



# Java Plug-in 1.4.2 Developer Guide

## Part I: Java Plug-in Basics

[Chapter 1: Overview—What Is Java Plug-in? What Does It Support?](#)

[Chapter 2: Using the Conventional Applet Tag](#)

[Chapter 3: Using OBJECT, EMBED and APPLET Tags in Java Plug-in](#)

[Chapter 4: Using the HTML Converter to Convert Applet Tags for Java Plug-in](#)

[Chapter 5: Proxy Configuration](#)

[Chapter 6: Protocol Support: HTTP, HTTPS, FTP, GOPHER, SOCKS, and NTLM](#)

[Chapter 7: Cookie Support](#)

[Chapter 8: Applet Caching](#)

[Chapter 9: Using the Java Plug-in Control Panel to Set Plug-in Behavior/Options](#)

## Part II: Deployment Schemes

[Chapter 10: Installation for Conventional Applets \(Microsoft Windows Only\)](#)

[Chapter 11: Intranet With `OBJECT` / `EMBED` Tag](#)

[Chapter 12: Internet Deployment](#)

[Chapter 13: Silent Installation](#)

[Chapter 14: Jar Indexing](#)

[Chapter 15: Java Server Pages](#)

## Part III: Security

[Chapter 16: Overview—Applet Security Basics](#)

[Chapter 17: How RSA Signed Applet Verification Works in Java Plug-in](#)

[Chapter 18: How to Sign Applets Using RSA-Signed Certificates](#)

[Chapter 19: How to Deploy RSA-Signed Applets in Java Plug-in](#)

## Part IV: Debugging Support

[Chapter 20: Debugging Support](#)

[Chapter 21: Java Plug-in Console](#)

[Chapter 22: Tracing and Logging](#)

## Part V: Advanced Topics

[Chapter 23: Supporting Multiple Versions of JRE/Java Plug-in](#)

[Chapter 24: Java-to-JavaScript Communication](#)

[Chapter 25: JavaScript-to-Java Communication \(Scripting\)](#)

[Chapter 26: Deploying Java Extensions](#)

[Chapter 27: Applet Persistence API](#)

[Chapter 28: Special Applet Attributes](#)

# Appendices

[Appendix 1: Netscape 6](#)

[Appendix 2: Frequently Asked Questions \(FAQ\)](#)

- [Basic Information FAQ](#)
- [Developer Information FAQ](#)
- [Troubleshooting FAQ](#)
- [About Applet Tag Support in Java Plug-in](#)

[Appendix 3: More About HTML Converter](#)

[Appendix 4: Microsoft VM and Java 2 Applet Compatibility Issues](#)

[Appendix 5: Complete Example—Deploying Java Media Framework as Java Extension](#)

[Appendix 6: Sun-Supported Specification-Version and Implementation-Version Formats](#)

---

# Overview—What Is Java Plug-in? What Does It Support?

---

This chapter includes the following topics:

- [What Is Java Plug-in?](#)
- [Supported Operating Systems and Browsers](#)

## What Is Java Plug-in?

Java Plug-in extends the functionality of a web browser, allowing applets or Java Beans to be run under Sun's Java 2 runtime environment (JRE) rather than the Java runtime environment that comes with the web browser. Java Plug-in is part of Sun's JRE and is installed with it when the JRE is installed on a computer. It works with both Netscape and Internet Explorer.

This functionality can be achieved in two different ways:

1. By [using the conventional APPLET tag](#) in a web page.
2. By replacing the APPLET tag with the OBJECT tag for Internet Explorer; by replacing the APPLET tag with the EMBED tag for Netscape 4. Note, however, that the OBJECT and EMBED tags must conform to a special format as described in the next chapter, [Using OBJECT, EMBED and APPLET Tags in Java Plug-in](#).

### **Note:**

Currently the OBJECT and EMBED tags do not work with Netscape 6 with Java Plug-in. You must use the APPLET tag with Netscape 6.

OBJECT and EMBED tags may be manually updated in web pages, but to facilitate updating web pages to this special format, an HTML Converter is provided. It is describe in the section called [Using the HTML Converter to Convert Applet Tags for Java Plug-in](#).

While the above constitutes the heart of Java Plug-in, there are many other related topics that you may want to understand. For instance, you may want to know how proxy configuration works in Java Plug-in, you may want to know what protocols Java Plug-in supports, or you may want to know about cookie support and caching. Such topics are included in *Part I: Java Plug-in Basics*.

You can determine some of the behavior of Java Plug-in and set options via the Java Plug-in Control Panel. How you do this is also discussed in Part I in the chapter called *Using the Java Plug-in Control*

*Panel to Set Plug-in Behavior/Options.*

Java Plug-in may be deployed in various ways—on the Internet, within an intranet, via Java Server Pages, etc. The various types and methods of deployments are discussed in the *Part II: Deployment Schemes*.

Applets must be run in a secure environment, and various security topics are discussed in the *Part III: Security*, including RSA signed applet verification. As signing applets can be a difficult topic for novice applet developers, this guide provides step-by-step instructions for signing applets in the chapter called [How to Sign Applets Using RSA-Signed Certificates](#).

Java Plug-in provides various kinds of debugging support for applets, and *Part IV: Debugging Support* describes them. Java Debugger support is discussed. So is the Java Plug-in Console, which includes various options that may be set for debugging, including ones for tracing and logging.

*Part V: Advanced Topics* discusses more esoteric matters such as supporting multiple JREs in the same environment, Java-toJavaScript communication, how to persist applets across browser sessions, special applet attributes for changing the applet window background color, etc.

Finally, there are Appendices that information about Netscape 6; a FAQ; additional information about the HTML Converter; and Microsoft VM versus Java 2 compatibility issues.

## **Supported Operating Systems and Browsers**

### **Operating System Support:**

Window 98 SE, ME, NT 4.0 (SP6), 2000 (SP2), and XP; Solaris 7, 8, and 9; and Linux 6.0 or later.

### **Browser Support:**

Internet Explorer 4 (SP2), 5.01 (SP2), 5.5 (SP2), and 6; Netscape 4.5, 4.6, 4.7, and 4.77; and Netscape 6.2.1 and higher.

---

# Using the Conventional `APPLET` Tag

---

This section includes the following topics:

- [Support for the `APPLET` Tag](#)
- [Compatibility with the `OBJECT` Tag](#)
- [Supported Platforms and Browsers](#)
- [Updating Old Class Files](#)
- [Other Enhancements](#)

## Support for the `APPLET` Tag

Java™ Plug-in now supports the HTML `APPLET` tag for launching applets. Users may configure their browsers so that Sun's JRE is the default runtime environment for handling `APPLET` tags.

Even though Java Plug-in now supports the `APPLET` tag, it does not support applets that make use of proprietary technologies. Examples of such technologies are

- CAB files
- Authenticode signing
- Java Moniker

## Compatibility with the `OBJECT` Tag (Internet Explorer on Windows Platform)

This release of Java Plug-in supports the use of `APPLET` tags for launching applets. However, it is also fully backward compatible with previous Java Plug-in releases in its support of the `OBJECT` tag for launching applets in Internet Explorer running on the Windows platform. As before, developers have the option of using the HTML Converter to set up their applet web pages to use the `OBJECT` tag.

