instance of a certain component is being used. If

the reference count of a component hits zero – which means that it is no longer used by any client – the component deletes itself. This is known as *component lifetime* or *object ownership.*

By using the nsISupports function *queryInterface*(), like we did in the above JavaScript code example, you can instantiate basically any implemented interface. Every component and interface is identified by unique keys.

### Contract ID vs. IID/CID

Interfaces can be addressed by name (contract ID) or by their IID/CID. The universally unique identifier (UUID) is a unique 128 bit number. In the case of XPCOM interfaces it is called interface identifier or IID, when we talk about components or classes it is called CID. But as a programmer it is much easier to address certain functionalities by their contract ID, which is a human readable string. In our first example we introduced the initialization of a cookie manager instance, identified by the

- Contract ID: *“@mozilla.org/cookiemanager;1”*
- IID: *“aaab6710-0f2c-11d5-a53b-0010a401eb10”*
- Interface name: *“nsICookieManager”*

### Component Registration

Before an application is able to access a new component it has to be registered. It does not matter whether the component is stored in shared libraries, JavaScripts or other files. The registration process provides all the information needed to use the new component properly. As a reminder, all new XPCOM components must implement the root interface, nsISupports, before being registered. In earlier Mozilla versions the registration process was done by a program called *regxpcom*, but it is now also possible to register the component using XPIDL or have the user install the component directly as an extension or add-on.

### 2.2.4 Criticism

Of course XPCOM does not only bring advantages, there are some things developers are moaning about for years. For example the “code bloat” in XPCOM-based systems. Mozilla has a rather large memory footprint. There is a lot of code involved in managing and monitoring all the components and objects, which could lead to a decrease in performance, especially in large applications. This is the main reason why Apple for example chose KHTML over Gecko for their Web browser.

At the moment Gecko developers are working on reducing the extensive use of XPCOM in the Gecko layout engine. This kind of work is - as a play of words - also known as “deCOMtamination” [Mill06].

## 2.3 A comparison between Java and Mozilla

The goal of this thesis is to show how a developer can link XPCOM-related technologies with existing programming and scripting languages to create simple, but powerful applications. This will be demonstrated with a series of nutshell examples. Ultimately XPCOM interfaces should be implemented in a Java application. The bridge, which will enable communication between Java and XPCOM, is called JavaXPCOM [OvXP09].

This section includes a comparison between the two main technologies behind JavaXPCOM, the programming language Java and Mozilla. After some background information on Java we will compare the functionalities and concepts of Java and Mozilla according to the following criteria:

- Java and XPCOM API
- Graphical user interfaces
- Multithreading
- Portability

### 2.3.1 Introduction to JAVA

Java is a multi-platform object-oriented language and is used for several types of (web-based) applications. It is generally known as the “language for the web” and for its great portability.

James Gosling created the Java language in June 1991, with the importance of easy portability already in mind. Java, first known as “Oak”, had a similar notation as C/C++. With Java 1.0 the first release was presented to the public in 1995. The internet and the development of Java 2 lead to its explosive rise in popularity.

In the Java creation process those were the primary goals: [NaMa09]

1. It should use the object-oriented programming methodology.
2. It should allow the same program to be executed on multiple operating systems.
3. It should contain built-in support for using computer networks.
4. It should be designed to execute code from remote sources securely.

5. It should be easy to use by selecting what were considered the good parts of other object-oriented languages.

**Use of Java**

Java earned the reputation as the "language for the web" primarily because of *Applets* – small programs that can be executed inside a web page. Although over the years, Applets have revealed their limits and are being pushed aside by technologies like Macromedia Flash.

*"Write once, run everywhere*", is the slogan most commonly associated with Java, which is being developed since 1997 by Sun Microsystems. New code can be written on any platform, compiled into a standardized byte code and expected to run on any system with a Java virtual machine (JVM), which ultimately interprets the generated byte code. The existence of a JVM or Java interpreter on desktops, laptops, mobile devices and chips has long become an industry standard.

The programmer no longer has to worry about the portability of applications, it does not matter whether the code is being accessed on a UNIX or Windows system, or the type of machine (desktop, router, mainframe, etc.). This feature guaranteed Java's large success, especially on the sector of mobile devices.

Another important reason in favor of Java is its technology pervasiveness. When Java lost the fight for client-side applets, the development of the Java 2 Enterprise Edition (J2EE) introduced a new opportunity, server-side development. J2EE provides a framework for secure and scalable applications and presents a counter-part to ASP and PHP, making it possible to write stand-alone programs, application plug-ins, applets for web pages and web applications using:

- *Applets* (Java programs embedded in a web page)
- *Servlets* (Java code with embedded HTML tags)
- *Java Server Pages* (HTML with embedded Java expressions)
- *Enterprise JavaBeans (*build the application logic in multi-tiered architectures)

Since the fall of 2006 Sun Microsystems Java is registered under the GNU General Public License (GPL) and is completely open-source. At http://www.sun.com/ it is

possible to download the standard Java Runtime Environment (JRE) and the Software Development Kit (SDK), targeted at Java programmers [Carb06].

### 2.3.2 Application Interfaces comparison

**Java API**

Both Java and Mozilla provide a rich set of APIs. While Java is used in many different domains, Mozilla interfaces usually specialize in expanding the functionality of web applications.

The Java SDK API provides amongst others the following functionalities:

- GUI
- Images
- Input/output on file/network/device
- Network Datagram, Sockets
- Remote Method Interface
- Security
- Text manipulation
- Sound
- XML parsing
- Abstract database connections

**XPCOM API**

Contrary to Java, Mozilla provides a language neutral API. New XPCOM Components can be written in several programming languages, as long as the interfaces are defined with a common Mozilla Interface Description Language, XPIDL.

The XPCOM API provides amongst others the following functionalities: [OvXP09]

- Access to web platform components like bookmarks, address book, etc.
- Collections, Sets, Dictionaries
- Directory service LDAP
- Mail

- Network
- RDF and XML
- SOAP, XML-RPC and WSDL.

**Graphical user interfaces**

Java offers two different GUI toolkits, the simple but fast AWT and the modern Swing components. The main advantage of AWT is its better performance, but in terms of native components and functionalities Swing is the much better choice. The Swing API is far larger than the older AWT and it provides a full object oriented architecture. Therefore a Java GUI is a selection of AWT/Swing components within a given Java code.

Mozilla, coming from a browser background, uses a completely different approach. The GUI is defined in XUL, a XML-like markup language. So, like the markup language HTML, it is possible for Javascript or other scripting languages to navigate through and manipulate the DOM tree. The Document Object Model (DOM) is the hierarchical order of all elements of a website, from the root window down to the attribute of a text label.

The layout engine Gecko is later used to render either HTML or XUL pages. Mozilla uses the Gecko engine to render both the GUI and the content of the web pages. Other applications, like e.g. Camino, use Gecko only for their web pages and rely on native GUI APIs.

**Multithreading**

Another important aspect of Java is multithreading, to run and manage different activities within runtime. This feature is handled by the Java virtual machine and theoretically enables multithreading even in systems that do not support it. Although today every system and operating system kernel is able to manage multiple activities or threads. The keywords wait, notify and synchronized realize this feature in Java.

Mozilla on the other hand relies on the use of object-oriented programming languages to realize multithreading. Therefore managing multiple activities is solely done in the respective source code. If we talk about a simple standalone program involving only XUL, XBL and JavaScript, then multithreading is very hard to achieve without bridges to other languages. JavaScript offers workarounds such as timers to

solve this problem in small programs, although this kind of programming/scripting is not recommended [Carb06].

### 2.3.3 Conclusion

There is no short answer to the question whether to use Java or Mozilla for your applications, the choice rather depends on what is really important for your program.

Comparison between Java and Mozilla

| | Mozilla | Java |
|---|---|---|
| Desktop | Good | Good |
| Mobile | Poor | Excellent |
| Server | Absent | Good |
| Web | Excellent | Good |
| License | Excellent | Poor |
| Community | Good | Good |
| Tools | Good | Excellent |
| Documentation | Good | Excellent |
| Portability | Good | Good |
| Multithreading | Poor | Good |

**Figure 3: Short comparison between Mozilla and Java [Carb06]**

Java is a programming language that enables you to write complex programs, be it for mobile clients, desktop clients or server-side programming. The included virtual machine is great as a layer between the source code and the underlying system and therefore guarantees great portability. But recent years showed that XML is getting more important in terms of internet technologies, covering areas that Java cannot handle.

Due to its modularity and many open source technologies like XPCOM and XUL Mozilla applications are getting key players in many internet domains. Mozilla is open to inter-operability and is language neutral, which means that components can be written and used by many different programming languages including Java. The internet is no longer a collection of documents, but rather a collection of services. And component-based applications are able to provide rich graphical user interfaces and the functionalities to access a variety of those web services.

Luckily, programmers do not need to make a choice between Java and Mozilla, since JavaXPCOM (bridge between Java and XPCOM) already combines those two technologies under one hood. In the following chapter we will learn how to write XUL GUIs, bring them to life using scripting languages and access XPCOM components in Java [Carb06].

## 2.4 XPCOM-related technologies

Mozilla supplies its development community with several proprietary and open source technologies, which aid you in the process of creating new Mozilla extensions or manipulating existing components. Furthermore, you can create interactive user interfaces with small effort.

### 2.4.1 XPIDL

Mozilla uses the *Cross-Platform Interface Description Language* (XPIDL) to define and manage all its interfaces. XPIDL stems from the *CORBA OMG Interface Definition Language* (IDL) and allows to define methods, attributes, constants and even the inheritance behavior of an interface.

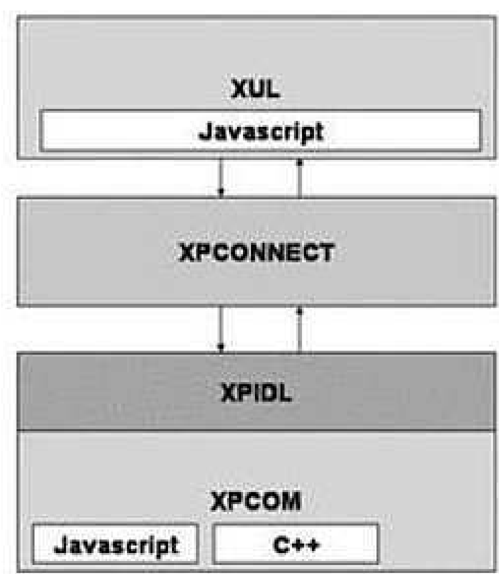



**Figure 4: XPCOM component architecture [PoHo07]**

XPIDL allows the programmer to generate *type libraries*, or *typelibs*. Those XPT files are a binary representation of an interface. When a programming language such as Java or C++ accesses an interface it communicates directly with its typelib to find out which methods and variables are available. This feature of XPCOM is called XPConnect and is the main layer between any programming or scripting language and the XPCOM core with its components and interfaces. In the following chapters we will learn more about XPIDL and its practical use, when dealing with XPCOM components [PoHo07].

### 2.4.2 XUL

The graphical user interfaces of all Mozilla applications are implemented in a XML-derivative called *XML User Interface Language* or XUL (spoken as “zool”). XUL is not a programming language, but a markup language like HTML or XML. As a markup language it is natively interpreted by Mozilla’s layout engine Gecko. Every visual part of Firefox or Thunderbird includes XUL-layouts, which determine the structure and design of the application front end. Those GUIs are then interpreted and rendered by Gecko.

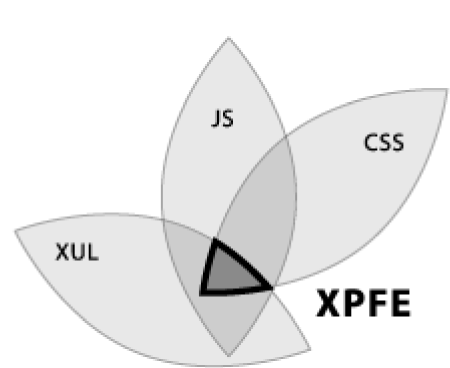



**Figure 5: Elements of an XPCOM GUI [PoHo07]**

Microsoft picked up on this concept and is expected to adopt a XUL-like markup language called XAML for its newest operating system Longhorn.

The graphical user interfaces are written in XUL, CSS provides the necessary design and Javascript handles the events of the GUI and brings the application to life. The scripts can communicate with the existing XPCOM architecture using the XPConnect layer [BuSm01].

An example for a XUL widget would be this short code sample.

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/global.css" type="text/css"?>

<dialog id="myDialog" title="My Dialog"
        xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
        buttons="cancel"

        buttonlabelcancel="Exit"
        buttonaccesskeycancel="E"
        ondialogcancel="doCancel();">

<script language="JavaScript" src="DialogScript.js"></script>
```

**Figure 6: Header information of a XUL document**

The XUL document begins like any other XML document with information about version and namespace. In line 2 is a reference to the standard style sheet provided by the Mozilla development center. The first XUL element is the *dialog*. The tag is structured like an HTML element. The tag name is followed by a series of attributes

identifying and creating the element at hand. The attribute *xmlns* includes the official XUL namespace, at least one namespace per XUL document is mandatory. *buttons* is a reference to the exit or cancel button, its behavior is documented in the last three attributes of the *dialog* element, the label is “Exit” and the action event triggers the JavaScript method doCancel(). At the moment there is no JavaScript included.

```
<menu label="Datei"  class="mymenu" >
        <menupopup>
        <menuitem label="Menu 1" class="mymenu" id="b1" oncommand="newTab('http://www.wu.ac.at/')" />
        <menuitem label="Menu 2" class="mymenu" id="b2" oncommand="newBookmarkFolder('XPCOM')" />
        <menuitem label="Menu 3" class="mymenu" id="b3" oncommand="doSomethingElse()" />
        </menupopup>
</menu>
</dialog>
```

**Figure 7: Body of a XUL document**

The actual body of the XUL document is a simple popup menu with three different menu items. Each item has its own label, ID, class name and action event. The document closes when the root tag element – in this case *dialog* – is closed. XUL documents can only be opened and rendered by Mozilla’s Gecko engine. In case you would open the document from Mozilla Firefox or by a double-click on the file, the following “web page” in Figure 8 would appear in the browser window.

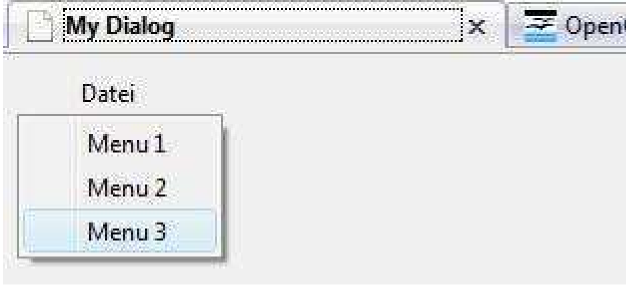



**Figure 8: Firefox displays the XUL document like this**

XUL elements are nice and easy to define, but they have no functionality whatsoever. This is the reason Mozilla provides its own scripting or binding language - XBL.

### 2.4.3 XBL

The *XML Binding Language* (XBL) is like XUL a XML-derived language. While as XUL only packs the components for graphical user interfaces, XBL is a way of dynamically changing those GUIs. When it comes to providing actual functionalities, XBL hands control over to more powerful scripting languages though. At the moment, developers have to work with XBL version 1.0, the coming version 2.0 is on its way to being standardized by the World Wide Web Consortium W3C.

So called bindings are attached to XUL elements and influence their behavior, for example the handling of certain events. This collection of bindings is stored in a separate XBL document and is integrated into the XUL document through CSS. In practice, this binding language is not very well known and has limited potential at the moment. The following short example shows how it is included into a XUL document.

```
<?xml version="1.0"?>
<?xml-stylesheet href="xbl.css" type="text/css"?>

<window
xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

    <box class="classname" />
</window>
```

**Figure 9: XBL example: The XUL document xbl.xul**

First the XUL root window is created. Note that the XML declaration includes our own style sheet *"xbl.css"* and not the standard reference *"chrome://global/skin/global.css"* like in the previous example. The *window* element only contains the XUL namespace and a *box* element with the class name "classname". The content of the box element is empty, but will later be filled by the corresponding XBL binding.

```
box.classname
{
    -moz-binding: url('xbl.xml#xblname');
}
```

**Figure 10: XBL example: The CSS document xbl.css**

The XBL document is integrated in the XUL document via the referenced cascading style sheet xbl.css. "*–moz-binding:*" links the *box* element with the class name "classname" to the binding *xblname* in the XBL document xbl.xml.

The content of the XBL file (Figure 11) is the actual code. The most important elements of all XBL documents are:

- <bindings>, the root tag of the XBL format. Contains namespace and the individual <binding> elements.
- Each <binding> is the definition of one XUL element.
- The <content> of such a binding declares further elements and behavior.

