Tobias Specht
Matr.-Nr.: 640 412
tobias.specht@student.uni-augsburg.de

BWS
Using Web-Browsers as Application Platform

Diploma Thesis

March 29, 2004

University of Augsburg
Chair for Business Administration, Business Information Systems III
Prof. Rony G. Flatscher

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
BSF	Bean Scripting Framework
BWS	BSF Web Scripting
Cf.	Confer
COM	Component Object Model
Corp.	Corporation
CSS	Cascading Style Sheets
CVS	Concurrent Versions System
DOM	Document Object Model
ECMA	European Computer Manufacturers Association
E.g.	Exempli gratia (for example)
Etc.	Et cetera
GNU	GNU's Not Unix
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transport Protocol
IBM	International Business Machines

I.e.	Id est/That is
IEC	International Electrotechnical Commission
Inc.	Incorporated
IP	Internet Protocol
ISO	International Organization for Standardization
JAXP	Java API for XML Processing
JSP	Java Server Pages
Ltd.	Limited
MS	Microsoft
OLE	Object Linking and Embedding
OS	Operating System
P.	Page
PP.	Pages
PC	Personal Computer
PDA	Personal Digital Assistent
Rexx	Restructured Extended Executor
RTE	Run Time Environment
SAX	Simple API for XML
SDK	Software Developer's Kit
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XPath	XML Path Language
XSLT	eXtensible Stylesheet Language Transformations

1 Introduction

The intent of this paper is to provide an overview of the technologies and the use of BSF Web Scripting (BWS), a framework for creating platform independent client side web applications in arbitrary scripting languages. Before this is done, a general introduction to the concept of web applications is given.

1.1 Client Side Web Applications

One of the most used parts of the Internet today is the World Wide Web, based on the HTTP protocol[1], the HTML markup language and the URL[2] developed by Tim Berners-Lee and Robert Cailliau in 1990.[3] And most of the people using the web are using web applications too: Search engines, library catalogs, on line shopping systems or web based email clients. Most of these web applications are server side applications, meaning that the browser is used only for presenting the user interface.[4] The *process logic* and the *data storage* tiers reside on the server side. The alternative to this approach is to move at least the application logic from the server to the client. This requires the client to be able to handle the process logic: There must be a way to run code on the client side within the web browser; see figure 1.1 for a depiction of server- and client-side

[1]A protocol for transferring files on the Internet.
[2]A standard for specifying the location of a document on the Internet.
[3]Cf. [Wik04d].
[4]Technically spoken the browser is used only as *user interface* tier of an application, cf. figure 1.1.

web application classified in a three-tier application architecture scheme.[5] As depicted in this figure, a browser based application that has the process logic residing on the client is a *client side web application*.

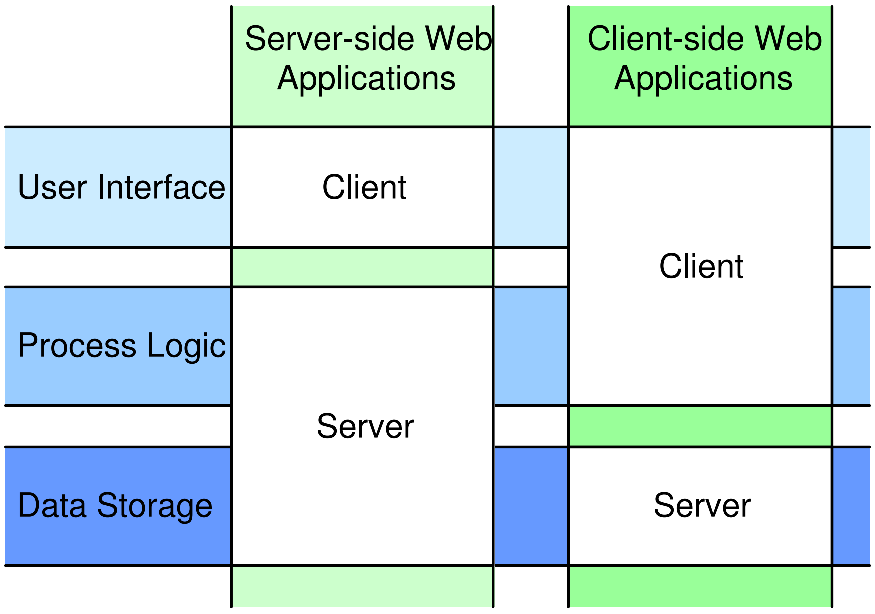


Figure 1.1: Web Applications in a Three-Tier Schema

The existing solutions for this problem all have one or another drawback making them inconvenient or impractical to use;[6] BWS is an attempt to overcome these drawbacks and provide a way to enable the creation of platform and scripting language independent client side web applications.

1.2 General remarks

All program code, class, method and variable names are set in a `monospaced typescript font`.

BWS is work in progress, this document relates to the 1.0 release of BWS available

[5]General Information on the three-tier application schema is for example available from [Fol98].
[6]Cf. section 3.

from the BWS website[7], further updates and changes will be documented on the website and, earlier, at the BWS documentation wiki[8], which is also linked from the BWS website.

BWS is licensed under the GNU General Public License[9] and is provided WITHOUT ANY WARRANTY, the documentation of BWS, including this document, is licensed under the GNU Free Documentation License[10].

Important terms are defined upon their first occurrence in footnotes.

1.3 Structure of this Paper

The section following this introduction, section 2, will present some advantages client side web applications have over conventional, operating system based applications. Section 3 will discuss some of the currently available solutions for creating such applications. Following this section, the technologies used as the basis of BWS are introduced. In section 5 a detailed, technical description of BWS is given before in section 6 some problems of BWS are discussed. Section 7 provides a step-by-step guide on how to create BWS applications. The final section recapitulates the major points of this paper and gives some information on future developments of BWS.

Readers only interested in using BWS and not in the inner workings of BWS will find all things relevant in section 7 and may want to skip all sections in front of this.

Despite being platform independent in theory, it can not be assured that BWS will work on any platform: In term of operating systems, BWS has been tested with Microsoft Windows operating systems and Linux and will probably be tested on the BSD-range of Unixes[11], OS/2 respective eCom-Station and BeOS. Concerning web browsers, BWS

[7]Cf. [Spe04a].
[8]Cf. [Spe04b].
[9]Cf. [Sta04].
[10]Cf. [Smi03].
[11]Including Darwin/MacOS X.

has been successfully tested on the Microsoft Internet Explorer[12], the Mozilla range of browsers[13], the Opera browser[14] and KDE Konqueror[15]. The term *relevant platform* in this paper means Microsoft Windows as well as Linux, *relevant browsers* are Internet Explorer, Mozilla/Firefox and Opera.

[12]Windows Version 6.
[13]Mozilla 1.4+ and Mozilla Firefox 0.7+.
[14]Opera 7.20.
[15]Konqueror 3.2.0. BWS on Konqueror only worked using the Gentoo Linux distribution, other distributions (SuSE 9.0, Fedora Core 1, Fedora Core 2 test 1) did not work and produced LiveConnect errors.

2 Economical and Technical Advantages of Client Side Web Applications

The various advantages of client side web applications can be separated to two categories: The advantages resulting from the use of web browsers for the presentation of an application and the advantages resulting from the client side execution of the applications.

2.1 Advantages of Web Based Applications

Compared with standard operating system based applications, applications running on browser platforms have several advantages:

Easy GUI Creation Using HTML and CSS as GUI markup languages, modern web browsers provide a standardized and easy way for creating GUIs available independent of browser and operating system platform. This allows a cost-effective development of GUIs for heterogeneous networks and avoids the risk of GUI based platform lock-ins.

Integrated Printing Functionality Web browsers generally provide printing functionality which consequently has not to be developed separately for each application.

Easy Porting of Existing Scripts Using HTML/CSS, existing script applications can be equipped with a modern, easy-to-use GUI with significantly less expense than with a conventionally programmed GUI.[1]

Central Maintenance As web based applications are usually retrieved from or run on a server, updating an application can be done in one central instance without the need to treat each client individually.

High Availability The basic requirement for web based applications, a web browser interpreting HTML/CSS documents, is fulfilled by all modern office PCs as well as many other devices possibly used as clients, e.g. PDAs, smartphones and set-top boxes.[2]

No Lock-In Effects As web based applications can be based on open standards supported by all major operating systems and browsers, lock-in effects requiring the use of a specific operating system or web browser, are avoided. Switching the underlying hardware or software platform of a web based application can be done without the need to specifically adjust a web based application to this platform.

High Usability Web browsers provide an environment almost all users know already. No special training for end-users is usually necessary because most users will know already how to do standard tasks like printing as this functionality is provided by the browser and not by individual applications.

[1]I.e. using the GUI programming capabilities provided by Java or a C++ class library.

[2]To be usable as a platform for BWS based applications, the client must also support several other requirements such as support for Java and LiveConnect (cf. section 4), which are currently not available on most *low-end* clients such as set-top boxes and smartphones.

2.2 Advantages of Client Side Applications

Web applications running on the client have multiple advantages over those running as server side applications:

Offline Usage Client side applications can be downloaded to the local client and then run without depending on network connections.[3]

Lower Network Load Client side applications do not have to exchange data as often with the web server as server side applications. While server side applications depend on the server for any small computation or integrity check, with client side web applications this is done on the clients and only the final results of client side applications have to be transferred over the network.

Lower Server Load Client side applications depend only to a very small extent on server resources. This allows the easy and cost-effective use of additional clients as server load increases minimally with each new client.

More Efficient Client side applications use primarily client side computing resources, which often are not as scarce as server resources. Additionally, because of lower requirements for stability, uptime, etc., client side resources are usually much cheaper than server side resources and especially for small web applications, these server resources are only used for data storage, not for running the applications themselves.

[3]If an application shall run without network access, it is of course necessary for the developer to consider this and to care that the application has all necessary code and data stored locally.

3 Current Solutions for Developing Client Side Web Applications

At the time of writing,[1] three solutions for developing client side browser based applications are existing:[2] JavaScript and Java are two platform independent options for developing such applications, Microsoft's ActiveScripting is an option for applications that only need to run on the MS Windows/Internet Explorer platform.

3.1 Platform Independent Solutions

Java and JavaScript both are platform independent solutions for creating client side web applications. Both are available on all relevant platforms; JavaScript is integrated in all relevant web browsers, Java comes separately and can be downloaded for free from Sun's Java website [Jav04b].

Despite the similarity in the name, Java and JavaScript are completely independent programming languages.

[1]First quarter of 2004.

[2]An additional option for creating client side web application is *Curl*, available from [Cur04]. Curl, developed by the Curl Corporation, is not based on HTML but uses a proprietary document format run inside the web browser using a Runtime Environment provided as a browser plug in. As Curl does not provide real integration with web browsers, it is not discussed further in this document, see [Cur02].

3.1.1 JavaScript

JavaScript/ECMAScript[3] is the common name for the scripting language defined in [ECM99]. It was originally developed by Brendan Eich at Netscape. The purpose of JavaScript was to give web developers the possibility to create interactive HTML documents; JavaScript is integrated with Netscape's Navigator web browser since version 2.0. An alternative implementation of a JavaScript interpreter was developed by Microsoft. This interpreter is not bound directly to Microsoft's Internet Explorer web browser but is integrated using OLE/ActiveX and COM. The Microsoft JavaScript interpreter however is not fully compatible to Netscape's interpreter; the Microsoft derivative of JavaScript is called *JScript*. The JScript interpreter is bundled with the Internet Explorer starting with its 3.0 release. Microsoft's approach to client side browser scripting, however, is not limited to JScript, section 3.2 gives more information on this.[4]

The development of the standard for the JavaScript language, standardized as ECMAScript, started in November 1996; the standard was adopted first in June 1997. It was also accepted as ISO/IEC standard 16262 in April 1998. As of February 2004, the standard is in its third edition and available from [ECM99], which corresponds to JavaScript 1.5.[5]

JavaScript is now available – to different extents – in all relevant web browsers. Additionally, stand-alone JavaScript interpreters exist that allow the use of JavaScript as a system script language like Perl or Python, i.e. outside of browser environments. One of the most prominent of these interpreters is the Rhino JavaScript interpreter maintained by and available from the Mozilla Foundation.[6]

For use as client side scripting language in web applications, JavaScript has some

[3]The language standardized in the ECMA-262 standard is called ECMAScript by the standard. This name came only up with the standardization of the language and never got used widely, JavaScript is still the more prominent name and will be used.

[4]Cf. [Wik04c].

[5]Cf. [ECM99, Section 'Brief History'].

[6]Cf. [Wik04c].

important disadvantages:

Limited Possibilites The possibilites provided by standard conform JavaScript are very limited. JavaScript only allows interaction with its parent document[7], not with any system or network resources. Tasks that require for example file system, database or network access are not possible with standard conform JavaScript.[8]

Missing Standard Conformance Another problem with JavaScript based solutions is that although JavaScript is standardized, different browsers interpret the same JavaScript code differently and any non-trivial application based on JavaScript has to be customized to the browser platform it will be used on. For an application that has to work on different platforms, for example Opera and Internet Explorer, some parts have to be coded for each browser separately. At run time, the script has to determine which platform it runs on and which code to use. A consequence of this is that JavaScript applications get too complex and expensive[9] if the resulting application shall be available on multiple browsers.

However, JavaScript also has some important advantages:

Easy to Learn JavaScript is a loose typed language, the syntax is a mix of elements from C and Java, a lot of simple examples and tutorials are available for free on the web.[10] Additionally, nothing but a web browser and a text editor is necessary to start programming with JavaScript.

Wide User Base JavaScript is integrated in all relevant browsers and thus provides a wide user base that can use JavaScript applications 'out of the box'. As JavaScript is always available when a web browser is installed, no costs of maintaining or installing JavaScript accumulate.

[7]The document that embeds or references the script.
[8]These limitations do not apply to Microsoft's JScript, cf. section 3.2.
[9]In terms of coding effort.
[10]E.g. at [W3S04].

Secure As JavaScript implementations conforming to the standard do not allow operations outside of their designated area – their parent document – potentially hazardous actions like reading content from other browser windows or accessing the local file system are theoretically not possible.[11]

3.1.2 Java

Java is an object oriented programming language developed by the Sun Microsystems engineer James Gosling as well as some other engineers at Sun. The development of Java was started in 1990; the first release of Java is available since November 1995. The design goals of Java are object-orientation, platform independence, distributed programming and security, especially the possibility to run remote code securely.

To reach the goal of platform independence, a 'two-step' approach is used: Java is not, as it is the case with conventional programming languages like C or C++, compiled into binary machine code but into an intermediate language called the Java *bytecode*. This bytecode is then interpreted by a bytecode interpreter, the *Java Virtual Machine*.

One of the most important reasons why Java was created platform independent was the intention to use Java as a programming language in heterogeneous environments, especially the Internet, where platform independence makes it possible to write and provide binary applications independent of the platform they shall be used on.[12]

Another feature of Java is that it allows the creation of applications that can be embedded in HTML documents: Java *applets*. These applets reside in and control a designated area of the HTML document. As Java, in contrast to JavaScript, is a full featured language that provides facilities for potentially hazardous tasks, these applets can be dangerous. To allow the use of Java applets from remote or unknown sources without the necessity of a security review by the user and without compromising system

[11]This is not the case for Microsoft JScript, cf. section 3.2.
[12]Cf. [Wik04a] and [Wik04b].

security, Sun created a secure environment, the *sandbox*, wherein all applets are run. This sandbox only allows safe operations by default; in order to allow hazardous tasks like file system access, this sandbox can be left when the user explicitly allows this. The security model of the Java sandbox is fine-grained, the user can specify exactly which applet should have which permission[13] depending on the location and the developer or vendor[14] of the applet.

The Java Standard Edition comes with a huge class library that provides classes for most standard tasks like reading files from local and remote locations, database access or TCP/IP connections. All of this functionality of Java is also available in Java applets.