For more information about using `OBJECT` tags to launch applets on Java Plug-in, see [Using `OBJECT`, `EMBED` and `APPLET` Tags in Java Plug-in](#) and [Using the HTML Converter to Convert Applet Tags for Java Plug-in](#).

# Supported Platforms and Browsers

The support for the `APPLET` tag in Java Plug-in is intended for use on the following operating systems:

- Windows 95
- Windows 98 (1st and 2nd editions)
- Windows ME
- Windows NT 4.0
- Windows 2000
- Windows XP
- Unix
- Linux

Java Plug-in provides support for the `APPLET` tag on the following web browsers:

- Internet Explorer 4.0 (4.01 recommended), 5.0 (5.01 recommended), 5.5 (Service Pack 2 recommended), 6.0
- Netscape 6.0, 6.1

## Note

Applet tag support currently does not work on Netscape 4.

## Updating Old Class Files

In some cases, the new Java Runtime Environment associated with this Java Plug-in release will not run class files that were generated with old compilers. The usual symptom is a `java.lang.ClassFormatError` that the virtual machine throws when it attempts to load such a class file. This failure has nothing specifically to do with the changes in this release. Rather, old bytecode compilers did not strictly adhere to proper class-file format in all situations when generating class files. Recent virtual machines are implemented to be strict in enforcing proper class file format, and this can lead to errors when they attempt to load old, improperly formatted class files. Some typical problems in some older class files are (this list is not exhaustive):

- There are extra bytes at the end of the class file;
- The class file contains method or field names that do not begin with a letter;
- The class attempts to access private members of another class;
- The class file has other format errors, including illegal constant pool indices and illegal UTF-8 strings;
- Some early (third-party) bytecode obfuscators produced class files that violated proper class-file format.

You can avoid this type of problem by recompiling your classes with the `Javac` bytecode compiler from

the current Java 2 SDK. If you choose to use a third-party obfuscator, be sure to use one that produces class files that respect proper class-file format.

## Other Enhancements

In addition to the support for the `APPLET` tag described [above](#), the Java Plug-in has many performance and architecture enhancements that will make it more suitable for widespread use on consumer client machines that typically are not as powerful as client platforms in an enterprise environment. Some of these enhancements are summarized below.

### Memory management

- Dynamic maximum heap size is scaled down from 128 MB to 96MB to avoid unnecessary paging on the system.
- Classloader implementation has been tuned to allow memory to be reclaimed more often by the garbage collector.
- Potential memory leak issue is addressed by using JNI/COM smart pointers in implementation.

### Performance

- Applet download time is significant reduced by relying on browser caching when possible. No connection will be opened on the server side unless it is absolutely necessary.
- Applet lifecycle is controlled asynchronously to allow very fast page switch.
- Sped up classloader object reclaim by removing its `finalize()` method.
- HTTPS read has been made significantly faster by increasing buffer size.
- HTTPS calls are significant faster by statically linking Microsoft's Wininet instead of dynamic function lookup every time.
- JavaScript performance is greatly enhanced by eliminating *BeanInfo* lookup over the network.
- Java Console performance is enhanced by using the Console Writer thread to avoid blocking `System.out` and `System.err` when possible.

### Design

- Default JavaScript/Java interaction is changed from "*not-scriptable*" to "*scriptable*", allowing any applets running through the `APPLET` tag to be used from JavaScript without modification.

### JDK 1.1 compatibility

- Java Plug-in can now access the `sun.audio` package.
- Some of applets are compiled with compilers that don't generate proper class file format which conforms the class-file specification. Such appletss may result in a `ClassFormatError` when Java Plug-in attempts to load them. To help address this problem, `PluginClassLoader`



provides a bytecode scanner to transform bad class files to conforming on-the-fly to allow the applets to run. Currently, only the bad class file with the following errors may be transformed

- Local variable name has bad constant pool index
- Extra bytes at the end of the class file
- Code segment has wrong length
- Illegal Field/Method name
- Illegal field/method modifiers
- Invalid start\_pc/length in local var table

See also [Updating Old Class Files](#) above.

# Using OBJECT, EMBED and APPLETTags in Java Plug-in

This chapter includes the following topics:

- [Introduction](#)
- [Java™ Plug-in in IE on Windows](#)
- [Java Plug-in in Navigator™ on Windows or Solaris™ operating environments](#)
- [Java Plug-in in IE and Navigator](#)
- [Java Plug-in Anywhere](#)
- [Summary](#)

## Introduction

This document explains the tagging structure, involving OBJECT and EMBED tags, required by Java Plug-in. It is intended for web authors who want to manually insert Java Plug-in tags in their HTML pages.

### Notes

1. There is a Java Plug-in [HTML Converter](#), available free-of-charge from Sun Microsystems, that can automatically generate the tagging structure. It is highly recommended that most web authors use it.
2. This converter can convert pages into any of the formats described below.
3. For a complete list of JRE releases that can be autdownloaded via .cab files, as mentioned below, see [Autodownload Files \(Windows Only\)](#).

Applets are normally specified in an HTML file as follows:

```
<APPLET code="XYZApp.class" codebase="html/" align="baseline"
width="200" height="200">
<PARAM name="model" value="models/HyaluronicAcid.xyz">
  No Java 2 SDK, Standard Edition v 1.4 support for APPLETT!
</APPLET>
```

And normally the APPLETT tag specifies information about the applet, while the <PARAM> tags, located between the <APPLETT> and </APPLETT> tags, store per-instance applet information.

However, an APPLETT is rendered by the browser and there is no easy way to interrupt the browser and force it to use Sun's Java Runtime Environment (JRE) to run the applet. To force the browser to do so, however, you may use a special Java Plug-in tagging structure—involving the OBJECT or EMBED tag or both, as described below—in place of the usual APPLETT tag in an your HTML pages. This will cause the browser to launch Java Plug-in, which will then run the applet using Sun's JRE.

For various combinations of browsers and platforms, the following sections tell you exactly what you need to do

### Note on Version Numbers

*Product version numbers* are of the form:

n1.n2.n3\_n4n5

where n1.n2 is the *major version number*, the n3 is the *minor version number* (also referred to as the *maintenance version number*), and n4n5 is the *patch version number* (also referred to as the *update version number*).

The version numbers used in the examples below refer to the 1.4 major release with minor release number of 1 and, at times, hypothetical patch number of mn. Do not put hypothetical values into actual HTML for web pages.