```
<?xml version="1.0"?>

<bindings xmlns="http://www.mozilla.org/xbl"
          xmlns:xul="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

  <binding id="xblname">
    <content>
        <xul:vbox style="margin:8px;">
                <xul:label value="Type a text:" />
                <xul:hbox>
                        <xul:textbox flex="8" />
                        <xul:button label="Enter"  oncommand="alert('enter');" />
                        <xul:button label="Clear"  oncommand="alert('clear');" />
                </xul:hbox>
        </xul:vbox>
    </content>
  </binding>

</bindings>
```

**Figure 11: XBL example: The XBL document xbl.xml**

The root tag *<bindings>* contains the namespaces for XBL and XUL elements and the different *<binding>* tags. In our case there is just one, the binding for the box element *classname*. The id attribute links the binding with the XUL component. The actual behavior is documented in the *<content>* tag (line 8 – 17). It creates a vertical box element, containing a label and a horizontal box with a text box and two buttons. Both buttons have action listeners (*oncommand*), which fire JavaScript functions, in this case only *alert().*

When you open the XUL document or when a certain user action triggers en event, the CSS file replaces the existing content of the XUL box with the content of the corresponding XBL binding. Such XBL files can be used to manipulate or create new parts of a XUL document on the fly, presenting a similar behavior like AJAX. The final result displayed by Mozilla Firefox 3.0 is displayed in Figure 12.

**Figure 12: XBL example: The final result of the XUL file call**

Now that we know how XUL/XBL interfaces gain access to scripts we should take a closer look at the underlying technology, which enables Javascript access to XPCOM components.

### 2.4.4 XPConnect

XPConnect is a XPCOM module and a bridge between Javascript and internal XPCOM interfaces. The script authorizes itself through XPConnect and enables access to the C++ code behind every interface. In fact, both languages do not need to know who is accessing their components, neither do they need to understand the other language, XPConnect works as a mediator. Likewise, the XPCOM components can also access Javascript documents. There are several third party development projects going on, aiming to extend the XPConnect functionality to other scripting languages. The authorization process of a Javascript can be as easy as adding one of the two following lines of code [MoSi09].

```
netscape.security.PrivilegeManager.enablePrivilege('UniversalXPConnect');
```

or

```
netscape.security.PrivilegeManager.enablePrivilege("UniversalPreferencesRead")
```

The Netscape security module grants the Javascript that calls one of the above functions access to internal XPCOM modules. What happens internally, is that Mozilla's Sign Tool creates a digital signature to the calling script. The signature and the responsible script are both temporarily stored in a Java archive. The access privileges are only granted for the actual scope of the calling Javascript functions, so this line of code has to be repeated in every separate function that wants to access XPCOM interfaces. On the front end of the application the user is asked to accept the signature of the script before access is granted. If you register your files with a digital signature before hand, this step can be avoided.

There is a reason for XPCOM to be sensible about which script requests the security privileges. Object-oriented programming languages like Java have several opportunities to hide and secure internal information. You can use key words like private/protected or data encapsulation to hide sensible information in external classes or interfaces. Plus Java code cannot be changed at runtime. All those security issues are not the case with Javascript or many other scripting languages, simply because they have other priorities than strict security. For our code examples

this is not an issue, we can simply access the script, knowing what it wants and what it is able to do. Would your *unsigned* Javascript try to get access to XPCOM components on another computer, it would fail the security checks.

To put these new ideas into perspective let us go back to the first XUL example in figure 7. The first menu item with the label “Menu 1” launches a Javascript function called doSomething() when it is clicked. The Javascript function in figure 13 is called and tries to create a new tab in the Mozilla Firefox window that is currently active. If no browser is opened yet, it will create a new browser instance with the requested target URL opened.

```
function doSomething() {
        var url = "http://www.wu.edu/";
        try {
                // get access to XPConnect module
                netscape.security.PrivilegeManager.enablePrivilege('UniversalXPConnect');
                // get access to XPCOM interface nsIWindowMediator
                var wm = Components.classes["@mozilla.org/appshell/window-mediator;1"]
                                   .getService(Components.interfaces.nsIWindowMediator);
                // get the current Firefox window and open URL
                var mainWindow = wm.getMostRecentWindow("navigator:browser");
                mainWindow.getBrowser().addTab(url);

        } catch (e) { alert(e); return false;}
}
```

**Figure 13: Javascript function doSomething adds a new tab to the current Firefox browser**

Let us take a look what happens exactly in the code in Figure 13. After the creation of var *url* we would request access privileges to XPConnect. At this point the end user is asked if he wants to trust the calling script, if no digital signature was created in advance. Through the *Components.classes* object we can access the component WindowMediator with the contract id *“@mozilla.org/appshell/window-mediator;1”* and use its service/interface *nsIWindowMediator* [CoCl08]. Like you may have guessed the interface is responsible to monitor and manipulate current browser windows and tabs. The method getMostRecentWindow() requests access to the currently opened browser window, if available. Then we can finally add our target URL as a new tab. Five lines of code that are able to achieve quite a lot.

### 2.4.5 XPT

Although we are now able to access XPCOM services there are some restrictions to what Javascript can do. We already talked about the precarious security issues, but it takes more than XPConnect to get access to an interface. Every time an internal XPCOM service is requested the script is handled by XPIDL, which checks if the requested interface exists in its own registry. In general, *Interface Description Languages* (IDLs) are used to describe interfaces in a language- and machine-independent way. IDLs make it possible to define interfaces which can then be processed by other applications.

When an interface is run through the XPIDL compiler, it produces an XPT or type library file, a language-independent representation of the interface code. Because XPConnect uses the information to allow scripts access to XPCOM interfaces, it is important to make sure they are generated and included with your code even if you are developing exclusively in C++, the native Mozilla XPCOM language. Not only is a big part of the browser implemented in Javascript, it is possible that in the future someone may wish to use scripts to interact with whatever components you created. If you did not register them with XPIDL, they basically do not exist for the rest of the development community.

# 3 Building powerful XPCOM applications

Our applications are now able to present a nice and clean user interface to the customer, via Javascript they come to life and using the XPConnect layer we can even access and manipulate XPCOM interfaces. But this is not enough for a full fledged software application. Due to security reasons we want our program to be implemented in a object oriented language, that offers us all the advantages that a scripting language like Javascript will never be able to. But as it stands now, we have no means to call Java classes or let them call our scripts. So it is time to introduce a bridge between Javascript and Java that will eliminate that barrier – the Bean Scripting Framework.

## 3.1 Bean Scripting Framework

"The Bean Scripting Framework (BSF) is a set of Java classes, which provide scripting language support within Java applications. It also provides access to Java objects and methods from supported scripting languages. [ApJa09]" That is the official definition of the BSF project. It provides a bridge between Java and most scripting languages, enabling a two-way communication. Java objects are encapsulated into beans, which can be accessed by any supported language. Vice versa, Java is able to process objects from other formats.

BSF was initiated by software giant IBM in 1999 to provide access to JavaBeans from other scripting languages. It was soon declared an open source project and was used both in IBM (e.g. Websphere) and Apache (e.g. Xalan) projects. In 2002, BSF was officially accepted as a subproject of Apache Jakarta, which deals with numerous open source technologies evolving around Java.

The current distribution of BSF is since late 2007 version 2.3, enabling access to the following scripting languages:

- JavaScript (Rhino)
- NetRexx
- Perl
- Python
- Tcl (using Jacl)

- XSLT Style sheets

It is possible to integrate more than those scripting languages using their respective BSF engines. This is the case for

- Open Object Rexx
- Groovy (Monkey)
- JLog
- JRuby
- JudoScript
- ObjectScript

**Installation & Use**

There are several ways to use the BSF classes. It can be used as a standalone, a class library or as part of an application server. Enabling access to BSF for the first two choices is very easy. The developer simply has to download the most current version of BSF's *bsf.jar* file from the Jakarta web page and include the archive in the classpath of the local machine, as well as the JAR archive for the scripting language you want to include in your Java application.

The advantages of BSF are pretty clear. First it bridges Java to your selected scripting engine. You therefore enable access to your application to end users, who might be familiar with simple languages like Visual Basic or Javascript, but have no real experience with object oriented programming languages. And it brings more possibilities for debugging the software at hand.

**Getting started**

The BSF architecture mainly exists of two components: BSFManager and BSFEngine. The BSFManager is used to access and execute scripts, BSFEngine loads and interacts with the scripts. First you have to import the BSF classes and instantiate the BSFManager.

```
import org.apache.bsf. *;
import org.apache.bsf. util.*;
```

```
BSFManager mgr = new BSFManager();
mgr.registerScriptingEngine("javascript",
        "org.apache.bsf.engines.javascript.JavaScriptEngine", null);
BSFEngine engine = mgr.loadScriptingEngine("javascript");
```

The newly created instance of the BSFManager registers the scripting language, using the name and full path of the package. Then the BSFEngine is instantiated using the registered scripting language. From this point on you have basically two ways of handling scripts. eval() evaluates the script and checks for errors. If you completely trust the script you can start it directly by calling exec().

### 3.1.1 Working with BSF

Now that we know a little bit about the underlying interfaces, we test it with our own Java class “ScriptInterpreter.java”. This Java class waits to be called, using the given argument as the file path of the script we want to include. To start it from the command line you have to call > *java ScriptInterpreter scriptname.js*. When the class is called without an argument, it assumes the script to be Javascript and execute a default script called *frame.js*. To instantiate the class from within Java you would have to call the constructor containing the script as the only parameter, e.g. *ScriptInterpreter si = new ScriptInterpreter (script).* This code example orients itself after one of the many BSF code samples, that come with the source code.

But first we have to include all necessary Java and BSF classes.

```
import java.awt.*;
import java.io.*;

import org.apache.bsf.*;
import org.apache.bsf.util.*;
```

**Figure 14: BSF-example: import of BSF classes**

The necessary path to include the BSFManager and the BSFEngine is *org.apache.bsf*, other utilities can be referenced by using the subdomain *.util.*

```
/* main() checks arguments (filename) and calls constructor
/***********************************************************/
public static void main (String[] args) throws Exception {

   if (args.length == 0) {
        scriptName = "frame.js";
    System.out.println ("Kein Skript uebergeben --> default: JavaScript "+scriptName);
  }
      else {
        scriptName = args[0];
    System.out.println ("Skript erfolgreich uebergeben --> "+args[0]);
        if (args.length > 1) System.out.println ("Error: Höchstens ein Parameter");
  }

  // initialize TestScript
  new ScriptInterpreter(scriptName);
}
```

**Figure 15: BSF-example: The main method of ScriptInterpreter.java**

When ScriptInterpreter.java is called, the main method in Figure 15 checks the given argument. Was no argument given (line 48), then the standard Javascript is called, did the main method receive more than one argument, the program exits due to misuse. The ideal scenario is one valid file name for the source of the script in line 44. The argument is stored as *String scriptName* and sent to the constructor method.

```
/* class constructor uses BSF to (evaluate and) launch a script
/**************************************************************/
public ScriptInterpreter(String scriptName) throws BSFException {

  // execute script
  try {
    BSFManager bsf = new BSFManager();

        //get script information
    String language = BSFManager.getLangFromFilename(scriptName);
    FileReader in = new FileReader(scriptName);
    String script = IOUtils.getStringFromReader(in);

        // launch script start
    bsf.exec (language, scriptName, 0, 0, script);

  }
      catch (BSFException e) { e.getMessage (); e.printStackTrace (); }
      catch (IOException e)  { e.getMessage (); e.printStackTrace (); }
}
```

**Figure 16: BSF-example: ScriptInterpreter's constructor method initializes BSFManager**

The first thing the constructor does is create a new instance of BSFManager called bsf. Lines 23 to 25 extract important information like the type of scripting language (*String language*) and reads the content via a FileReader to the *String script*. At this point it would be possible to instantiate the BSFEngine and check the validity of the script with *eval(),* but we elect to execute the script right away using *exec().* This method receives all the information we have about the script and launches it. Now let us take a look at what happens within the script.

```
/**      JavaScript Rhino imports Java packages
/****************************************************/
importPackage(java.awt);
importPackage(java.lang);
importPackage(java.net);
importPackage(org.apache.bsf);
```

**Figure 17: BSF-example: Import of Java packages within Javascript**

Again, we have to include all the Java and BSF classes we need in the script. In this we include BSF and Java's .awt, .lang and .net packages.

```
/**      Create GUI components
/************************************/
frame = new Frame("AWT Frame, implementiert in JavaScript");
panel = new Panel(new BorderLayout());
scroll = new Scrollbar(Scrollbar.HORIZONTAL);
```

**Figure 18: BSF-example: The construction of an AWT Frame within Java**

As you can see in Figure 18, we can now create Java components without any restrictions from within the script. We would even be able to do this, if we haven't had included the necessary Java packages. In that case, we would have to address the full path of the package every time. The creation of the main frame would have taken the following line of code: *var frame = new java.awt.Frame ("…");* I will not include the rest of the frame creation, since it is a pretty simple task of putting AWT components together.

You see that it is very easy to bridge a complex object oriented language like Java to a much simpler construct like Javascript. I mentioned that we can use pretty much any aspect of Java we would like. That is even the case for Event listeners, as shown in Figure 19.

```
/**      Add event handler to components
/*****************************************/
frame.addWindowListener( function(event, methodName) {
        if (methodName == "windowClosing") { System.exit(0);}
        if (methodName == "windowDeiconified") { label.setText("Willkommen zurück"); }
});

color.addItemListener( function(event, methodName) {
        if (methodName == "itemStateChanged")
        {
                switch (color.getSelectedIndex()) {
                        case 0: label.setBackground(Color.green); break;
                        case 1: label.setBackground(Color.blue); break;
                        case 2: label.setBackground(Color.red); break;
                        default: break;
                }
                label.setFont(new Font("Dialog", Font.BOLD, 14));
        }
});
```

**Figure 19: BSF-example: Event handling from within the script**

The approach to adding the event listeners is exactly the same as it is in Java. The target component is attached to an event listener (WindowListener, ItemListener, etc.). When the action is observed the inner function catches the event and processes it according to the script code. In our case we have two window listeners (line 37, 38), who close the application (event *windowClosing*) or change the content of the label element *label (*event *windowDeiconified*). The item listener is attached to the choice element *color*. When the event *itemStateChanged* is launched, the background color of the label element changes to the selected item in the choice box. Now we *pack()* the frame and *show()* it or *setVisible(true).* Figure 20 shows the final result, an responsive AWT frame written purely in JavaScript.

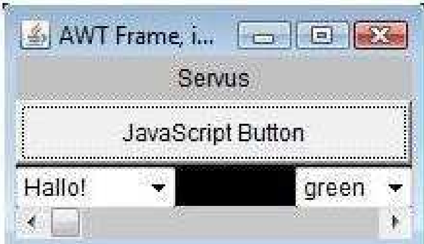



**Figure 20: BSF-example: AWT Frame in JavaScript**

When the choice elements at the bottom are selected, either the color of the label background (choice element on the right), or the content of the label element (choice element on the left side) will be changed (see Figure 21).

**Figure 21: BSF-example: AWT Frame in JavaScript**

The Javascript button in the center has a special use and will be the topic of the next chapter. Bean Scripting Framework is more than just the execution of a script in an unknown language. It is about communication and the exchange of objects. But before objects can be transferred to a foreign script they have to be encapsulated. Like the name of the BSF project probably hints, those encapsulated objects are known as *beans*.

### 3.1.2 Working with Beans

The idea for BSF beans evolved out of JavaBeans, which was the first step to encapsulate and transfer Java objects. BSF beans work in a very similar way. First they are added to a common registry, containing a key and a value. Then they can be loaded at any place you like. The registering as well as the loading can take place in Java or any scripting language you desire. We should take a few steps back to the constructor method of the original BSF example and include our first bean.

```
/* class constructor uses BSF to (evaluate and) launch a script
/*************************************************************/
public ScriptInterpreter(String scriptName) throws BSFException {

  // execute script
  try {
    BSFManager bsf = new BSFManager();

      // register Java String as BSF bean
    String btnLabel = "Label stammt aus Java";
    bsf.declareBean("btnLabel", btnLabel, String.class);

    String language = BSFManager.getLangFromFilename(scriptName);
    FileReader in = new FileReader(scriptName);
    String script = IOUtils.getStringFromReader(in);

      // launch script start
    bsf.exec (language, scriptName, 0, 0, script);

  }
      catch (BSFException e) { e.getMessage (); e.printStackTrace (); }
      catch (IOException e)  { e.getMessage (); e.printStackTrace (); }
}
```

**Figure 22: Bean-example: The new StringInterpreter constructor**

This is pretty much the same code as before, the only difference are the two lines that declare the BSF bean. First the *String btnLabel* is initialized containing a new value for our label element created with Javascript. The String is registered to the BSFManager using the method *declareBean()*, and is given a keyword to find the bean (“btnLabel”), the content of the bean (*btnLabel*) and the class type (String) as arguments. Alternatively, you could use *registerBean()* with almost the same parameters. The bean is now fully registered and can be accessed wherever we want.