With current versions of Java and web browsers, it is also possible for applets to 'leave' their designated area and interact directly with the currently loaded document using the possibilities provided by LiveConnect[15]. However, compared to JavaScript, interaction with document elements is much more complicated.

In a nutshell, the disadvantages of Java as a client side application language are:

Hard to Learn As Java is a fully-fledged programming language that is built on the object-oriented development paradigm and uses strong typing, it is not a programming language that is as easy to learn as most scripting languages.

Badly Suited to Small Problems The object orientation and strong typing of Java often make it necessary to write much more code for small problems compared to scripting languages that do not demand object oriented developing or strong typing.

Complicated DOM[16]Access Accessing the 'live' DOM of a document currently loaded by a web browser using LiveConnect is much more complicated than accessing it

[13]Permissions might be for example to open a specific file or to connect to a specified host on the Internet.

[14]Identification of users and companies is done using certificates.

[15]Cf. section 4.2.

[16]The DOM is an hierarchical object model of a XML document, cf. section 4.1.

with languages natively supported by the browser, for example JavaScript.[17]

Compilation Necessary Java code must be compiled to become platform independent bytecode, it can not be changed at runtime. Additionally, a Java compiler as well as all classes used in the source code to compile the code must be available on the client to enable the compilation of Java source code.

Compared with the other available solutions, the advantages of using Java are:

Real Platform Independence Java is a truly platform independent programming language available on all major operating system and hardware platforms. Incompatibilities or different implementations that make the cross-platform development of JavaScript based applications comparably complicated do not exist in Java.[18]

Class Library Java comes with an enormous class library that provides functionality for tasks from string manipulation to HTTP transfers and from image rendering to XML transformation.[19]

Speed A program written in Java will run in most cases much faster than a comparable program written in a scripting language like JavaScript or Rexx. Java execution speed is usually comparable to object-oriented native C++.[20]

Secure As described above, Java applets always run within the Java sandbox, which can only be left if the user explicitly grants specific rights.

[17]Cf. figure 4.6 and section 4.2.

[18]An exception to this is Microsoft's 'Java' Virtual Machine. This is, however, neither developed nor distributed or supported any more. Cf. [MSJ03].

[19]Cf. [Jav03b].

[20]Cf. [SH03a, german] and [SH03b, german].

3.2 A Proprietary Solution: Microsoft ActiveScripting

ActiveScripting, also called ActiveX Scripting, is a technology developed by Microsoft. It can be used for client side scripting in web browsers. It is a proprietary solution available only for the Microsoft platform, that is for the Windows operating systems and the Internet Explorer web browser.[21]

ActiveScripting is based on COM interfaces that can be used to communicate with the various supported scripting engines. It is not limited to a specific scripting language, instead scripting engines for various languages are available from Microsoft and other vendors, see table 3.1 for an overview of some available languages.[22]

Language	**Vendor** **URL**
VBScript	Microsoft `http://www.microsoft.com/scripting/vbscript/`
JScript	Microsoft `http://www.microsoft.com/scripting/jscript/`
Perl	ActiveState `http://www.activestate.com/Products/ActivePerl/`
Python	Python.org `http://www.python.org/windows/win32com/ActiveXScripting.html`
Object Rexx	IBM `http://www-306.ibm.com/software/awdtools/obj-rexx/`

Table 3.1: Active Scripting Engines

Used as a client side web scripting language, some of the advantages of ActiveScripting are:

Language Independent ActiveScript is not depending on a specific programming language developers must use; instead, it provides the possibility to choose one of several available languages like JScript, VBScript or Perl.

[21] ActiveScripting depends on the Windows Scripting Host and thus is not available on the Internet Explorer for MacOS.

[22] Developing a scripting language for languages other than these is possible, information on this is available at [MSW04].

Powerful ActiveScripting languages are based on the windows COM and can therefore use all parts of a system that provide COM interfaces, including system resources and most Windows applications.

Easy to Use Accessing a currently loaded document's DOM using an ActiveScripting language works in the same way as it works using JavaScript.[23] The learning curve for the use of a programming language other than JavaScript in browser environments is very shallow for any developer already knowing JavaScript.

In contrast to these advantages, ActiveScripting also has some severe disadvantages:

Insecure ActiveScripting within HTML documents can only be switched on or off, [24] a differentiated approach to allow or prohibit specific actions of a script is not feasible. It also is impossible to run scripts in a secure environment similar to the Java sandbox.

Platform Dependent ActiveScripting is limited to the Microsoft Windows/Internet Explorer platform and thus the creation of applications based on this technology creates massive lock-in effects and raises switching costs to other platforms as solutions based on this platform would have to be rewritten at least partially.

3.3 Comparison

Table 3.2 gives an overview of the afore mentioned assets and drawbacks of JavaScript, Java and ActiveScripting.

[23]Microsoft's JScript itself is an ActiveScripting language.
[24]The decision if scripting shall be allowed can be made based on signatures and URLs.

	Assets	**Drawbacks**
JavaScript	Easy to learn	Limited possibilities
	Wide user base	Missing standard conformity
	Secure	
Java	Real platform independence	Difficult to learn
	Huge class library	Badly suited to small problems
	Speed	Complicated DOM access
	Secure	Compilation necessary
ActiveScripting	Language Independence	Insecure
	Powerful	Platform Dependence
	Easy to use	

Table 3.2: Assets and Drawbacks of Available Client Side Scripting Solutions

4 Base Technologies of BWS

BWS is an approach to integrate existing technologies into one solution for developing platform independent client side scripted HTML/XML documents.

These technologies are Java, the Document Object Model (DOM), Java DOM communication with LiveConnect and the Bean Scripting Framework (BSF) in connection with the BSF supported scripting languages. The structure of these technologies is depicted in figure 4.1.

4.1 DOM

The DOM

> "is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents."[1]

It is standardized by the W3C DOM working group which accepts submissions from member companies and tries to implement their wishes in a interoperable and scripting-language independent solution.[2]

[1] [HW04, What is the Document Object Model?].

[2] The DOM standards are available from the W3C's DOM Technical Reports page [Hég04].

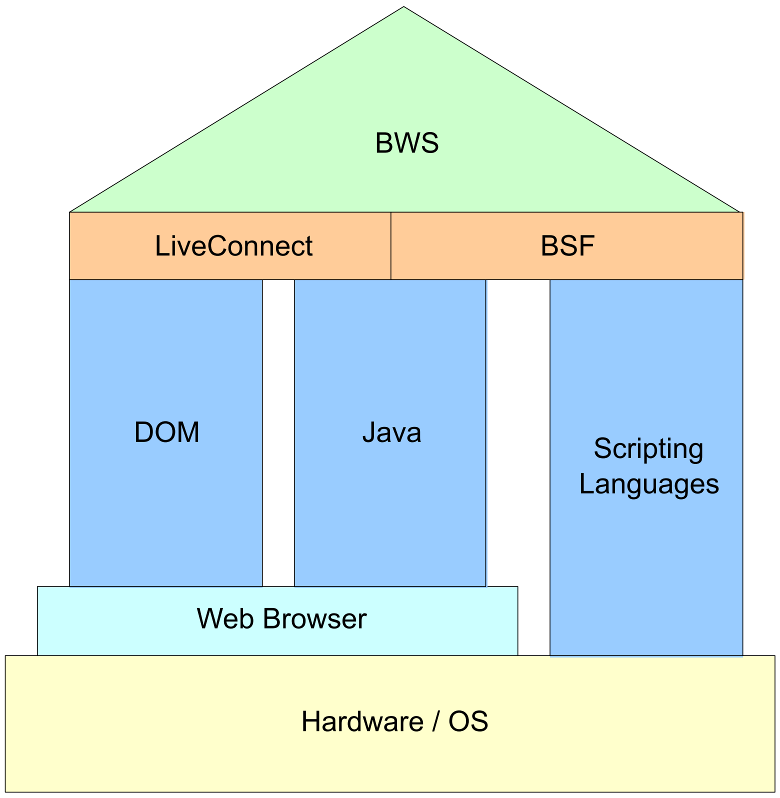


Figure 4.1: Structure of Technologies Used in BWS

4.1.1 DOM Basics

DOM objects are hierarchically organized into a DOM tree with the root element of a XML document being the root object of the DOM tree. Every part of a XML document corresponds to a DOM object: Figure 4.2 shows a simple XML document, figure 4.3 a graphic representation of the mapping of this document to a DOM tree.

The figure shows three types of DOM 'classes':

1. Element Nodes: These correspond to XML tags and are characterized by their type, e.g. a `h1` or a `title`.
2. Attribute Nodes: These correspond to XML attributes and are name-value pairs, e.g. the paragraph's `style` value is `Heading`.
3. Content Nodes: The part of the document between the tags that is the content of

```
<html>
 <head>
  <title>DocumentTitle</title>
 </head>
 <body>
  <h1 id="aHeading">Heading</h1>
  <br />
  <p id="aParagraph" style="fontFamily:sans-serif;">Content</p>
 </body>
</html>
```

Figure 4.2: A Simple XML Document.

the document.

Properties and methods available for these DOM classes are defined in the DOM standard; DOM for example defines that each element node may have an attribute `id` and that every element node supports the `appendChild()` method that expects another node as argument and attaches this node to the node the method is invoked on.

4.1.2 Navigating a DOM Tree Hierarchically

There are two standard ways of navigating a DOM tree in order to access a specific DOM element: The first one is to walk the DOM tree hierarchically: Iterate over all elements of the lowest layer, check if an element of this layer matches the desired criteria, then select all of this element's children and do the same for these an all their descendants until the wished elements are reached.

For example if in figure 4.3 the heading's content is wanted, the procedure would have to look like this:[3]

1. Select all child elements of the `html` node.

2. Check if the elements type is `body`.

[3]In order to access a element this way, the position of the element in the DOM tree must be known exactly.

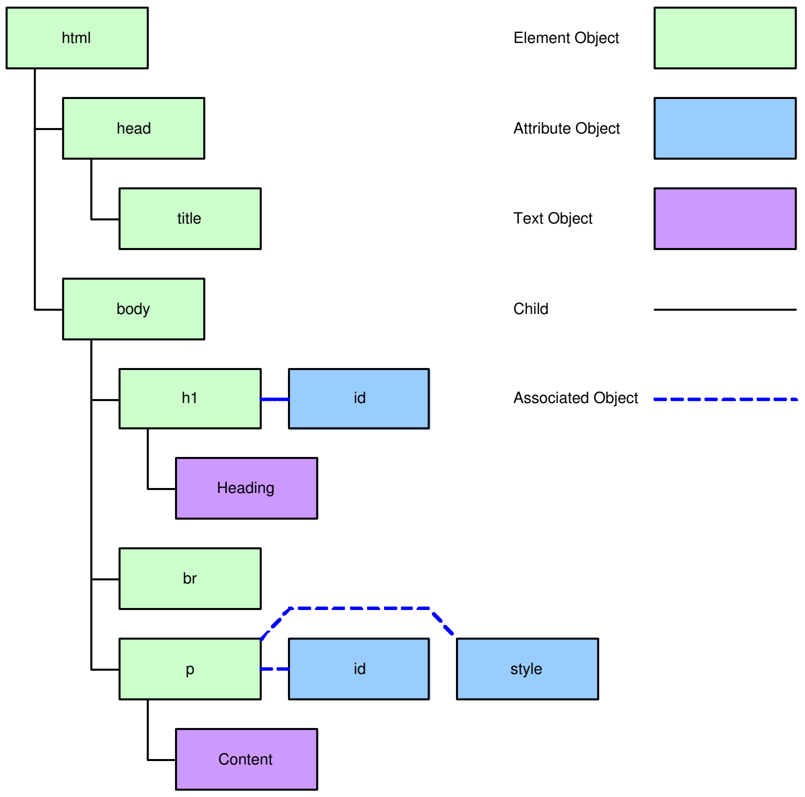


Figure 4.3: Graphic Representation of the DOM Tree of Figure 4.2.

3. If it is `body`, select all child elements of this element.

4. Iterate over these elements and check if they are `h1` elements.

5. If a element is of type `h1`, select its content.

As this method is very laborious and time consuming, especially for complex and large XML documents, another way of selecting specific elements of a DOM tree is necessary: the XML Path Language (XPath).

4.1.3 Selecting DOM Elements: XPath

XPath, standardized by the W3C in [CD99], is a XML element query language: It allows to specify certain criteria that are then used to select all elements of a XML document matching these criteria. The easiest way is to specify the path to an element: To address the `h1` element in figure 4.3, the appropriate XPath expression would be `/html/body/h1`, that is, the root element (`/`) and all relevant child types separated by forward slashes.

For selecting the `h1` content, the procedure would be this:

1. Select the `h1` element with XPath using `/html/body/h1`.
2. Get this element's content.

However, this way of selecting elements is still based on the full path of the elements within the DOM tree and does not allow accessing elements for example only by type.[4] This is necessary if the exact structure of the document is not known in advance. For this purpose, XPath provides the possibility to select elements independent of their path: XPath expressions using this method start with a double slash (`//`) instead of the single root slash. The expression for selecting all `h1` elements in a document e.g. is `//h1`.

This allows to use the following procedure for selecting the `h1` content:

1. Select all `h1` elements with XPath using `//h1`.
2. Get all selected elements' content.

The difference between the two examples using XPath above is that the second example would return any `h1` element within the document independent of its position, for example it would also return a `h1` lying under the `p` element in the given document, whereas the first expression would only return `h1` elements that are exactly in the third layer of the document and have `html/body` as parents.

Besides, these selections based on the element type, XPath supports selecting elements by the existence of specific attributes, by attribute values and several other criteria. A tutorial on the miscellaneous ways of selecting XML nodes using XPath is available from [NJ00].

[4]I.e. without specifying a full path.

4.1.4 Working With the DOM in Java Using dom4j

In order to work with these DOM objects, it is necessary that they are available in the programming language used; in this case the programming language is Java.

One of the solutions available for mapping DOM objects to Java objects is the open-source solution dom4j.[5] This framework enables the parsing of XML documents to Java objects, the navigation and modification of parts of the represented XML document as well as the creation of a XML document from the Java objects. dom4j provides hierarchical as well as XPath based access to DOM elements.

The first step in modifying an existing[6] DOM in Java is parsing a XML document and building its DOM tree in Java. dom4j uses its own implementation of a SAX reader for this purpose.

Reading the XML document is done by using a `org.dom4j.io.SAXReader` object's `read()` method on the source of the XML document; the source may be a `String`, an `URL`, a `InputStream`, a `Reader` or a `org.sax.InputSource`.[7] This method returns a DOM tree as an `org.dom4j.Document`.

The root element of the parsed document can then be obtained using the `getRootEle ment()` method on the document returned from the `SAXReader`. DOM Elements in dom4j are represented as objects of the class `org.dom4j.Element`.

To allow access to child elements of the current element, each `Element` provides a standard Java `Iterator`[8] that allows iteration over all child elements. Figure 4.4 shows a minimal dom4j example that provides two methods: `parseDocFromURL()` to read an XML document from an URL and `printRootChildren()` to print the element types

[5]dom4j is available from [SCM03], an introduction to dom4j is [RS01]; alternative frameworks for Java XML processing are for example Sun's Java API for XML Processing (JAXP) and the JDOM Project's JDOM framework.

[6]dom4j also provides the possibility to build DOM trees/XML documents from scratch. As this possibility is not used in BWS, it is not described in this paper.

[7]For details on these classes see their respective API documentation at [dom03] and [Jav03b].