## Java Plug-in in IE on Windows

To use Java Plug-in in IE on Windows, use the OBJECT tag. The following is an example of mapping an APPLETT tag to a Java Plug-in tag:

**Original APPLETT tag:**

```
<APPLET code="XYZApp.class" codebase="html/" align="baseline"
width="200" height="200">
<PARAM name="model" value="models/HyaluronicAcid.xyz">
No Java 2 SDK, Standard Edition v 1.4.1 support for APPLETT!
</APPLET>
```

## New OBJECT tag:

```
<OBJECT classid="clsid:CAFEEFAC-0014-0001-0000-ABCDEFEDCBA"
width="200" height="200" align="baseline"

codebase="http://java.sun.com/products/plugin/autodl/jinstall-1_4-windows-i586.cab#Version=1,4,1,mn">
<PARAM name="code" value="XYZApp.class">
<PARAM name="codebase" value="html/">
<PARAM name="type" value="application/x-java-applet;jpi-version=1.4.1">
<PARAM name="model" value="models/HyaluronicAcid.xyz">
<PARAM name="scriptable" value="true">
No Java 2 SDK, Standard Edition v 1.4.1 support for APPLETT!
</OBJECT>
```

## Note

The codebase value show above is hypothetical; do not use it in actual HTML. Currently the working codebase for autodownload is [http://java.sun.com/products/plugin/autodl/jinstall-1\\_4\\_0-win.cab](http://java.sun.com/products/plugin/autodl/jinstall-1_4_0-win.cab). It is for the 1.4.0 version of the product. The autodownload cab for 1.4.1 will not be available until FCS for 1.4.1. And note that the autodownload cab can change as patch releases are released. See below for more about the interpretation of [codebase](#).

Note that the OBJECT tag contains similar information to the APPLETT tag. It is sufficient to launch Java Plug-in in IE. The classid in the OBJECT tag is the class identifier for Java Plug-in itself. When IE renders this class identifier in the OBJECT tag, it will try to load Java Plug-in into the browser. See [classid attribute](#) below for information about versioning.

There are several attributes in the OBJECT tag, such as width, height and align, that are mapped directly from the corresponding attributes in the APPLETT tag. These contain formatting information that IE will use to position Java Plug-in. Since this information is mapped directly without changes, the position and appearance of the applets using Java Plug-in should be the same as those applets using the APPLETT tag.

Not all attributes in the APPLETT tag can be mapped to the OBJECT tag attributes. For example, the attributes code and codebase in the APPLETT tag are not mapped into the OBJECT tag attribute. Instead, the attribute code is mapped into the PARAM code because, according to the w3c.org HTML specification, the attribute code does not exist in the OBJECT tag. There are other attributes that do not correspond in the OBJECT tag attributes. These attributes, with one exception, should be mapped to PARAM tags.

## Note

Duplicate parameter names should never be used with the OBJECT tag in Java Plug-in.

The one exception is the codebase attribute. In the APPLETT tag, the codebase attribute represents the location from which to download additional class and jar files. However, in the OBJECT tag, the codebase attribute represents the location from which to download Java Plug-in when it is not found on the local machine. Because the codebase attribute has two different meanings in the APPLETT and OBJECT tags, the codebase attribute in the APPLETT tag is mapped into a PARAM codebase in the OBJECT tag to resolve the conflict.

In the above example, the code and codebase attributes in the APPLETT tag are mapped into the OBJECT tag parameters. The PARAM code identifies the applet, and its value should be the same as the code attribute in the APPLETT tag. The PARAM codebase identifies the codebase of the applet. Java Plug-in knows where to download the applet or JavaBeans component because it can read this information from the parameters. The parameter type is not mapped from the APPLETT tag, but it is required in the OBJECT tag. It identifies the type of the Java executable, such as an applet or a JavaBean, so that Java Plug-in knows how to initialize the Java executable. These three PARAM tags (code, codebase, and type) in the above example are specified for Java Plug-in. They do not exist in the PARAM of the original APPLETT tag. Note that the model parameter within the OBJECT tag is identical to the model parameter inside the APPLETT tag. Except for these first three parameters specified for Java Plug-in, the remainder of the parameters are the same as those inside the APPLETT tag.

A new addition for Java Plug-in 1.3 (and valid in 1.4.x) was the PARAM scriptable. This was added to improve performance of applets that do not require the use of JavaScript or VBScript. The value should be true if the applet requires scripting support and false if it does not. The value is false by default.

Please note that PARAM scriptable is not the same as the PARAM mayscript. mayscript provides support for communication from Java applets to JavaScript only, while scriptable allows communication from JavaScript to Java applets in Internet Explorer only.

The text "No Java 2 SDK, Standard Edition v 1.4.1 support for APPLETT!" in the APPLETT tag is mapped inside the <OBJECT> and </OBJECT> tags. Originally, this text is displayed only if the browser does not have Java support. By mapping it inside the OBJECT tag, this text will be displayed if the browser does not support the OBJECT tag.

The APPLETT-to-OBJECT tag attributes mapping is as follows:

| Attribute Name | APPLETT Tag Support | OBJECT Tag Support | Attribute Mapping to OBJECT Tag |
|----------------|---------------------|--------------------|---------------------------------|
| align          | X                   | X                  | OBJECT attribute align          |
| alt            | X                   |                    |                                 |

|           |   |   |   |
|-----------|---|---|---|
| archive   | X |   | PARAM element archive   |
| code      | X | X | PARAM element code  |
| codebase  | X | X | PARAM element codebase  |
| height    | X | X | OBJECT attribute height   |
| hspace    | X | X | OBJECT attribute hspace   |
| name      | X | X | OBJECT attribute name,<br>PARAM element name<br><br>This case is an exception. The name attribute in the original APPLELET element must be mapped both to the name attribute in the OBJECT element and to a nested PARAM. If the original applet were as follows:<br><br><APPLET name="clock"<br>...><br></APPLET><br><br>Then the mapping would be:<br><br><OBJECT name="clock"<br>...><br><PARAM name="name"<br>value="clock"><br></OBJECT> |
| object    | X |   | PARAM element object  |
| title     | X | X | OBJECT attribute title  |
| vspace    | X | X | OBJECT attribute vspace   |
| width     | X | X | OBJECT attribute width  |
| mayscript | X |   | PARAM element mayscript   |

Some OBJECT attributes and nested PARAM elements are special to the OBJECT tag. They are the following:

| Attribute/PARAM    | Meaning in OBJECT tag  |
|--------------------|--|
| Attribute classid  | <p>It should always have the same value for dynamic version support, i.e. clsid:8AD9C840-044E-11D1-B3E9-00805F499D93. For static version support it will have a unique value for the version, e.g., clsid:CAFEEFAC-0014-0001-0000-ABCDEFFEDCBA.</p> <p>For more information about static vs. dynamic versioning, see <a href="#">Encountering OBJECT, EMBED, and APPLELET Tags With Different Plug-in Versions and Browsers</a>.</p> <p><b>Note:</b> The example above uses static versioning.</p>   |
| Attribute codebase | <p>It should be a full URL pointing to a CAB file somewhere on the network. Its purpose is to allow downloading and installation of Plug-in if either no version of Plug-in present on a user's computer or if no appropriate version of Plug-in is installed. The URL should, by default, point to a page on the Java Software website.</p> <p><b>Examples:</b></p> <p>In the <a href="#">above example</a>, codebase is:</p> <pre>codebase="http://java.sun.com/products/plugin/autodl/jinstall-1_4-windows-i586.cab#Version=1,4,1,mn"</pre> <p>Note that this was for static versioning and that the version number appended to the CAB file indicates that 1.4.1_mn version of Plug-in needs to be installed on the user's computer, otherwise an attempt will be made to download and install it.</p> <p>For dynamic versioning, codebase would be as follows:</p> <pre>codebase="http://java.sun.com/products/plugin/autodl/jinstall-1_4-windows-i586.cab#Version=1,4,0,0"</pre> <p>The appended version number indicates that if the user has version 1.4.0 or higher of the Plug-in installed on the computer, no download will occur. In other words, if the installed Plug-in is in the 1.4 family (major version 1.4), then it will run. If the installed version is lower, however, then an attempt will be made to download and install a newer version via the CAB file.</p> |
| PARAM element type | <p>The value for parameter type should be as follows:</p> <pre>"application/x-java-<object_type&gt;;&lt;version_type&gt;=&lt;implementation_version&gt;"< pre=""> <p>where</p> <ul style="list-style-type: none"> <li>object_type is applet or bean;</li> <li>version_type is jpi-version for static versioning or version for dynamic versioning;</li> <li>implementation_version is Implementation-Version as defined in <a href="#">Appendix 6: Sun-Supported Specification-Version and Implemenation-Version Formats</a>.</li> </ul> <p>For more information about static vs. dynamic versioning, see <a href="#">Encountering OBJECT, EMBED, and APPLELET Tags With Different Plug-in Versions and Browsers</a>.</p> </object_type&gt;;&lt;version_type&gt;=&lt;implementation_version&gt;"<></pre>   |

