



# JExplorer Programmer's Guide

Version 2.6  
April 03, 2012

## Table of Contents

<b>1. Introduction</b> .....	<b>1</b>
<b>2. JExplorer Architecture</b> .....	<b>2</b>
2.1. General IWebBrowser2 Interface Functionality .....	2
2.2. Exposing DOM Model.....	2
2.3. Listening to Events .....	2
2.4. JavaScript Execution .....	2
<b>3. Using Internet Explorer in a Java Application</b> .....	<b>3</b>
<b>4. Document Object Model Implementation in JExplorer</b> .....	<b>4</b>
<b>5. DOM Element Attributes</b> .....	<b>5</b>
<b>6. DOM Element Styles</b> .....	<b>6</b>
<b>7. Navigating to URL and Submitting a Form</b> .....	<b>7</b>
<b>8. Working with Pop-ups</b> .....	<b>8</b>
<b>9. Working with Events from DOM Elements</b> .....	<b>9</b>
<b>10. Working with Browser Events</b> .....	<b>10</b>
<b>11. Working with Cookies</b> .....	<b>12</b>
<b>12. Working with Proxy Settings</b> .....	<b>13</b>
<b>13. Executing Document Operations (Save, Print)</b> .....	<b>14</b>
<b>14. Setting Custom Context Menu</b> .....	<b>15</b>
14.1. Configuring Swing to Use Heavyweight Pop-up Menus .....	15
14.2. Implementing Context Menu Provider .....	15
14.3. Assigning Context Menu Provider .....	15
14.4. Getting Context Element in Action Code .....	15
14.5. Disabling Context Menu.....	16
<b>15. Working with Native Peers and Avoiding Thread Issues</b> .....	<b>17</b>
<b>16. Pop-up windows management</b> .....	<b>18</b>
<b>17. Using JExplorer in Applets</b> .....	<b>20</b>
<b>18. Support</b> .....	<b>22</b>
18.1. JExplorer Forum .....	22
18.2. Reporting Problems.....	22
18.3. Troubleshooting .....	22
<b>19. Where to Get a New Version</b> .....	<b>23</b>

## Chapter 1. Introduction

All Java programmers who have ever developed a Web site testing tool or desktop Java application with an HTML viewer need a library that could load, display an HTML document and access its content. Of course, the implementation of such a library from scratch may take a lot of programming and testing efforts. Therefore, integrating the existing browser is a preferable way.

The JExplorer library is based on the Microsoft Internet Explorer browser and provides the base `Web-Browser` interface and its two implementations: the `Browser` component and `HeadlessBrowser` class. The `Browser` component displays HTML documents in Java Swing applications. The `HeadlessBrowser` class works like the `Browser` component, except that it doesn't have a visual presentation and can be used for automated testing.

Also, it would be useful if the library could provide access to the document object model (DOM). For example, some testing script may need to find a form in the HTML document, fill its controls with values and then submit the form to the web server. The JExplorer library provides such functionality. Besides, it can handle various browser and document events, read and write elements' attributes, execute JavaScript code, and even listen to script error events, etc. The DOM implementation in JExplorer provides access to elements inside frames.

The JExplorer distribution contains a demo application (`JExplorerDemo`) allowing you to browse HTML documents in tabbed mode and view the DOM model of a currently opened page. You can find the application in the `samples\JExplorerDemo` folder.

Each chapter below provides code samples which you can also find in the `samples\Guide` folder of the JExplorer distribution.

## Chapter 2. JExplorer Architecture

As mentioned above, JExplorer uses Internet Explorer, i.e. JExplorer is a Java wrapper for the WebBrowser COM component and is based on the COM integration of the JNIWrapper library.

The functionality of the WebBrowser component can be divided into several parts described in the sections below.

### 2.1. General IWebBrowser2 Interface Functionality

JExplorer provides the WebBrowser interface that is implemented by both Browser and HeadlessBrowser classes. Using this interface, you can:

- Load a new document into the browser.
- Use history commands.
- Disable error, alert, and confirmation dialogs.