[8]Available by calling the `Element`'s `elementIterator()` method.

of all first-generation descendants of the root element of the document read with the `parseDocFromURL()` method.

```
import java.util.Iterator;
import java.net.URL;

import org.dom4j.*;
import org.dom4j.io.SAXReader;

public class Dom4jExample {
  private Document dom4jDocument;

  /* SAXReader.read() throws a DocumentException when problems reading
   * or parsing an XML document occur, for example if the document is
   * not available at the specified place or if it is not well-formed.
   */
  public void parseDocFromURL(URL anURL) throws DocumentException {
    SAXReader xmlReader = new SAXReader();
    this.dom4jDocument = xmlReader.read(anURL);
  }

  public void printRootChildren() {
    Element documentRoot = this.dom4jDocument.getRootElement();

    Iterator elementIterator = documentRoot.elementIterator();

    // iterate over all children
    while(elementIterator.hasNext()) {
      Element currentElement = (Element)elementIterator.next();
      System.out.println(currentElement.getName());
    }
  }
}
```

Figure 4.4: A minimal dom4j example

As stated above, this iterative approach is not practicable for larger documents or documents with a previously unknown structure. In these cases, the XPath support of dom4j is helpful. It allows to select elements based on XPath queries and returns the results of this queries as a `java.util.Collection`. A shortened example of printing the number of `h1` elements in a document is given in figure 4.5.

```
import org.dom4j.XPath;
import org.dom4j.DocumentHelper;

...

  // create XPath query
  XPath xpathSelector = DocumentHelper.createXPath("//h1");

  // select elements from document
  List results = xpathSelector.selectNodes(document);

  // print number of elements
  System.out.println(results.size());

...
```

Figure 4.5: Using XPath with dom4j

Once the wished element has been found, several methods are available for modifying the element itself, its attributes and its content. See table 4.1 for an overview of the most important ones.

4.2 LiveConnect

LiveConnect is a technology that provides interfaces usable for interaction between Java, JavaScript and browser plug ins. It enables calls of Java methods from JavaScript as well as access from Java to the functionality of JavaScript.

The development of LiveConnect was started by Netscape Inc. in 1996; it was first available with the release of the Netscape Navigator 3.0. For the Internet Explorer, interaction between Java and the browser is done using the COM/ActiveX interfaces of Sun's Java Plug in. From a web application developer's point of view, there is no

[9] I.e. any change in the attribute is done directly to the referenced Attribute, not to a clone.
[10] Inherited unchanged from `org.dom4j.Node`.

Return Value	Method Head Description
`void`	`addAttribute(String name, String value)` Sets the attribute `name` to the value `value`.
`Attribute`	`attribute(int index)` Returns the attribute at `index`.
`Attribute`	`attribute(String name)` Returns the attribute `name`.
`List`	`attributes()` Returns as a backed[9]`List` of all attributes.
`String`	`attributeValue(String name)` Returns the value of the attribute `name`.
`String`	`getText()` returns the text value of the element.
`void`	`setAttributes(List attributes)` Sets the attributes of the element.
`void`	`setText(String text)`[10] Sets the Text of the Attribute to `text`.

Table 4.1: Important `org.dom4j.Element` Methods

difference between LiveConnect on Netscape/Mozilla or the Internet Explorer, interaction works using the same classes and methods. The other relevant browsers, Opera and Konqueror, provide their own implementation of LiveConnect.

For invoking JavaScript and modifying the DOM from Java, Netscape provides the `netscape.javascript.JSObject` Java class that capsules all non-primitive data types passed between JavaScript and Java. This class also provides the only way for Java to access the current documents DOM.[11] Due to this very generic DOM interface, modifying the DOM from Java is comparatively elaborate. See figure 4.6 for a comparison of modifying a DOM element from Java compared to modifying it from JavaScript.[12]

[11]From Java 1.5 on, Sun plans to provide another, better adjusted possibility for accessing the DOM called the *Common DOM API*, cf. section 8.

[12]Cf. [Jav03c, Chapters 24 and 25].

Setting the background color of the first heading to red in JavaScript and Java.

```
firstHeading=window.getElementsByTagName("h1")[0];
firstHeading.style.backgroundColor="red;"
```

JavaScript

```
Object[] objectArray=new Object[1];
JSObject appletWindow=this.getWindow();
objectArray[0]="h1";
JSObject headingArray=
  (JSObject)appletWindow.call(getElementsByTagName,objectArray);
JSObject firstHeading=(JSObject)headingArray.getSlot(0);
JSObject styleAttribute=(JSObject)firstHeading.getAttribute("style");
objectArray[0]="red";
styleAttribute.setMember("backgroundColor",objectArray)
```

Java

Figure 4.6: Comparison of DOM access in Java and JavaScript

4.3 BSF

The Bean Scripting Framework (BSF) is

> "a set of Java classes which provides scripting language support within Java applications. It also provides access to Java object and methods from supported scripting languages."[13]

The main accent of BSF usage at the moment[14] is the use of scripting languages within JSPs and to allow these scripting languages access to the Java class library. Additionally, BSF enables the creation of Java applications completely or partially in scripting languages. To enable these functionalities, BSF provides an API that allows the invocation of scripting engines from Java and an central object registry, the BSF registry, that allows to register and retrieve objects from scripting languages as well as from Java.

[13][Pro02b, What is Bean Scripting Framework?].
[14]February 2004.

BSF was initially developed at an IBM research center with the motivation to provide access to Java Beans from scripting languages. It was moved to IBM's AphaWorks[15] site[16] where it found significant interest. This led IBM to move it on to its developerWorks[17] site[18] where it was developed as an Open Source project until the release of BSF 2.2. BSF was incorporated into IBM's application server WebSphere and also into the Xalan XSLT processor developed by the Apache project.

The interest of the Apache project in BSF finally led IBM to hand over the project to the Apache foundation where it is developed since 2002 as an subproject of The Apache Jakarta Project[19]. As of February 2004, the current release of BSF is 2.3.

4.3.1 BSF Architecture

BSF consists of two primary components, the `BSFManager` and the `BSFEngine`. The `BSFManager` maintains the BSF object registry and handles the scripting engines. To use the BSF, a Java application must instantiate a `BSFManager` and then either load a scripting engine directly using the `BSFManager`'s `loadScriptEngine()` method[20] or indirectly by passing a script directly to the `BSFManager` that loads the appropriate scripting engine transparently. The `BSFManager` also caches scripting engines so they only have to be instantiated once.[21]

For an overview of the structure of BSF and the two methods of executing scripts see figure 4.7. An overview of the methods provided by BSF is available at [Pro02a].

[15]Cf. [Alp01], homepage: [Alp04].
[16]A website whereon IBM provides developers access to its latest innovations.
[17]developerWorks 'is IBM's technical resource for developers' [Dev04b]; it provides technical information and code for developers using IBM and open standards technologies like WebSphere, DB2, Java, Linux, etc.
[18]Cf. [Dev04a].
[19]Cf. [Pro04].
[20]Loading scripting engines explicitly requires one instance of `BSFEngine` for each scripting language.
[21]Cf. [Pro02a].

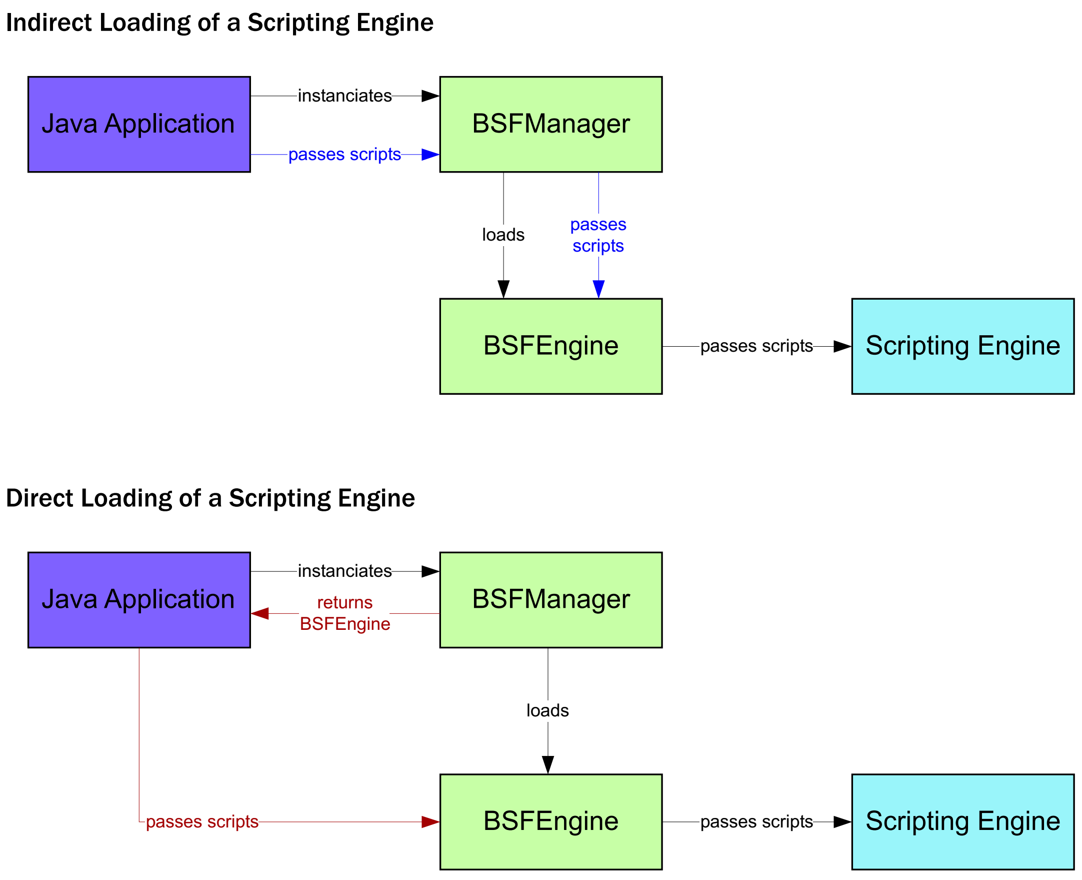


Figure 4.7: BSF Overview

4.3.2 BSF4Rexx

BSF4Rexx enables the binding of the Object Rexx[22] and the Regina[23] rexx interpreters to the BSF by providing a Rexx scripting engine for use with the BSF. BSF4Rexx was developed initially by Peter Kalender, a student at the University of Essen, in the winter semester of 2000/2001, for a term paper.[24] BSF4Rexx has since been extended and is maintained by Prof. Rony Flatscher of the WU Wien; it is available from [BSF03]. Prof. Flatscher also wrote two papers on BSF4Rexx, one about the early "Essener" Version[25]

[22]An object-oriented rexx interpreter by IBM that also supports classic rexx. Object Rexx is available for Windows, AIX, OS/2, Sun Solaris and Linux.
[23]An open source rexx interpreter developed by Mark Hessling and available on around twenty different operating system platforms.
[24]Cf. [Kal00].
[25]Cf. [Fla01].

and one about the current "Augsburg" version[26].

4.4 Relations Between the Base Technologies

The technologies described above are used in BWS in the following constellations:

For document rewriting, dom4j is used to transform the user created BWS document to a Java object tree. Using other methods provided by dom4j, these objects are then modified to represent a browser interpretable document. This resulting object tree is then serialized to a HTML document.

During execution of the document, LiveConnect is used to provide DOM access to the runtime environment applet, BSF is used to run the scripts and forward their DOM method calls.

For a graphical representation of these relations see figure 5.1.

[26]Cf. [Fla03].

5 BWS in Detail

Technically, BWS is split in two parts: The first part transforms a BWS document[1] to a browser interpretable BWS/HTML document, the second part provides the runtime environment that connects the different technologies necessary for script execution and DOM access.

As depicted in figure 5.1 the first part of BWS, document rewriting, can be integrated into the second one so that working with BWS documents for the user is similar to working with conventional HTML documents.

5.1 Classes and Their Relations

BWS consists of two separated sets of classes:

- The first set is based on the `BWSDocument` class that represents an BWS document and provides methods to transform it to a browser interpretable HTML document and is used by the `BWSRewriter`, the `BWS2XHTML` and the `RewriterApplet` class.[2] `BWSRewriter` reads the BWS document from an specified URL, rewrites the document and prints messages useful for debugging a failing document; `BWS2XHTML` also

[1]The term *BWS document* refers to a document meeting the requirements stated in section 7.3.1, p. 64, a *BWS/HTML* document is a BWS document transformed to a browser interpretable document as described in this section.

[2]`BWSRewriter` and `BWS2XHTML` are command line applications that expect the URL of the BWS document as their first parameter.

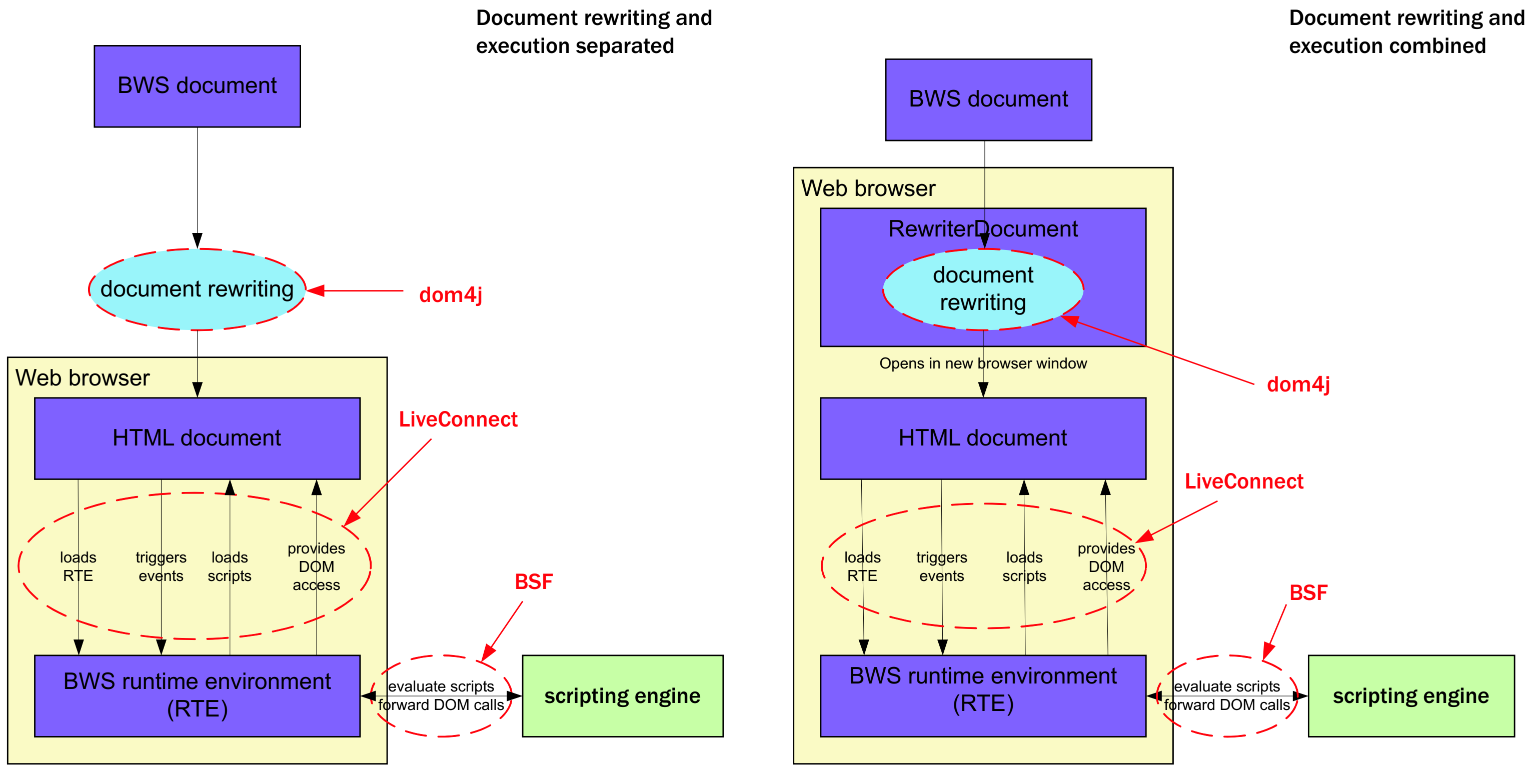


Figure 5.1: Technical Overview of BWS

reads the document from a specified URL and prints only the transformed document to the standard output. Another option for BWS to BWS/HTML transformation is the `RewriterApplet` that reads a BWS document from an URL, transforms it to a BWS/HTML document and directly opens it in a new browser window.[3]

- The second set is based on the `BWSApplet` class that provides the runtime environment for BWS scripts. This class utilizes the `JSNode` class, a capsule class for HTML/XML nodes[4], and the `ScriptString` class that is used for the interpretation and evaluation of script calls from events.