**Examples:**

For an applet deployed for static versioning with Plug-in version 1.4.1 patch release 02, value for type would be as follows:

```
"application/x-java-applet;jpi-version=1.4.1_02"
```

For a bean deployed for dynamic versioning with Plug-in version 1.4.1 patch release 03, value for type would be:

```
"application/x-java-bean;version=1.4"
```

This is because for dynamic versioning only the major version number is checked; i.e., if the installed Plug-in is in the major version family, no download is required.

|                          |   |
|--------------------------|---|
| PARAM element codebase   | Specifies the base URL of the applet. The URL can be relative or absolute but it should be in the domain of the current document. This PARAM element is required only if the applet is not located in the same directory as the document. |
| PARAM element code       | Specifies the name of the Java applet or JavaBeans component. It cannot be used with PARAM element object nested inside the same OBJECT tag.  |
| PARAM element scriptable | Specifies whether the applet is scriptable from the HTML page using JavaScript or VBScript. The value can be either true or false. This attribute is new in Java Plug-in 1.4.   |
| PARAM element object     | Specifies the name of the serialized Java applet or JavaBeans component. It cannot be used with PARAM element code nested inside the same OBJECT tag. This attribute is optional.   |
| PARAM element archive    | Specifies the name of the Java archive. This attribute is optional.   |
| PARAM element mayscript  | Specifies whether the applet is allowed to access netscape.javascript.JSObject. The value can be either true or false. This attribute is optional.  |

If the original APPLET element uses attributes type, codebase, code, object or archive, and also has nested PARAM element attributes type, codebase, code, object or archive, there will be a problem with the direct mapping of the APPELET attributes to PARAM attributes, as there will then be PARAM elements with duplicate name attributes, e.g.,

If the original applet were as follows:

```
<APPLET codebase = "a/b/c ...">
<PARAM name="codebase" value="my.jar">
</APPLET>
```

Then the direct mapping would be:

```
<OBJECT ...>
<PARAM name="codebase" value="a/b/c">
<PARAM name="codebase" value="my.jar">
</OBJECT>
```

To avoid this problem, any APPLET attribute that has a corresponding PARAM element in the original applet should be mapped as shown below.

| Original attribute in the APPLET element | New PARAM attribute with OBJECT element |
|--|---|
| code                                     | java_code                               |
| codebase                                 | java_codebase                           |
| archive                                  | java_archive                            |
| object                                   | java_object                             |
| type                                     | java_type                               |

These new PARAM element attribute names should be used only when necessary. If both the new and original PARAM names exist in the same OBJECT tag, the values associated with the new PARAM element names are used by Java Plug-in to load the applet or JavaBean.

## Example

Suppose the original applet was as follows:

```
<APPLET codebase = "a/b/c ...">
<PARAM name="codebase" value="my.jar">
</APPLET>
```

Then the mapping should be as shown below:

```
<OBJECT ...>
<PARAM name="java_codebase" value="a/b/c">
<PARAM name="codebase" value="my.jar">
</OBJECT>
```

## Java Plug-in in Netscape Navigator on Windows or Solaris™ operating environments

To use Java Plug-in with Netscape Navigator 4.x on Windows or Solaris operating environments, you must use the EMBED tag. The following example shows the mapping of a traditional APPLET tag to a Java Plug-in EMBED tag:

### Original APPLET tag:

```
<APPLET code="XYZApp.class" codebase="html/" align="baseline"
width="200" height="200">
<PARAM name="model" value="models/HyaluronicAcid.xyz">
No Java 2 SDK, Standard Edition v 1.4.1 support for APPLELET!!
</APPLET>
```

## New EMBED tag:

```
<EMBED type="application/x-java-applet;jpi-version=1.4.1" width="200"
height="200" align="baseline" code="XYZApp.class"
codebase="html/" model="models/HyaluronicAcid.xyz"
pluginspage="http://java.sun.com/j2se/1.4.1/download.html">
<NOEMBED>
No Java 2 SDK, Standard Edition v 1.4.1 support for APPLELET!!
</NOEMBED>
</EMBED>
```

Note that the EMBED tag contains similar information to the APPLELET tag. It is sufficient to launch Java Plug-in in Navigator. The attribute `type` in the EMBED tag is used to identify the type of the Java executable, such as an applet or a bean. When Navigator renders this attribute in the EMBED tag, it will try to load Java Plug-in into the browser. Note that a version number is also appended. It is used for comparing the version installed on a user's computer and deciding if another version needs to be installed. The second table below describes this in greater detail.

In the above example, several attributes in the EMBED tag, such as `width`, `height` and `align`, map directly from the corresponding attributes in the APPLELET tag. These contain formatting information that Navigator uses to position Java Plug-in. Since this information is mapped directly without changes, the position and appearance of the applets using Java Plug-in should be the same as those applets using the APPLELET tag.

Unlike the OBJECT tag, all information must be stored inside the <EMBED> tag. No PARAM elements are used with the EMBED tag. Therefore, all APPLELET attributes and related PARAM elements must be mapped as attribute `name="value"` pairs inside the EMBED tag.

In the above example, the `code` and `codebase` attributes in the APPLELET tag are mapped into the EMBED tag attributes. Attribute `code` identifies the applet. Its value should be the same as the `code` attribute in the APPLELET tag. Attribute `codebase` identifies the codebase of the applet. Java Plug-in knows where to download the applet or JavaBeans component because it can read this information from the attributes. Also notice that the `model` attribute within the EMBED tag is mapped from the nested PARAM element `model` inside the APPLELET tag.

Like the `codebase` attribute in the OBJECT tag, attribute `pluginspage` in the EMBED tag is used by Navigator if Java Plug-in is not installed on the user's computer, or if the wrong version of Java Plug-in is installed. It should point to the Java Plug-in Download Page on the Java Software website; or, for intranet deployment, to a page on the intranet from which Java Plug-in may be downloaded.

The text "No Java 2 SDK, Standard Edition v 1.4.1 support for APPLELET!!" in the APPLELET tag is mapped inside the <NOEMBED> and </NOEMBED> tags. With the traditional APPLELET tag this text is displayed only if the browser does not have internal support for Java. By mapping it inside the NOEMBED tag, this text will be displayed if the browser does not support the EMBED tag, or if Navigator fails to start the Java Plug-in.