```
button.addActionListener( function(event) {
        try {
                btnLabel = bsf.lookupBean('btnLabel');
                button.setLabel(btnLabel);
        } catch (e) {  }
});
```

**Figure 23: Bean-example: Javascript ActionListener accesses bean**

The ActionListener for our button contains only two lines of code, that change the content of the button label, which was "Javascript Button". As you can see it is not necessary to instantiate the BSFManager within Javascript as it was already created in the Java class constructor and is now available to any script that is loaded into the BSFManager or BSFEngine. We access the bean by calling the BSFManager function *lookupBean()* and can get it using the identifier we specified while declaring the bean. The imported bean contains the String "Label stammt aus Java", which will be changed to the new button label (see Figure 24).

**Figure 24: Bean-example: The Java String is now used within Javascript**

We have learned that with very few lines of code it is possible to connect the powerful programming language Java with many different scripting languages and avoid complicated and dangerous workarounds. But what if Javascript is not the scripting language our customers use, in fact they rely on a scripting format, which is not supported by BSF. In this case we would need another bridge to solve the problem. One example for such a scripting language is Open Object Rexx. We will learn what it is and how we can use it in combination with Java in the following chapter.

## 3.2 Open Object Rexx

We have learned how an object-oriented programming language like Java can interact with Javascript using the Bean Scripting Framework. It is also possible to connect Java applications with other scripting languages, like Open Object Rexx (ooRexx). ooRexx's main use is the automation of processes, dealing for example with component object models and applications running on different operating systems.

### 3.2.1 Introduction to Open Object Rexx

Open Object Rexx celebrates 30 years of information technology history this year. The predecessor “Restructured Extended Executor” (REXX) was introduced by IBM in 1979. It was created as a “human-centric” format and is a typeless language since its beginning. Operating systems like the Amiga OS or OS/2 use REXX as their standard scripting language. In 1987 it was redefined to be IBM's main procedural scripting language for their operating systems. REXX was standardized by the American National Standards Institute (ANSI) in 1996.

Due to the strong desire of the IT market to incorporate object oriented paradigms into their application, IBM developers started working on the open source version of REXX, Open Object Rexx (ooRexx). The final product was introduced as part of “OS/2 Warp” in 1997. It is built on an object model and has many object oriented features like multiple inheritance, but is at the same time backwards compatible with REXX. With the decline of IBM's success in the desktop market, REXX and ooRexx became increasingly less used [Fla02].

**Advantages of ooRexx**

ooRexx has a standardized API, which enables you to call ooRexx methods from within applications written in compiled programming languages. It is even possible to extend the existing ooRexx API using those languages. Combining procedural with object oriented script features, this human-centric language can solve many problems in a very effective way. ooRexx is especially used in the operating systems of many IBM mainframes. Therefore it is a vital component in connecting client operating systems like Windows or Linux with the existing mainframes [OnLa09].

**Getting started**

ooRexx comes in the form of a simple installer, Windows users can download an executable installer, users of other operating systems have to download the binary installer. The current version is 3.2, with 4.0 available as a beta version since April 2009.

ooRexx itself is implemented in C++ and has a very small, but powerful application programming interfaces. 18 different classes (Array, Class, Directory, etc.), that all inherit from the same root class “Object”. ooRexx messages are very simple and the typelessness of the language makes it even easier to work with. Printing a string to the command line can be as easy as:

```
say "Hello World"
```

As you can see, there is very little code involved. Every ooRexx statement is closed with a semicolon like in Java or Javascript, but in case it is missing the ooRexx interpreter automatically adds one at the end of every line [Fla02].

Within ooRexx everything is an object and every object can receive and send messages. Functions are called by using a message operator, the tilde character (~).

```
name = person~getName
```

There are several ways to structure the code. Certain keywords like function, call, do over, when then, etc. initiate methods, loops and conditional behavior. How they are used will be demonstrated in this chapter.

### 3.2.2 Working with ooRexx

RexxUtil is a set of ooRexx classes that allow the scripting and automation of operating system services and allow access to COM components. Later on we will use ooRexx to access Mozilla XPCOM components. The short script in Figure 25 accesses the Windows COM service “Windows Management Instrumentation” (WMI). It provides an operating system interface, through which scripts and programming languages gain access to internal information. Win32_Process is a WMI module that contains detailed information about all active processes [MiTe09].

```
objWMIService = .OLEObject~GetObject("winmgmts:\\.\root\CIMV2")
do objItem over objWMIService~ExecQuery("Select * from Win32_Process")
    say "_______________________________________________________________"
    say "Name:" objItem~Name
    say "Creation Date:" objItem~CreationDate
    say "Execution State:" objItem~ExecutionState
    say "Install Date:" objItem~InstallDate
    say "Parent Process Id:" objItem~ParentProcessId
    say "Priority:" objItem~Priority
    say "Process Id:" objItem~ProcessId
    say "Read Operation Count:" objItem~ReadOperationCount
    say "Virtual Size:" objItem~VirtualSize
    say "_______________________________________________________________"
end
```

**Figure 25: Rexx Script listing data about all active Windows processes**

All windows processes that are currently active are listed in *root\CIMV2\Win32_Process*, a copy of this database is stored to the ooRexx object *objWMIService*. The *do over* loop is similar to an *for each* loop, meaning that for every result of the SQL method *ExecQuery(“Select * ..”)* the steps until the keyword *end* are repeated. *ExecQuery(“Select * ..”)* is sent to our ooRexx object, launching the function with the corresponding name and extracting all results through the supplied SQL statement. *objItem* represents in every single loop iteration the current process object and displays their attribute values.

```
________________________________________________
Name: firefox.exe
Creation Date: 20090615185119.280819+120
Execution State: The NIL object
Install Date: The NIL object
Parent Process Id: 4320
Priority: 8
Process Id: 6096
Read Operation Count: 32403
Virtual Size: 343293952
________________________________________________
________________________________________________
Name: gvim.exe
Creation Date: 20090615185955.803819+120
Execution State: The NIL object
Install Date: The NIL object
Parent Process Id: 4320
Priority: 8
Process Id: 4788
Read Operation Count: 202
Virtual Size: 74326016
________________________________________________
```

**Figure 26: The Windows command line prints all active processes**

When the ooRexx script is started, it prints the requested information about the entire result set to the command line. Figure 21 shows one of the differences to other (scripting) languages. *The NIL object (.nil)* represents the same data type as *null* usually does.

ooRexx is not only suited to deal with the underlying operating system, it can also be used to automate processes of different applications. The Open Office applications for example are open source and provide a vast API to create or change Open Office documents. What ooRexx cannot do is communicate natively with Java. For this reason we need a bridge similar to BSF. This gap is closed by BSF4ooRexx.

### 3.2.3 BSF4ooRexx

The Bean Scripting Framework for Rexx (BSF4ooRexx) is an extension, which enables ooRexx to communicate with Java objects. This way any script can take advantage of the huge Java function library. Vice versa, any Java application can use ooRexx as a scripting language. The current version of BSF4ooRexx[1] is 4.0 and was released in October of 2009.

A set of Java classes and the external function package "BSF4ooRexx.dll" translate ooRexx code into Java-readable commands. Java field objects can be collected in a wrapper and accessed by ooRexx objects. Scripts using the BSF4ooRexx bridge are therefore fully portable to any system that has a Java and Rexx interpreter installed. At the moment BSF4ooRexx provides ready-to-go support for Windows and Linux [WURe09].

**Installation**

First make sure you have the ooRexx interpreter installed correctly on your system. The fastest way to do this would be to invoke a simple script from the command line. Download the installer source code[2] at the Vienna University of Economics and Business (WU). If you have an older version of BSF4ooRexx installed, uninstall the source code by invoking

```
uninstallBSF4ooRexx.cmd or uninstallBSF.cmd
```

in the Windows command line or executing

```
uninstallBSF4ooRexx.sh or uninstallBSF.sh
```

[1] The current version of BSF4ooRexx is also referenced as the „Vienna version of BSF4ooRexx", since the development took place at the Vienna University of Economics and Business. Earlier work on BSF4ooRexx were the "Augsburger" and "Essener" versions.

[2] Most current BSF4ooRexx Installer: http://wi.wu-wien.ac.at/rgf/rexx/bsf4oorexx/current/

on a Linux system. Unzip the installer archive *BSF4ooRexx_install.zip* and change into the target directory. The execution of the command

```
rexx setupBSF.rex
```

will create customized install scripts. Afterwards invoke either

```
installBSF.cmd or ./installBSF.sh
```

depending on the underlying operating system. In the last step call

`setEnvironment4BSF.cmd` or `setEnvironment4BSF.sh`

which will add the Java archives *bsf-rexx-engine.jar* and *bsf-v400-20090910.jar* (depending on the current version) to the classpath of your systems environment variables. If not, please provide your system with the necessary path information manually.

Test the successful installation by invoking rexx infoBSF.rex on your machine. You should now see detailed information about your ooRexx, BSF4ooRexx and Java installation [WURe09].

### 3.2.4 Working with BSF4ooRexx

Before we begin with the practical code example for BSF4ooRexx we should take a few steps back to chapter 3.1.1 (Working with BSF) and 3.1.2 (Working with Beans). In these code samples we implemented a Java AWT Frame completely within Javascript, including several event listeners. The script was invoked by a Java class, which created the BSFManager, a BSF bean and executed the script. We will repeat the same process for ooRexx and look at the language-specific differences between the two scripting engines.

```
/** Klassen-Namen und statische Klassen-Variablen */
BorderLayout = 'java.awt.BorderLayout'
Button = 'java.awt.Button'
Choice = 'java.awt.Choice'
Color = 'java.awt.Color'
Frame = 'java.awt.Frame'

EAST = .bsf~bsf.getStaticValue(BorderLayout, "EAST")
WEST = .bsf~bsf.getStaticValue(BorderLayout, "WEST")
SOUTH = .bsf~bsf.getStaticValue(BorderLayout, "SOUTH")
NORTH = .bsf~bsf.getStaticValue(BorderLayout, "NORTH")
```

**Figure 27: BSF4ooRexx-example: Definition of class names and constants**

The first step in figure 27 is optional and serves only to make the coming ooRexx code easier to read. We define all full class names, class constants and static variables we later need. To get to the constant variables of the respective classes we need to address the global *bsf* object and invoke the method *getStaticValue(class name, variable name).* Now we are able to build the frame.

```
/**     Create GUI components
/************************************/
frame = new Frame("AWT Frame, implementiert in JavaScript");
panel = new Panel(new BorderLayout());
scroll = new Scrollbar(Scrollbar.HORIZONTAL);
```

```
/** Komponenten erstellen */
frame = .bsf~new(Frame, "AWT Frame, implementiert in open Object Rexx ")
layout = .bsf~new(BorderLayout)
panel = .bsf~new(Panel, layout)

scroll = .bsf~new(Scrollbar, HORIZONTAL)
label = .bsf~new(Label, "Servus")
button = .bsf~new(Button, "ooRexx Button")

color = .bsf~new(Choice)
color ~~add("green") ~~add("blue") ~~add("red")
```

**Figure 28: BSF4ooRexx-example: The frame construction**

Figure 28 illustrates how BSF4ooRexx handles the invocation of Java methods. The method name is sent to the *bsf* object, supplying the full class name and optional values for the class constructor. The first line creates a new java.awt.Frame with the caption “AWT Frame, […]”.

```
/** Verschiedenste Event-Handler */
frame~bsf.addEventListener('window', 'windowClosing', 'call BSF "exit"')
frame~bsf.addEventListener('window', 'windowDeiconified', 'label~setText("Willkommen zurück")')

color~bsf.addEventListener('item', 'itemStateChanged', 'call changeColor color, label')
text~bsf.addEventListener('item', 'itemStateChanged', 'label~setText(text~getSelectedItem())')
button~bsf.addEventListener('action', 'actionPerformed', 'call switchLabel button')
```

**Figure 29: BSF4ooRexx-example: The ooRexx event listeners**

When the frame is completed, we can implement the event listeners. In Javascript we pretty much copied the Java style of writing event listeners, including the hidden functions. This step is done much faster in ooRexx. We simple send the method *addEventListener(a, b, c*) to the corresponding Java object*,* with

- **a** being the triggered event set,
- **b** being the specific event and
- **c** being the ooRexx statement/method to call.

The event listener for Frame *frame* would simply close the application, while the event listeners for the choice elements *color* and *button* invoke the ooRexx methods changeColor and switchLabel.

```
/** bsf.pollEventText ist der allgemeine Rexx-Listener */
do forever
        INTERPRET .bsf~bsf.pollEventText
        if result="SHUTDOWN, REXX !" then leave
end
exit
```

**Figure 30: BSF4ooRexx-example: The pollEventText Interpreter**

Figure 31 shows another difference to Javascript, or many other scripting languages for that matter. We include a loop which iterates until the termination of the program and interprets the current *pollEventText*. This makes it possible for us to catch the events like we did in figure 29. The keyword *leave* terminates the loop and *exit* closes the application.

```
/** EventListener-Handler für Choice-Feld "color" */
changeColor: procedure
        use arg color, label

        label~setBackground( .bsf~bsf.getStaticValue('java.awt.Color', color~getSelectedItem()))
        label~setFont(.bsf~new('java.awt.Font', 'Dialog', .bsf~bsf.getStaticValue('java.awt.Font', 'BOLD'), 14))
return

/** EventListener-Handler für Choice-Feld "color" */
switchLabel: procedure
        use arg button

        btnLabel = bsf('lookupBean','btnLabel')
        button~setLabel(btnLabel)
return

::requires BSF.cls
```

**Figure 32: BSF4ooRexx-example: ooRexx methods for event handling**

Here we see the implementation of the two event handler. The *procedure changeColor* is triggered by the choice component *color* and receives the object *color* and *label* as parameters. It sets the background color of *label* to the selected color and changes the font style of label to “Dialog”, “bold” and size “14”. The second event handler is *switchLabel* and receives the button *button*. By providing the method name and the bean name as arguments, we get a copy of the String created in Java. The second statement changes the value of the label to the received String. In the last line of our ooRexx programs we include all necessary BSF4ooRexx libraries. Every script that wants to access BSF4ooRexx functions has to include the main library BSF.cls via:

```
::requires BSF.cls
```

If you want to get access to the interfaces of Star Office/Open Office applications, you would have to include *UNO.cls*.

Getting into ooRexx can be a bit awkward at first, since many of its principles differ from the more popular scripting languages. But the possibilities you have in remote controlling operating systems and other applications are very useful. It is now possible for us to communicate with XPCOM (or COM) components from within Javascript and ooRexx and both languages are able to communicate with Java. But our Java applications are not yet able to call XPCOM interfaces directly. To do this we have to get to know XULRunner, a JavaXPCOM runtime environment.

## 3.3 JavaXPCOM and XULRunner

JavaXPCOM is a technology aiming to connect Java libraries with XPCOM components. The Mozilla Development Center (MDC) describes the bridge this way. “JavaXPCOM allows for communication between Java and XPCOM, such that a Java application can access XPCOM objects, and XPCOM can access any Java class that implements an XPCOM interface. JavaXPCOM is very similar to XPConnect (JavaScript-XPCOM bridge), and uses XPIDL [JaXP06]”.

It is available as part of the XULRunner project, which is development framework and runtime environment for JavaXPCOM components. In this chapter XULRunner will provide us with the necessary interfaces to communicate with the XPCOM architecture, while in chapter 3.4 we will build standalone applications with its help.

**Installation**

The current version[3] of XULRunner can be downloaded at the Mozilla Development Center[4]. At the moment XULRunner is available for Windows, Linux and Macintosh operating systems. When the download is completed unzip the source code to your target directory and execute –register-global (register for all users) or –register-user (register for this user) in the command line/shell. This should include the following .jar archives to your classpath:

- *XULRunnerDirectory*\javaxpcom.jar;
- *XULRunnerDirectory*\xulrunner\MozillaInterfaces.jar;
- *XULRunnerDirectory*\xulrunner\MozillaInterfaces-src.jar;
- *XULRunnerDirectory*\xulrunner\MozillaGlue.jar;

Furthermore, a new user environment variable called “GRE_HOME” (Gecko Rendering Environment path or GRE) with the path of the XULRunner application should be created. If not, please do this manually to get a dynamic access to the JavaXPCOM functions. This GRE path plays a crucial part in initiating the XPCOM embedding.

[3] The current XULRunner version is 1.9.2, for Mozilla Firefox 3.6

[4] XULRunner Download: http://releases.mozilla.org/pub/mozilla.org/xulrunner/releases/

To test the successful installation, try to invoke

```
xulrunner –version
```

to get the current XULRunner info dialog. By calling a XUL application through

```
xulrunner filename
```

the XUL document should open in a standalone frame, instead of in the browser [XuRu09].

### 3.3.1 Working with JavaXPCOM

Setting up the XPCOM environment for Java is more complex than accessing it from Javascript. The following code example describes the implementation of a startup application which loads an URL in an independent browser window. We will be using the following XPCOM interfaces:

- nsIServiceManager: The XPCOM service manager is one of the most important interfaces, through it we will reference all other necessary XPCOM services.
- nsIAppStartup: The interface is intended to be used as an application startup service. Applications are started with run().
- nsIWindowCreator: Gecko uses this interface to create new windows.
- nsIWindowWatcher: This interface is used to display and manipulate Gecko/DOM Windows. Must implement a nsIWindowCreator object.
- nsIDOMWindow: Primary interface for a DOMWindow object. Represents a single window outside the Firefox browser environment.

```
1    import java.io.*;
2    import java.net.*;
3
4    import org.mozilla.xpcom.*;
5    import org.mozilla.interfaces.nsIAppStartup;
6    import org.mozilla.interfaces.nsIDOMWindow;
7    import org.mozilla.interfaces.nsIServiceManager;
8    import org.mozilla.interfaces.nsIWindowCreator;
9    import org.mozilla.interfaces.nsIWindowWatcher;
```

**Figure 33: JavaXPCOM-example: Import of all necessary classes.**

We start by importing all necessary classes. First we get the Java packages *java.io* and *java.net*. *org.mozilla.xpcom.** enables us to set up the XPCOM environment with

the Mozilla singleton class. *org.mozilla.interfaces.** contains the specific XPCOM interfaces we later implement.

```
11  public class URLOpener {
12
13      /** Opens a new nsIAppStartup instance to create a new DOMWindow */
14      public static void main(String [] args) throws Exception {
15
16      URL targetUrl = null;
17      File grePath = null;
18      String startupCID = "@mozilla.org/toolkit/app-startup;1";
19      String startupIID = nsIAppStartup.NS_IAPPSTARTUP_IID;
20      String creatorIID = nsIWindowCreator.NS_IWINDOWCREATOR_IID;
21      String watcherCID = "@mozilla.org/embedcomp/window-watcher;1";
22      String watcherIID = nsIWindowWatcher.NS_IWINDOWWATCHER_IID;
```

**Figure 34: JavaXPCOM: The main method and CID/IID definition**

Our application "URLOpener.java" only contains a main method and is invoked through the command line, the web page which should be opened is supplied as the sole argument. How the requested URL (*targetUrl*) is verified is not included in the code example, since it has nothing to do with JavaXPCOM. *grePath* is the directory of the current XULRunner installation and is essential for XPCOM embedding. In the following lines of code we define all necessary contract ID's (CID) to request components, and IID's to instantiate the interfaces.