Figure 5.2 shows which elements of a document are represented by which classes, figure 5.3 shows the relations between the classes and their relations to their environment.

5.2 BWS Implementation[5]

The following section will describe the two parts of BWS in detail as they are implemented. To follow the description of the individual processes, it is recommended to have the source code of the described classes at hand.

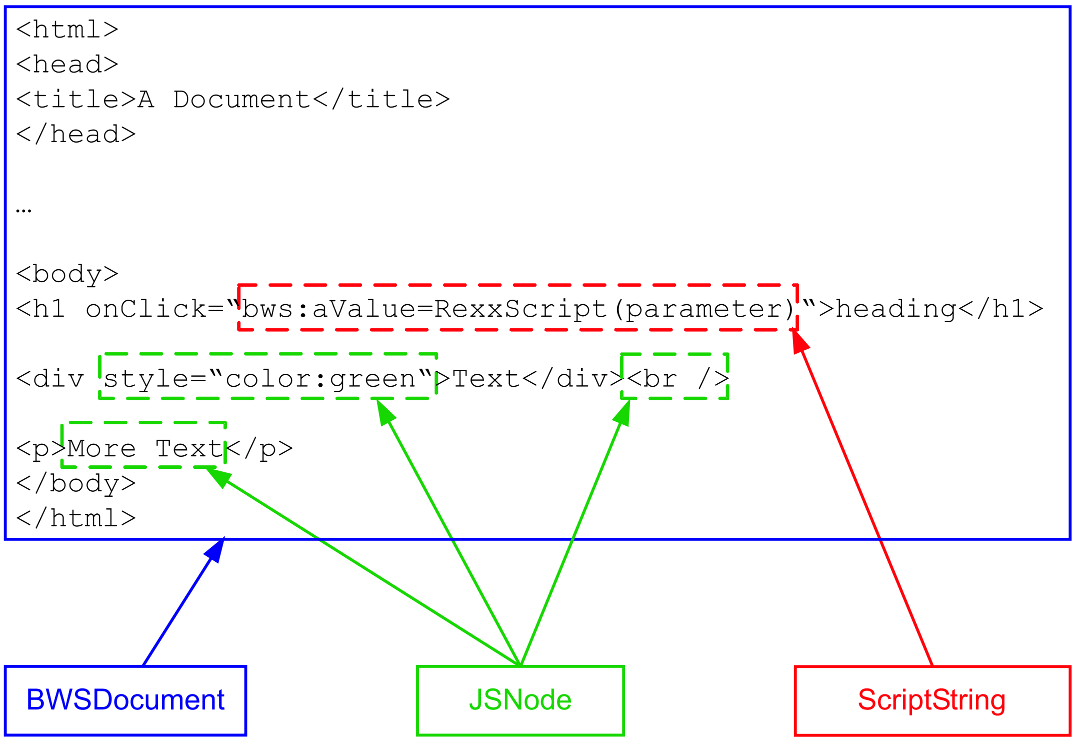
[3]This is done using the DOM `document.write()` method. It does not work on the Internet Explorer as the Internet Explorer does not load applets referenced from tags it gets passed using this method.

[4]DOM objects are capsuled independent of the node type, i.e. element, text and attribute nodes are all represented as `JSNode`; this is the same as it is with the DOM's `node` 'class'.

[5]The code of BWS as it is described here is available via anonymous CVS. The CVS server is `cvs.berlios.de`, the repository is `bsfws` and the 1.0 relase described in this paper is contained in the `org` module and tagged as `Release-Branch-1_0`. It can be checked out with these commands:

```
cvs -d:pserver:anonymous@cvs.bsfws.berlios.de:/cvsroot/bsfws login
cvs -z3 -d:pserver:anonymous@cvs.bsfws.berlios.de:/cvsroot/bsfws co org -r Release-Branch-1_0.
```

There is no password set for the anonymous account.



```
<html>
<head>
<title>A Document</title>
</head>

...

<body>
<h1 onClick="bws:aValue=RexxScript(parameter)">heading</h1>

<div style="color:green">Text</div><br />

<p>More Text</p>
</body>
</html>
```

Figure 5.2: BWS Documents - Representation in Java

5.2.1 Details of Document Rewriting

Document rewriting is the process of rewriting a BWS document conforming to the requirements defined in section 7.3, p. 63 to a BWS/HTML document interpretable by the browser. To achieve this, BWS script calls must be transformed to JavaScript calls to the `BWSApplet`'s `executeScript()` method; additionally, the BWS runtime environment, provided by the `BWSApplet` class, must be embedded in the document. As shown in figure 5.1, this process can be run separately from the application execution itself or it can be run directly at runtime by using the `RewriterApplet`; then of course a *starter document* embedding this applet must be used.[6]

[6]An example for such a starter document is the `RewriterDocument.html` contained in the BWS distribution. As stated above, this does not work with the Internet Explorer.

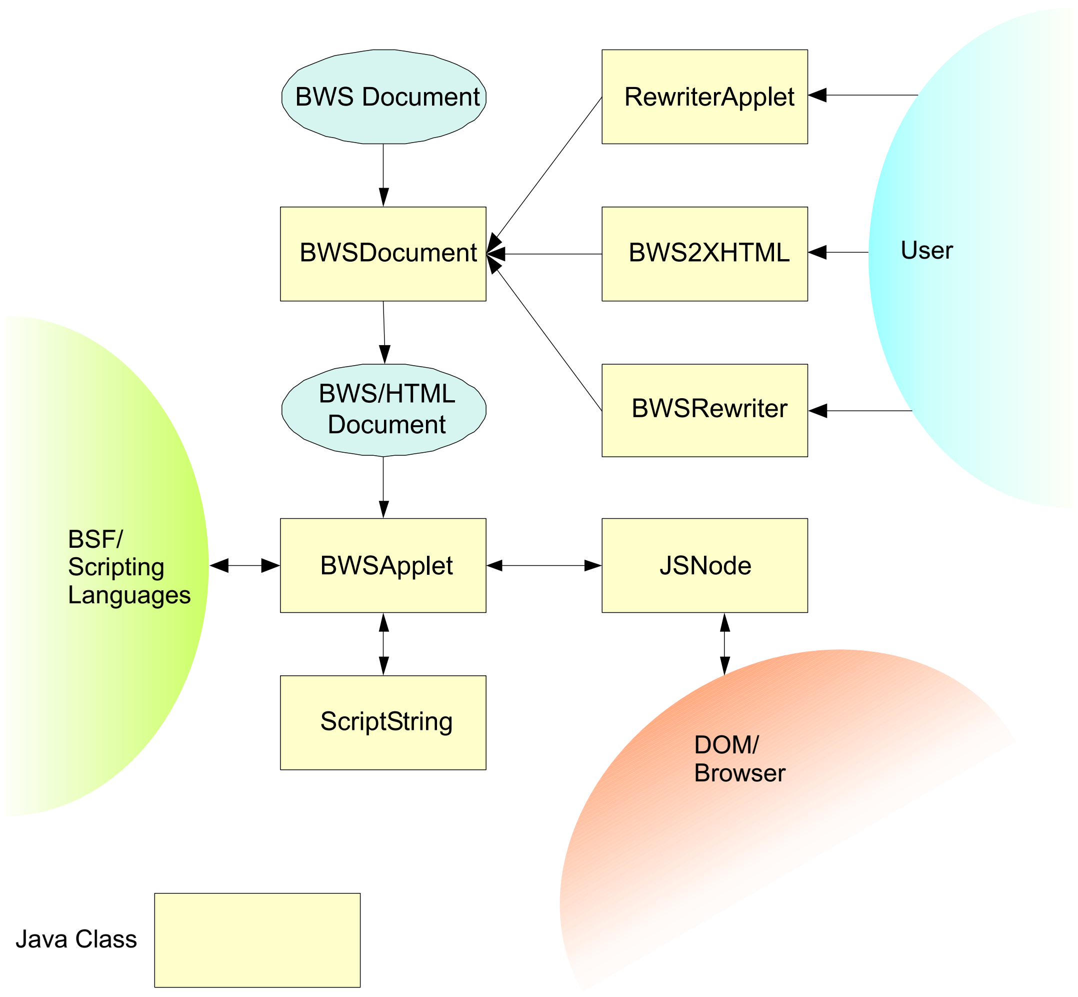


Figure 5.3: BWS Classes Overview

5.2.1.1 Using BWSDocument

BWS provides two command line applications and one applet for document rewriting: Both command line applications, `BWSRewriter` and `BWS2XHTML`,[7] expect the URL of the BWS document as the first argument of their invocation and both print the resulting BWS/HTML document to the standard output from which it can be redirected for example to a file. The difference between both applications is that `BWSRewriter` prints

[7]Despite the name, `BWS2XHTML` at the moment does not produce valid XHTML documents. The reason for this is that the `applet` tag is not allowed in XHTML; `applet` however currently is the only working way to embed an applet regardless which browser is used to open the embedding document.

out additional debugging information and is not be of use for document rewriting; in contrast to this, `BWS2XHTML` only prints out the final BWS/HTML document and thus can be used for example to save the standard output to a file.

The `RewriterApplet`, contrary to the command line applications, is a more end-user oriented solution: In combination with the `RewriterDocument`, it offers a HTML user interface where the user can specify the location of a BWS document as an URL. This document is then retrieved and transformed to a BWS/HTML document. The resulting document is instantly opened in a new browser window where the user can immediately work with it.[8]

5.2.1.2 BWSDocument Class

The `BWSDocument` class is, as shown in figure 5.2, a Java representation of a BWS document. It provides methods to transform a BWS document to a browser interpretable HTML document by rewriting the script calls from BWS script calls to JavaScript calls calling an applet method that runs the desired script. Additionally, it embeds the runtime environment applet in the HTML document.

For these purposes, the class provides nine methods, an overview of these method is available in the class diagram in figure 5.4. The `printDocumentSource()` method is used to print the source code of the currently represented DOM tree to the standard output; the `printVector()` method can be used to check if all BWS script calls are gathered correctly by the `BWSDocument` class. The other methods are used for reading a document from an URL and rewriting it.

The usual work flow for transforming documents is as follows:

1. Read and parse the document.

2. Gather all script `id`s occurring in the document.

[8]This does not work with the Microsoft Internet Explorer at the moment, see above.

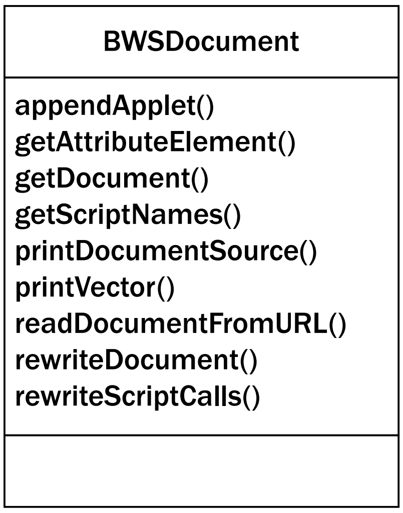


Figure 5.4: Class diagram: `BWSDocument`

3. Search all attributes of all elements if they reference a BWS script and replace the script call if this it the case.

4. Embed the runtime environment applet in the document.

This work flow corresponds to the following method calls of `BWSDocument`:

1. `readDocumentFromURL(String URL)`[9].

2. `getScriptNames()`[10].

3. `rewriteScriptCalls()`[11].

[9]The `readDocumentFromURL()` method creates an `URL` and a `SAXReader`. The `SAXReader`'s `read()` method is then used on the `URL` and its result is assigned to the `xmlDocument` property of `BWSDocument`.

[10]`getScriptNames()` creates a `XPath` object selecting all script elements (The XPath expression for this is `//script`, see section 4.1.3, p. 29). This `XPath` expression is then used to select the matching nodes which are referenced in the `results List`. Using an `Iterator`, all scripts are parsed for their `id` attribute. The value of the `id` attribute is then added to the `scriptNames Vector`, which is a `private` property of the `BWSDocument` class. If an `id` is not available, the `name` attribute is used instead, if this also is missing, the script is ignored.

[11]`rewriteScriptCalls()` is a very short method that iterates over the `scriptNames Vector` and calls the `getAttributeElement()` method for each of the `Vector`'s elements.

`getAttributeElement()` is the method used to actually rewrite the script calls: It gets the name of a script embedded in the document and replaces all occurrences of BWS calls to this script with JavaScript calls to the `BWSApplet`'s `executeScript()` method.

4. `appendApplet()`[12]

Instead of calling `getScriptNames()`, `rewriteScriptCalls()` and `appendApplet()` individually, the `rewriteDocument()` method which calls these three methods can be used.

Attribute	**Value**
`code`	`org.tsp.bws.BWSApplet`
`id`	`BWSApplet`
`width`	`0`
`height`	`0`
`mayscript`	`true`

Table 5.1: Attributes and Values of the `applet` Element

The resulting final document[13] can then be printed out to the standard output using the `printDocumentSource()` method; alternatively, the complete document can be obtained in a `String` using the `getDocument()` method.

5.2.2 Details of Application Execution

Running BWS applications is enabled by the BWS runtime environment, consisting of the `BWSApplet` applet and the `ScriptString` and `JSNode` 'helper' classes.

5.2.2.1 BWSApplet

The `BWSApplet` class is the centerpiece of the BWS runtime environment. It is embedded automatically during document rewriting in BWS/HTML documents and provides

Upon invocation, `getAttributeElement()` queries the document for all attributes of all elements (XPath query: `//@*`). It then iterates over these attributes and checks if the attribute's value matches `bws:ScriptString` or `#:ScriptString`; if this is the case, the attribute value is replaced with the correspondent JavaScript call of the form `document.getElementById('BWSApplet').executeScript('scriptString',this)`.

[12] `appendApplet()` first retrieves the `body` element of the document (Using XPath; query: `/html/body`) and then adds a new `applet` element and sets the attributes specified in table 5.1 to the values specified in this table.

[13] The document is stored in the `BWSDocument`'s `xmlDocument` property.

methods for running scripts embedded in or referenced from BWS/HTML documents as well as methods to easily obtain nodes of the current document. It is, as any applet must be, an extension of the `java.applet.Applet` class.

`BWSApplet` provides six public methods and a standard constructor.[14]

Two methods, `init()` and `destroy()`[15] are overwritten `Applet` methods that are automatically executed on loading respective unloading of the applet.

The `init()` method creates a new `BSFManager` and stores three objects to the BSF registry so these are always easily available in any BWS script. These objects are:

- The BWS applet itself.[16]
- The `java.lang.System.out` object to allow scripts access to the Java console.[17]
- The window the applet resides in.[18]

Of the remaining four methods one, `getNode()`, is a shortcut to obtain a DOM node as a `JSNode` object using the applet,[19] the other three methods are described in the following part of this section in detail.

5.2.2.1.1 Executing Scripts The `executeScript()` method is the core of the `BWSApp let`. It is called from the BWS/HTML document by the JavaScript/DOM event handlers and upon calling, runs the scripts specified with the method call. The process flow of script execution is shown in the sequence diagram in figure 5.5.

It can be called either with one or with two parameters. If it is called with one parameter, this parameter must be a `String` specifying a script call string; the two

[14]The constructor only prints a startup notification to the standard output.
[15]`destroy()` is a rudimentary method that sets the `BSFManager` `null` and prints a string to the standard output.
[16]Registry key: `BWSApplet`.
[17]Registry key: `SystemOut`.
[18]Registry key: `DocumentWindow`.
[19]This method expects the node's `id` as the only parameter to the method call.