### 2.2. Exposing DOM Model

The DOM model of Internet Explorer is exposed through the IWebBrowser2::Document property as a tree of COM objects that represent elements in the document. The WebBrowser interface from the JExplorer library has the `getDocument()` method that returns a Java object (a wrapper for the native document). The returned object implements the `org.w3c.dom.html.HTMLDocument` interface, so the document object contains methods for accessing the elements in a document. These methods return objects that wrap native COM objects representing elements or element collections. The element wrapper implements the `org.w3c.dom.htmlHTMLElement` interface, and the element collection wrapper implements the `org.w3c.dom.NodeList` interface.

### 2.3. Listening to Events

JExplorer allows you to set up listeners for getting events related to navigation to another page, creation of a new Internet Explorer window, status bar text change, etc. You can find these listeners in the `event` subpackage.

JExplorer uses its own dispatch server that implements the `DWebBrowserEvents2` interface and receives notifications from the WebBrowser component. This dispatch server calls appropriate methods of Java listeners and handlers. Method parameters in the listeners and handlers are of Java types: the library creates corresponding Java objects for native objects. For example, `BSTR` is converted to a `java.lang.String` object, `IWebBrowser2` interface pointer to a Java object implementing the WebBrowser interface, and so on.

### 2.4. JavaScript Execution

To execute JavaScript code, use the `executeScript()` method provided by the WebBrowser interface.

## Chapter 3. Using Internet Explorer in a Java Application

JExplorer provides two main classes for working with Internet Explorer: the `Browser` class, which is a visual Swing component, and the `HeadlessBrowser` class, which has no visual presentation. Both classes reside in the `com.jniwrapper.win32.ie` package. The example below shows how to add an instance of the `Browser` component to `JFrame`:

```
JFrame frame = new JFrame();
frame.setSize(800, 600);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
Browser browser = new Browser();
frame.getContentPane().add(browser);
frame.setVisible(true);
```

The `Browser.navigate()` method loads a specified HTML document:

```
browser.navigate("http://www.teamdev.com");
```

The `Browser.navigate()` method starts document loading, but it doesn't guarantee that the document is completely loaded after the method is executed. Note that you will not find elements or get the contents for the document if it is partly loaded. Use the `Browser.waitReady()` method to force the calling thread to wait until the document is loaded completely:

```
browser.waitReady();
String content = browser.getContent();
System.out.println("Document: " + content);
```

Again, you should call the `Browser.waitReady()` method after `Browser.navigate()` and before `Browser.getContent()` and `Browser.getDocument()` calls. Do not call the `WebBrowser.waitReady()` method in the Swing thread because this causes deadlock. For the same reason, do not execute time-consuming tasks in the Swing thread.

Now let's try to create a `HeadlessBrowser` instance:

```
WebBrowser browser = new HeadlessBrowser();
```

The `HeadlessBrowser` class doesn't require any container as a parent, nor does it create visible Swing windows. The `HeadlessBrowser` class uses an invisible Swing window in its implementation, therefore a Swing thread runs when the program creates `HeadlessBrowser` instances. A Java application does not terminate when the main thread exits with the Swing thread still running. Therefore, call the `System.exit(0)` method to terminate the program.

The `com.jniwrapper.win32.ie.WebBrowser` interface provides `navigate()`, `waitReady()`, `getContent()` and other useful methods. Therefore, the `Browser` and `HeadlessBrowser` implementations are interchangeable.

The following code demonstrates the technique of loading an HTML document from a `java.lang.String` object:

```
WebBrowser browser = new HeadlessBrowser();

String content = "<html><body><h1>Simple document</h1></body></html>";

browser.setContent(content);
browser.waitReady();

String content = browser.getContent();
System.out.println("Document: " + content);
```

## Chapter 4. Document Object Model Implementation in JExplorer

In the previous chapter, we called the `browser.getContent()` method and obtained a document as a String. You can also manipulate the document as a Document Object Model (DOM):

```
HTMLDocument document = browser.getDocument();
Element form = document.getElementById("mainForm");
```