```
27      /** get GRE path grePath instance using the GREVersionRange */
28      // The range of supported XULRunner installations
29      GREVersionRange[] range = new GREVersionRange[1];
30      range[0] = new GREVersionRange("1.8", true, "1.9+", true);
31      // grePath = XULRunner Installation Dir
32      grePath = Mozilla.getGREPathWithProperties(range, null);
33  }
34  catch (FileNotFoundException e) { System.out.println("File Not Found"); }
35  catch (MalformedURLException e) { System.out.println("Malformed URL");  }
36
37  if (grePath == null) {
38    System.out.println("no GRE PATH found");
39    System.exit(0);
40    return;
41  }
```

**Figure 35: JavaXPCOM: Initialization of the Gecko Runtime Engine**

In Figure 36 we begin with the embedding process. The GREVersionRange array *range[]* contains the range of supported Gecko Runtime Engines (GRE). The Mozilla singleton class receives the range array and should return a valid GRE path. This singleton class will provide us later with all necessary methods to access XPCOM components. Can the GRE path not be found, then the application throws a "File Not Found Exception" and terminates.

```
49 /** Work with the validated GRE path */
50 try {
51     // get instance of JavaXPCOM class Mozilla
52     Mozilla mozilla = Mozilla.getInstance();
53
54     // try embedding the XPCOM environment into the Mozilla class
55     mozilla.initialize(grePath);
56     mozilla.initXPCOM(grePath, null);
57     System.out.println("--> JavaXPCOM initialized");
58
59     // Now we need to start an XUL application, so we get an instance of the XPCOM service manager
60     nsIServiceManager serviceManager = mozilla.getServiceManager();
61     // Now we need to get the @mozilla.org/toolkit/app-startup;1 service:
62     nsIAppStartup appStartup = (nsIAppStartup)serviceManager.
63                                 getServiceByContractID(startupCID, startupIID);
64     // Get the nsIWindowWatcher interface to the above
65     nsIWindowCreator windowCreator = (nsIWindowCreator)appStartup.queryInterface(creatorIID);
66     // Get the window watcher service
67     nsIWindowWatcher windowWatcher = (nsIWindowWatcher)serviceManager.
68                                     getServiceByContractID(watcherCID, watcherIID);
```

**Figure 37: JavaXPCOM: Using the Mozilla instance and XPCOM services**

Now that we have validated the GRE path we can create a new instance of the Mozilla singleton class. The new instance, *mozilla*, has to be initialized with the GRE path and the XPCOM embedding is complete when either mozilla.initXPCOM() or mozilla.initEmbedding() are called.

In line 60 we use *mozilla* to create a new Service Manager. There are basically two ways of getting hold of an XPCOM interface. Either using the Service Manager to create a new instance of the desired service by CID/IID or using *queryInterface(IID),* which is a method inherited by all XPCOM objects by the root interface nsISupports. If *queryInterface(IID)* is invoked, only a pointer to the requested interface is returned.

The Service Manager *serviceManager* will provide us with new instances of nsIAppStartup and nsIWindowWatcher. nsIAppStartup is the interface used to launch and quit different kind of applications and provides us with a pointer to nsIWindowCreator by querying the interface ID. Now we have created all necessary XPCOM objects to run the application.

```
71 // Set the window creator
72 windowWatcher.setWindowCreator(windowCreator);
73 // Create the root XUL window:
74 nsIDOMWindow win = windowWatcher.openWindow(null, targetUrl.toString(), "mywindow",
75                                  "chrome,resizable,centerscreen", null);
76 // Set this as the active window
77 windowWatcher.setActiveWindow(win);
78 // Hand over the application to xpcom/xul, this will block:
79 System.out.println("Opening "+targetUrl);
80 appStartup.run();
```

**Figure 38: JavaXPCOM-example: Running the startup application**

Before *windowWatcher* can open a new window it has to be initialized using the *windowCreator*. When this is done, we can open the new window with the provided URL and a set of window parameters. The result will be stored in the form of a DOMWindow, the simplest way to realize windows using Mozilla. We associate the windowWatcher with the DOMWindow *win* and run the application. Now the Java application will be blocked until the startup application terminates. To prevent this you would have to implement multithreading by using the XPCOM interface nsIEventQueue.

```
83          // XPCOM cleanup
84          System.out.println("JavaXPCOM shuts down -->");
85          System.gc();
86          mozilla.shutdownXPCOM(null);
87
88      }
89      catch (XPCOMException e) { e.printStackTrace(); }
90      }
91
```

**Figure 37: JavaXPCOM-example: XPCOM shutdown**

When the Java application is done using the XPCOM environment all resources have to be freed. Calling the garbage collector with System.gc() is optional, since Java should do this automatically. But the XPCOM embedding has to be shutdown. If you have used initEmbedding(), then termEmbedding() has to be called. Was XPCOM initialized using initXPCOM(), then shutdownXPCOM(Service Manager) ends the embedding process. When the application is invoked, a new window loads the requested target URL.

Now we are able to connect Java with several scripting languages and let the scripts communicate directly to Java using the Mozilla components. Furthermore XPCOM can be accessed directly by script. Usually a Mozilla application or extension gets started with a static XUL GUI and enabling it with Javascript. If your application needs Java in the backend, the script can call the necessary Java classes with the Bean Scripting Framework. Out of the box Javascript can only call the standard Java API. If you want to address your own Java classes from within your scripts, then you would have to pack your classes to a .JAR archive and include them into your application with a Java Class Loader. The use of such a class loader will be explained in chapter 3.4.3 “Extending the application with JavaXPCOM”.

This chapter described how the XULRunner installation embeds XPCOM into a native Java environment. In the next chapter we will learn how to implement a JavaXPCOM environment with Open Object Rexx.

### 3.3.2 Implementing JavaXPCOM with ooRexx

Most XPCOM implementations use Javascript and XPConnect to request certain services and enhance existing applications. We have already learned how such an implementation could be done using the Java programming language. Since Java can be connected to a vast number of scripting languages with interfaces like BSF or BSF4ooRexx, it should be possible to let Open Object Rexx (ooRexx) to interact directly with XPCOM components. In a way we are now connecting JavaXPCOM and BSF4ooRexx to make our scripts more powerful.

In the previous chapter about JavaXPCOM we worked with the “URLOpener” example, completely written in Java. When we translate the code into an ooRexx script it becomes obvious how similar it will look. Which should not be surprising, since we are using the same JavaXPCOM libraries. “URLOpener.rex” will have the same task as its Java pendant, to receive an URL and open it with XPCOM interfaces.

```
.bsf~bsf.import('java.io.File','File')
.bsf~bsf.import('java.lang.System','System')
.bsf~bsf.import('org.mozilla.xpcom.Mozilla','Mozilla')
.bsf~bsf.import('org.mozilla.xpcom.GREVersionRange','GREVersionRange')
.bsf~bsf.import('org.mozilla.interfaces.nsIAppStartup','nsIAppStartup')
.bsf~bsf.import('org.mozilla.interfaces.nsIDOMWindow','nsIDOMWindow')
.bsf~bsf.import('org.mozilla.interfaces.nsIServiceManager','nsIServiceManager')
.bsf~bsf.import('org.mozilla.interfaces.nsIWindowCreator','nsIWindowCreator')
.bsf~bsf.import('org.mozilla.interfaces.nsIWindowWatcher','nsIWindowWatcher')
```

**Figure 38: ooRexx and JavaXPCOM: “URLOpener.rex” - The import step**

The first step is to import all necessary Java classes and XPCOM interfaces. As opposed to Java, ooRexx does not yet support the import of whole packages. This is one of the main differences to the previous Java example. The other one would be the initialization of the XPCOM embedding. On certain operating systems and with a wrong GRE registration, the getGREPathWithProperties() method might return a nulltype object instead of the right path to your javaxpcom.jar file in the XULRunner installation directory. There are a few ways to solve this problem.

You could try to get the GRE path from your systems properties, from your registry or include the Java archive into your application and use a relative path. In our test case, we look for a corresponding system property that we have set up (System.getProperty()). If this fails, we use a path where javaxpcom.jar can be found. The ooRexx variable *grePathName* contains this path.

```
/** Target URL to open (explicit or by BSF bean) */
targetUrl = .bsf~bsf.lookupBean('targetUrl')
if targetUrl = .nil then targetUrl = 'http://derstandard.at'

/** Initiate XPCOM embedding */
path = .System~getProperty('GRE_PATH')
 -- set grePathName manually (see first line)
if path = .nil then grePath = .File~new(grePathName)
else grePath = .File~new(path)
say 'Gecko Runtime Engine path:  ' grePath~getPath

mozilla = .Mozilla~getInstance
        mozilla~initialize(grePath)
        mozilla~initXPCOM(grePath, .nil)
say 'Mozilla XPCOM initialized!'
```

**Figure 39: ooRexx and JavaXPCOM: "URLOpener.rex" – XPCOM embedding**

First we check if our script was launched by a Java class that could have registered a targetURL as a BSF bean. If not we will open the homepage of the newspaper derStandard by default. Finding a correct GRE path is next. As we already know there are a number of ways to acquire it. One foolproof way would be to include javaxpcom.jar to your script and reference it, which is done here by *grePathName*.

When this is done, we get a new Mozilla instance from the Singleton class and call the functions *initialize()* and *initXPCOM().*

```
/** Get the Service Manager (responsible for acquiring XPCOM objects) */
serviceManager = mozilla~getServiceManager

/** Retrieve necessary property values and XPCOM interface IIDs */
appStartupID    = .bsf~bsf.getStaticValue(.nsIAppStartup, 'NS_IAPPSTARTUP_IID')
windowCreatorID = .bsf~bsf.getStaticValue(.nsIWindowCreator, 'NS_IWINDOWCREATOR_IID')
windowWatcherID = .bsf~bsf.getStaticValue(.nsIWindowWatcher, 'NS_IWINDOWWATCHER_IID')
winProps = "width=1000, height=650, centerscreen, scrollbars='yes', status='yes'"
```

**Figure 40: ooRexx and JavaXPCOM: "URLOpener.rex" – get the Interface IDs**

The actual implementation of the XPCOM services is a near identical copy of the previous Java example. We get a new Service Manager and then all the necessary interface IIDs with the BSF4ooRexx subfunction *getStaticValue(class, property name).*

You should already be familiar with the code from figure 41, since it was already explained in the previous chapter. When we are done with our task, we terminate the XPCOM embedding again by calling *shutdownXPCOM*(). The last line of this code example includes the Java support for ooRexx by including the Java wrapper program *BSF.cls*, which is part of the BSF4ooRexx distribution.

```
/** Set up the application and load the new window with interface nsIWindowWatcher */
appStartup = serviceManager~getServiceByContractID
                        ('@mozilla.org/toolkit/app-startup;1', appStartupID)
windowCreator = appStartup~queryInterface(windowCreatorID)
windowWatcher = serviceManager~getServiceByContractID
                        ('@mozilla.org/embedcomp/window-watcher;1', windowWatcherID)
windowWatcher~setWindowCreator(windowCreator)

window = windowWatcher~openWindow(.nil, targetUrl, 'URL Opener', winProps, .nil)
windowWatcher~setActiveWindow(window)
appStartup~run

/** Terminate XPCOM embedding */
mozilla~shutdownXPCOM(.nil)
say 'Mozilla XPCOM embedding finished!'


::requires BSF.cls  -- adds BSF support to Java and ooRexx scripts
```

**Figure 41: ooRexx and JavaXPCOM: “URLOpener.rex” – implement the XPCOM services**

The ooRexx script does not need to be called by Java, the user can simply start the script and BSF4ooRexx takes care of the embedding on its own. But XPCOM can do much more than facilitate certain web services. It is also possible to do handle the file system, I/O operations, XML parsing and so on.

Since we know how to initiate the XPCOM embedding from within Java and ooRexx, the next example will only cover the use of other interfaces. “DirCreator.rex” demonstrates how to create new files and directories with the XPCOM interfaces nsIProperties and nsILocalFile.

```
/** cross platform path retrieval of directories */
properties = serviceManager~getServiceByContractID("@mozilla.org/file/directory_service;1", propsID)
dir = properties~get("Desk", fileID) -- references the users Desktop directory

/** Append directory name and check for existence/create */
dir~append("Ordner aus ooRexx")
if dir~exists=.false
then dir~create(dirType, 0777)
else say "Directory '"dir~bsf.getPropertyValue("path")"' exists already!"
```

**Figure 42: ooRexx and JavaXPCOM: “DirCreator.rex” – create a new desktop directory**

XPCOM offers a way to retrieve platform independent directory paths, this is done using the *nsIProperties* interface within the *directory_service* component. By calling the *get*()-method with the keyword “Desk” we get the path to the current user’s desktop directory, this path will of course vary depending on the operating system.

Subsequently, we append the name of our new directory and create it with the *nsILocalFile* property *“DIRECTORY_TYPE”.* If this directory exists already, the user is informed and the operation aborted.

```
/** Create nsILocalFile object and init with directory path */
file = serviceManager~getServiceByContractID("@mozilla.org/file/local;1", localID)
file~initWithPath(dir~bsf.getPropertyValue("path"))

/** Append file name and check for existence/create */
file~append("Datei aus ooRexx.txt")
if file~exists=.False
then file~create(fileType, 0666)
else say "File '"file~bsf.getPropertyValue("path")"' exists already!"

file~launch -- open the file
```

**Figure 43: ooRexx and JavaXPCOM: "DirCreator.rex" – create a new file and launch**

Figure 43 demonstrates how to do the same with files. All objects descending from the interface *nsILocalFile* have to be initialized with *initWithPath*(). Analog to the creation of our directory we append the file name to the existing directory path with the interface property "*NORMAL_FILE_TYPE".* The method *launch*() will open the file with the default program for this file type. The complete XPCOM API available with XULRunner version 1.9.2 can be found at the Mozilla Development Center [Mozi10] or the Oxymoronical Blog [Oxym10] , which covers current Mozilla development news.

This chapter showed how to embed XPCOM in an ooRexx environment, taking advantage of BSF4ooRexx. Once the XPCOM embedding is successful, it is easy to write effective code with just a few lines of code. In the next chapter we will learn how to create platform independent standalone applications.

## 3.4 XULRunner Applications

In the last chapter we have already established that our XULRunner installation is working. Our aim now is to pack a XUL dialog and a script file into a standalone XULRunner application, which can be started without a browser on any platform. The program will be able to download and install a file by supplying a target URL and a target path to store the file locally. The user can choose to download the target or download and install at the same time. Let's start with the entry dialog, realized in XUL.

### 3.4.1 The Javascript Application Content

We have already learned how a XUL document is formed, so we concentrate on the actual content. The elements are arranged using horizontal (*hbox*) and vertical boxes (*vbox*) that serve as containers for the other elements. Two labels are describing the kind of input or text fields are expecting. The first text field *text1* receives the URL of a downloadable file and *text2* expects the target path on the local hard drive to save the file. Both text fields are mandatory.

```
<hbox>
    <vbox>
            <label id="label2" align="right" value="Enter Download URL, e.g:" />
            <label id="label1" value="------------------------->" />
            <label id="label2" align="right" value="Enter the target path,  e.g:" />
    </vbox>
    <vbox>
         <textbox id="text1" value="http://www.rarlab.com/rar/wrar380d.exe" size="40"/>
         <textbox id="text2" value="c:\\winrar3.zip" size="40"/>
    </vbox>
    <vbox>
        <button id="button1" label="Just Download"                    oncommand="downloadFile('download')" />
        <button id="button2" label="Download &amp; Install" height="40px" oncommand="downloadFile('install')" />
    </vbox>
</hbox>
```

**Figure 44: XULRunner-example: The main XUL document**

In Figure 45 you can see the resulting dialog window.

**Figure 45: XULRunner-example: The dialog window**

From this point on the user has two options: Either to just download the file or to execute right after the download has finished. These options are realized by buttons, their event handlers are implemented by the *oncommand* attribute. Both events launch the same Javascript function, but with different arguments.

```
55  function downloadFile(method) {
56  
57      // get download download Url and target Path from the XUL dialog
58      var url = document.getElementById('text1').value;
59      var targetPath = document.getElementById('text2').value;
60  
61      // check if both have values
62      if (!url || !targetPath) {
63          alert("Please enter download url AND target path!");
64          return false;
65      }
66  
67      // Just download
68      if (method == "download") {
69          download(url, targetPath);
70          alert("Download finished");
71      } else {
72      // Download and install
73      if (method == "install") {
74          download(url, targetPath);
75          alert("Executing file"+ targetPath);
76          execute(targetPath);
77      } };
78  }
```

**Figure 46: XULRunner-example: Javascript main function downloadFile()**

Figure 46 shows the Javascript function *downloadFile(method,)* which determines what the program does with the supplied paths. First we get the two path values by looking for the XUL elements in the DOM tree with getElementById(). When both variables were entered the script continues to check the given argument method, with the value:

- 'download', if the file should only be downloaded. In this case the function *download()* is invoked.
- 'install', when the file should be executed right after downloading. Here both functions *download()* and *execute()* are called.