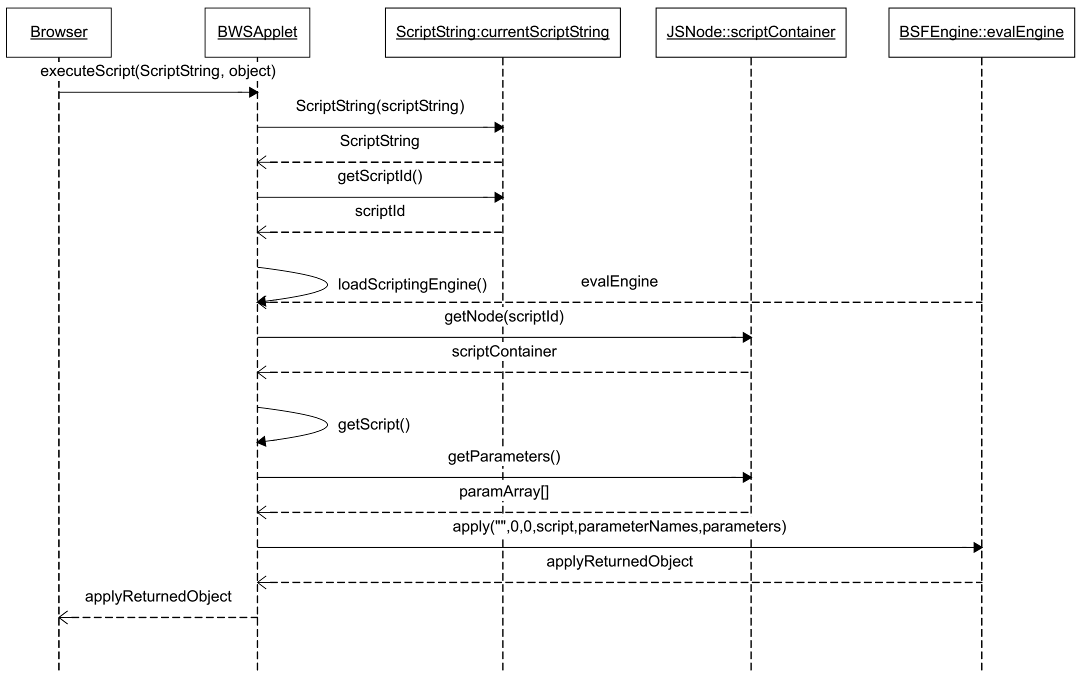


Figure 5.5: Script Execution Sequence Diagram

parameter variant expects an DOM node, represented as a `JSObject`, as additional second parameter.[20]

The first step in the execution of a BWS script is the interpretation of the script call string passed with the script call. This is done using the `ScriptString` class. After the script `id` has been determined by `ScriptString`, it is used to load the appropriate scripting engine and look up the script code using the `loadScriptingEngine()`[21] and `getScript()` methods.

After the engine has been loaded and the script is available, the script parameters are evaluated. This is done by first reading the parameters keys from the `ScriptString` with `getParameters()` and then passing these keys to the `evaluateParameters()` method described below.[22]

[20]This second parameter may be accessed within the script string's parameters section using `this` as a shortcut; it usually is a `JSObject` representation of the HTML event that triggered the execution of the script, if `executeScript()` is invoked from another script and not from a DOM event handler, it may be an arbitrary object. Cf. section 7.4.1, p. 69.

[21]This is a `private` method and therefore not shown in the API documentation.

[22]Additionally, a `Vector` is created that contains the names of the arguments; as the original names

As scripts may possibly need to leave the Java sandbox, they have to be executed within a privileged environment[23] which is available in Java using the `java.security.AccessController`'s `doPrivileged()` method with an anonymous instance of the `java.security.PrivilegedAction` interface.[24] Script execution is done using the `apply()` method provided by the `BSFEngine`.[25] The object returned from the script, i.e. from `apply()`, is stored in an `Object` created before script execution.

After script execution, the returned object as well as the return key[26] is checked if it contains a value or `null`. If a return key was specified and the script returned a non-`null` value, this object is stored in the BSF registry under the specified key and returned to the calling method. If no return key was specified or the script returned no value, nothing is stored to the registry and the returned value, independent if it is `null` or an actual object, is returned to the caller.

5.2.2.1.2 Loading a Scripting Engine Loading a scripting engine in BWS is done in the `loadScriptingEngine()`[27] method and works in two steps:

1. The language of the script and the according scripting engine is determined by reading the respective script's `type` attribute. This is done by calling the `BWSApplet`'s `getScriptingEngine()` method.[28]

of these objects are not known, they are named `argumentX` where `X` is an increasing integer index starting at zero; i.e. the arguments are named `argument0`, `argument1`, `argument2`, ...

[23]Privileged environments can be equipped with less restrictions, depending on the Java security policies set, cf. section 7.2, pp. 62.

[24]The portal page for Java 2 SDK 1.4.2 security is [Jav04a]; for details on Java security architecture see [Jav03a], for information on the Privileged Block API [Jav01].

[25]This method expects six parameters: The first three are not used and therefore left empty respective zero in BWS, the fourth is a `String` containing the complete script code, the fifth and sixth are `Vectors` containing the names of the parameters passed and the parameters for the script.

[26]The return key is obtained from the `ScriptString` object using its `getRetKey()` method just before script execution.

[27]This is a private method of `BWSApplet` and therefore is not shown in the generated API documentation of BWS. It expects the `id` of the script that is to be invoked as its only parameter.

[28]`getScriptingEngine()` gets passed the script `id`; it creates a `JSNode` of the script element and reads the script element's `type` attribute. A substring of this type attribute, the part behind the slash stored in the `engineString`, specifies the `BSFEngine` that shall be used as the interpreter of the script. This `engineString` is then returned to the `loadScriptingEngine()` method.

2. The engine itself is loaded explicitly as a `BSFEngine`[29] with using the `BSFManager`'s `loadScriptingEngine()` method.

After the appropriate scripting engine has been determined, the loading procedure starts: As the scripting engine may be based on native code,[30] the JNI might be used for loading the engine. This is normally not allowed for applets and thus this code must be granted the rights to do so. This is again done here using a privileged environment[31]

Within this privileged block, only the direct loading of the scripting engine takes place; after the engine has been loaded, it is returned out of the privileged environment and from there on directly to the method that called `loadScriptingEngine()`.

5.2.2.1.3 Loading Script Code Loading the script code is done using the method `getScript()`:[32] This method expects a `JSNode` of the script element and returns the script code as a `String`.

The script code can be either embedded between the `script` tags or referenced specifying the scripts location as the `src` attribute of the `script` tag. The location may be specified as an absolute or relative path or as an URL.

The first action of the `getScript()` method is to determine if the script is embedded or referenced: This is done by checking if the `script` element has got a `src` attribute that is not `null` or empty. If it has not got one, the text within the script tag is read[33] and returned. If a `src` attribute is available, the content of the `script` tag is ignored and the method tries to read the script code from the specified location.

If the script is referenced, it has to be determined if the specified `src` value is an URL

[29]Cf. figure 4.7, p. 37.
[30]I.e. written not in Java but in C/C++ or any platform specific, compiled code. An example for such an engine is the Rexx engines of BSF4Rexx which uses ObjectRexx or Regina Rexx as interpreters.
[31]See section 5.2.2.1.1 directly above.
[32]`getScript()` is a `private` method and therefore not shown in the API documentation.
[33]This is either done using the DOM conform `getData()` method, or it this fails using the `getInnerHTML()` method; both methods are methods of `JSNode`. `getData()` at the moment only works on Mozilla browsers.

or an absolute or relative path: This is done by checking if the value of the `src` attribute

- contains a colon followed by a double forward slash (`://`),
- starts with a slash (`/`) or
- matches neither of these criteria.

Values of `src` containing `://` are treated as URLs, values starting with `/` are treated as absolute paths[34] while values matching neither criteria are treated as relative paths.[35]

After the type of the reference has been determined, the script is read from the specified location to a `String` and returned to the method that called `getScript`.[36]

5.2.2.1.4 Mapping Objects to Parameters As described in section 5.2.2.2, script calls may contain parameters passed to scripts upon execution. Since BWS only expects a script call string for method invocation, objects that shall be passed to the script can not be passed directly to the script, but only be referenced by a key within this call string.[37] The look-up of the objects referenced by the key is always done at runtime of the script, therefore all objects are always used in their current state.

The `BWSApplet` provides the `evaluateParameters()` method for mapping these keys

[34]When an absolute path is used, the protocol, host and port of the embedding document's URL are obtained using the applet's `getDocumentBase()` method. The specified path is then appended to the `String` representations of protocol, host and port of the document's URL.

[35]For a relative path, the embedding document's URL is read; this URL is then converted to a `String`. The URL of the script is then obtained concatenating the applet's URL with the relative path specified as the `src` attribute.

[36]This is done using conventional stream reading: i.e. a stream is created on the specified location, read to a byte array and a `String` is created from this array.

[37]The reason for this limitation is that both possibilities theoretically available for passing parameters directly are not feasible: The first possibility would be to overload the `executeScript()` method with any possible combination of objects that could be passed to the script. This option would lead to a nearly unlimited number of `executeScript()` methods and thus is not a real possibility. The second theoretical option would be to wrap all objects that shall be passed into one container (as e.g. the `BSFEngine`'s `apply()` method does with `Vector`s). This however would have to be done by the person writing BWS documents and scripts and would make the creation of these scripts and documents overly complex and thus also is not a real alternative to the parameter mapping used currently.

to their objects.[38] When this method is called, keys are mapped to objects in the following order:

1. If the key is `this`, the object that invoked this script, e.g. the HTML element that triggered the execution of a script, is used.
2. If the key is in quotation marks, its quotation marks are stripped off and the key itself is used.[39]
3. If an object in the BSF registry is available under the key, this object is retrieved and used.
4. If a DOM node has the `id` specified as the key, this DOM node is used.[40]
5. If none of this criteria match, the key itself is used.[41]

As soon as the first of these criteria matches, the matching object is used and the next key evaluated. After all keys have been mapped to their respective objects, they are returned to the method that called `evaluateParameters()`.[42]

5.2.2.2 ScriptString Class

`ScriptString` is the class providing interpretation of BWS script call strings for the runtime environment. These call strings are the strings used in DOM event handlers to call BWS scripts. They generally are formed according to this example: `returnValue= scriptId(parameter1,parameter2)`. Script call strings are passed to the `BWSApplet`

[38]This also is a `private` method and therefore not shown in the API documentation. The method expects a `String` array containing the parameter keys and returns a `Vector` holding references to the objects found under the specified keys. The objects are always mapped and passed to the script in the exact order in which they were specified in the script call string.

[39]As a `String` object.

[40]A `JSNode` reference to this node is created and used.

[41]As a `String` object.

[42]During evaluation, all objects that were mapped are put on a `Vector`, this `Vector` is returned to the calling method.

using the `executeScript()` method and then have to be interpreted by the runtime environment to call the appropriate script, lookup the specified parameters and store the return value to the specified place after script execution. A class diagram of the `ScriptString` class is depicted in figure 5.6.

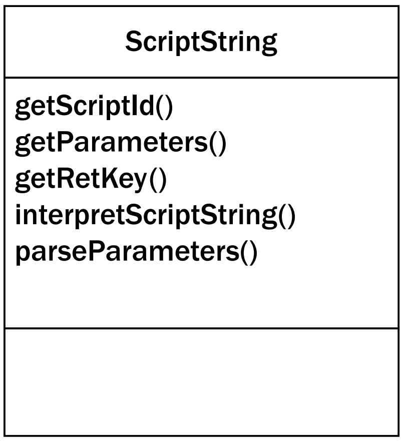


Figure 5.6: Class Diagram: `ScriptString`

The `get` methods[43] of this class make the results of the interpretation available for other classes. The interpretation itself is done in the `ScriptString`'s `interpretScriptString()` method using the `parseParameters()` method to interpret the parameter part of the string.

Interpretation of the script strings works according to the following description:

1. Before the string is interpreted it is tested if it assigns a return value and if it has got parameters that are to be passed to the script.[44]
2. If the string contains neither return value nor parameters, it is interpreted as only the script `id` and stored in the `ScriptString` property `scriptId`, available using `getScriptId()`.
3. The part of the string recognized as the parameters part is passed to the `parseParameters()` method, the `String` array returned by this method is stored as

[43]`getParameters()`, `getRetKey()` and `getScriptId()`.
[44]The string is tested for the assignment operator (`=`) and a left parenthesis (`(`).

the `parameters` property and can be obtained by calling the `getParameters()` method.[45]

4. If available, the return value key[46] is stored as `returnValue`. It is available with `getRetKey()`.

The assignment of values to the keys, i.e. the creation of references to DOM nodes etc. is not done during `ScriptString` interpretation but at runtime using the keys obtained during interpretation.

5.2.2.3 JSNode Class

`JSNode` is a class providing the functionality of the DOM's node object directly in Java using LiveConnect for Java to DOM communication.[47] It can be used to reference and modify already existing nodes or to create new ones. In addition to the methods defined for the DOM node, it also provides some helpful methods not available there.

It is important that `JSNode`s are only references to DOM nodes, they are not the nodes themselves. Therefore, creating a new `JSNode` object *does not* implicate the creation of a new DOM node, but only the creation of a reference to such a node. Creating DOM nodes must be done explicitly using the `create` methods of `JSNode`.[48]

5.2.2.3.1 JSNode Method Overview A `JSNode` object can be created using one of three available constructors, see table 5.2.

Additionally, a node can be obtained without specifying a window by calling an existing `JSNode`'s `getJSNode()` method. This method expects a reference to an existing

[45]The `parseParameters()` method takes the string it gets passed, removes everything not in parentheses (`()`) and splits the remaining string using commas as delimiters. The resulting parameter keys are the returned as a `String` array.
[46]The part of the script call string in front of the equals sign (`=`).
[47]`JSNode` implements all core methods of the DOM Level 1 `node` object as well as the methods of the `CharacterData` interface and `Element` interface except the `getElementsByTagName()` and `normalize()` methods as these are not available in the current JavaScript implementations.
[48]Cf. section 5.2.2.3.1.

Method Head	**Description**
`JSNode(JSObject window, String nodeRef)`	Creates a reference to the node with the `id` `nodeRef` in the `window`.
`JSNode(JSObject window, JSObject existingNode)`	Creates a reference to the node passed as `existingNode`.
`JSNode(JSObject window, JSObject existingNode, String id)`	Creates a reference to the node passed as `existingNode` and sets its `id` to the specified `id`.

Table 5.2: JSNode Constructors

node of the type `JSObject` as its argument. It returns a new JSNode with the window of the called node.

Other methods provided by `JSNode` can be used to obtain the nodes window (`getWindow()` and document (`getDocument()`), its `id`(`getIdentifier()`), the full HTML content of an element (`getInnerHTML()`) and the node object referenced by the `JSNode` (`getNode()`).

`JSNode` also provides three methods for creating new nodes of different types:

- `createAttribute(String attributeType, String attributeValue)`, creates an attribute named `attributeType`, e.g. `style` or `name`, set to the value `attributeValue`.
- `createElement(String elementType)`, creates an element of the type `elementType`, e.g. `h1` or `div`.
- `createTextNode(String elementText)`, creates a text node with the specified content (`elementText`).

5.2.2.3.2 Advantages of JSNode Over Conventional LiveConnect DOM Access

The purpose of `JSNode` is to ease the access to DOM elements at runtime in comparison to LiveConnect. As LiveConnect handles all DOM interaction over `JSObject` which provides very limited functionality, DOM interaction only using LiveConnect directly usually leads to long and error-prone code. Especially when compared to JavaScript DOM interaction, direct Java DOM interaction is unnecessary complicated.[49]

One of the most problematic and 'code-consuming' aspects of `JSObject` is the `call()` method. This method can be used to call methods of the underlying DOM node in Java, for example, it can be used to call a nodes `getAttributeNode()` method. The syntax for calling these DOM methods is `Object returnedObject = nodeObject.call(method Name, arguments)` where `methodName` is a string representing the name of the method to be invoked, e.g. `getAttributeNode` and `arguments` is an array of `Object`s that contains the arguments that are to be passed to the method. This means that for every DOM method call, an `Object` array has to be created and the individual arguments have to be stored in this array. If an argument is a primitive value, it additionally has to be converted to an object before it can be put in the array.