The APPLELET-to-EMBED tag attributes mapping is as follows:

| Attributes             | APPLET tag support | EMBED tag support | Attribute map to EMBED tag       |
|------------------------|--------------------|-------------------|----------------------------------|
| <code>align</code>     | X                  | X                 | Attribute <code>align</code>     |
| <code>alt</code>       | X                  | X                 | Attribute <code>alt</code>       |
| <code>archive</code>   | X                  |                   | Attribute <code>archive</code>   |
| <code>code</code>      | X                  |                   | Attribute <code>code</code>      |
| <code>codebase</code>  | X                  |                   | Attribute <code>codebase</code>  |
| <code>height</code>    | X                  | X                 | Attribute <code>height</code>    |
| <code>hspace</code>    | X                  | X                 | Attribute <code>hspace</code>    |
| <code>name</code>      | X                  | X                 | Attribute <code>name</code>      |
| <code>object</code>    | X                  |                   | Attribute <code>object</code>    |
| <code>title</code>     | X                  | X                 | Attribute <code>title</code>     |
| <code>vspace</code>    | X                  | X                 | Attribute <code>vspace</code>    |
| <code>width</code>     | X                  | X                 | Attribute <code>width</code>     |
| <code>mayscript</code> | X                  |                   | Attribute <code>mayscript</code> |

Some attributes are special to the EMBED tag. These attributes are:

| Attribute | Meaning in EMBED tag |
|-----------|----------------------|
|           |                      |

|                       |  |
|-----------------------|--|
| Attribute type        | <p>type should be as follows:</p> <pre>"application/x-java-&lt;object_type&gt;;&lt;version_type&gt;=&lt;implementation_version&gt;"</pre> <p>where</p> <ul style="list-style-type: none"> <li>object_type is applet or bean;</li> <li>version_type is jpi-version for static versioning or version for dynamic versioning;</li> <li>implementation_version is Implementation-Version as defined in <a href="#">Appendix 6: Sun-Supported Specification-Version and Implemenation-Version Formats</a>.</li> </ul> <p>For more information about static vs. dynamic versioning, see <a href="#">Encountering OBJECT, EMBED, and APPLET Tags With Different Plug-in Versions and Browsers</a>.</p> <p><b>Examples:</b></p> <p>For an applet deployed for static versioning with Plug-in version 1.4.1 patch release 02, type would be as follows:</p> <pre>"application/x-java-applet;jpi-version=1.4.1_02"</pre> <p>For a bean deployed for dynamic versioning with Plug-in version 1.4.1 patch release 03, type would be:</p> <pre>"application/x-java-bean;version=1.4"</pre> <p>This is because for dynamic versioning only the major version number is checked; i.e., if the installed Plug-in is in the major version family, no download is required).</p> |
| Attribute codebase    | Specifies the base URL of the applet or JavaBeans component. The URL can be relative or absolute but it should be in the domain of the current document. This attribute is required only if the applet is not located in the same directory as the document.   |
| Attribute code        | Specifies the name of the Java applet or JavaBeans component. It cannot be used with param object inside the same EMBED tag.   |
| Attribute object      | Specifies the name of the serialized Java applet or JavaBeans component. It cannot be used with the code attribute inside the same EMBED tag. This attribute is optional.  |
| Attribute archive     | Specifies the name of the Java archive. This attribute is optional.  |
| Attribute pluginspage | It should be a full URL pointing to an HTML page from which Java Plug-in can be downloaded. If there is no Plug-in installed on a user's computer, or if the version of the Plug-in is not the one specified via the the type attribute, then this page will be presented to the user.   |
| Attribute mayscript   | Specifies whether the applet is allowed to access netscape.javascript.JSObject. The value can be either true or false. This attribute is optional.   |

Similar to the case of the OBJECT tag, if the original APPLET tag has PARAM element type, codebase, code, object, or archive, mapping it to the EMBED tag attribute will cause a problem. To avoid this, Java Plug-in also supports the same new set of attribute names for the EMBED tag, as specified below:

Similar to the case of the OBJECT element, if the original APPLET element uses attributes type, codebase, code, object or archive, and also has nested PARAM element attributes type, codebase, code, object or archive, there will be a problem with the direct mapping of the APPELET attributes to EMBED attributes. To avoid this problem, Java Plug-in also supports the same new set of attribute names for the EMBED tag, as specified below:

| Original Attribute Names | New Attribute Names |
|--------------------------|---------------------|
| code                     | java_code           |
| codebase                 | java_codebase       |
| archive                  | java_archive        |
| object                   | java_object         |
| type                     | java_type           |

These new attribute names should be used only when necessary. If both new and original attribute names exist in the same EMBED tag, the value associated with the new attribute name is used by Java Plug-in to load the applet or bean.

## Java Plug-in in IE and Navigator

The new OBJECT element scheme described above allows a web page with an applet to use Java Plug-in with Internet Explorer, and the new EMBED element scheme allows a web page with an applet to use Java Plug-in with Netscape. However, changes are a web page will be browsed by both browsers. Hence, there needs to be a way to use the OBJECT scheme when a page is browser by Internet Explorer, and the EMBED scheme when the page is browsed by Netscape. Such a scheme is described below.

### Original APPLET element:

```
<APPLET code="XYZApp.class" codebase="html/" align="baseline"
width="200" height="200">
<PARAM NAME="model" VALUE="models/HyaluronicAcid.xyz">
No Java 2 SDK, Standard Editoin v 1.4 support for APPLET!!
</APPLET>
```

## New OBJECT element with nested EMBED element:

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
width="200" height="200" align="baseline"

codebase="http://java.sun.com/products/plugin/1.4/jinstall-14-win32.cab#Version=1,4,0,mn">
<PARAM NAME="code" VALUE="XYZApp.class">
<PARAM NAME="codebase" VALUE="html/">
<PARAM NAME="type" VALUE="application/x-java-applet;jpi-version=1.4">
<PARAM NAME="model" VALUE="models/HyaluronicAcid.xyz">
<PARAM NAME="scriptable" VALUE="true">
<COMMENT>
  <EMBED type="application/x-java-applet;jpi-version=1.4" width="200"
height="200" align="baseline" code="XYZApp.class"
codebase="html/" model="models/HyaluronicAcid.xyz"
pluginspage="http://java.sun.com/products/plugin/1.4/plugin-install.html">
  <NOEMBED>
    No Java 2 SDK, Standard Edition v 1.4 support for APPLETT!!
  </NOEMBED>
</EMBED>
</COMMENT>
</OBJECT>
```

Because IE understands the <OBJECT> tag, it will try to launch Java Plug-in. Notice that the <COMMENT> tag is a special HTML tag understood only by IE. IE ignores text between the <COMMENT> and </COMMENT> tags. In effect, the above tags actually become:

Note that the <COMMENT> element is unique to IE. IE ignores text between the <COMMENT> and </COMMENT> elements. For IE, then, the above structure of elements looks like this:

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
width="200" height="200" align="baseline"

codebase="http://java.sun.com/products/plugin/1.4/jinstall-14-win32.cab#Version=1,4,0,mn">
<PARAM NAME="code" VALUE="XYZApp.class">
<PARAM NAME="codebase" VALUE="html/">
<PARAM NAME="type" VALUE="application/x-java-applet;jpi-version=1.4">
<PARAM NAME="model" VALUE="models/HyaluronicAcid.xyz">
<PARAM NAME="scriptable" VALUE="true">
</OBJECT>
```

This is identical to the new OBJECT tag example described above.