```
2   function download(url, targetPath) {
3   try {
4
5       // get XPCOM access privileges
6       netscape.security.PrivilegeManager.enablePrivilege('UniversalXPConnect');
7
8       //new obj_URI object
9       var urlFile = Components.classes["@mozilla.org/network/io-service;1"]
10                      .getService(Components.interfaces.nsIIOService)
11                          .newURI(url, null, null);
12      //new file object
13      var nsIFile = Components.classes["@mozilla.org/file/local;1"]
14                          .createInstance(Components.interfaces.nsILocalFile);
15
16      //set file with path
17      nsIFile.initWithPath(targetPath);
18      //if file doesn't exist, create
19      if(!nsIFile.exists()) {
20          nsIFile.create(0x00,0644);
21      }
```

**Figure 4739: XULRunner-example: First part of the Javascript function download()**

The function *download()* is called in both use cases. First we have to get access to the XPCOM layer XPConnect, enabling Javascript to reference the different interfaces. We need to implement the XPCOM interface nsIIOService with the target URL and create a new local file with the nsILocalFile interface. We get both interfaces using *Components.classes,* which is a read-only object. Each object within Components.classes represents one of the XPCOM components that can be accessed. When our XPCOM file is created, we have to initiate it with the target path on our hard drive. In case it does not exist we just create an empty file at the desired location [CoCl08].

```
23      //new persitence object
24      var persist = Components.classes["@mozilla.org/embedding/browser/nsWebBrowserPersist;1"]
25                      .createInstance(Components.interfaces.nsIWebBrowserPersist);
26
27      // with persist flags if desired See nsIWebBrowserPersist page for more PERSIST_FLAGS.
28      const nsIWBP = Components.interfaces.nsIWebBrowserPersist;
29      const flags = nsIWBP.PERSIST_FLAGS_REPLACE_EXISTING_FILES;
30       persist.persistFlags = flags | nsIWBP.PERSIST_FLAGS_FROM_CACHE;
31
32      //save file to target
33      persist.saveURI(urlFile,null,null,null,null,nsIFile);
34      alert(targetPath+" was created");
35      return true;
36
37  } catch (e) { alert(e); return false;}
38  }
```

**Figure 48: XULRunner-example: Second part of the Javascript function download()**

Now that we have set up the URL and the target path in XPCOM, we can start downloading. This is the duty of the interface nsIWebBrowserPersist, so we create an instance called *persist*. nsIWebBrowserPersist saves DOM documents and URLs either to local or remote drives. Before we can call *saveURI()* to save the file we have

to declare the necessary flags. Those flags are constants, which determine the behavior of nsIWebBrowserPersist and are stored in the object attribute *persistFlags*. The only necessary arguments for *saveURI*() are the file name and the URL, all other parameters can be set to null. When *saveURI*() completes without errors, the file is done downloading .

```
40 function execute(targetPath) {
41 try {
42
43     // get XPCOM access privileges
44     netscape.security.PrivilegeManager.enablePrivilege('UniversalXPConnect');
45
46     // get new XPCOM nsIFile
47     var file = Components.classes["@mozilla.org/file/local;1"]
48                          .createInstance(Components.interfaces.nsILocalFile);
49     // store target path and execute
50     file.initWithPath(targetPath);
51     file.launch();
52     return true;
53 } catch (e) { alert(e); return false;}
54 }
```

**Figure 49: Javascript function execute() opens the downloaded file**

Depending on the user choices the function *execute()* is called. Its goal is to execute the downloaded file. Because the security privileges are only given within the scope of a Javascript function, we have to call it here again. Now we repeat the steps from function *download()* and create and initiate *nsIFile* with the given file name. The nsILocalFile function *launch()* executes the file. This is all the functionality we need for our little desktop application. Now we just have to pack it and it is ready for distribution.

### 3.4.2 The XULRunner Packaging Process

XULRunner handles standalone applications with a predefined folder structure and three key files that define the loading behavior of the program and make sure that every file is registered [GeXU09].

1. ***application.ini***: The entry point for the application.
2. ***chrome.manifest***: defines the content, including text files and JAR archives
3. ***prefs.js****:* Defines the path and preferences of all documents of the application.

When all files are in their right place, the package looks like the structure in Figure 50. Our documents are stored in the subfolder /myapp/chrome/content/. The structure within the subfolder /content can be chosen by the developer as long as chrome.manifest has all the necessary path information [XuTu05].

```
+ /myapp
|
+-+ /chrome
| |
| +-+ /content
| | |
| | +- main.xul
| |
| +- chrome.manifest
|
+-+ /defaults
| |
| +-+ /preferences
|   |
|   +- prefs.js
|
+- application.ini
```

**Figure 50: Folder structure of our XULRunner application**

*application.ini* will be the entry point for our application and contains general information as well as the supported XULRunner versions. Setting up requirements for the supported Gecko Runtime Engines prevents the scripts from crashing, since the XPCOM libraries may change over time.

```
[App]
Vendor = andi
Name = xulrunner
Version = 1.0
BuildID = 20090411
[Gecko]
MinVersion = 1.8
MaxVersion = 1.9.*
```

In our case, the XULRunner installation that starts the program has to be at least of version 1.8.0[5] [XuSo09].

Chrome.manifest only contains the following line:

```
content xulrunner file:xulrunner/
```

The line is telling application.ini that the folder *content* contains the necessary documents, the name of the application (*xulrunner*) and the type of documents in folder content (*file:xulrunner*) [GeXU09].

The declaration of our XUL document as the main window is done in prefs.js, where all preferences are stored in a Javascript file. It is also possible to define the attributes of all windows in this script. Our application has the following information stored in prefs.js.

```
pref("toolkit.defaultChromeURI", "chrome://xulrunner/content/dialog.xul");
pref("toolkit.defaultChromeFeatures","chrome,resizable=yes,dialog=no");
```

The first line describes the relative path of our XUL document. In the second line we refer to object *chrome* (the main window) and define the attributes *resizable* and *dialog*.

Now that the XUL document dialog.xul and the Javascript dialogScript.js are in the folder content and we have written application.ini, chrome.manifest and prefs.js, we are ready to test the application. Open the command line and change to the directory, in which application.ini resides. The program is called with

```
xulrunner application.ini
```

and the application starts in a separate window. You see the resulting GUI in Figure 51.

**Figure 51: The graphical user interface of our XULRunner application**

The package is ready to be distributed as an archive and runs on any platform that is familiar with the XPCOM library. Even though the last two chapters contained a bit more code than the scripting examples you can see that Mozilla enables software developers to use the XPCOM component architecture to build powerful platform independent applications with relatively small effort.

---

[5] The current version of XULRunner – as of 1st of June 2009 – is 1.9.2 pre.

### 3.4.3 Extending the Application with JavaXPCOM

In the previous two chapters we used the XULRunner framework to create a stand-alone application with just a few configuration files and the application content, consisting of a XUL template, which served as the application's entry point, and Javascript. We already know that it is possible to use Java to interact with any number of scripting languages.

Now we will extend this application to embed XPCOM within an ooRexx script, which will be managed and launched by a custom Java class. The configuration files of our new application are basically identical with the previous example, with the exception of the path to the XUL template. So we will concentrate solely on the new, more complex application content, consisting of:

- **"UrlOpener.xul"** – the XUL application entry point.
- **"JavaClassLoader.js"** – the Javascript that implements a Java Class Loader object and will give the necessary permissions to certain Java archives.
- **"javaFirefoxExtensionUtils.jar"** – the Java archive that contains certain policy setting classes and our custom *JavaBSF.class*.
- **"JavaBSF.java"** – the custom Java class with BSF support, that will register some BSF beans and launch *Window.rex*.
- **"Window.rex"** – the ooRexx script that will implement the XPCOM services.

All of the above content files will reside in the application subdirectory */xulrunner/chrome/xulrunner*.

#### UrlOpener.xul

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/global.css" type="text/css"?>
<?xml-stylesheet href="dialog.css" type="text/css"?>

<window id="myDialog" title="URL Opener with Java Class Loader"
        onload="window.sizeToContent()"
        xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

<script language="JavaScript" src="JavaClassLoader.js"> </script>

<hbox>
    <vbox>
            <label id="label1" align="center" value="Enter URL:" />
            <textbox id="urlField" value="http://www.wu.edu/" size="40"/>
            <button id="button1" label="Launch URL via Java and ooRexx"
                                oncommand="useClassLoader();" />
    </vbox>
</hbox>


</window>
```

**Figure 52: JavaXPCOM XULRunner: "UrlOpener.xul"**

The XUL template “*UrlOpener.xul”* looks almost like the one in our first XULRunner application. It sets up the XUL xml namespace, some other window options and includes the Javascript “*JavaClassLoader.js*”. The content only consists of a label, a textbox containing the URL to be called and a button that executes the Javascript function *useClassLoader().*When invoked, our XUL file will produce the following output (see Figure 53).

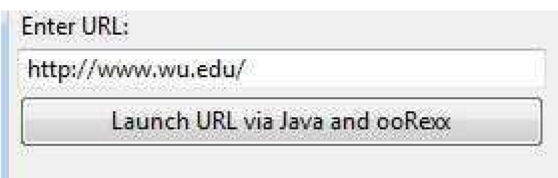



**Figure 53: The output of “UrlOpener.xul”**

## JavaClassLoader.js

This Javascript uses a Java Class Loader to register certain Java archives to the system. By default a Java application can only use the standard API. If you want to use other libraries or frameworks you have to include the corresponding Java archives into the system’s classpath. But this approach does not work with a custom Java class that you would write, so we have to make our class available to Java, which is done by using a Java Class Loader object. Our Javascript contains three functions.

- **useClassLoader**(): This will generate a new Class Loader object and add certain Java archives. In our case we need the support for our custom Java class, BSF and JavaXPCOM.
- **policyAdd**(): This function sets the permissions and access privileges for our Class loader files.
- **launchUrl**(): If everything is set up, we can instantiate our custom Java class *JavaBSF* and call its methods.

In our application we take advantage of the Java archive “javaFirefoxExtensionUtils.jar”, which is part of the open source MIT Simile project “Java Firefox Extensions” [MITS10]. On the Mozilla Development Center you will find instructions on how to utilize this Java archive for other XULRunner applications [Mozi10]. The Javascript function policyAdd() will need this JAR file to set the permissions. If the paths to the necessary Java archives are not set correctly, the whole application will not work.

```
function useClassLoader () {

// Get path to the following JAR files
try {
        // get the desktop directory
        var file = Components.classes["@mozilla.org/file/directory_service;1"].
                        getService(Components.interfaces.nsIProperties).
                        get("Desk", Components.interfaces.nsIFile);
        var desktop = file.path;

// You must add this utilities JAR (javaFirefoxExtensionUtils.jar) to give your application f
var myJar  = "file:///"+desktop+"/xulrunner/chrome/xulrunner/javaFirefoxExtensionUtils.jar";
var xpcom1  = "file:///C:/xulrunner-1.9.2/bin/javaxpcom.jar";
var xpcom2  = "file:///C:/xulrunner-1.9.2/sdk/lib/MozillaGlue.jar";
var xpcom3  = "file:///C:/xulrunner-1.9.2/sdk/lib/MozillaInterfaces.jar";
var bsf1  = "file:///C:/Program Files/bsf4oorexx/bsf-rexx-engine.jar";
var bsf2  = "file:///C:/Program Files/bsf4oorexx/bsf-v400-20090910.jar";

        // Builds a regular JavaScript array (LiveConnect will auto-convert to a Java array)
        var urlArray = [];
        urlArray[0] = new java.net.URL(myJar);
        urlArray[1] = new java.net.URL(xpcom1);
        urlArray[2] = new java.net.URL(xpcom2);
        urlArray[3] = new java.net.URL(xpcom3);
        urlArray[4] = new java.net.URL(bsf1);
        urlArray[5] = new java.net.URL(bsf2);
        var cl = java.net.URLClassLoader.newInstance(urlArray);

        // Set security policies using the above policyAdd() method
        policyAdd(cl, urlArray);

        var url = document.getElementById("urlField").value;
        alert("Opening this URL: " + url);
        // launch URL
        launchUrl(cl, url, desktop);
}
catch(e) {alert(e+' ::useClassLoader:: '+e.lineNumber);}
}
```

**Figure 54: JavaXPCOM XULRunner: “JavaClassLoader.js” – useClassLoader()**

Figure 54 shows the function useClassLoader(), which is invoked by the XUL button. Since the application is installed in the desktop directory, we get the path using the XPCOM interface nsIProperties, like we did in the “DirCreator.rex” example in chapter 3.3.2. At the moment the Java Class Loader needs to include the following JAR files in order for the application to work.

Archive with the policy setting classes and our custom Java class:

- javaFirefoxExtensionUtils.jar

Archives for JavaXPCOM support:

- javaxpcom.jar
- MozillaGlue.jar
- MozillaInterfaces.jar

Archives for BSF support:

- bsf-rexx-engine.jar
- bsf-v400-20090910.jar

Those paths have to be edited if the application is distributed to other systems. If you want to free the end user of the need to change all those paths himself, you could add those Java archives to your application directory and reference them with relative paths. But it is better to point to the correct installation directories.

Pack the paths into an "URL-Array" and create a new Class Loader object with *URL-ClassLoader.newInstance(array).* Then you can add the necessary permissions by calling *policyAdd()* and get access to your custom Java file by calling *launchUrl().*

```
// launches the target URL, using the Java class loader and JavaBSF.class
function launchUrl (loader, url, desktop) {

try {
        var myClass = loader.loadClass('edu.mit.simile.javaFirefoxExtensionUtils.JavaBSF');
        var myObj = myClass.newInstance(); // instantiates JavaBSF class
        // alert("launching JavaBSF class");
        var response = myObj.launchUrl(url); // calls JavaBSF, pass whatever arguments you
        alert(response);
}
catch(e) {alert(e+' ::launchUrl:: '+response);}

}
```

**Figure 55: JavaXPCOM XULRunner: "JavaClassLoader.js" – launchUrl()**

The function *launchUrl()* is pretty straight forward. You reference your custom Java class with *URLClassLoader.loadClass(),* create a new instance of the class with *newInstance()* and call any method you want. Method parameters are sent just like any other Javascript function. From this point on, the Java/ooRexx part of the XULRunner application is active. But before your Java class can call the ooRexx script you will need to set up a "grant"-permission for the JAR file containing your custom Java class.

### Setting permissions in the Java policy file

Depending on the kind of application, which will use a custom Java class, we also have to give the right permissions. In Java, every program runs in a so called Sandbox, which shelters the application from your operating system and the file system. Different access privileges can be applied for certain programs. This is especially important for the security of your system if you deal with applets or other web-based Java services. If you use custom Java classes outside of your classpath, you will also need to grant permissions to the application directory.

There are two configuration files responsible for your Java security settings, java.security and java.policy. The first one tells your Java installation where to look for the specific policy files. The default java.security is located in the subdirectory /lib/security/ of your Java Runtime Environment installation and contains the following two entries:

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

The first entry is the path to your systemwide policy file, the second entry specifies the security settings for the current user. In our case, we want to provide this XULRunner application to all the users on the machine, so we edit /lib/security/java.policy.

Although it is a simple text file you should never edit the security permissions manually, but use the policytool.exe program in the /bin/ directory of your Java installation. On Windows systems always run policytool.exe as an Administrator, otherwise you may not be able to edit the permissions.

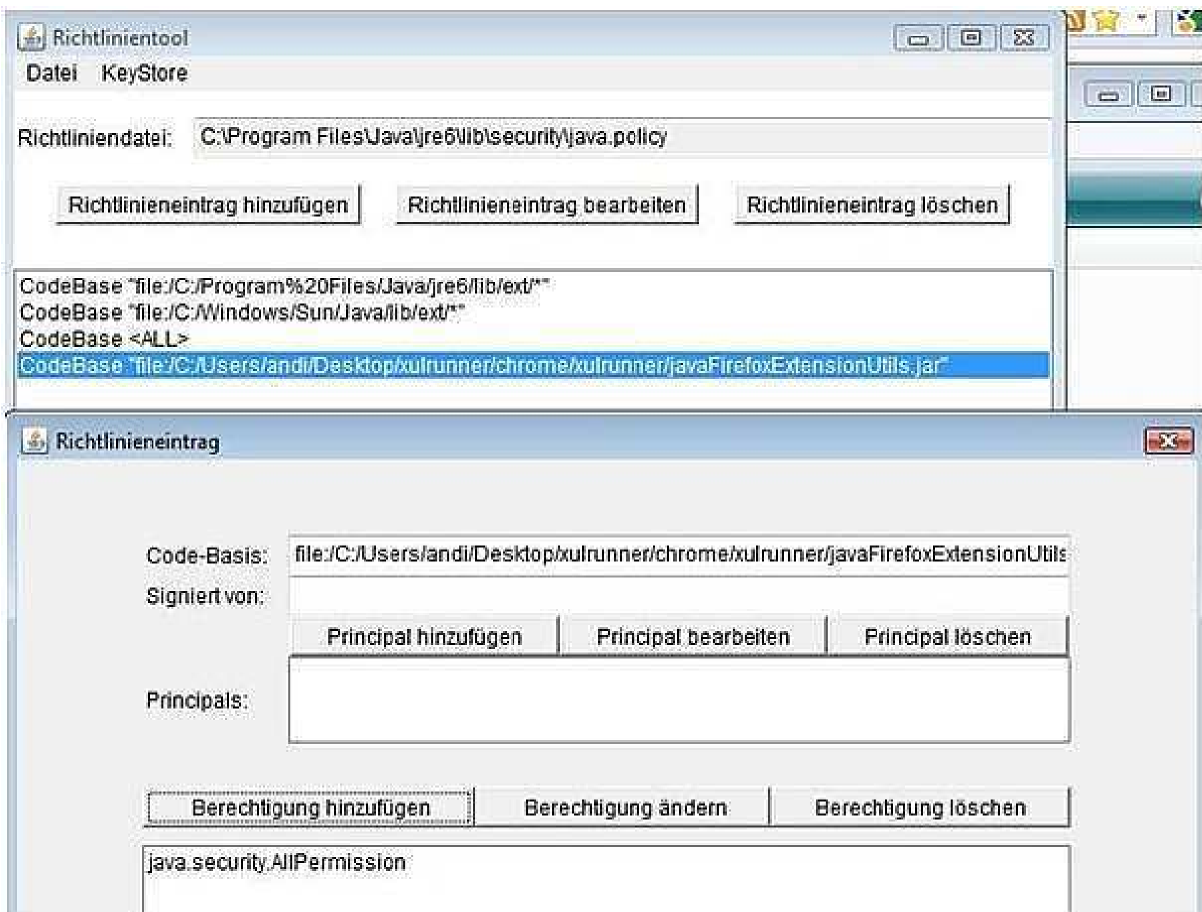



**Figure 56: Grant Java Access permissions with policytool.exe**

Figure 56 demonstrates how to edit you policy file. Open java.policy and add a new permission (dt.: Richtlinieneintrag). The code base is the absolute path to the directory of your XULRunner application directory, in our case it resides on the current user's desktop. Add all the necessary permissions and save the policy file. Since we do not know if the application gets extended in the future we will grant all available permissions with *java.security.AllPermission*.