The above code returns a `org.w3c.dom.html.HTMLDocument` object that represents a document loaded into the browser and finds the element by ID. The found element is an `org.w3c.dom.Element` instance. A DOM element wrapper is an object of `org.w3c.dom.html.HTMLElement` type, so the form object supports such functionality as clicking an element, firing an event on some element, getting an element position, size, styles and other:

```
HTMLElement htmlElement = (HTMLElement) form;
String className = htmlElement.getClassName();
```

Another way to find elements in the document is to search them by tag name:

```
NodeList forms = document.getElementsByTagName("form");
HTMLElement form1 = (HTMLElement) forms.item(0);
```

The forms object is a list of all forms in the document. The `NodeList.item()` method returns one element from this list. Another useful method of the `NodeList` interface is `getLength()` that returns the number of elements in the list.

Apart from finding elements by ID or tag name, you can get the children or parent of the element. The `org.w3c.dom.Node` interface provides the `getChildNodes()` and `getParentNode()` methods for these tasks. The `org.w3c.dom.Document` and `org.w3c.dom.Element` interfaces extend the `Node` interface.

And finally, the `org.w3c.dom.Element` interface contains the `getElementsByName()` method, enabling you to search for an element inside another element. For example, you can search input elements in the form:

```
NodeList inputs = mainForm.getElementsByName("input");
```

If you need more information about DOM, please refer to [W3C Document Object Model](#).

*For the complete samples, please refer to the following files:*

`dom/DomNavigationSample.java`, `dom/SubmitFormSample.java`

## Chapter 5. DOM Element Attributes

Element attributes can be used to fill the form controls on the HTML page with values, get the link text or retrieve the type of an input element. The `Element.getAttribute()` method returns an attribute value. The following code finds the "Groups" link on a Google page and gets the link text:

```
WebBrowser browser = new HeadlessBrowser();

browser.navigate("http://www.google.com");
browser.waitReady();

Element groupsLink = (Element) browser.getDocument().getElementsByTagName("a").item(1);
String linkText = groupsLink.getAttribute("innerHTML");
```

Use the `type` attribute of the input element to determine its type:

```
browser.navigate("http://www.google.com");
browser.waitReady();

Element input = (Element) browser.getDocument().getElementsByTagName("input").item(0);
String inputType = input.getAttribute("type");
```

The `Element.setAttribute()` method sets the element's attribute value. The following sample fills the input field on the Google page:

```
browser.navigate("http://www.google.com");
browser.waitReady();

Element textInput = getTextInput(browser.getDocument());
textInput.setAttribute("value", "JExplorer");
```

Use the `value` attribute to fill the text and password fields, text areas and select controls with values as demonstrated in the sample above. Use the `checked` attribute to work with check boxes and radio button controls:

```
// select a checkbox
checkbox.setAttribute("checked", "true");

// select a radio button
radio1.setAttribute("checked", "true");
```

Note that setting a value in the file chooser control through the `value` attribute does not work because of the security restrictions of the `WebBrowser COM` component.

You can find additional information about DHTML attributes in [HTML and DHTML Reference on MSDN](#).

## Chapter 6. DOM Element Styles

Each DOM element has style properties (or styles) responsible for its visual presentation in an HTML page. Each style determines a certain aspect of visual presentation. For example, the border style contains information about the border type, its thickness, and color.

JExplorer provides methods for managing styles in the `HTMLElement` interface. The sample below finds the font element in the document and changes its background color:

```
browser.navigate("http://www.google.com ");
browser.waitReady();

HTMLElement font = getFontElement(browser.getDocument());
DOMUtils.setStyle(font, "background", "green");
```

The `DOMUtils.getStyle()` method returns the element's style value.