Now because Navigator understands neither the OBJECT nor the COMMENT elements, Navigator 4 views the above structure as follows:

```
<EMBED type="application/x-java-applet;jpi-version=1.4" width="200"
height="200" align="baseline" code="XYZApp.class"
codebase="html/" model="models/HyaluronicAcid.xyz"
pluginspage="http://java.sun.com/products/plugin/1.4/plugin-install.html">
<NOEMBED>
  No Java 2 SDK, Standard Edition v 1.4 support for APPLETT!!
</NOEMBED>
</EMBED>
```

This is identical to the new EMBED element example described above.

Thus you can use the new element structure—OBJECT element with nested EMBED element—to invoke Java Plug-in when either IE or Netscape encounters a page with an applet. This is the recommended format to use.

## Java Plug-in Anywhere

### **Warning!**

The material in this section is out of date. It will be updated in the future. The description of the script below corresponds to the Extended option with the HtmlConverter. It is recommended that you do not use this option at this time. The Standard option is the recommended option.

In most environments, Internet or intranet, an HTML page is likely to be viewed on different browsers on different platforms. Ideally, then, you should activate Java Plug-in only on the right browser-platform combination. Otherwise, an applet should use the browser's default JVM. You can achieve this using the following Java Plug-in format:

### Original APPLET element:



```
<APPLET code="XYZApp.class" codebase="html/" align="baseline"
width="200" height="200">
<PARAM NAME="model" VALUE="models/HyaluronicAcid.xyz">
No Java 2 SDK, Standard Edition v 1.4 support for APPLETT!
</APPLET>
```

## New-style format:

The following is an example of an equivalent Java Plug-in format. (The example includes comments.)

```
<!-- The following code is specified at the beginning of the <BODY> tag. -->
<SCRIPT LANGUAGE="JavaScript"><!--
var _info = navigator.userAgent;
var _ns = false;
var _ie = (_info.indexOf("MSIE") > 0 && _info.indexOf("Win") > 0
&& _info.indexOf("Windows 3.1") < 0);
/--></SCRIPT>
<COMMENT><SCRIPT LANGUAGE="JavaScript1.1"><!--
var _ns = (navigator.appName.indexOf("Netscape") >= 0
&& ((_info.indexOf("Win") > 0 && _info.indexOf("Win16") < 0
&& java.lang.System.getProperty("os.version").indexOf("3.5") < 0)
|| _info.indexOf("Sun") > 0));
/--></SCRIPT></COMMENT>

<!-- The following code is repeated for each APPLETT tag -->
<SCRIPT LANGUAGE="JavaScript"><!--
if (_ie == true) document.writeln('
<OBJECT
classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
width="200" height="200" align="baseline"
codebase="http://java.sun.com/products/plugin/1.4/jinstall-14-win32.cab#Version=1,4,0,nn">
<NOEMBED><XMP>');
else if (_ns == true) document.writeln('
<EMBED
type="application/x-java-applet;jpi-version=1.4" width="200" height="200"
align="baseline" code="XYZApp.class" codebase="html/"
model="models/HyaluronicAcid.xyz"
pluginspage="http://java.sun.com/products/plugin/1.4/plugin-install.html">
<NOEMBED><XMP>');
/--></SCRIPT>
<APPLETT code="XYZApp.class" codebase="html/" align="baseline"
width="200" height="200">
</XMP>
<PARAM NAME="java_code" VALUE="XYZApp.class">
<PARAM NAME="java_codebase" VALUE="html/">
<PARAM NAME="java_type" VALUE="application/x-java-applet;jpi-version=1.4">
<PARAM NAME="model" VALUE="models/HyaluronicAcid.xyz">
<PARAM NAME="scriptable" VALUE="true">
No Java 2 SDK, Standard Edition v 1.4 support for APPLETT!
</APPLETT></NOEMBED></EMBED>
</OBJECT>

<!--
<APPLETT code="XYZApp.class" codebase="html/" align="baseline"
width="200" height="200">
<PARAM NAME="model" VALUE="models/HyaluronicAcid.xyz">
No Java 2 SDK, Standard Edition v 1.4 support for APPLETT!
</APPLETT>
-->
```

Although this tag seems complicated compared to the old APPLETT tag, it is not. Most of the Java Plug-in tag is the same regardless of the applet used. For the majority of cases, a webmaster can copy and paste the Java Plug-in tag.

The first block of the script extracts the browser and platform. You must determine the browser and platform on which the applet is running. You do this by using JavaScript™ to extract first the browser name, then the platform. This is done once per HTML document.

The second block of the script replaces the APPLETT tag. You must replace each APPLETT tag with a similar block of code. The script replaces the APPLETT tag with either an EMBED tag or OBJECT tag, depending on the browser. You use the OBJECT tag for IE and the EMBED tag for Netscape Navigator. Finally, the original APPLETT tag is included as a comment at the end. It is always a good idea to keep the original APPLETT tag in case you want to remove the Java Plug-in invocation.

The first JavaScript establishes the browser and the platform on which the browser is running. You must do this because, currently, Java Plug-in supports only Windows 95, Windows 98, Windows NT 4.0, and Solaris. Note that Windows NT 3.51 is the only Win32 platform that Java Plug-in does not support. Java Plug-in should be invoked only on the supported browser and platform. The script sets the variable `_ie` to true if the browser is Internet Explorer. It sets the variable `_ns` to true if the browser is Navigator. (Note that all variable names in the JavaScript start with `"_"`. This is done to avoid conflicting with other JavaScript variables in the same page.)

To detect the right browser, the JavaScript evaluates three strings that are within the JavaScript's Navigator object: `userAgent`, `appName`, and `appVersion`. These strings

contain information about the browser and the platform. By looking at some examples of the string `userAgent`, you can see how to evaluate `userAgent` and use it to determine the browser. The following are some examples of the `userAgent` string for different platforms as it appears in Internet Explorer.

| Platform and Browser             | JavaScript's Navigator .userAgent string            |
|----------------------------------|---|
| Windows 2000 w/IE 5.0            | "Mozilla/4.0 (compatible; MSIE 5.01; Window NT5.0)" |
| Windows NT 4.0 w/IE 4.0          | "Mozilla/4.0 (compatible; MSIE 4.0; Windows NT)"    |
| Windows NT 4.0 w/IE 3.02         | "Mozilla/2.0 (compatible; MSIE 3.02; Windows NT)"   |
| Windows 95 w/IE 4.0              | "Mozilla/4.0 (compatible; MSIE 4.0; Windows 95)"    |
| Windows 95 w/IE 3.02             | "Mozilla/2.0 (compatible; MSIE 3.02; Windows 95)"   |
| Windows NT 3.51 w/IE 4.0         | "Mozilla/4.0 (compatible; MSIE 4.0; Windows 3.1)"   |
| Windows 3.1 w/IE 4.0             | "Mozilla/4.0 (compatible; MSIE 4.0; Windows 3.1)"   |
| Windows NT 4.0 w/Navigator 4.04  | "Mozilla/4.04 [en] (WinNT; I)"                      |
| Windows NT 4.0 w/Navigator 3.04  | "Mozilla/3.04 (WinNT; I)"                           |
| Windows NT 3.51 w/Navigator 4.04 | "Mozilla/4.04 [en] (WinNT; I)"                      |
| Windows NT 3.51 w/Navigator 3.04 | "Mozilla/3.04 (WinNT; I)"                           |
| Windows 95 w/Navigator 4.03      | "Mozilla/4.03 [en] (Win95; I)"                      |
| Windows 95 w/Navigator 3.03      | "Mozilla/3.03 (Win95; I)"                           |
| Solaris 2.6 w/Navigator 4.02     | "Mozilla/4.02 [en] (X11; I; SunOS 5.6 sun4u)"       |