To overcome these drawbacks of 'pure' LiveConnect, `JSNode` capsules the methods normally used in JavaScript using the DOM node object. This capsuling is implemented using the methods provided by LiveConnect's `JSObject`. An example of how these methods are implemented is given in figures 5.7 and figure 5.8.[50]

The `getAttributeNode()` method shown in figure 5.7 expects a string that identifies the attribute to be returned, e.g. the `name` or `src` attribute, and returns this attribute as a `JSNode` object. To do this, `getAttributeNode()` first creates a new `Object` array with one field and assigns this field the string that was passed as `attributeName`. It then calls the `call()` method of the `JSNode`'s `node` property to obtain a `JSObject` representation of the attribute node. This `JSObject` representation is assigned to the `attributeNode` variable which is then used to create a new `JSNode` containing the `attributeNode`. This `JSNode` is then return to the calling function.

`setStyleAttribute()` of figure 5.8 is a shorter method as it does not involve calling

[49]Cf. figure 4.6, p. 35.
[50]Debug messages were omitted in these examples to improve readability.

```
public JSNode getAttributeNode(String attributeName) {
  Object[] callArgs=new Object[1];
  callArgs[0]=attributeName;
  JSObject attributeNode=(JSObject)node.call("getAttributeNode",callArgs);
  JSNode attributeJSNode=this.getJSNode(attributeNode);
  return attributeJSNode;
}
```

Figure 5.7: Implementation getAttributeNode()

the `JSObject`'s `call()` method and thus creating and filling an `Object` array is not necessary here. The method expects two strings, `styleAttribute` specifying the attribute that is to be set, e.g. `fontFamily` or `color`, and `value` specifying the value this attribute shall be set to.

```
public int setStyleAttribute(String styleAttribute, String value) {
  JSObject styleNode=(JSObject)node.getMember("style");
  styleNode.setMember(styleAttribute,value);
  return 0;
}
```

Figure 5.8: Implementation setStyleAttribute

6 Discussion

The 1.0 release of BWS still has got some issues resulting often from shortcomings of integrated technologies as well as misinterpretation and ignoration of web standards in some situations.

DOM Standard Conformance One of the most critical of these problems is the Internet Explorer's failure to correctly interpret HTML code that is not read from a file or an URL but is passed to it directly from a JavaScript or via LiveConnect using the `document.write()` method. In these cases, the Internet Explorer does not load applets referenced from the passed HTML code. As this method of directly passing HTML code to the browser is used in the `RewriterApplet`, it is at the moment not possible to load and run BWS applications end-user friendly using the Internet Explorer.

Another one of these problems is the direct embedding of scripts in BWS/HTML documents. While reading this embedded script code is possible with Mozilla-based browsers using standard DOM methods and the Internet Explorer at least supports this using the `innerHTML` property[1] of DOM nodes on the `script` node, Opera and Konqueror support neither of these methods when they are invoked from LiveConnect. Consequently, applications that are required to run on all of these browsers must not use embedded scripts but only external, referenced ones; as retrieval of these scripts is done

[1]`innerHTML` was originally created by Microsoft; nowadays (March 2004) is can be considered a de facto standard which is available in most browsers' JavaScript implementation. It is available in all considered browsers but apparently not accessible via LiveConnect on Opera and Konqueror.

browser independent using Java methods, referencing scripts works browser independent.

Other issues resulting from misinterpretation or ignorance of DOM standard are the event model,[2] especially of the Internet Explorer[3] or the implementation of the `attributes` property.[4] An excellent overview on which browser interprets which browser standard to which extent is available in [Koc04].

The BSF's `apply()` Method A problem not related to browsers but to the BSF engine implementations is the insufficient support of the `BSFEngine`'s `apply()` method that is used to pass scripts to their engines. Some standard `BSFEngine`s, for example the Rhino engine, do not support the method to its full extent but only evaluate the script passed to them using the `BSFEngine`'s `eval()` method that does not support passing parameters to scripts. As a consequence to this, passing parameters to scripts is only available for scripting languages that support the `apply()` method, for example the BSF4Rexx engines.

Client Side Rewriting Another discussable point of BWS is the incorporation of document rewriting directly into the BWS runtime environment in order to enable the loading of only the BWS document from a server without the need to transform it on the server or through an intermediary applet. This however would make the creation of a BWS document more complicated as the BWS applet with all its parameters would have to be embedded by the developer of a BWS application. The provided `RewriterApplet` solution that is based on a HTML document and an intermediary applet in contrast

[2]The DOM event model defines how DOM events should be handled, it allows for example the dynamic registration of event handlers (e.g. `onClick`).

[3]The Internet Explorer even in its current release (Internet Explorer 6 Service Pack 1) does not support the DOM event interface at all, but still uses Microsoft's proprietary event interface.

[4]`attributes` provides a numbered array of all attributes of a DOM node. On the Internet Explorer, these array contains values for all theoretically available attributes of the node while on other browsers, it only contains the values of attributes actually set; consequently, e.g. the fifth element of `attributes` will contain different values on Internet Explorer and other browsers and makes `attributes` unusable in cross-browser environments.

to this is a generic alternative that can be used with any BWS document without the need to embed the applet directly in the HTML document. The drawbacks of the `RewriterDocument` are that at the moment it does not work with the Internet Explorer and it requires the specification of the full URL of a document.[5]

Concerning document rewriting, there is also the problem that large parts of dom4j are necessary on the computer doing the rewriting. If client side rewriting is used, be it using the `RewriterApplet`, the command line utilities or any other theoretical option, it always is necessary to have dom4j installed and available for Java on the client or in the Java archive containing the BWS runtime environment. This, however, blows up the size of this Java archive and make BWS less comfortable to use, especially when only a low-bandwidth connection to the server is available.

Consequently, it is recommended to do document rewriting preferably on the server side or computers that have dom4j installed permanently; however, a solution to the problem of user friendly document rewriting is considered an absolute necessity and will be the next improvement of BWS.

[5]Cf. section 7.3.4, pp. 68.

7 Using BWS

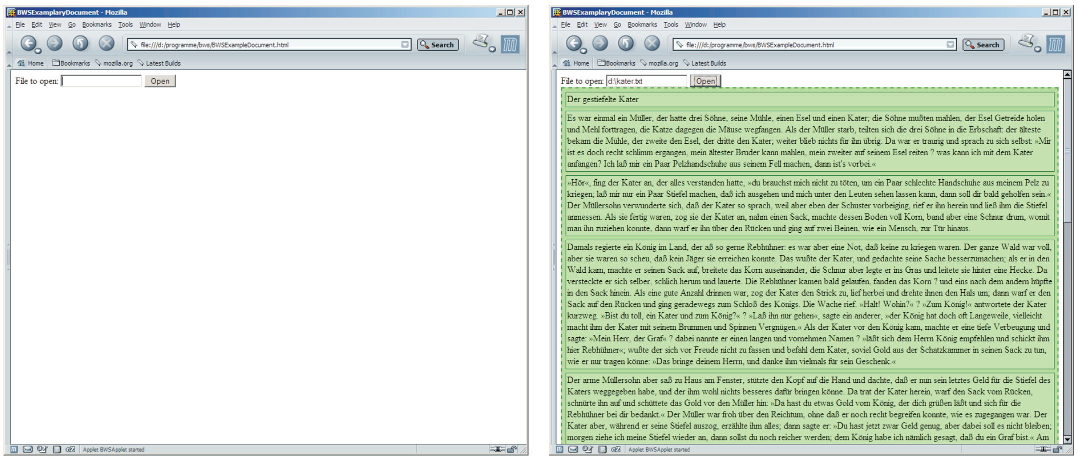
The creation of BWS applications works mostly like the creation of conventional client side web applications realized in JavaScript. This section will provide information on how BWS applications are created, what requirements must be met and what limitations do apply. To improve the comprehensibility of this section, all of the explanations will be based on an example document.

It is assumed that the reader will have BSF as well as the wished scripting languages installed. Additionally, a basic knowledge of HTML is assumed.

7.1 An Example of BWS in Action

The example document is a BWS document that allows the user to specify the name and path of a local text[1] file. This file is then read by a rexx script, the paragraphs of it are created as HTML tags[2] and the resulting HTML code is shown in a designated area of the document. Figure 7.1 shows two screen shots: One depicts the screen when the document has been loaded, the other after the script has been run. The BWS document and rexx script used are shown in figures 7.2 and 7.3.

[1]The file type of the file is not checked, so in theory using a binary file is also possible. Using a binary file will probably result in various results, including crashes of the browser and is not recommended.
[2]I.e. they are embedded withing `<div>` tags.


BWSExamplaryDocument - Mozilla
File Edit View Go Bookmarks Tools Window Help
file:///d:/programme/bws/BWSExampleDocument.html
Search
Home Bookmarks mozilla.org Latest Builds
File to open:
Open
Applet BWSApplet started

File to open: d:\kater.txt

Der gestiefelte Kater

Es war einmal ein Müller, der hatte drei Söhne, seine Mühle, einen Esel und einen Kater; die Söhne mußten mahlen, der Esel Getreide holen und Mehl forttragen, die Katze dagegen die Mäuse wegfangen. Als der Müller starb, teilten sich die drei Söhne in die Erbschaft: der älteste bekam die Mühle, der zweite den Esel, der dritte den Kater; weiter blieb nichts für ihn übrig. Da war er traurig und sprach zu sich selbst: »Mir ist es doch recht schlimm ergangen, mein ältester Bruder kann mahlen, mein zweiter auf seinem Esel reiten ? was kann ich mit dem Kater anfangen? Ich laß mir ein Paar Pelzhandschuhe aus seinem Fell machen, dann ist's vorbei.«

»Hör«, fing der Kater an, der alles verstanden hatte, »du brauchst mich nicht zu töten, um ein Paar schlechte Handschuhe aus meinem Pelz zu kriegen; laß mir nur ein Paar Stiefel machen, daß ich ausgehen und mich unter den Leuten sehen lassen kann, dann soll dir bald geholfen sein.« Der Müllersohn verwunderte sich, daß der Kater so sprach, weil aber eben der Schuster vorbeiging, rief er ihn herein und ließ ihm die Stiefel anmessen. Als sie fertig waren, zog sie der Kater an, nahm einen Sack, machte dessen Boden voll Korn, band aber eine Schnur drum, womit man ihn zuziehen konnte, dann warf er ihn über den Rücken und ging auf zwei Beinen, wie ein Mensch, zur Tür hinaus.

Damals regierte ein König im Land, der aß so gerne Rebhühner: es war aber eine Not, daß keine zu kriegen waren. Der ganze Wald war voll, aber sie waren so scheu, daß kein Jäger sie erreichen konnte. Das wußte der Kater, und gedachte seine Sache besserzumachen; als er in den Wald kam, machte er seinen Sack auf, breitete das Korn auseinander, die Schnur aber legte er ins Gras und leitete sie hinter eine Hecke. Da versteckte er sich selber, schlich herum und lauerte. Die Rebhühner kamen bald gelaufen, fanden das Korn ? und eins nach dem andern hüpfte in den Sack hinein. Als eine gute Anzahl drinnen war, zog der Kater den Strick zu, lief herbei und drehte ihnen den Hals um; dann warf er den Sack auf den Rücken und ging geradewegs zum Schloß des Königs. Die Wache rief. »Halt! Wohin?« ? »Zum König!« antwortete der Kater kurzweg. »Bist du toll, ein Kater und zum König?« ? »Laß ihn nur gehen«, sagte ein anderer, »der König hat doch oft Langeweile, vielleicht macht ihm der Kater mit seinem Brummen und Spinnen Vergnügen.« Als der Kater vor den König kam, machte er eine tiefe Verbeugung und sagte: »Mein Herr, der Graf« ? dabei nannte er einen langen und vornehmen Namen ? »läßt sich dem Herrn König empfehlen und schickt ihm hier Rebhühner«; wußte der sich vor Freude nicht zu fassen und befahl dem Kater, soviel Gold aus der Schatzkammer in seinen Sack zu tun, wie er nur tragen könne: »Das bringe deinem Herrn, und danke ihm vielmals für sein Geschenk.«

Der arme Müllersohn aber saß zu Haus am Fenster, stützte den Kopf auf die Hand und dachte, daß er nun sein letztes Geld für die Stiefel des Katers weggegeben habe, und der ihm wohl nichts besseres dafür bringen könne. Da trat der Kater herein, warf den Sack vom Rücken, schnürte ihn auf und schüttete das Gold vor den Müller hin: »Da hast du etwas Gold vom König, der dich grüßen läßt und sich für die Rebhühner bei dir bedankt.« Der Müller war froh über den Reichtum, ohne daß er noch recht begreifen konnte, wie es zugegangen war. Der Kater aber, während er seine Stiefel auszog, erzählte ihm alles; dann sagte er: »Du hast jetzt zwar Geld genug, aber dabei soll es nicht bleiben; morgen ziehe ich meine Stiefel wieder an, dann sollst du noch reicher werden; dem König habe ich nämlich gesagt, daß du ein Graf bist.« Am

Figure 7.1: Screen shots of the Example Document

7.2 Installing BWS

Just for running a simple BWS document that uses only Java based script engines[3] and does not need to leave the Java sandbox, an installation is not necessary. Depending on the way BWS is distributed, it may or may not contain the necessary scripting engines within the BWS' Java archive. If the scripting engines are not contained in the archive, their Java classes must be made available by incorporating them in the Java `CLASSPATH` environment variable.

For non-Java based scripting engines, like the BSF4Rexx engine, it is also necessary to make the libraries containing the engine available by providing them in the library path of the Java Runtime Environment. On Microsoft Windows systems, this can be done by setting the `PATH` environment variable to include the path of the scripting engine's library; on Unix systems, either the libraries must be copied or linked to the `jre/lib/i386/` and `jre/lib/i386/client/` subdirectories of the Java installation or the `java.library.path` of the Java Runtime Environment must be adapted to include the engine library's paths.

If the BWS applications needs to leave the Java sandbox or if a native scripting engine

[3]E.g. Rhino or NetRexx.

```
parse arg filename, outputArea

filename=.bsf~lookupbean(filename)
outputArea=.bsf~lookupbean(outputArea)

file=.stream~new(filename~getAttribute('value'))
do while file~lines<>0
  text=file~linein
  anElementNode=outputArea~createElement('div')
  anElementNode~addEventListener('onClick','alertMe')
  contentText=outputArea~createTextNode(text)
  anElementNode~appendChild(contentText)
  anElementNode~setStyleAttribute('padding','2px')
  anElementNode~setStyleAttribute('backgroundColor','#aaffaa')
  anElementNode~setStyleAttribute('margin','5px')
  anElementNode~setStyleAttribute('border','#00cc00 2px ridge')
  outputArea~appendChild(anElementNode)
end

outputArea~setStyleAttribute('backgroundColor','#99ff99')
outputArea~setStyleAttribute('border','#009900 2px dashed')

::REQUIRES BSF.CLS
```

Figure 7.2: Example Document: Rexx script

is used, it is necessary to create appropriate Java security policies: Java for this purpose provides the `policytool` available in the `bin` subdirectory of the Java installation. Using this tool, the necessary permissions can be set; the resulting policy file must be saved to the user's home directory[4] under the name `.java.policy`.

7.3 Creating BWS Applications

After BWS has been installed successfully, the creation of a BWS applications can begin with the creation of a BWS document.