## Chapter 7. Navigating to URL and Submitting a Form

JExplorer provides the ability to click a link or button in the form. Use the `DOMUtils.click()` method to emulate the user click action on an element. The sample below shows how to click a link on the HTML page:

```
browser.navigate("http://www.google.com");  
browser.waitReady();
```

```
HTMLInputElement groupsLink = (HTMLInputElement) browser.getDocument().getElementsByTagName("a").item(1);  
DOMUtils.click(groupsLink);
```

The `DOMUtils.click()` method used on the form's submit button sends the form data to the server. The following sample runs Google search:

```
browser.navigate("http://www.google.com");  
browser.waitReady();
```

```
Element textInput = getTextInput(browser.getDocument());  
textInput.setAttribute("value", "JExplorer");
```

```
HTMLInputElement button = getSearchButton(browser.getDocument());  
DOMUtils.click(button);
```

## Chapter 8. Working with Pop-ups

Sometimes you need to click a button that opens a pop-up window, fill a form in that pop-up window, submit the form and then continue working with the document that opened the pop-up window. JExplorer allows you to wait for a pop-up window to be loaded and get a reference to it:

```
// Start tracking creation of a child window
browser.trackChildren();

HTMLInputElement element = (HTMLInputElement) browser.getDocument().getElementById("theLink");

// click the link that opens a pop-up window
element.click();

// get a reference to the pop-up window
WebBrowser child = browser.waitForChildCreation();
child.waitForReady();

String childTitle = child.getDocument().getTitle();
```

## Chapter 9. Working with Events from DOM Elements

Each DOM element wrapper in JExplorer's Document Object Model implements the `org.w3c.dom.events.EventTarget` interface. Therefore, you can add a listener to an element:

```
HTMLElement link = (HTMLElement) links.item(i);

EventListener listener = new EventListener() {
    public void handleEvent(Event evt) {
        Element target = (Element) event.getTarget();
        System.out.println("Event from " + target.tagName());
    }
};
((EventTarget)link).addEventListener("onclick", listener, false);
```

The event listener implements the `org.w3c.dom.events.EventListener` interface with one method that is called when a specified event is fired. The `Event` object passed to the method contains information about the event type, source of the event, coordinates of the mouse pointer for the mouse event, and pressed keys.

The sample below adds a listener for the `onchange` event from the `select` element:

```
private void setupChangeListener(final Element select) {
    EventTarget eventTarget = (EventTarget) select;
    eventTarget.addEventListener("onchange", new EventListener() {
        public void handleEvent(Event evt) {
            Element target = (Element) evt.getTarget();
            // Gets current value of select target
            String value = target.getAttribute("value");
        }
    }, false);
}
```

The event handler gets the current value of the `select` element and prints it. The `org.w3c.dom.Element` does not extend the `org.w3c.dom.events.EventTarget` interface, so you need to cast the `select` element to the `EventTarget` type.

## Chapter 10. Working with Browser Events

Internet Explorer sends notifications about its various events related to navigation, document load completion, document title or status bar text change, etc.

`NavigationEventListener`, `StatusEventListener`, `NewWindowEventHandler`, `NewWindowEventListener`, and `WebBrowserEventsHandler` provide methods that are called when the browser events occur. You can find these interfaces and their basic implementations in the `com.jniwrapper.win32.ie.event` package.

The following sample demonstrates how to add a document completed event listener to the browser:

```
browser.addNavigationListener(new NavigationEventAdapter() {
    public void documentCompleted(WebBrowser webBrowser, String url) {
        System.out.println("Document completed: " + url);
    }
});
```

`NavigationEventListener` handles navigation-related events, such as the start and completion of navigation and its progress. The `documentCompleted()` method is called when the document is completely loaded to the browser.

`StatusEventListener` handles browser state change events such as changes of the document title, status text, navigation button state, etc.

The `NavigationEventAdapter` class implements the `NavigationEventListener` interface and contains only methods with empty bodies. You can extend the adapter class instead of implementing a listener. This way is convenient when you create a listener that handles only one event.

Instances of the `Browser` and `HeadlessBrowser` classes can have several listeners. However, each instance of `WebBrowser` cannot have more than one event handler, because the handler methods return values that affect the browser's behavior. For example, you can cancel navigation by a link.

The code below shows how to add a handler for a navigation event:

```
browser.setEventHandler(new DefaultWebBrowserEventsHandler() {
    public boolean beforeNavigate(
        WebBrowser webBrowser,
        String url,
        String targetFrameName,
        String postData,
        String headers) {
        System.out.println("url = " + url);
        System.out.println("postData = " + postData);
        System.out.println("headers = " + headers);

        // proceed
        return false;
    }
});
```

The `beforeNavigate` event is fired when some link is clicked or a form is submitted. If the handler's method returns `true`, navigation is cancelled. Otherwise, navigation or the form submission is performed. The `DefaultWebBrowserEventsHandler` class is the basic implementation of `WebBrowserEventsHandler` that enables navigation by links, navigation to error pages, etc.