Note that in each case the substring "MSIE" is always in the `userAgent` string in Internet Explorer. Also, the `userAgent` string in IE under Windows 3.1 and Windows 3.51 would contain the substring "Windows 3.1" because IE in these platforms is 16-bit. While IE 4 is also available on Solaris and Mac, in these versions the `userAgent` string does not contain the substring "Win". In addition, IE on Windows CE does not support JavaScript. This can be summarized as follows:

| userAgent string / Browsers    | Windows 3.1 w/ IE 3/4 | Windows NT 3.51 w/ IE 3/4 | Windows 95 w/ IE 3/4 | Windows NT 4.0 w/ IE 3/4 | Windows CE w/ IE | Mac w/ IE | UNIX w/ IE | Other browsers on any platform |
|--------------------------------|-----------------------|---------------------------|----------------------|--------------------------|------------------|-----------|------------|--------------------------------|
| contains "MSIE"                | X                     | X                         | X                    | X                        |                  | X         | X          |                                |
| contains "Win"                 | X                     | X                         | X                    | X                        |                  |           |            |                                |
| does not contain "Windows 3.1" |                       |                           | X                    | X                        |                  | X         | X          | X                              |

The above table shows that only Windows 95 and Windows NT 4.0 with IE can pass the Java Plug-in browser and platform requirements. However, this logic makes no assumptions about future releases of IE or future releases of Windows with IE. As long as the `userAgent` string contains "MSIE" and "Win", the above code should work in future releases of IE on Win32.

The above logic summarizes into the following:

```
<SCRIPT LANGUAGE="JavaScript"><!--
  var _info = navigator.userAgent;
  var _ie = (_info.indexOf("MSIE") > 0 && _info.indexOf("Win") > 0
    && _info.indexOf("Windows 3.1") < 0);
  //--></SCRIPT>
```

It is harder to detect Navigator on the right platform. Using just JavaScript, there is no way to determine if the browser is running on the Windows NT 3.51 or Windows NT 4.0 operating platform. (Refer to the above table and examine the `userAgent` string. Notice that the `userAgent` strings in Windows NT 3.51 and Windows NT 4.0 operating platforms are the same in Navigator.) It is important to make this distinction because Java Plug-in supports only the Windows NT 4.0 operating platform. To run Java Plug-in on the right platform, you must use LiveConnect in Navigator to determine the OS version number. This can be summarized as follows:

| Testing logic / Browsers           | Windows 3.1 w/ NS 3/4 | Windows NT 3.51 w/ NS 3/4 | Windows 95 w/ NS 3/4 | Windows NT 4.0 w/ NS 3/4 | NS on Solaris | IE on Solaris | NS on other platform | IE on other platform | Other browsers on any platform |
|------------------------------------|-----------------------|---------------------------|----------------------|--------------------------|---------------|---------------|----------------------|----------------------|--------------------------------|
| appName contains "Netscape"        | X                     | X                         | X                    | X                        | X             |               | X                    |                      |                                |
| userAgent contains "Win"           | X                     | X                         | X                    | X                        |               |               |                      | X                    |                                |
| userAgent does not contain "Win16" |                       | X                         | X                    | X                        |               |               | X                    | X                    |                                |
| os.version does not contain 3.5    | X                     |                           | X                    | X                        |               |               | Depends on OS        |                      |                                |

|                                |  |  |  |  |   |   |  |  |  |  |
|--------------------------------|--|--|--|--|---|---|--|--|--|--|
| userAgent<br>contains<br>"Sun" |  |  |  |  | X | X |  |  |  |  |
|--------------------------------|--|--|--|--|---|---|--|--|--|--|

The above logic translates into the following code:

```
<SCRIPT LANGUAGE="JavaScript"><!--
    var _info = navigator.userAgent; var _ns = false;
//--></SCRIPT>
<COMMENT><SCRIPT LANGUAGE="JavaScript1.1"><!--
    var _ns = (navigator.appName.indexOf("Netscape") >= 0
        && ((_info.indexOf("Win") > 0 && _info.indexOf("Win16") < 0
            && java.lang.System.getProperty("os.version").indexOf("3.5") < 0)
            || _info.indexOf("Sun") > 0));
//--></SCRIPT></COMMENT>
```

Referring to the previous table, note that only Windows 95, Windows NT 4.0, and Solaris operating environments with Navigator pass all the tests. Because LiveConnect is used to get the OS version number and only Navigator supports LiveConnect, a JavaScript that accesses LiveConnect will not be understood by IE. To prevent this from causing a problem, you block out this piece of the script using the COMMENT tag since COMMENT is an IE-specific comment tag. The text between the COMMENT tag is ignored by IE but not by Navigator. In addition, you must specify the script language as JavaScript1.1 to block this out if the browser is Navigator 2.

At this point, the above logic for IE and Navigator summarizes to a script that should look as follows:

```
<!-- The following code is specified at the beginning of the <BODY> tag. -->
<SCRIPT LANGUAGE="JavaScript"><!--
    var _info = navigator.userAgent; var _ns = false;
    var _ie = (_info.indexOf("MSIE") > 0 && _info.indexOf("Win") > 0
        && _info.indexOf("Windows 3.1") < 0);
//--></SCRIPT>
<COMMENT><SCRIPT LANGUAGE="JavaScript1.1"><!--
    var _ns = (navigator.appName.indexOf("Netscape") >= 0
        && ((_info.indexOf("Win") > 0 && _info.indexOf("Win16") < 0
            && java.lang.System.getProperty("os.version").indexOf("3.5") < 0)
            || _info.indexOf("Sun") > 0));
//--></SCRIPT></COMMENT>
```

Remember that this block of JavaScript should be put at the top of the <BODY> of the HTML file. It is put at the top so that other JavaScripts can reference the variables \_ie and \_ns. This JavaScript is the same in all HTML files, and it is only needed once for each HTML body.

The second block of HTML tags are actually the corresponding OBJECT and EMBED tags that are mapped from the data in the APPLETTAG. Note that JavaScript outputs the OBJECT tag when the browser is IE running on the Windows 95, Windows 98 or Windows NT 4.0 operating environments. If the browser is Navigator 3/4 on Windows 95, Windows 98, Windows NT 4.0, or Solaris operating environments, then JavaScript also outputs the EMBED tag, though with a slightly different syntax. Recall that the mechanism for detecting the browser and the platform has been described in the above section. (Tags <!-- and --> are used for comments in HTML.)