### JavaBSF.java

Now that we are able to access our custom Java class from within Javascript we can initiate the BSF support, load the ooRexx script and execute it. You should already be familiar with the BSF Manager and its ability to call different scripting languages.

```
package edu.mit.simile.javaFirefoxExtensionUtils;

import java.lang.*;
import java.util.logging.*;
import java.io.*;
import org.apache.bsf.*;
import org.apache.bsf.util.*;

public class JavaBSF {

        private Logger logger = Logger.getLogger("");
        String targetUrl;
        String currentDir;

/** launchUrl() receives the targetUrl from JavaScript and launches the ooRexx script  */
public String launchUrl (String url) {

    try {
        // Get instance of BSFManager (beans and launching the script)
        BSFManager bsf = new BSFManager();

        // package targetUrl as a BSF bean
        if (url != null) targetUrl = url;
        else             targetUrl = "http://www.orf.at/";
        bsf.registerBean("targetUrl", targetUrl);

        // current directory
        // when called by xulrunner app, it is the directory of application.ini (top level)
        currentDir = new File(".").getAbsolutePath();
        System.out.println(currentDir);

        // retrieve information from script
        String scriptName = currentDir + "/chrome/xulrunner/Window.rex";
        String language = bsf.getLangFromFilename(scriptName);
        FileReader in = new FileReader(scriptName);
        String rexxCode = IOUtils.getStringFromReader(in);

        // launch script and terminate BSF environment
        bsf.exec (language, scriptName, 0, 0, rexxCode);
    }
        catch (BSFException e) { e.getMessage(); e.printStackTrace(); logError(e); return e.getMessage(); }
        catch (IOException e)  { e.getMessage(); e.printStackTrace(); logError(e); return e.getMessage(); }

return "Java/ooRexx ist fertig!";
 }
```

**Figure 57: JavaXPCOM XULRunner: “JavaBSF.java” – launchUrl()**

The first step is to add this Java class to a package (in our case: edu.mit.simile.javaFirefoxExtensionUtils.jar) and import all necessary classes. The method launchUrl(url) will call the ooRexx script. We get a new BSF Manager, register the target Url, which was supplied by Javascript, as a BSF bean and start the script. There are no customizing steps needed for the Java class to work on other systems.

### Window.rex

The ooRexx script is the last step of our application and will implement the XPCOM embedding and will open a new XPCOM window with the provided target Url. It is the same script as the “UrlOpener.rex” example in chapter 3.3.2. Therefore it is not necessary to explain the code again, if you have any questions regarding the implementation of XPCOM from within ooRexx please see chapter 3.3.2 or the Appendix. The script will be explained there in full detail.

There is only one word of advice. Since it does not seem to be possible to get the GRE path dynamically with Mozilla.getGREPathWithProperties() with the current XULRunner version 1.9.2 please provide your “Window.rex” script in the first line with the path to the directory of javaxpcom.jar! Usually this JAR file can be found in the /bin directory of your XULRunner installation. This is very important, otherwise the XPCOM embedding process fails.

If you have followed all the steps you should be ready to go. Start your program by invoking

```
xulrunner.exe application.ini
```

in your application directory or double-click on start.bat (Windows) or start.sh (Linux).

# 4 Conclusion

This bachelor thesis aimed to explain the working principle of XPCOM and component object models in general. It showed in how many ways the developer can access complete interfaces to enrich the application at hand. We used Javascript, ooRexx and Java to implement XPCOM interfaces and finally learned how to build standalone applications with XULRunner, enabling easy distribution of our programs. The fact that the XPCOM architecture is open source and platform independent makes this technology even more appealing.

The Mozilla project and XPCOM are continuously growing. Over the years the technologies matured and new languages evolved. XUL/XBL, the xml-based and intuitive user interface languages, and other web languages quickly accelerated this movement.

Although XPCOM-related formats like XUL or the binding language XBL are not yet fully recognized by standardizing consortiums like the W3C, those technologies are expected to soon emerge from their shadowy existence.

Like Mozilla, every open source project needs the support of an avid community to prosper. The funding and research of projects is directly correlated to the project’s popularity. So, if you have taken help from the countless forums and newsgroups you should contribute as well and help other people.

# 5 Literature

[ApJa09] Apache Jakarta Project – BSF: http://jakarta.apache.org/bsf/faq.html#what-is-bsf. Last visited on May 24th, 2009.

[BoOe02] Boswell David, Oeschger Ian and Murphy Eric: “Creating Applications with Mozilla“, 2002 O'Reilly

[BrMa09] Browser1.de – “50% Marktanteil für Firefox”: http://www.browser1.de/content/view/369/112/. Last visited on April 9th, 2009.

[BuSm01] Bullard Vaughn, Smith Kevin and Daconta Michael: “Essential XUL Programming”. John Wiley & Sons, 2001.

[Carb06] Carboni Davide: “Mozilla: a development platform under the hood of your browser”. Free Software Magazine, 09/2006

[CoCl08] MDC – Components.classes: https://developer.mozilla.org/en/Components.classes. Last visited on May 21st, 2009.

[Flat02] Flatscher Rony G.:“Automatisierung von Windows Anwendungen“. (http://wi.wu-wien.ac.at/rgf/rexx/misc/ecoop06/ECOOP2006_RDL_Workshop_Flatscher_Paper.pdf).

[Flat03] Flatscher Rony G.: “The Augsburg Version of BSF4ooRexx”. http://wi.wu-wien.ac.at/rgf/rexx/orx14/orx14_BSF4ooRexx-av.pdf.
Last visited on May 21st, 2009.

[GeXU09] MDC – Getting started with XULRunner: https://developer.mozilla.org/en/Getting_started_with_XULRunner. Last visited on June 1st, 2009.

[InNe09] InternetNews - “Mozilla.org Unleashes Mozilla 1.0”: http://www.internetnews.com/xSP/article.php/1299381. Last visited on April 8th, 2009.

[JaXP06] MDC – JavaXPCOM: https://developer.mozilla.org/en/JavaXPCOM. Last visited on June 9th, 2009.

[MaSh09] Market share Net Applications – “Firefox market”: http://marketshare.hitslink.com/firefox-market-share.aspx?qprid=0&sample=28. Last visited on April 9th, 2009.

[McFa03] McFarlane Nigel: “Rapid Application Development with Mozilla“. Prentice Hall, 2003.

[Mill06] Millenium X: “COM in Mozilla”, 05.08.2006. http://www.bengoodger.com/2006/08/com_in_mozilla.html. Last visited on May 10th, 2009.

[MiTe09] Microsoft TechNet – Rexx Script Center: http://www.microsoft.com/technet/scriptcenter/scripts/rexx/default.mspx?mfr=true. Last visited on May 26th, 2009.

[MITS10] MIT Simile Project – Java Firefox Extension: http://simile.mit.edu/ , Last visited on March 22th, 2010

[MoSi09] Mozilla.org – Signed Scripts in Mozilla: http://www.mozilla.org/projects/security/components/signed-scripts.html. Last visited on May 25th, 2009.

[MoXP09] Mozilla Development Center (MDC) - XPCOM project page: www.mozilla.org/projects/xpcom. Last visited on May 15th, 2009.

[MoXV08] mozdev.org - XPCOMViewer extension: http://xpcomviewer.mozdev.org/. Last visited on May 1st, 2009.

[Mozi10] Mozilla Development Center Homepage: https://developer.mozilla.org/en , Last visited on March 10th, 2010

[NaMa09] NationMaster Encyclopedia - Java (programming language): http://www.nationmaster.com/encyclopedia/Java-(programming-language). Last visited on May 5th, 2009.

[OnLa09] O’Reilly ONLamp - System Administration with ooRexx: http://www.onlamp.com/pub/a/onlamp/2006/03/02/oorexx_gui.html?page=last&x-maxdepth=0. Last visited on May 26th, 2009.

[OvXP09] MDC - Overview of XPCOM: https://developer.mozilla.org/En/Creating_XPCOM_Components/An_Overview_of_XPCOM. Last visited on April 20th, 2009.

[Oxym10] http://www.oxymoronical.com/experiments/apidocs/platform/1.9.2a1pre , Last visited on March 18th, 2010

[PoHo07] Pohja Mikko, Honkala Mikko, Penttinen Miemo, Vuorimaa Petri and Panu Ervamaa: “Web User Interaction”. Springer, 2007.

[ThSma09] MDC - The Smart Pointer Guide: http://www.mozilla.org/projects/xpcom/nsCOMPtr/. Last visited on May 20th, 2009.

[WUCo09] WU Wien - BSF4ooRexx Source Code Installer: http://wi.wu-wien.ac.at/rgf/rexx/BSF4ooRexx/current/BSF4ooRexx_install.zip. Last visited on May 21st, 2009.

[WURe09] WU-Wien - BSF4ooRexx Readme.txt: http://wi.wu-wien.ac.at/rgf/rexx/BSF4ooRexx/current/readmeBSF4ooRexx.txt. Last visited on May 21st, 2009.

[XPCo09] MDC - Information on XPConnect and scriptable components: http://www.mozilla.org/scriptable. Last visited on June 10th, 2009.

[XPID09] MDC - XPIDL Reference: http://www.mozilla.org/scriptable/xpidl/. Last visited on May 10th, 2009.

[XulP09] XULPlanet's online XPCOM reference: http://www.xulplanet.com/references/xpcomref/. Last visited on April 29th, 2009.

[XuRu09] Mozilla.org - XULRunner Releases Download Site: http://releases.mozilla.org/pub/mozilla.org/xulrunner/releases/. Last visited on May 10th, 2009.

[XuSo09] Mozilla.org – XULRunner Source Repository: http://ftp.mozilla.org/pub/mozilla.org/xulrunner/nightly/. Last visited on May 4th, 2009.

[XuTu05] XULRunner Tutorial: http://blogs.acceleration.net/ryan/archive/2005/05/06/1073.aspx. Last visited on June 1st, 2009.

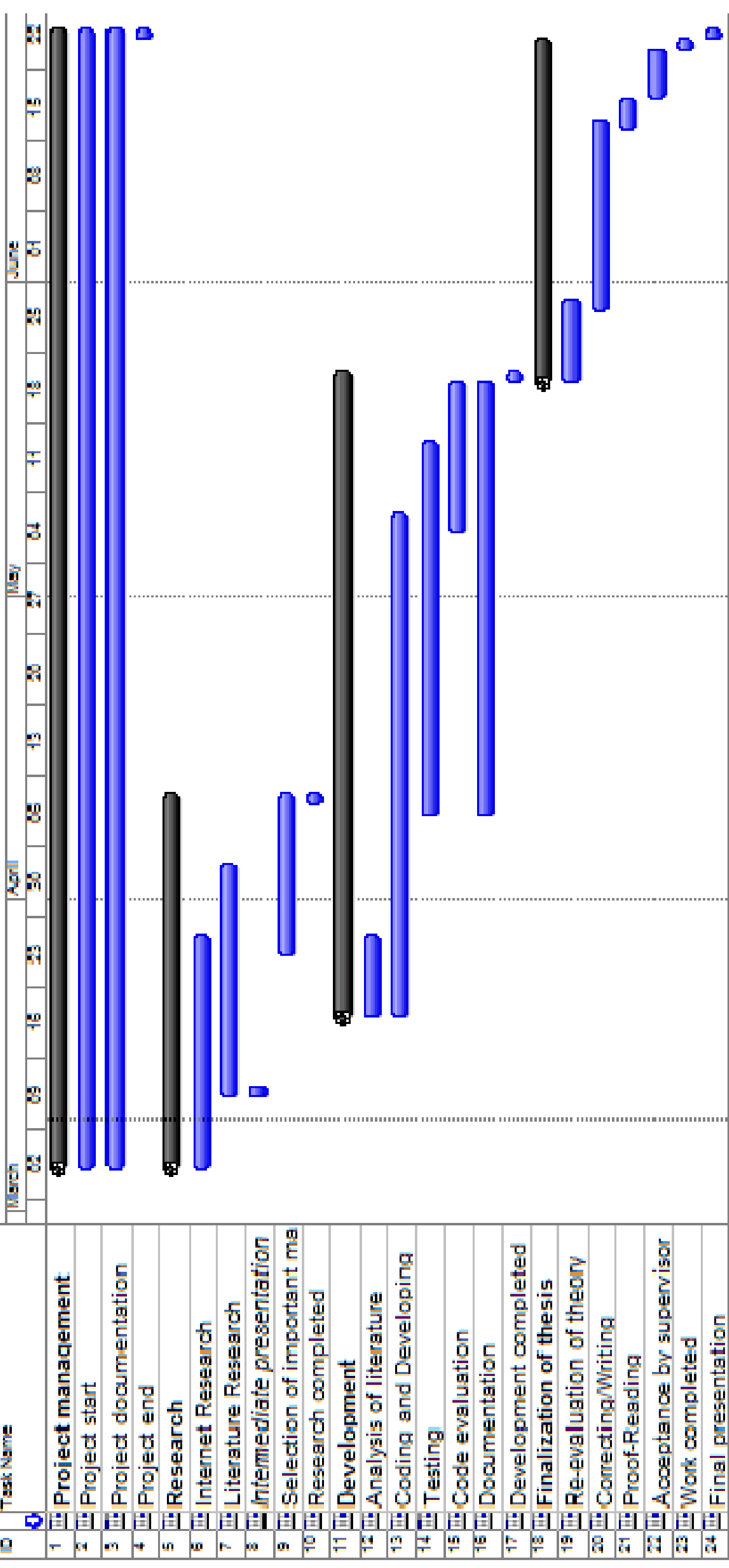
# 6 List of Figures

Figure 1: Firefox closes the market share gap to the Internet Explorer [MaSh09]...... 7
Figure 2: In some countries, Firefox is already the most popular browser [MaSh09] . 8
Figure 3: Short comparison between Mozilla and Java [Carb06]............................. 18
Figure 4: XPCOM component architecture [PoHo07] .............................................. 20
Figure 5: Elements of an  XPCOM GUI [PoHo07] .................................................. 21
Figure 6: Header information of a XUL document ................................................... 21
Figure 7: Body of a XUL document.......................................................................... 22
Figure 8: Firefox displays the XUL document like this ............................................. 22
Figure 9: XBL example: The XUL document xbl.xul ................................................ 23
Figure 10: XBL example: The CSS document xbl.css ............................................. 23
Figure 11: XBL example: The XBL document xbl.xml ............................................. 24
Figure 12: XBL example: The final result of the XUL file call ................................... 24
Figure 13: Javascript function doSomething adds a new tab to the current browser 26
Figure 14: BSF-example: import of BSF classes ..................................................... 30
Figure 15: BSF-example: The main method of ScriptInterpreter.java....................... 31
Figure 16: BSF-example: ScriptInterpreter's constructor initializes BSFManager .... 31
Figure 17: BSF-example: Import of Java packages within Javascript....................... 32
Figure 18: BSF-example: The construction of an AWT Frame within Java............... 32
Figure 19: BSF-example: Event handling from within the script .............................. 32
Figure 20: BSF-example: AWT Frame in JavaScript ............................................... 33
Figure 21: BSF-example: AWT Frame in JavaScript ............................................... 33
Figure 22: Bean-example: The new StringInterpreter constructor ........................... 34
Figure 23: Bean-example: Javascript ActionListener accesses bean....................... 35
Figure 24: Bean-example: The Java String is now used within Javascript ............... 35
Figure 25: Rexx Script listing data about all active Windows processes .................. 38
Figure 26: The Windows command line prints all active processes.......................... 38
Figure 27: BSF4ooRexx-example: Definition of class names and constants............ 40
Figure 28: BSF4ooRexx-example: The frame construction ..................................... 41
Figure 29: BSF4ooRexx-example: The ooRexx event listeners .............................. 41

Figure 30: BSF4ooRexx-example: The pollEventText Interpreter ............................ 42
Figure 31: BSF4ooRexx-example: ooRexx methods for event handling .................. 42
Figure 32: JavaXPCOM-example: Import of all necessary classes. .......................... 45
Figure 33: JavaXPCOM-example: The main method and CID/IID definition ............ 46
Figure 34: JavaXPCOM-example: Initialization of the Gecko Runtime Engine path. 46
Figure 35: JavaXPCOM-example: Embedding XPCOM into Java........................... 47
Figure 36: JavaXPCOM-example: Running the startup application.......................... 47
Figure 37: JavaXPCOM-example: XPCOM shutdown............................................. 48
Figure 38: ooRexx and JavaXPCOM: “URLOpener.rex” - The import step .............. 49
Figure 39: ooRexx and JavaXPCOM: “URLOpener.rex” – XPCOM embedding....... 53
Figure 40: ooRexx and JavaXPCOM: “URLOpener.rex” – get the Interface IDs ...... 54
Figure 41: ooRexx and JavaXPCOM: “URLOpener.rex” – XPCOM services ........... 55
Figure 42: ooRexx and JavaXPCOM: “DirCreator.rex” – create new desktop dir..... 55
Figure 43: ooRexx and JavaXPCOM: “DirCreator.rex” – create new file and launch 56
Figure 44: XULRunner-example: The main XUL document..................................... 57
Figure 45: XULRunner-example: The dialog window ............................................. 58
Figure 46: XULRunner-example: Javascript function - downloadFile()………………..54
Figure 47: XULRunner-example: 1st part of Javascript function download()…………55
Figure 48: XULRunner-example: 2nd part of Javascript function download()………..55
Figure 49: Javascript function execute() opens the downloaded file…………………..56
Figure 50: Folder structure of our XULRunner application………………………………57
Figure 51: The graphical user interface of our XULRunner application………………..58
Figure 52: JavaXPCOM XULRunner: “UrlOpener.xul”…………………………………..59
Figure 53: The output of “UrlOpener.xul”Figure…………………………………………..60
Figure 54: JavaXPCOM XULRunner: “JavaClassLoader.js” – useClassLoader()……61
Figure 55: JavaXPCOM XULRunner: “JavaClassLoader.js” – launchUrl()……………62
Figure 56: Grant Java Access permissions with policytool.exe…………………………64
Figure 57: JavaXPCOM XULRunner: “JavaBSF.java” – launchUrl()…………………..65

# 7 Project Management

## Gantt-Chart



# 8 Appendix

The appendix is a collection of all programs that were used to illustrate the working principles of the discussed languages. You will find the entire code and instructions on how to run the mentioned programs.