`NewWindowEventHandler` contains the `newWindow()` method that you can use to create a tabbed browser. Please see `JExplorerDemo` for more details.

Using `JExplorer`, a Java application can listen to JavaScript errors that may occur in a document. Use `ScriptErrorListener` with the `errorOccured()` method for this purpose. This method is called on JavaScript error in the browser document. The code below demonstrates how to assign a script error listener for the current document:

```
browser.setScriptErrorListener(new ScriptErrorListener() {
```

```
public void errorOccured(ScriptErrorEvent event) {
    int errorCode = event.getErrorCode();
    String url = event.getUrl();
    String message = event.getMessage();

    System.out.println("errorCode = " + errorCode);
    System.out.println("url = " + url);
    System.out.println("message = " + message);
}
});
```

Any script error produces a dialog that blocks the execution of the calling thread until the dialog is closed. This is inconvenient for code that uses the `HeadlessBrowser` class. The code below demonstrates how to disable the error, alert, and confirmation dialogs:

```
WebBrowser browser = new HeadlessBrowser();
browser.setSilent(true);
```

Note that the `WebBrowser.setSilent()` call sets up its own `DialogEventHandler`. Therefore, if a dialog event handler is already set up, it will be removed.

## Chapter 11. Working with Cookies

WebBrowser provides methods for working with document cookies. The `WebBrowser.getCookies(URL url)` method returns `java.util.Set` of `com.jniwrapper.win32.ie.dom.Cookie` objects that contain the name, value, domain of a cookie, and other information:

```
URL domain = new URL("http://www.google.com");
Set<Cookie> cookies = browser.getCookies(domain);
if (cookies.size() > 0) {
    Cookie cookie = cookies.iterator().next();
}
```

If you need to set a cookie, follow these steps:

1. Create a `Cookie` object.
2. Assign its value and other needed parameters.
3. Add the `Cookie` object to an empty set.
4. Pass this set to the `WebBrowser.setCookies(URL url, Set<Cookie> cookies)` method.

The cookie will be modified after the user navigates to a different page or sends the form data to the server.

## Chapter 12. Working with Proxy Settings

The `ProxyManager` class provides methods for working with global proxy settings and the ones available only for the current process. The following sample code demonstrates how to update global proxy settings:

```
ProxyManager proxyManager = ProxyManager.getInstance();

// changes global proxy settings
ProxyConfiguration globalProxySettings = proxyManager.getProxyConfiguration(ProxyConfiguration.Type.GLOBAL);
globalProxySettings.setConnectionType(ProxyConfiguration.ConnectionType.PROXY);
globalProxySettings.setProxy("<http proxy address>", ProxyConfiguration.ServerType.HTTP);
globalProxySettings.setUserName("<user name>", ProxyConfiguration.ServerType.HTTP);
globalProxySettings.setPassword("<password>", ProxyConfiguration.ServerType.HTTP);
globalProxySettings.setIgnoreProxyForLocalAddresses(true);

proxyManager.updateProxyConfiguration(globalProxySettings, ProxyConfiguration.Type.GLOBAL);
```

To get global proxy information, you need just to pass the `ProxyConfiguration.Type.GLOBAL` value to the `ProxyManager.getProxyConfiguration()` method. To get the proxy settings available only for the current process, pass the `ProxyConfiguration.Type.PROCESS` value to this method.

**Note:** By default, the current process uses global proxy settings. If you update proxy settings for the current process using `proxyManager.updateProxyConfiguration`, you will be unable to use global proxy setting unless the current process is complete. This is standard behavior of Internet Explorer and there's no known way to go back to using global proxy settings.