[4]On Unix-based system this usually is `/home/<username>/`, on Windows XP it is `c:\documents and settings\<username>\` or something similar, depending on with localization of Windows is used.

```
<html>
 <head>
  <title>
   BWSExamplaryDocument
  </title>
  <script id="readFile" type="bws/rexx" src="readFileScript.rex" />
 </head>

 <body>
  File to open: <input type="text" id="filename" />
  <input type="button" onclick="bws:readFile(filename)" value="Open!" />
  <div id="output">
  </div>
 </body>
</html>
```

Figure 7.3: Example Document: BWS Document

7.3.1 HTML Documents as Starting Point for BWS Applications

The starting point of a BWS document is a standard well-formed HTML document fulfilling the following requirements:[5]

- For every start tag, there must be a corresponding closing tag, i.e. for every `<li>` tag there must be a `</li>` tag; an exception to this rule is allowed for tags that have no content, e.g. `<br>`, these tags may alternatively be specified in the combined notation `<br/>`. As some browsers do not accept the later variant, it is recommended to use explicit start and end tags or, in cases where this is not possible, separate the closing slash by a space from the tag name, e.g. `<br />`. This notation works with any of the browsers BWS was tested on.
- Elements can have sub elements, they must use strict nesting, 'overlapping' tags, for example `<h1>Text<center>Centered</h1>also centered</center>`, are not

[5]There are some additional requirements that do not apply to BWS documents, see [CMP01], the exact requirements of well-formedness for XML documents is [BPSMM00, Section 2.1]. These requirements are not 'native' requirements of BWS but are only necessary for enabling the parsing of the document using a conventional XML parser. Cf. section 5.2.1.2, pp. 44.

allowed.

- Tags and attributes are case-sensitive, for example `<h1>` and `</H1>` are not matching.
- Attributes must have exactly one value, empty attributes that are allowed in standard HTML, e.g. the `mayscript` attribute of objects must be given as `<object mayscript="true" />`, not only `<object mayscript>`, all attributes must be enclosed in quotes.
- The document must have exactly one root element, i.e. a construction `<html><!-- htmlcode --></html><somethingElse></somethingElse>` is not allowed.

Any document conforming to these requirements can be used as a BWS document and extended with BWS scripts; generally, an XHTML compliant document will work.[6] Figure 7.4, p. 73 shows the basic example document first in an incorrect and then in a corrected version.[7]

7.3.2 Scripts and Script Calls

Once a BWS compliant document as described above has been created, scripts can be embedded. This step consists of two parts:

1. Embedding or referencing the scripts in the document.
2. Attaching script calls to the events that shall be handled.

Both steps are done almost exactly like they would be done using JavaScript scripts.

[6]The resulting BWS/HTML, however, will never be a XHTML conforming document, as the `applet` tag is not allowed in XHTML.
[7]Changed lines are marked with an asterisk (`*`).

7.3.2.1 Embedding and Referencing Scripts

For every script you want to use, create a `<script>` tag.[8] This script tag must have at least two attributes:

1. The `type` attribute that specifies that this script is a script that shall be interpreted using BWS and the programming language the script is written in; this attribute is given as `bws/scriptengine`, for example it is `bws/netrexx` for the NetRexx interpreter or `bws/javascript` for the Rhino JavaScript interpreter.

2. The `id` attribute under which this script is referenced in script calls. This `id`, as well as any other `id` used in the document, must be unique, it may consist of only alphanumeric characters (A-Z, a-z, 0-9), dashes (-), underscores (_), dots (.) and colons (:).

If only these two attributes are specified, the script code must be embedded within the script element (see figure 7.5, p. 74); if the script code shall not be contained within the document, it is also possible to reference scripts contained in extra files. To do this, the optional source attribute `src` can be specified. This attribute has to be in the form of an absolute[9] or a relative[10] path or an URL; using this method, the script file is loaded only on execution. If the `src` attribute is specified, the code under the script element is not inspected.

Other tags may be specified too, but are not evaluated by BWS.[11] The example document with the reference to the script is shown in figure 7.5, p. 74.[12]

[8]It is not possible to embed two scripts within one script tag.
[9]Relative to the root path of the current document's server, e.g. `/scripts/aScript.rex` when the script and the document reside on the same server.
[10]Relative to the path of the current document, e.g. if the document is available from `http://www.mydomain.com/docs/aDocument.html` and the script file is available from `http://www.mydomain.com/scripts/aScript.bws`, it can also be referenced as `../scripts/aScript.bws`.
[11]They may be evaluated by the browser.
[12]Changed line is marked with an asterisk (`*`).

7.3.2.2 Attaching Script Calls to Events

After all necessary scripts are embedded or referenced in the document, script calls can be attached to events, i.e. scripts can be triggered by DOM events, for example by clicking a part of text. Attaching a script to a specific event is done by setting the correspondent script handler, e.g. `onClick`, to `bws:` followed by the script id.[13] To attach a script to the button that calls the `readFile` script when the button is clicked, the button's `onClick` attribute is set to `bws:readFile` in figure 7.6.[14] For an overview which events are available see [Pix00].

7.3.3 Accessing And Modifying DOM Elements

Scripts will usually have to access the document, for example for outputting the results of an computation, for setting certain styles of a document or for reading input values of a user. To access parts of the document, the first step is to obtain access to the BWS runtime environment, specifically the `BWSApplet`. This applet is always available in the BSF registry[15] and can be retrieved by calling the `BSF` object's `lookupBean()` method and specifying the applet's key: `BWSApplet`, as the parameter to this function call. An example call for obtaining the applet in rexx is line 1 of the example script in figure 7.7, p. 75.[16]

Once the applet is available in the script, the applet's `getJSNode()` method can be used to obtain document nodes represented as objects of the `JSNode` class. To obtain a specific node, the node's `id` attribute must be passed as parameter of the method call. In the example, retrieving a node using this method is done in lines 3 and 4.

[13]Instead of `bws:`, a hash sign (`#`) followed by a colon and the script id is also allowed, e.g. `#:scriptId`.
[14]Changed line is marked with an asterisk (`*`).
[15]The BSF registry is a central object storage were objects can be stored and retrieved using a unique key. The BSF registry is provided by the BSF and available in all BWS scripts. Cf. section 4.3, pp. 35.
[16]This script is a shortened version of the example rexx script in figure 7.3, p. 64 that only reads the file and does no formating.

The retrieved nodes can then be modified using the methods provided by the `JSNode` class[17], these methods are exactly the same as the methods provided by the DOM node object in DOM supporting web browsers. For example in lines 11 and 12, the `appendChild()` method is used to first add text to a existing node and then append this node to the document. `JSNode` also provides methods for creating new nodes, for example the creation of a new `div` element in line 9 and the creation of a new text node in line 10 of figure 7.7, p. 75.

7.3.4 Running BWS Applications

Once a BWS document has been completed, it can be run directly using the `Rewriter Document` HTML document. [18] This document allows the user to specify the location of a BWS document in the form of an URL: On Windows systems, this URL is created by using `file:///`, the drive the document is located on, e.g. `c:` and the full path of the document. In contrast to the usage of backslashes for separating directories in windows, for URLs directories have to be separated with forward slashes.[19] On Unix systems, the URL is `file://` followed by the full path of the document, e.g. `file:///home /myhome/bws/testApplication.bws`. A click on the `Open` button loads the specified document to a new browser window where it is run.

Figure 7.8, p. 76 shows a screenshot of the rewriter document.

[17]Cf. API Documentation, appendix A.

[18]It is necessary to have dom4j installed and available in the Java classpath to use the `BWSDocument` class, which is necessary for document rewriting.

[19]If the BWS document is `c:\documents and settings\admin\my documents\test Application.bws`, the correspondent URL is `file:///c:/documents and settings/admin/my documents/testApplication.bws`.

7.4 Advanced Possibilities of BWS

In addition to the basics of BWS mentioned earlier that will be sufficient for many applications, BWS also provides two more advanced possibilites:

- Passing parameters to and returning values from scripts.
- Calling BWS scripts, embedded or referenced, within other scripts.

7.4.1 Passing Parameters and Returning Values from BWS Scripts

BWS scripts that run on an engine supporting the `apply()` method of BSF, for example BSF4Rexx engines, can pass parameters to scripts and access their return values in other scripts. Instead of using only the `script id` in the script call string, script call strings that shall pass parameters or receive return values must specify the script call using a C/Java style syntax:

Only Parameters For calling a script with parameters but without a return value, parameters should be appended to the script id within parentheses and separated by commas. The final version of the example document shown in figure 7.10 shows such a invocation in the `onclick` attribute in line 12.

Only Return Value When only a return value is necessary, the key for the return value may be specified by putting it in front of the script id followed by an equals sign (`=`).

Parameters and Return Value Specifying both, parameters and return value works with the combination of both possibilites, i.e. the return value key followed by an equals sign (`=`), the script id and the comma-separated values in parameters: `returnValue=script Id(parameters)`.

Parameters specified with script calls are evaluated always on script invocation, they are evaluated to the following values:

1. If the parameter is `this`, the DOM object that triggered the event is passed, for example in figure 7.10, p. 77 a `JSObject` referencing the `input` element would be used.[20]

2. Parameters specified in quotations marks are stripped of their quotation marks and passed as `String`s.

3. If the parameter is the key to a object in the BSF registry, the corresponding object is passed.

4. Parameters that are neither in quotation marks nor match a BSF registry key are tested if they match the id of a DOM object. If this is the case, the correspondent DOM object is passed as a `JSNode`.

5. If none of these criteria matches, the parameter is passed as a `String`.

The return value of the script is then stored in the BSF registry under the specified key. Independent of the method invocation, the return value is always returned directly to the method calling the script.

7.4.2 Calling Scripts from Other Scripts

Calling scripts from other scripts is significantly different from calling scripts from HTML event handlers: Script invocation within scripts is done using one of `BWSApplet`'s `executeScript()` methods,[21] either with only a script string as parameter or with an

[20]This applies only to script calls from HTML element event handlers. If scripts are called from other scripts, the object that will be referenced as `this` can be specified by the developer of the calling script. Cf. section 7.4.2.

[21]Cf. API documetation appendix A.

additional arbitrary object as second parameter. If such an object is specified with the method call, this object will be passed to the invoked script and can be referenced using the `this` keyword in the parameters part of the script string. If no object is passed, `this` will be a reference to the `script` node that holds the invoked script.

The script string that is passed to the applet as parameter of `executeScript()` method must at least contain the script id of the script to be invoked, it may additionally specify parameters or a return value as any script invocation that is done with HTML event handlers; evaluation of parameters works the same as it is the case with HTML event handler calls. In contrast to the value of an event handler, script strings passed directly to the BWS applet must not start with `bws:` or `#:`.

An example of such a script invocation within a rexx script is shown in figure 7.11, p. 77. This example starts by retrieving a reference to the `BWSApplet` in line 1. Additionally, another object stored under the key `anObjectKey` is retrieved from the BSF registry and referenced as `anObject` in line 2. Line 4 executes the script `otherScript` that must be existent under this `id` in the currently loaded BWS/HTML document. This script is invoked with the `anObject` as additional parameter; the `anObject` can be referenced from the script string with the `this` keyword, which is done. Consequently, the BWS runtime environment will pass the `anObject` object to the `otherScript` as the only parameter of the script call.[22]

[22]The BWS parameter interpretation also makes it possible to omit retrieving the object from the registry explicitly to use it in calling the script: As strings that are the keys of registry objects are automatically resolved to the registry objects, the more sensible way in this case would be to omit line 2 of the example and replace line 4 with `otherScriptReturnValue=theApplet~executeScript('otherScript(anObjectKey)')`.

7.5 Code Comparison: BWS and JavaScript

Figure 7.12[23] shows four different ways of modifying DOM elements using different approaches: JavaScript provided by the browser, Java using LiveConnect and BWS with the Rhino and the BSF4Rexx[24] scripting engine. As can be seen in this figure, BWS code is much easier to read and to write compared to Java, it also is only slightly different and not more complicated than comparable code written using the browser supplied JavaScript engine.

[23]This is a slightly modified version of the comparison shown in figure 4.6, p. 35.
[24]With Object Rexx as interpreter.

```
<html>
 <head>
  <Title>
   BWSExamplaryDocument
  </title>
 </head>

 <body>
  File to open: <input type="text" id="filename">
  <input type="button" value="Open!">
  <div id="output">
  </div>
 </body>
</html>
```

The incorrect document.

```
  <html>
   <head>
*   <title>
     BWSExamplaryDocument
    </title>
   </head>

   <body>
*   File to open: <input type="text" id="filename" />
*   <input type="button" value="Open!" />
    <div id="output">
    </div>
   </body>
  </html>
```

A corrected version.

Figure 7.4: Correct And Incorrect XHTML Document

Example document with reference to the script

```
<html>
 <head>
  <title>
   BWSExamplaryDocument
  </title>
* <script id="readFile" type="bws/rexx" src="readFileScript.rex" />
 </head>

 <body>
  File to open: <input type="text" id="filename" />
  <input type="button" value="Open!" />
  <div id="output">
  </div>
 </body>
</html>
```

Example of an embedded script

```
<html>
...
<script type="bws/netrexx" id="aRexxScript">
-- some netrexx code
</script>
...
</html>
```

Figure 7.5: Embedding and Referencing Scripts

```
<html>
 <head>
  <title>
   BWSExamplaryDocument
  </title>
  <script id="readFile" type="bws/rexx" src="readFileScript.rex" />
 </head>

 <body>
  File to open: <input type="text" id="filename" />
  <input type="button" value="Open!" onclick="bws:readFile" />
  <div id="output">
  </div>
 </body>
</html>
```

Figure 7.6: Attaching a Script Call

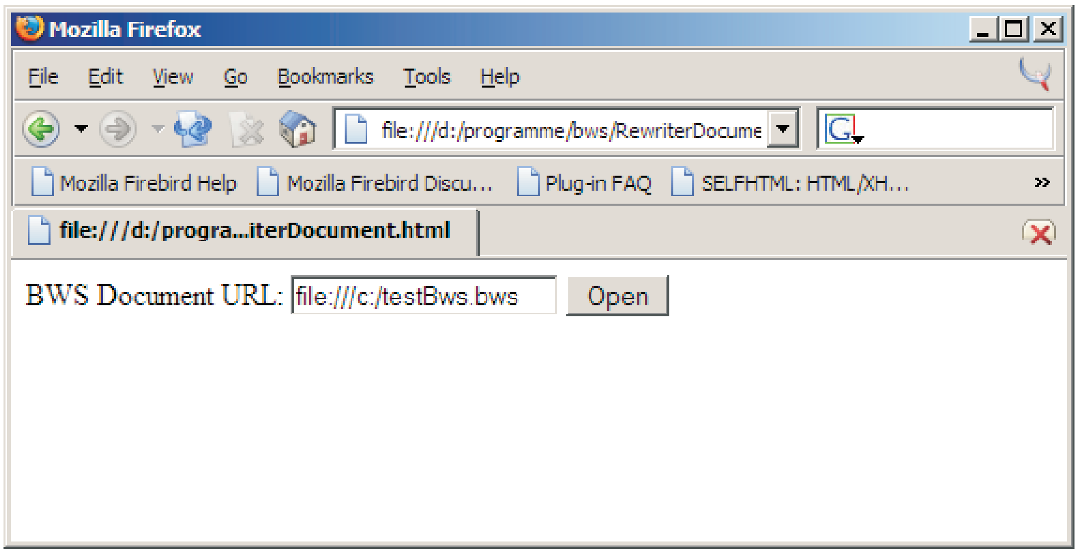
```
theBWSApplet=.bsf~lookupbean('BWSApplet')

filename=theBWSApplet~getJSNode('filename')
outputArea=theBWSApplet~getJSNode('outputArea')

file=.stream~new(filename~getAttribute('value'))
do while file~lines<>0
  text=file~linein
  anElementNode=outputArea~createElement('div')
  contentText=outputArea~createTextNode(text)
  anElementNode~appendChild(contentText)
  outputArea~appendChild(anElementNode)
end

::REQUIRES BSF.CLS
```