## 8.1 XUL example

Instructions: The XUL document dialog.xul shows a typical structure of a XUL document. The behavior of the XUL window is controlled by the CSS script "dialogScript.js". When the function newTab(url) is called, dialogScript.js adds a new browser tab to Firefox. newBookmarkFolder(folder) adds a new bookmark folder to the bookmark service of Mozilla Firefox.

### dialog.xul

```
<?xml version="1.0"?>
<?xml-style sheet href="chrome://global/skin/global.css" type="text/css"?>

<dialog id="myDialog" title="My Dialog"
xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
      buttons="cancel"
      buttonlabelcancel="Exit"
      buttonaccesskeycancel="E"
      ondialogcancel="doCancel();">

<script language="JavaScript" src="DialogScript.js"></script>


<menu label="Datei"  class="mymenu" >
      <menupopup>
         <menuitem label="Menu 1" class="mymenu" id="b1" oncommand="
newTab(‘http://www.wu.ac.at/’) " />
         <menuitem label="Menu 2" class="mymenu" id="b2" oncommand="
newBookmarkFolder(folder) " />
         <menuitem label="Menu 3" class="mymenu" id="b3"
oncommand="doSomethingOther()" />
      </menupopup>
</menu>
</dialog>
```

## dialogScript.js

```
function downloadFile(url) {
try {
netscape.security.PrivilegeManager.enablePrivilege('UniversalXPConnect');

    if (url==null)
    var url = "http://tinyurl.com/koko5l";  // WinRAR Downloader

    //new obj_URI object
    var urlFile = Components.classes["@mozilla.org/network/io-service;1"]
                        .getService(Components.interfaces.nsIIOService)
                          .newURI(url, null, null);
    //new file object
    var nsIFile = Components.classes["@mozilla.org/file/local;1"]
                          .createInstance(Components.interfaces.nsILocalFile);
    //set file with path
    nsIFile.initWithPath("c:\\winrar.exe");
    //if file doesn't exist, create
    if(!nsIFile.exists()) {
       nsIFile.create(0x00,0644);
    }

    //new persitence object
    var persist =
Components.classes["@mozilla.org/embedding/browser/nsWebBrowserPersist;1"]
                        .createInstance(Components.interfaces.nsIWebBrowserPersist);

    // with persist flags if desired
    const nsIWBP = Components.interfaces.nsIWebBrowserPersist;
    const flags = nsIWBP.PERSIST_FLAGS_REPLACE_EXISTING_FILES;
    persist.persistFlags = flags | nsIWBP.PERSIST_FLAGS_FROM_CACHE;

    //save file to target
    persist.saveURI(urlFile,null,null,null,null,nsIFile);

    alert(flags); alert(nsIFile);

} catch (e) { alert(e); return false;}
}


function newTab(url) {
try {
    // gBrowser is only accessible from the scope of
    // the browser window (browser.xul)    gBrowser.addTab("http://derstandard.at");
netscape.security.PrivilegeManager.enablePrivilege('UniversalXPConnect');

var wm = Components.classes["@mozilla.org/appshell/window-mediator;1"]
                .getService(Components.interfaces.nsIWindowMediator);
var mainWindow = wm.getMostRecentWindow("navigator:browser");
mainWindow.getBrowser().addTab(url);

} catch (e) { alert(e); return false;}
}
```

```
function newBookmarkFolder(newFolder) {
// https://developer.mozilla.org/en/Code_snippets/Bookmarks
netscape.security.PrivilegeManager.enablePrivilege('UniversalXPConnect');

try {
    // before you can use the bookmarks service, you need to get access to it
    var bmsvc = Components.classes["@mozilla.org/browser/nav-bookmarks-service;1"]
                    .getService(Components.interfaces.nsINavBookmarksService);

    var menuFolder = bmsvc.bookmarksMenuFolder; // existing Bookmarks menu folder
    var newFolderId = bmsvc.createFolder(menuFolder, newFolder, bmsvc.DEFAULT_INDEX);

    alert("Der neue Bookmark-Folder '"+newFolder+"' hat die ID "+newFolderId);

} catch (e) { alert(e); return false;}
}


function showInterfaces(component) {
    netscape.security.PrivilegeManager.enablePrivilege('UniversalXPConnect');
    // |component| is the XPCOM component instance
   for each (i in Components.interfaces) {
          if (component instanceof i) { alert(i); }
    }
}
```

## 8.2 XBL example

Instructions: The XBL example shows how XBL bindings can change the structure of XUL documents.

When xbl.xul is started, parts of its body are replaced with the content of the corresponding XBL binding. The XBL file xbl.xml is connected to the XUL document by a CSS statement in xbl.css.

### xbl.xul

```
<?xml version="1.0"?>
<?xml-style sheet href="xbl.css" type="text/css"?>

<window
      xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

   <box class="classname" />
</window>
```

### xbl.css

```
box.classname
{
   -moz-binding: url('xbl.xml#xblname');
}
```

### xbl.xml

```
<?xml version="1.0"?>


<bindings xmlns="http://www.mozilla.org/xbl"
         xmlns:xul="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

  <binding id="xblname">
   <content>
    <xul:vbox style="margin:8px;">
        <xul:label value="Type a text:" />
        <xul:hbox>
            <xul:textbox flex="8" />
            <xul:button label="Enter"  oncommand="alert('enter');" />
            <xul:button label="Clear"  oncommand="alert('clear');" />
        </xul:hbox>
    </xul:vbox>
   </content>
  </binding>

</bindings>
```

## 8.3 BSF example

Instructions: This example shows how Java can access scripts directly.

Different scripting languages (e.g. Javascript, ooRexx) create a Java GUI within the script by using BSF.

To start: 1. cmd.exe/shell: java TestScript "script filename"

2. create new instance: new ScriptInterpreter(filename)

### ScriptInterpreter.java

```
import java.awt.*;
import java.awt.*;
import java.io.*;

import org.apache.bsf.*;
import org.apache.bsf.util.*;

public class ScriptInterpreter {

public static String scriptName;

/* class constructor uses BSF to (evaluate and) launch a script
 /**************************************************************/
 public ScriptInterpreter(String scriptName) throws BSFException {

   // execute script
   try {
     BSFManager bsf = new BSFManager();

      //create "Java-String"
      String btnLabel = "Label stammt aus Java";
     bsf.declareBean("btnLabel", btnLabel, String.class);

     String language = BSFManager.getLangFromFilename(scriptName);
     FileReader in = new FileReader(scriptName);
     String script = IOUtils.getStringFromReader(in);

      // launch script start
     bsf.exec (language, scriptName, 0, 0, script);

   }
    catch (BSFException e) { e.getMessage (); e.printStackTrace (); }
    catch (IOException e)  { e.getMessage (); e.printStackTrace (); }
 }
```

```
  /* main() checks arguments (filename) and calls constructor
  /*******************************************************************/
  public static void main (String[] args) throws Exception {

     if (args.length == 0) {
       scriptName = "frame.js";
      System.out.println ("Kein Skript uebergeben --> default: JavaScript "+scriptName);
    }
     else {
       scriptName = args[0];
      System.out.println ("Skript erfolgreich uebergeben --> "+args[0]);
       if (args.length > 1) System.out.println ("Error: Es kann höchstens ein Parameter
übergeben werden");
    }

    // initialize TestScript
    new ScriptInterpreter(scriptName);
  }
}
```

## frame.js

```
 /**    JavaScript Rhino imports Java packages
/*******************************************************/
importPackage(java.awt);
importPackage(java.lang);
importPackage(java.net);
importPackage(org.apache.bsf);

/**  Create GUI components
/***********************************/
frame = new Frame("AWT Frame, implementiert in JavaScript");
panel = new Panel(new BorderLayout());
scroll = new Scrollbar(Scrollbar.HORIZONTAL);

label = new Label("Servus");
button = new Button("JavaScript Button");
color = new Choice();
    color.add("green"); color.add("blue"); color.add("red");
text = new Choice();
    text.add("Hallo!"); text.add("Ciao."); text.add("Bis bald...");

panel.setBackground(Color.black);
label.setBackground(Color.lightGray);
label.setAlignment(Label.CENTER);

/**  Pack the GUI
/************************/
panel.add(BorderLayout.EAST, color);
panel.add(BorderLayout.WEST, text);
panel.add(BorderLayout.SOUTH, scroll);

frame.add(BorderLayout.NORTH, label);
frame.add(BorderLayout.CENTER, button);
frame.add(BorderLayout.SOUTH, panel);

/**  Add event handler to components
```

```
/******************************************/
frame.addWindowListener( function(event, methodName) {
    if (methodName == "windowClosing") { System.exit(0);}
    if (methodName == "windowDeiconified") { label.setText("Willkommen zurück"); }
});

color.addItemListener( function(event, methodName) {
    if (methodName == "itemStateChanged")
    {
       switch (color.getSelectedIndex()) {
           case 0: label.setBackground(Color.green); break;
           case 1: label.setBackground(Color.blue); break;
           case 2: label.setBackground(Color.red); break;
           default: break;
       }
       label.setFont(new Font("Dialog", Font.BOLD, 14));
    }
});

text.addItemListener( function(event) {
    label.setText(text.getSelectedItem());
});

button.addActionListener( function(event) {
    try {
       btnLabel = bsf.lookupBean('btnLabel');
       java.lang.System.out.println(45);
       button.setLabel(btnLabel);
       } catch (e) {  }
});

/**  Display frame with properties
/******************************************/
frame.resize(400, 400);
frame.setLocation(300, 300);
frame.pack();
frame.setVisible(true);
```

## frame.rex

```
/**   Define "ooRexx shortcuts" to create Java components  **/
/***********************************************************/
BorderLayout = 'java.awt.BorderLayout'
Button = 'java.awt.Button'
Choice = 'java.awt.Choice'
Color = 'java.awt.Color'
Frame = 'java.awt.Frame'
Label = 'java.awt.Label'
Panel = 'java.awt.Panel'
Scrollbar = 'java.awt.Scrollbar'

EAST = .bsf~bsf.getStaticValue(BorderLayout, "EAST")
WEST = .bsf~bsf.getStaticValue(BorderLayout, "WEST")
SOUTH = .bsf~bsf.getStaticValue(BorderLayout, "SOUTH")
NORTH = .bsf~bsf.getStaticValue(BorderLayout, "NORTH")
```

```
HORIZONTAL = .bsf~bsf.getStaticValue(Scrollbar, "HORIZONTAL")
CENTER = .bsf~bsf.getStaticValue(BorderLayout, "CENTER")
LCENTER = .bsf~bsf.getStaticValue(Label, "CENTER")

black = .bsf~bsf.getStaticValue("java.awt.Color", "black")
blue = .bsf~bsf.getStaticValue("java.awt.Color", "blue")
green = .bsf~bsf.getStaticValue("java.awt.Color", "green")
red = .bsf~bsf.getStaticValue("java.awt.Color", "red")
lightGray = .bsf~bsf.getStaticValue("java.awt.Color", "lightGray")

/**  Create GUI components    **/
/***********************************/
frame = .bsf~new(Frame, "AWT Frame, implementiert in open Object Rexx ")
layout = .bsf~new(BorderLayout)
panel = .bsf~new(Panel, layout)

scroll = .bsf~new(Scrollbar, HORIZONTAL)
label = .bsf~new(Label, "Servus")
button = .bsf~new(Button, "ooRexx Button")

color = .bsf~new(Choice)
color ~~add("green") ~~add("blue") ~~add("red")

text = .bsf~new(Choice)
text ~~add("Hallo!") ~~add("Ciao.") ~~add("Bis bald...")

panel~setBackground(black)
button~setBackground(lightGray)
label~setAlignment(LCENTER)

/**  Pack the GUI    **/
/************************/
panel~add(EAST , color)
panel~add(WEST, text)
panel~add(SOUTH, scroll)

frame~add(NORTH, label)
frame~add(CENTER, button)
frame~add(SOUTH, panel)

/**    Add event handler to components    **/
/*****************************************/
frame~bsf.addEventListener('window', 'windowClosing', 'call BSF "exit"')
frame~bsf.addEventListener('window', 'windowDeiconified', 'label~setText("Willkommen
zurück")')

color~bsf.addEventListener('item', 'itemStateChanged', 'call changeColor color, label')
text~bsf.addEventListener('item', 'itemStateChanged',
'label~setText(text~getSelectedItem())')
button~bsf.addEventListener('action', 'actionPerformed', 'call switchLabel button')

/**  Display frame with properties       **/
/******************************************/
frame~setSize(400, 400)
frame~setLocation(300, 300)
frame~pack()
frame~show()

/** pollEventText (any event) gets interpreted costantly  **/
```

```
/****************************************************************/
do forever
    INTERPRET .bsf~bsf.pollEventText
    if result="SHUTDOWN, REXX !" then leave
end
exit

/** event handler for choice field "color"    **/
/********************************************** **/
changeColor: procedure
    use arg color, label

    label~setBackground( .bsf~bsf.getStaticValue('java.awt.Color', color~getSelectedItem()))
    label~setFont(.bsf~new('java.awt.Font', 'Dialog', .bsf~bsf.getStaticValue('java.awt.Font',
'BOLD'), 14))
return

/** event handler for button "button"       **/
/**********************************************  **/
switchLabel: procedure
    use arg button

    btnLabel = bsf('lookupBean','btnLabel')
    button~setLabel(btnLabel)
return


::requires BSF.cls  -- adds BSF support to Java and scripts
```

### listAllPro.rex

```
/** ooRexx script accessing and displaying all active processes of a Windows machine  **/

/** get Windows management database containing all active processes */
objWMIService = .OLEObject~GetObject("winmgmts:\\.\root\CIMV2")

/** do for every result  */
do objItem over objWMIService~ExecQuery("Select * from Win32_Process")
   say "________________________________________________________________________________"
   say "Name:" objItem~Name
   say "Creation Date:" objItem~CreationDate
   say "Execution State:" objItem~ExecutionState
   say "Install Date:" objItem~InstallDate
   say "Parent Process Id:" objItem~ParentProcessId
   say "Priority:" objItem~Priority
   say "Process Id:" objItem~ProcessId
   say "Read Operation Count:" objItem~ReadOperationCount
   say "Virtual Size:" objItem~VirtualSize
end
```

## 8.4 JavaXPCOM example

Instructions: WindowCreator.java

Java program embeds XPCOM environment and explains the necessary steps.

Different XPCOM services open a URL and a startup application displays the content in a separate DOM window.

Just start the main method, no arguments required at the moment.