## **Chapter 13. Executing Document Operations (Save, Print)**

The `IWebBrowser2.execWB()` native interface call allows you to execute such commands as invoking Save, Print, and Print Preview dialogs. JExplorer has the `WebBrowser.execute(BrowserCommand command)` method that provides the same functionality. You create a `BrowserCommand` instance and pass it to the `WebBrowser.execute()` method:

```
browser.execute(new SaveAsCommand());
```

The `com.jniwrapper.win32.ie.command` package contains classes for working with browser commands.

## Chapter 14. Setting Custom Context Menu

The `Browser` component allows you to display a custom Java pop-up menu instead of the default one displayed by Internet Explorer. Because the contents of the menu depends on what element of a page is under the mouse pointer, the menu is dynamically constructed by the application.

To customize the browser context menu, you need to:

1. Configure Swing to use heavyweight pop-up menus.
2. Implement the `ContextMenuProvider` interface.
3. Assign the context menu provider to the browser.

Below is given a detailed description of these steps.

### 14.1. Configuring Swing to Use Heavyweight Pop-up Menus

Since the browser is a native window inserted into Swing UI, pop-up menus will not be displayed over the browser component unless the menus are made heavyweight, as the code line below shows:

```
JPopupMenu.setDefaultLightWeightPopupEnabled(false);
```

### 14.2. Implementing Context Menu Provider

The purpose of the context menu provider is to supply the browser control with an instance of `JPopupMenu` that provides actions for the DOM element clicked in the browser. The `ContextMenuProvider` interface defines a single `getPopupMenu()` method that accepts the `org.w3c.dom.Element` instance.

The following code demonstrates a simple provider that changes a menu item text if the context menu is invoked for a link element:

```
ContextMenuProvider contextMenuProvider = new ContextMenuProvider() {
    public JPopupMenu getPopupMenu(Element contextElement) {
        final boolean isLink = "A".equals(contextElement.getTagName());
        String openItemText = isLink ? "Open in New Tab" : "New Tab";

        _openItem.setText(openItemText);

        return _contextMenu;
    }
};
```

Note that the `Browser` component invokes the menu provider's method each time before displaying the context menu. It is necessary that the implementation does not contain time-consuming operations to ensure that the menu is displayed promptly at the user's request.

### 14.3. Assigning Context Menu Provider

To assign the context menu provider that was created at the previous step, the `setContextMenuProvider()` method of the `Browser` component should be called:

```
browser.setContextMenuProvider(contextMenuProvider);
```

## **14.4. Getting Context Element in Action Code**

To get an element the context name was called for, use the `getContextElement()` method of the browser component:

```
Element contextElement = browser.getContextElement();
```

## **14.5. Disabling Context Menu**

To disable the context menu, you need to implement `ContextMenuProvider` that returns `null` in the `getPopupMenu()` method and assign the provider to the browser.

## Chapter 15. Working with Native Peers and Avoiding Thread Issues

The Web browser control is a thread unsafe component, therefore, you should call its methods from one thread, i.e. from the thread that created this component (its OLE message loop). JExplorer API calls (WebBrowser interface, DOM wrappers and other) invoke native methods from this thread, so you don't need to invoke these methods from the OLE message loop. The direct use of native peers' methods should be performed in the OLE message loop. You should consider this when working with native peer objects (interface method calls and interface querying) like shown in the code sample below:

```
IWebBrowser2 peer = (IWebBrowser2)getBrowser().getBrowserPeer();
IDispatch dispatch = peer.getDocument();
IHTMLDocument2 doc2 = new IHTMLDocument2Impl(peer.getDocument());
String title = doc2.getTitle().getValue();
```

The native peer functionality is available in the JExplorer Professional version only. You can find a Java wrapper for the IWebBrowser2 COM interface.

If you need to work with a native object, do it in the browser OLE message loop:

```
getBrowser().getOleMessageLoop().doInvokeLater(new Runnable() {
    public void run() {
        IWebBrowser2 peer = (IWebBrowser2)getBrowser().getBrowserPeer();
        IDispatch dispatch = peer.getDocument();
        IHTMLDocument2 doc2 = new IHTMLDocument2Impl(peer.getDocument());
        String title = doc2.getTitle().getValue();
    }
});
```

The OleMessageLoop class provides two methods for executing methods in this thread: doInvokeLater() and doInvokeAndWait(). The difference between these methods is that the former returns immediately, and the latter blocks further execution while the task defined by Runnable method parameter is being executed. Use doInvokeLater() in a Swing thread for time-consuming tasks to prevent this thread from blocking.