Figure 7.7: Example: Simple Rexx Script


Mozilla Firefox
File Edit View Go Bookmarks Tools Help
file:///d:/programme/bws/RewriterDocume
Mozilla Firebird Help
Mozilla Firebird Discu...
Plug-in FAQ
SELFHTML: HTML/XH...
file:///d:/progra...iterDocument.html
Open

BWS Document URL: file:///c:/testBws.bws

Figure 7.8: Rewriter Document Screenshot

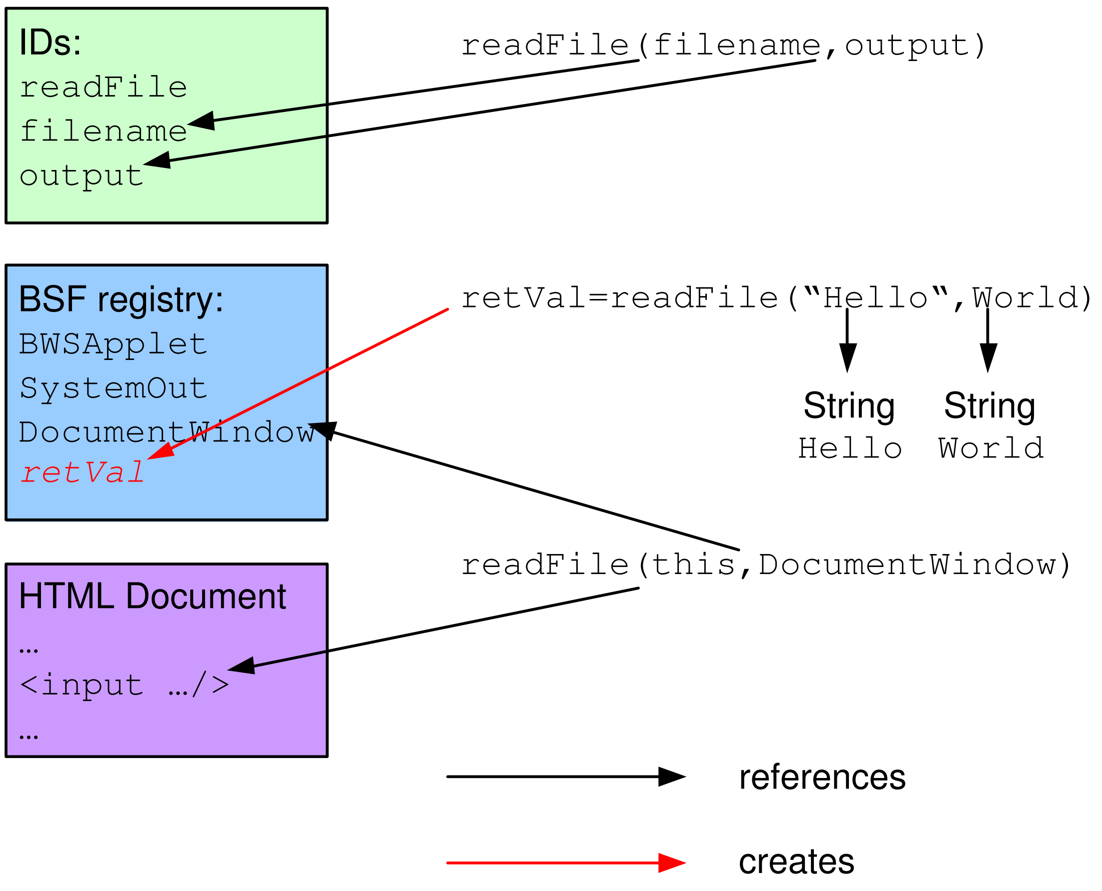


Figure 7.9: Parameter Interpretation

```
 <html>
  <head>
   <title>
    BWSExamplaryDocument
   </title>
   <script id="readFile" type="bws/rexx" src="readFileScript.rex" />
  </head>

  <body>
   File to open: <input type="text" id="filename" />
*  <input type="button" value="Open!"
*    onclick="bws:readFile(filename,output)" />
   <div id="output">
   </div>
  </body>
 </html>
```

Figure 7.10: Final Version of the Example Document

```
theApplet=.bsf~lookupBean('BWSApplet')
anObject=.bsf~lookupBean('anObjectKey')

otherScriptReturnValue=theApplet~executeScript('otherScript(this)',anObject)

::REQUIRES BSF.cls
```

Figure 7.11: Calling a Script Within Another Script in Rexx

Setting the heading's background color to red, the heading's id must be `aHeading`.
JavaScript

```
firstHeading=window.getElementById("aHeading");
firstHeading.style.backgroundColor="red;"
```

Java

```
Object[] objectArray=new Object[1];
JSObject appletWindow=this.getWindow();

objectArray[0]="aHeading";
JSObject heading=
   (JSObject)appletWindow.call(getElementsyTagName,objectArray);

JSObject styleAttribute=(JSObject)heading.getAttribute("style");
objectArray[0]="red";
styleAttribute.setMember("backgroundColor",objectArray)
```

BWS with Rhino

```
theBWSApplet=BSF.lookupBean('BWSApplet');

firstHeading=theBWSApplet.getJSNode('aHeading');
firstHeading.setStyleAttribute('backgroundColor','red');
```

BWS with ObjectRexx

```
theBWSApplet=.bsf~lookupBean('BWSApplet')

firstHeading=theBWSApplet~getJSNode('aHeading')
firstHeading~setStyleAttribute('backgroundColor','red')

::REQUIRES BSF.cls
```

Figure 7.12: DOM Scripting Comparison: JavaScript, Java And BWS

8 Wrap-Up and Outlook

Compared to JavaScript, BWS overcomes JavaScript's limitations and enables any task within the browser that can be done with a supported scripting language and that doesn't violate the safety measures of the Java sandbox. BWS, using BSF, also provides access to the full Java class library within the user's preferred scripting language. And in comparison to ActiveScripting, BWS allows an as easy integration of scripts and documents, but in a browser and operating system independent and more secure way.

BWS scripts can be edited ad hoc and immediately tested in a web browser, without the need to compile them or do anything besides editing; if they are not embedded directly in the BWS/HTML document but referenced, it is not even necessary to reload the document in the browser.

Concerning the possible fields of usage of BWS, one of the most interesting areas is the use of BWS for cryptographic applications that shall run inside web browsers. This is an area that is problematic with server side applications as in this case, the keys for encryption or decryption have to be available on the server, which is a potential security hazard as keys may be compromised by the server administrator easily. With the use of client side scripting, there is no need to store keys or decrypted files on the server, all cryptographic operations can take place on the client.

The future of BWS to a large part depends on the future development of Java and web browsers. One of the most important task is the use of the Common DOM API instead of LiveConnect. The Common DOM API will provide a direct, DOM conform, way of accessing the DOM of the document currently loaded in the web browser. It was originally planned to be available with Java 1.4. However, Java 1.4 only provided a extremely limited and practically unusable version of the Common DOM API. Sun, despite announcing to do so, did not change that in later releases of Java 1.4 and now plans to have a working version with Java 1.5 which will probably be available in the third quarter of 2004. Another planned improvement is to provide a unified interface to the DOM event model if this is still necessary with the availability of the Common DOM API and if it is realizable using Java.

For ease of use, it is also planned to provide a server-side rewriting mechanism in form of PHP and JSP scripts that would do document rewriting on the server without any interaction from the user and without the need to embed the applet in the document by hand.

However, the most important change that will be made to BWS in the next time will be the creation of an end-user friendly way of transforming BWS documents to BWS/HTML documents. This aspired solution shall work on the client, be completely independent of the browser and further diminish the difference between browser native and BWS based client side scripting.

Bibliography

[Alp01] IBM AlphaWorks, editor. *About alphaWorks.* IBM Corp., http://alphaworks.ibm.com/about, 2001.

[Alp04] *IBM alphaWorks : emerging technologies.* IBM Corp., http://alphaworks.ibm.com/ (visited: 2004-03-23), 2004.

[BPSMM00] Tim Bray, Jean Paoli, C. M. Sperber-McQueen, and Eve Maler, editors. *Extensible Markup Language (XML) 1.0 (Second Edition).* W3C, http://www.w3.org/TR/2000/REC-xml-20001006#sec-well-formed (visited: 2004-03-13), 2000.

[BSF03] BSF4Rexx homepage. http://wi.wu-wien.ac.at/rgf/rexx/bsf4rexx/ (visited: 2004-03-94), 2003.

[CD99] James Clark and Steve DeRose, editors. *XML Path Language (XPath) Version 1.0.* W3C XSL Working Group and W3C XML Linking Working Group, http.//www.w3.org/TR/xpath (visited: 2004-03-03), 1999.

[CMP01] *The Well-Formed XML Document.* CMP Media LLC, http://www.intelligenteai.com/XML Repository/well_formed_xml_document.htm (visited: 2004-03-13), 2001.

[Cur02] *The Curl Client/Web Platform.* Curl Corporation, http://www.curl.com/pdf/The_Curl_ClientWeb_Platform.pdf (visited: 2004-03-08), 2002.

[Cur04] *Curl Corporation Website.* Curl Corporation, http://www.curl.com/ (visited: 2004-03-08), 2004.

[Dev04a] *Developer Works : IBM's resource for developers.* IBM Corp., http.//www.ibm.com/developerworks (visited: 2004-03-23), 2004.

[Dev04b] IBM DeveloperWorks, editor. *About developerWorks.* IBM, http://www.ibm.com/developerworks/aboutdw/, 2004.

[dom03] *dom4j 1.4 API.* MetaStuff Ltd., http://www.dom4j.org/apidocs/index.html (visited: 2004-03-23), 2003.

[ECM99] ECMAScript language specification. Technical Report Standard ECMA-262, European Computer Manufacturers Association, http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf (visited: 2004-03-08), 1999.

[Fla01] Rony G. Flatscher. Java bean scripting with rexx. In *Proceedings of the "12th International Rexx Symposium"*, Raleigh N.C., http://wi.wu-wien.ac.at/rgf/rexx/orx12/JavaBeanScriptingWithRexx*xxx*orx12.pdf (visited: 2004-03-04), 2001. The Rexx Language Association.

[Fla03] Rony G. Flatscher. The augsburg version of BSF4Rexx. In *Proceedings of the "2003 International Rexx Symposium"*, Raleigh N.C., http://wi.wu-wien.ac.at/rgf/rexx/orx14/orx14_bsf4rexx-av.pdf (visited: 2004-03-04), 2003. The Rexx Language Association.

[Fol98] *Three-Tier.* Imperial College London, http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?three-tier (visited: 2004-03-03), 1998.

[Hég04] Philippe Le Hégaret. *Document Object Model (DOM) Technical Reports.* W3C DOM Working Group, http://www.w3.org/DOM/DOMTR (visited: 2004-03-03), 2004.

[HW04] Philippe Le Hégaret and Ray Whitmer. *W3C Document Object Model.* W3C DOM Working Group, http://www.w3.org/DOM/ (visited: 2004-03-03), 2004.

[Jav01] *API for Privileged Blocks.* Sun Microsystems, Inc., http://java.sun.com/j2se/1.4.2/docs/guide/security/doprivileged.html (visited: 2004-03-05), 2001.

[Jav03a] *Security.* Sun Microsystems, Inc., http://java.sun.com/j2se/1.4.2/docs/guide/security/ (visited: 2004-03-05), 2003. Security enhancements for the Java 2 SDK, Standard Edition, v 1.4.2.

[Jav03b] *Java 2 Platform, Standard Edition, V 1.4.2 API Specification.* Sun Microsystems, http://java.sun.com/j2se/1.4.2/docs/api/index.html (visited: 2004-03-14), 2003.

[Jav03c] *Java Plug-in 1.4.2 Developer Guide.* Sun Microsystems, Inc., http://java.sun.com/j2se/1.4.2/docs/guide/plugin/developer_guide/contents.html (visited: 2004-03-23), 2003.

[Jav04a] *Security and the Java Platform.* Sun Microsystems, Inc., http://java.sun.com/security (visited: 2004-03-05), 2004.

[Jav04b] *Java Technology Website.* Sun Microsystems, Inc., http://java.sun.com/ (visited: 2004-03-08), 2004.

[Kal00] Peter Kalender. *A Concept for and an Implementation of the Bean Scripting Framework for Rexx.* University of Essen, http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws0001/PKalender/Seminararbeit.pdf (visited: 2004-03-04), 2000.

[Koc04] Peter-Paul Koch. *QuirksMode.* http://www.quirksmode.org/ (visited: 2004-03-14), 2004.

[MSJ03] *MSJVM Transition FAQ.* Microsoft Corp., http://www.microsoft.com/mscorp/java/faq.asp (visited: 2004-03-14), 2003.

[MSW04] *Microsoft Windows Script Interfaces - Introduction.* Microsoft Corp., http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/scripting.asp (visited: 2004-03-14), 2004.

[NJ00] Miloslav Nic and Jiri Jirat. *XPath Tutorial.* Zvon.org, http://www.zvon.org/xxl/XPathTutorial/General/examples.html (visited: 2004-03-03), 2000.

[Pix00] Tom Pixley, editor. *Document Object Model Events.* W3C, http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113/events.html (visited: 2004-03-09), 2000.

[Pro02a] The Apache Jakarta Project, editor. *BSF Documentation.* Apache Software Foundation, http.//jakarta.apache.org/bsf/manual.html (visited: 2004-03-04), 2002.

[Pro02b] The Apache Jakarta Project, editor. *BSF FAQ.* Apache Software Foundation, http://jakarta.apache.org/bsf/faq.html, 2002.

[Pro04] The Apache Jakarta Project, editor. *The Jakarta Site.* The Apache Software Foundation, http://jakarta.apache.org/index.html, 2004.

[RS01] Tobias Rademacher and James Strachan. *Dom4j Cookbook.* MetaStuff Ltd., http://www.dom4j.org/cookbook/cookbook.pdf (visited: 2004-03-03), 2001.

[SCM03] James Strachan, Maarten Coene, and Bob McWhirter. *dom4j: The Flexible XML Framework for Java.* MetaStuff Ltd., http.//www.dom4j.org/ (visited 2004-03-03), 2003.

[SH03a] Arne Schäpers and Rudolf Huttary. Daniel Düsentrieb, C#, Java, C++ und Delphi im Effizienztest, Teil 1. *c't magazin für computertechnik*, (19):204, 2003.

[SH03b] Arne Schäpers and Rudolf Huttary. Daniel Düsentrieb, C#, Java, C++ und Delphi im Effizienztest, Teil 2. *c't magazin für computertechnik*, (21):222, 2003.

[Smi03] Brett Smith. *GNU Free Documentation License.* Free Software Foundation, Inc., http://www.gnu.org/copyleft/fdl.html, 2003.

[Spe04a] Tobias Specht. *BSFWebScripting (BWS).* BerliOS, http://bsfws.berlios.de/ (visited 2004-03-03), 2004.

[Spe04b] Tobias Specht. *BWS Documentation Wiki.* BerliOS, http://openfacts.berlios.de/index-en.phtml?title=BSFWebScripting (visited: 2004-03-03), 2004.

[Sta04] Richard M. Stallman. *GNU General Public License.* Free Software Foundation, Inc., http://www.gnu.org/copyleft/gpl.html (vistited: 2004-03-05), 2004.

[W3S04] *Introduction to JavaScript.* Refsnes Data, http://www.w3schools.com/js/js_intro.asp (visited: 2004-03-08), 2004.

[Wik04a] *Wikipedia Article: Java Programming Language.* Wikipedia, http://en.wikipedia.org/wiki/Java_programming_language (visited: 2004-03-08), 2004.

[Wik04b] *Wikipedia Article: Java Platform.* Wikipedia, http://en.wikipedia.org/wiki/Java_platform (visited: 2004-03-08), 2004.

[Wik04c] *Wikipedia Article: JavaScript.* Wikipedia, http://en.wikipedia.org/wiki/JavaScript (2004-03-16), 2004.

[Wik04d] *Wikipedia Article: World Wide Web.* Wikipedia, http://en.wikipedia.org/wiki/Www (visited: 2004-03-03), 2004.