### WindowCreator.java

```
import java.io.*;
import java.util.*;
import org.mozilla.xpcom.*;

import org.mozilla.interfaces.nsIAppStartup;
import org.mozilla.interfaces.nsIDOMWindow;
import org.mozilla.interfaces.nsIFile;
import org.mozilla.interfaces.nsILocalFile;
import org.mozilla.interfaces.nsIServiceManager;
import org.mozilla.interfaces.nsISimpleEnumerator;
import org.mozilla.interfaces.nsISupports;
import org.mozilla.interfaces.nsIWindowCreator;
import org.mozilla.interfaces.nsIWindowWatcher;


public class WindowCreator {

   /* main() embeds XPCOM environment and opens an URL in a DOMWindow
  /*************************************************************/
   public static void main(String []args) throws Exception {

   String targetURL = "http://www.kurier.at";

   GREVersionRange[] range = new GREVersionRange[1];
  range[0] = new GREVersionRange("1.8", true, "1.9+", true);

   File grePath = null;
   /** get a Mozilla class instance and the path to the Gecko Runtime Environment (GRE)    */
  try {
    grePath = Mozilla.getGREPathWithProperties(range, null);
  }
      catch (FileNotFoundException e) { }

  if (grePath == null) {
    System.out.println("found no GRE PATH");
    return;
  }
  System.out.println("GRE PATH = " + grePath.getPath());
```

```
    Mozilla mozilla = Mozilla.getInstance();

    /** try embedding the XPCOM environment using the GRE path  */
   try {
       mozilla.initialize(grePath);
       mozilla.initXPCOM(grePath, null);
   }
       catch (IllegalArgumentException e) {
           System.out.println("no javaxpcom.jar found in given path");
           return;
       }
       catch (Throwable t) {
           System.out.println("initXPCOM failed");
           t.printStackTrace();
           return;
       }

    /**  XPCOM is successfully embedded  */
    System.out.println("\n--> initialized\n");

    try {

    // To get access to interfaces we get an instance of the XPCOM service manager
    nsIServiceManager serviceManager = mozilla.getServiceManager();
    // Use contract ID (@mozilla.org/toolkit/app-startup;1) and IID to get startup application nsIAppStartup
    nsIAppStartup appStartup = (nsIAppStartup)serviceManager.getServiceByContractID
                              ("@mozilla.org/toolkit/app-startup;1",
nsIAppStartup.NS_IAPPSTARTUP_IID);
    // Get the nsIWindowCreator interface through appStartup
    nsIWindowCreator windowCreator =
(nsIWindowCreator)appStartup.queryInterface(nsIWindowCreator.NS_IWINDOWCREATOR_IID
);
    // Get the nsIWindowWatcher interface
    nsIWindowWatcher windowWatcher =
(nsIWindowWatcher)serviceManager.getServiceByContractID
                                  ("@mozilla.org/embedcomp/window-watcher;1",
nsIWindowWatcher.NS_IWINDOWWATCHER_IID);

    // Set the window creator
    windowWatcher.setWindowCreator(windowCreator);
    // Create the DOMWindow with the supplied URL
    nsIDOMWindow win = windowWatcher.openWindow(null, targetURL, "mywindow",
"chrome,resizable,centerscreen", null);
    // DOMWindow win is active window
    windowWatcher.setActiveWindow(win);
    // Start the XPCOM startup application
    appStartup.run();

    }
    catch (XPCOMException e) { e.printStackTrace(); }

    // shut down XPCOM embedding
    mozilla.shutdownXPCOM(null);

    }
}
```

## 8.5 XULRunner Javascript example

Instructions: Standalone application that is launched using the XULRunner framework.

XUL document dialog.xul and JS script dialogScript.js display a standalone GUI giving the user the option to download (and install) a file by providing a download URL and a target path to save the file.

Start application: go to cmd.exe/shell and application directory and invoke:

--> xulrunner(.exe) application.ini

### dialog.xul

```
<?xml version="1.0"?>

<?xml-style sheet href="chrome://global/skin/global.css" type="text/css"?>
<?xml-style sheet href="dialog.css" type="text/css"?>

<window id="myDialog" title="Download Applet"
    xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
    onload="window.sizeToContent()">

<script language="JavaScript" src="dialogScript.js"> </script>


<hbox>
   <vbox>
       <label id="label2" align="right" value="Enter Download URL, e.g:" />
       <label id="label1" value="------------------------->" />
       <label id="label2" align="right" value="Enter the target path,  e.g:" />
   </vbox>
   <vbox>
     <textbox id="text1" value="http://www.rarlab.com/rar/wrar380d.exe" size="40"/>
     <textbox id="text2" value="c:\\winrar3.zip" size="40"/>
   </vbox>
   <vbox>
    <button id="button1" label="Just Download"
oncommand="downloadFile('download')" />
    <button id="button2" label="Download &amp; Install" height="40px"
oncommand="downloadFile('install')" />
   </vbox>
</hbox>

</window>
```

## dialogScript.js

```
function download(url, targetPath) {
try {

    // get XPCOM access privileges
    netscape.security.PrivilegeManager.enablePrivilege('UniversalXPConnect');

    //new obj_URI object
    var urlFile = Components.classes["@mozilla.org/network/io-service;1"]
                       .getService(Components.interfaces.nsIIOService)
                          .newURI(url, null, null);
    //new file object
    var nsIFile = Components.classes["@mozilla.org/file/local;1"]
                          .createInstance(Components.interfaces.nsILocalFile);

    //set file with path
    nsIFile.initWithPath(targetPath);
    //if file doesn't exist, create
    if(!nsIFile.exists()) {
       nsIFile.create(0x00,0644);
    }

    //new persitence object
    var persist =
Components.classes["@mozilla.org/embedding/browser/nsWebBrowserPersist;1"]
                       .createInstance(Components.interfaces.nsIWebBrowserPersist);

    // with persist flags if desired See nsIWebBrowserPersist page for more PERSIST_FLAGS.
    const nsIWBP = Components.interfaces.nsIWebBrowserPersist;
    const flags = nsIWBP.PERSIST_FLAGS_REPLACE_EXISTING_FILES;
     persist.persistFlags = flags | nsIWBP.PERSIST_FLAGS_FROM_CACHE;

    //save file to target
    persist.saveURI(urlFile,null,null,null,null,nsIFile);
    alert(targetPath+" was created");
    return true;

} catch (e) { alert(e); return false;}
}

function execute(targetPath) {
try {

    // get XPCOM access privileges
    netscape.security.PrivilegeManager.enablePrivilege('UniversalXPConnect');

    // get new XPCOM nsIFile
    var file = Components.classes["@mozilla.org/file/local;1"]
                 .createInstance(Components.interfaces.nsILocalFile);
    // store target path and execute
    file.initWithPath(targetPath);
    file.launch();
    return true;
} catch (e) { alert(e); return false;}
```

```
}

function downloadFile(method) {

      // get download download Url and target Path from the XUL dialog
    var url = document.getElementById('text1').value;
    var targetPath = document.getElementById('text2').value;

     // check if both have values
     if (!url || !targetPath) {
         alert("Please enter download url AND target path!");
         return false;
     }

     // Just download
     if (method == "download") {
         download(url, targetPath);
         alert("Download finished");
     } else {
     // Download and install
     if (method == "install") {
         download(url, targetPath);
         alert("Executing file"+ targetPath);
         execute(targetPath);
     } };
}
```

## 8.6 XULRunner ooRexx example

Instructions: Standalone application that is launched using the XULRunner framework.

XUL document UrlOpener.xul invokes Javascript JavaClassLoader.js. The script includes a custom Java class “JavaBSF.java” and several other JAR archives for BSF support. When the Java class method launchUrl() is called by the Javascript, Java will start an ooRexx script Window.rex, that implements an XPCOM environment and opens an URL provided by the XUL dialog.

Start application: go to cmd.exe/shell and application directory and invoke:

--> xulrunner(.exe) application.ini

### UrlOpener.xul

```
<?xml version="1.0"?>

<?xml-stylesheet href="chrome://global/skin/global.css" type="text/css"?>
<?xml-stylesheet href="dialog.css" type="text/css"?>

<window id="myDialog" title="URL Opener with Java Class Loader"
      onload="window.sizeToContent()"
      xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

<script language="JavaScript" src="JavaClassLoader.js"> </script>

<hbox>
  <vbox>
        <label id="label1" align="center" value="Enter URL:" />
        <textbox id="urlField" value="http://www.wu.edu/" size="40"/>
        <button id="button1" label="Launch URL via Java and ooRexx"
oncommand="useClassLoader();" />
  </vbox>
</hbox>


</window>
```

## JavaClassLoader.js

```
/*****************************************************************************
** This script is part of a XULRunner application:
** (see Readme.text in top level directory for more information)
**
** This JS script receives a call by the XUL template, creates a Java class loader using the
** following jar Files: javaFirefoxExtensionUtils.jar
**                 javaxpcom.jar
**                 MozillaGlue.jar
**                 MozillaInterfaces.jar
**                 bsf-rexx-engine.jar
**                 bsf-v400-20090910.jar
** The Jars are collected in a Java Class Loader, which gives the proper permissions.
** Now we can instantiate and call our custom Java class "JavaBSF.class".
**
** Customize: change vars myJar, bsf1, bsf2, xpcom1, xpcom2 and xpcom3 to your installation dirs !
**
** Output: calls JavaBSF.launchUrl(String targetUrl), which launches ooRexx script
*****************************************************************************/

function useClassLoader () {

// Get path to the following JAR files
try {
      // get the desktop directory
      var file = Components.classes["@mozilla.org/file/directory_service;1"].
                  getService(Components.interfaces.nsIProperties).
                  get("Desk", Components.interfaces.nsIFile);
      var desktop = file.path;

// You must add this utilities JAR (javaFirefoxExtensionUtils.jar) to give your application full privileges
var myJar  = "file:///"+desktop+"/xulrunner/chrome/xulrunner/javaFirefoxExtensionUtils.jar";
var xpcom1  = "file:///C:/xulrunner-1.9.2/bin/javaxpcom.jar";
var xpcom2  = "file:///C:/xulrunner-1.9.2/sdk/lib/MozillaGlue.jar";
var xpcom3  = "file:///C:/xulrunner-1.9.2/sdk/lib/MozillaInterfaces.jar";
var bsf1  = "file:///C:/Program Files/bsf4oorexx/bsf-rexx-engine.jar";
var bsf2  = "file:///C:/Program Files/bsf4oorexx/bsf-v400-20090910.jar";

      // Builds a regular JavaScript array (LiveConnect will auto-convert to a Java array)
      var urlArray = [];
      urlArray[0] = new java.net.URL(myJar);
      urlArray[1] = new java.net.URL(xpcom1);
      urlArray[2] = new java.net.URL(xpcom2);
      urlArray[3] = new java.net.URL(xpcom3);
      urlArray[4] = new java.net.URL(bsf1);
      urlArray[5] = new java.net.URL(bsf2);
      var cl = java.net.URLClassLoader.newInstance(urlArray);

      // Set security policies using the above policyAdd() method
      policyAdd(cl, urlArray);

      var url = document.getElementById("urlField").value;
      alert("Opening this URL: " + url);
      // launch URL
      launchUrl(cl, url, desktop);
```

```
}
catch(e) {alert(e+' ::useClassLoader:: '+e.lineNumber);}
}


// launches the target URL, using the Java class loader and JavaBSF.class
function launchUrl (loader, url, desktop) {

try {
      var myClass = loader.loadClass('edu.mit.simile.javaFirefoxExtensionUtils.JavaBSF');
      var myObj = myClass.newInstance(); // instantiates JavaBSF class
      // alert("launching JavaBSF class");
      var response = myObj.launchUrl(url); // calls JavaBSF, pass whatever arguments you need
      alert(response);
}
catch(e) {alert(e+' ::launchUrl:: '+response);}

}


// This function will give the necessary privileges to the JAR files in your ClassLoader
function policyAdd (loader, urls) {
try {
       var str = 'edu.mit.simile.javaFirefoxExtensionUtils.URLSetPolicy';
       var policyClass = java.lang.Class.forName(str, true, loader);

       var policy = policyClass.newInstance();
       policy.setOuterPolicy(java.security.Policy.getPolicy());
       java.security.Policy.setPolicy(policy);
       policy.addPermission(new java.security.AllPermission());

       for (var j=0; j < urls.length; j++) {
          policy.addURL(urls[j]);
       }
}
catch(e) {alert(e+' ::policyAdd:: '+e.lineNumber);}
}
```

## JavaBSF.java

```
/***********************************************************************
** This Java class is part of a XULRunner application:
** (see Readme.text in top level directory for more information)
**
** The class JavaBSF is called by the JavaScript "ClassLoader.js" receives an URL,
** implements BSF (BSFManager and BSFEngine) and launches the ooRexx script "Window.rex"
**
** Customize: Usually no configuration required.
**         If either the ooRexx script "Window.rex" or the Error Logger file "java/ErrorLog.txt"
**         can not be found, change the paths for the variables "scriptName" and "log"
**
** Paramaters: receives String "targetUrl" from ClassLoader.js
**
** Output: launches ooRexx script "Window.rex"
**
** Warning: If you change this Java class, you have to load the new
**         .class file into the JAR archive "javaFirefoxExtensionUtils.jar"
***********************************************************************/

package edu.mit.simile.javaFirefoxExtensionUtils;

import java.lang.*;
import java.util.logging.*;
import java.io.*;
import org.apache.bsf.*;
import org.apache.bsf.util.*;

public class JavaBSF {

      private Logger logger = Logger.getLogger("");
      String targetUrl;
      String currentDir;

/** launchUrl() receives the targetUrl from JavaScript and launches the ooRexx script  */
public String launchUrl (String url) {

   try {
      // Get instance of BSFManager (beans and launching the script)
       BSFManager bsf = new BSFManager();

      // package targetUrl as a BSF bean
      if (url != null) targetUrl = url;
      else           targetUrl = "http://www.orf.at/";
      bsf.registerBean("targetUrl", targetUrl);

      // current directory
      // when called by xulrunner app, it is the directory of application.ini (top level)
      currentDir = new File(".").getAbsolutePath();
      System.out.println(currentDir);

      // retrieve information from script
      String scriptName = currentDir + "/chrome/xulrunner/Window.rex";
      String language = bsf.getLangFromFilename(scriptName);
       FileReader in = new FileReader(scriptName);
```

```
      String rexxCode = IOUtils.getStringFromReader(in);

      // launch script and terminate BSF environment
      bsf.exec (language, scriptName, 0, 0, rexxCode);
   }
      catch (BSFException e) { e.getMessage(); e.printStackTrace(); logError(e); return
e.getMessage(); }
      catch (IOException e)  { e.getMessage(); e.printStackTrace(); logError(e); return
e.getMessage(); }

return "Java/ooRexx ist fertig!";
 }

/** Simple Error Logger, receives possible errors from method launchUrl(url) */
public void logError (Exception ex) {

      try {
      // current directory
      // when called by xulrunner app, it is the directory of application.ini (top level)
      currentDir = new File(".").getAbsolutePath();
      System.out.println(currentDir);

      String log = currentDir + "/chrome/xulrunner/java/ErrorLog.txt";
       FileHandler handler = new FileHandler(log);

      logger.addHandler(handler);
       logger.setLevel(Level.ALL);
       logger.info("Error logs:");
       logger.log(Level.INFO, "", ex);
       logger.fine("");
   }
   catch (Exception e) { e.printStackTrace(); }
}

/** main method for command line test purposes */
/**
public static void main (String[] args) {
JavaBSF a = new JavaBSF();
a.launchUrl("http://www.zdf.de");
}
*/

}
```

## Window.rex

```
/***********************************************************************
** This script is part of a XULRunner application:
** (see Readme.text in top level directory for more information)
**
** This ooRexx script receives an URL, implements XPCOM embedding and calls
** several XPCOM services to open a new window.
**
** Customize: change String "grePathName" in the first line to your XULRunner/bin directory!
**
** Paramaters: receives String "targetUrl" as a BSF bean
**
** Output: opens a new window using "targetUrl"
***********************************************************************/

/** Insert Path to your XULRunner installation directory, which contains the JavaXPCOM library "javaxpcom.jar"
*/

grePathName = "C:/xulrunner-1.9.2/bin"
/*************************************/

.bsf~bsf.import('java.io.File','File')
.bsf~bsf.import('java.lang.System','System')
.bsf~bsf.import('org.mozilla.xpcom.Mozilla','Mozilla')
.bsf~bsf.import('org.mozilla.xpcom.GREVersionRange','GREVersionRange')
.bsf~bsf.import('org.mozilla.interfaces.nsIAppStartup','nsIAppStartup')
.bsf~bsf.import('org.mozilla.interfaces.nsIDOMWindow','nsIDOMWindow')
.bsf~bsf.import('org.mozilla.interfaces.nsIServiceManager','nsIServiceManager')
.bsf~bsf.import('org.mozilla.interfaces.nsIWindowCreator','nsIWindowCreator')
.bsf~bsf.import('org.mozilla.interfaces.nsIWindowWatcher','nsIWindowWatcher')

/** Target URL to open (explicit or by BSF bean) */
targetUrl = .bsf~bsf.lookupBean('targetUrl')
if targetUrl = .nil then targetUrl = 'http://derstandard.at'

/** Initiate XPCOM embedding */
path = .System~getProperty('GRE_PATH')
 -- set grePathName manually (see first line)
if path = .nil then grePath = .File~new(grePathName)
else grePath = .File~new(path)
say 'Gecko Runtime Engine path:  ' grePath~getPath

mozilla = .Mozilla~getInstance
      mozilla~initialize(grePath)
      mozilla~initXPCOM(grePath, .nil)
say 'Mozilla XPCOM initialized!'

/** Get the Service Manager (responsible for acquiring XPCOM objects) */
serviceManager = mozilla~getServiceManager

/** Retrieve necessary property values and XPCOM interface IIDs */
appStartupID    = .bsf~bsf.getStaticValue(.nsIAppStartup, 'NS_IAPPSTARTUP_IID')
windowCreatorID = .bsf~bsf.getStaticValue(.nsIWindowCreator,
'NS_IWINDOWCREATOR_IID')
windowWatcherID = .bsf~bsf.getStaticValue(.nsIWindowWatcher,
```

```
'NS_IWINDOWWATCHER_IID')
winProps = "width=1000, height=650, resizable, centerscreen, scrollbars='yes', status='yes'"

/** Set up the application and load the new window with interface nsIWindowWatcher */
appStartup = serviceManager~getServiceByContractID('@mozilla.org/toolkit/app-startup;1',
appStartupID)
windowCreator = appStartup~queryInterface(windowCreatorID)
windowWatcher =
serviceManager~getServiceByContractID('@mozilla.org/embedcomp/window-watcher;1',
windowWatcherID)
windowWatcher~setWindowCreator(windowCreator)

window = windowWatcher~openWindow(.nil, targetUrl, 'URL Opener', winProps, .nil)
windowWatcher~setActiveWindow(window)
appStartup~run

/** Terminate XPCOM embedding */
mozilla~shutdownXPCOM(.nil)
say 'Mozilla XPCOM embedding finished!'


::requires BSF.cls  -- adds BSF support to Java and ooRexx scripts
```