The HTMLInputElement interface provides the getElementPeer() method that returns an IHTMLInputElement instance (a Java wrapper for a COM interface). Therefore, you can call COM interface methods of an element to work with the functionality not provided by JExplorer:

```
WebBrowser browser = new HeadlessBrowser();

browser.setContent("<html><body><div width='400' height='300'>div text</div></body>");
browser.waitReady();

HTMLInputElement element = (HTMLInputElement) browser.getDocument().getElementsByTagName("div").item(0);
final IHTMLInputElement peer = (IHTMLInputElement) element.getElementPeer();

browser.getOleMessageLoop().doInvokeAndWait(new Runnable() {
    public void run() {
        BStr outerHTML = peer.getOuterHTML();
        System.out.println("outerHTML = " + outerHTML);
    }
});
```

## Chapter 16. Pop-up windows management

A lot of web-pages use the `window.open` JavaScript function to open a new pop-up window. This function allows creating pop-up window with the features such as width, height, top, left, toolbar, status bar, full screen, menu bar etc. For example, the following code opens a resizable window with menu bar, status bar and size 350 X 250:

```
window.open("about:blank","mywindow","menubar=1,statusbar=1,resizable=1,width=350,height=250");
```

In JExplorer 1.9.1 was introduced new functionality that allows receiving these features through `com.jniwrapper.win3`. For example, the following sample code shows how to configure pop-up window according to the features that were passed to the `window.open` JavaScript function:

```
Browser browser = new Browser();

JFrame frame = new JFrame();
frame.setContentPane(browser);
frame.setSize(400, 200);
frame.setLocationRelativeTo(null);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);

browser.waitReady();
browser.setNewWindowHandler(new DefaultNewWindowHandler(browser));
browser.executeScript("window.open('about:blank', 'name',
'resizable=no, fullscreen=no, top=200, left=400, width=300, height=200');");
```

Where the `DefaultNewWindowHandler` looks like:

```
private static class DefaultNewWindowHandler implements NewWindowEventHandler {
    private Browser parent;

    private DefaultNewWindowHandler(Browser parent) {
        this.parent = parent;
    }

    public NewWindowEventHandler.NewWindowAction newWindow() {
        final JFrame frame = new JFrame();
        final Browser browser = new Browser(parent.getOleMessageLoop());

        browser.addBrowserWindowListener(new BrowserWindowAdapter() {
            public void onWindowResize(final BrowserWindowEvent event) {
                SwingUtilities.invokeLater(new Runnable() {
                    public void run() {
                        frame.setBounds(event.getWindowBounds());
                    }
                });
            }

            public void onWindowResizable(final BrowserWindowEvent event) {
                SwingUtilities.invokeLater(new Runnable() {
                    public void run() {
                        frame.setResizable(event.isWindowResizable());
                    }
                });
            }

            public void onFullScreen(final BrowserWindowEvent event) {
                SwingUtilities.invokeLater(new Runnable() {
                    public void run() {
```

```
        boolean fullScreen = event.isFullScreen();
        int state = fullScreen ? Frame.MAXIMIZED_BOTH : Frame.NORMAL;
        frame.setExtendedState(state);
    }
});

public void onVisible(BrowserWindowEvent event) {
    frame.setVisible(event.isVisible());
}
});

frame.setContentPane(browser);
frame.setVisible(true);

NewWindowEventHandler.NewWindowAction result = new NewWindowEventHandler.NewWindowAction(
    NewWindowEventHandler.NewWindowAction.ACTION_REPLACE_BROWSER);
result.setBrowser(browser);

return result;
}
}
```

## Chapter 17. Using JExplorer in Applets

To use JExplorer in applets, follow these instructions:

1. Copy `jniwrap.dll` to the folder of a web server computer where the applet resides (it is recommended, but you can place `jniwrap.dll` to some other folder on the web server).

2. Prepend the following code to the `init()` method of the applet:

```
// passes the instance of the current object as parameter
AppletHelper.getInstance().init(this);
```

This call downloads `jniwrap.dll` from the web server and copies it to the Windows `system32` folder in the client's computer. The presence of `jniwrap.dll` on the client side is required, otherwise the applet will not work.

If you copied `jniwrap.dll` to a folder with no residing applet, you should provide JNIWrapper library URL, for example:

```
AppletHelper.getInstance().init("http://applets.com/native/jniwrap.dll");
```

3. Prepend the following code to the method to start the applet method:

```
AppletHelper.getInstance().start();
```

This call starts `NativeResourceCollector` thread of `JNIWrapper`.

4. Append the following code to the method to stop the applet method:

```
AppletHelper.getInstance().stop();
```

This call stops `NativeResourceCollector` thread of `JNIWrapper`.

**Important:** You must dispose all Browser component instances in this method. In a different way it will cause incorrect behavior during reloading the web page.

5. The license files (`jniwrap.lic`, `comfyj.lic` and `jexplorer.lic`) can be included into any JAR file of a JWS application, into the `META-INF` subfolder.
6. `application.jar` should be signed and should reference the JExplorer libraries in its manifest file by setting the class path variable. Signing JExplorer libraries is not necessary as they are already signed.

The sample build file that prepares an applet library is shown below:

```
<project name="Applet Sample" default="build" basedir=".">
  <property name="jniwrapper-licenseFile" value="jniwrap.lic"/>
  <property name="comfyj-licenseFile" value="comfyj.lic"/>
  <property name="jexplorer-licenseFile" value="jexplorer.lic"/>

  <property name="native-library" value="jniwrap.dll"/>

  <property name="certificate" value=".keystore"/>
  <property name="applet-classes" value="classes"/>
  <property name="output-jar" value="sample.jar"/>

  <target name="build">
    <jar destfile="${output-jar}" index="true">
      <fileset dir="${applet-classes}" includes="AppletSample.class"/>
      <zipfileset file="${jniwrapper-licenseFile}" prefix="META-INF"/>
      <zipfileset file="${comfyj-licenseFile}" prefix="META-INF"/>
      <zipfileset file="${jexplorer-licenseFile}" prefix="META-INF"/>
      <zipfileset file="${native-library}" />
    </jar>
  </target>
</project>
```

```
<signjar jar="{output-jar}" alias="your_alias" keystore="{certificate}"
  storepass="your_storepass" keypass="your_keypass"/>
</target>
</project>
```

Below is given the applet usage sample:

```
<html>
<body><h1>Applet Sample</h1>
<p>
<applet docbase="http://your_url" code="AppletSample.class" width="800" height="600"
  archive="sample.jar,jniwrap-3.6.1.jar,comfj-2.4.jar,jexplorer-1.8.jar,winpack-3.6.jar"/>
</body>
</html>
```

## Chapter 18. Support

If you have any problems or questions regarding JExplorer, please e-mail us at:

[jexplorer-support@teamdev.com](mailto:jexplorer-support@teamdev.com)

### 18.1. JExplorer Forum

If you want to discuss topics related to JExplorer, please visit a specialized forum on the TeamDev integrated customer support and troubleshooting center at:

<http://support.teamdev.com/category.jspa?categoryID=11>

### 18.2. Reporting Problems

If you find any bugs in JExplorer, please submit the issue to us using a special report form on the TeamDev integrated customer support and troubleshooting center at:

<http://support.teamdev.com/>

The form will help you provide all necessary information.

### 18.3. Troubleshooting

To find a solution to your problem, please visit the Troubleshooting page on our site at:

<http://www.teamdev.com/explorer/tshoot.jsf>

This page is regularly updated using information from support requests.

If you don't find a solution, please e-mail us at [jexplorer-support@teamdev.com](mailto:jexplorer-support@teamdev.com) or report the problem as described in the previous section.

## **Chapter 19. Where to Get a New Version**

To get the latest version of JExplorer, please visit:

<http://www.teamdev.com/jexplorer/downloads.jsf>