**Original APPLETTAG tag:**

```
<APPLET code="XYZApp.class" codebase="html/" align="baseline"
width="200" height="200">
<PARAM NAME="model" VALUE="models/HyaluronicAcid.xyz">
    No Java 2 SDK, Standard Edition v 1.4 support for APPLETTAG!!
</APPLET>
```

**New style tag:**

```
<SCRIPT LANGUAGE="JavaScript"><!--
    if (_ie == true) document.writeln('<OBJECT
classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
width="200" height="200" align="baseline"
codebase="http://java.sun.com/products/plugin/1.4/jinstall-14-win32.cab#Version=1,4,0,mn">
<NOEMBED><XMP>');
    else if (_ns == true) document.writeln('<EMBED
type="application/x-java-applet;jpi-version=1.4" width="200" height="200"
align="baseline" code="XYZApp.class" codebase="html/"
model="models/HyaluronicAcid.xyz"
pluginspage="http://java.sun.com/products/plugin/1.4/plugin-install.html">
<NOEMBED><XMP>');
//--></SCRIPT>
<APPLET code="XYZApp.class" codebase="html/" align="baseline"
width="200" height="200"></XMP>
<PARAM NAME="java_code" VALUE="XYZApp.class">
<PARAM NAME="java_codebase" VALUE="html/">
<PARAM NAME="java_type" VALUE="application/x-java-applet;jpi-version=1.4">
<PARAM NAME="model"
VALUE="models/HyaluronicAcid.xyz">LUE="models/HyaluronicAcid.xyz">
<PARAM NAME="scriptable" VALUE="true">
    No Java 2 SDK, Standard Edition v 1.4 support for APPLETTAG!!
</APPLET></NOEMBED></EMBED></OBJECT>
```

Note that the original APPLET tag is also mapped in the new Java Plug-in tag. This is done because Java Plug-in is intended to be used only on supported platforms. Leaving the APPLET tag in the script ensures that browsers that do not support Java Plug-in, or browsers that do not support JavaScript can gracefully handle the applet using the default JVM. HotJava Browser, IE, and Navigator on non-Java Plug-in supported platforms, or browsers without JavaScript support, read the above tags as follows:

```
<APPLET code="XYZApp.class" codebase="html/" align="baseline"
width="200" height="200"></XMP>
<PARAM NAME="java_code" VALUE="XYZApp.class">
<PARAM NAME="java_codebase" VALUE="html/">
<PARAM NAME="java_type" VALUE="application/x-java-applet;jpi-version=1.4">
<PARAM NAME="model" VALUE="models/HyaluronicAcid.xyz">
<PARAM NAME="scriptable" VALUE="true">
    No Java 2 SDK, Standard Edition v 1.4 support for APPLET!!
</APPLET></NOEMBED></EMBED></OBJECT>
```

These browsers ignore the tags </XMP>, </OBJECT>, </EMBED>, and </NOEMBED> as well because there is no corresponding <XMP>, <OBJECT>, <EMBED>, and <NOEMBED> tags. Because Java Plug-in is targeted for features in the Java 2 SDK, Standard Edition v 1.4 or future releases, those browsers without full Java 2 SDK 1.4 support and who do not support Java Plug-in will display the message "No Java 2 SDK, Standard Edition v 1.4 support for APPLET".

Unlike the previous examples, the mapped PARAM names contain java\_code, java\_codebase, and java\_type instead of code, codebase, and type. This is necessary because specifying code and codebase in the <PARAM> inside the <APPLET> and </APPLET> tag causes problems in some browsers.

IE on Windows reads the tags as follows:

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
width="200" height="200" align="baseline"
codebase="http://java.sun.com/products/plugin/1.4/jinstall-14-win32.cab#Version=1,4,0,mn">
<NOEMBED><XMP>
<APPLET code="XYZApp.class" codebase="html/" align="baseline"
width="200" height="200"></XMP>
<PARAM NAME="java_code" VALUE="XYZApp.class">
<PARAM NAME="java_codebase" VALUE="html/">
<PARAM NAME="java_type" VALUE="application/x-java-applet;jpi-version=1.4">
<PARAM NAME="model" VALUE="models/HyaluronicAcid.xyz">
<PARAM NAME="scriptable" VALUE="true">
    No Java 2 SDK, Standard Edition v 1.4 support for APPLET!!
</APPLET></NOEMBED></EMBED>
</OBJECT>
```

Be careful when you use the <XMP> tag. Because IE renders the <OBJECT> tag, you must disable the <APPLET> tag. If not disabled, two applets will simultaneously show up in the browser—one applet will be running in Microsoft's JVM, and the other will be running in Sun's JVM using Java Plug-in. The <XMP> tag provides a solution. The <XMP> and </XMP> tags basically transform any HTML tag that occurs between them into a stream of static text. In the above example, the <XMP> and </XMP> tags cause the browser to treat the <APPLET> tag as static text instead of an HTML tag. Because the browser ignores any static text between the <OBJECT> tag and the <PARAM> tag, the above tags actually become:

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
width="200" height="200" align="baseline"
codebase="http://java.sun.com/products/plugin/1.4/jinstall-14-win32.cab#Version=1,4,0,mn">
<PARAM NAME="java_code" VALUE="XYZApp.class">
<PARAM NAME="java_codebase" VALUE="html/">
<PARAM NAME="java_type" VALUE="application/x-java-applet;jpi-version=1.4">
<PARAM NAME="model" VALUE="models/HyaluronicAcid.xyz">
<PARAM NAME="scriptable" VALUE="true">
    No Java 2 SDK, Standard Edition v 1.4 support for APPLET!!
</OBJECT>
```

This is identical to the OBJECT tag example outlined above. Note that the <OBJECT> tag ignores the <NOEMBED>, </NOEMBED>, and </EMBED> tags.

Navigator on Windows operating environments reads tags as follows:

```
<EMBED type="application/x-java-applet;jpi-version=1.4" width="200" height="200"
align="baseline" code="XYZApp.class" codebase="html/"
model="models/HyaluronicAcid.xyz"
pluginspage="http://java.sun.com/products/plugin/1.4/plugin-install.html">
<NOEMBED><XMP>
<APPLET code="XYZApp.class" codebase="html/" align="baseline"
width="200" height="200"></XMP>
<PARAM NAME="java_code" VALUE="XYZApp.class">
<PARAM NAME="java_codebase" VALUE="html/">
<PARAM NAME="java_type" VALUE="application/x-java-applet;jpi-version=1.4">
<PARAM NAME="model" VALUE="models/HyaluronicAcid.xyz">
<PARAM NAME="scriptable" VALUE="true">
    No Java 2 SDK, Standard Edition v 1.4 support for APPLET!!
</APPLET></NOEMBED></EMBED></OBJECT>
```

Note that the <XMP> tag is used again in the <EMBED> tag to also disable the <APPLET> tag. The <EMBED> tag ignores the <PARAM> and </OBJECT> tags as well. In effect, the above tags actually become:

```
<EMBED type="application/x-java-applet;jpi-version=1.4" width="200" height="200"
  align="baseline" code="XYZApp.class" codebase="html/"
  model="models/HyaluronicAcid.xyz"
  pluginspage="http://java.sun.com/products/plugin/1.4/plugin-install.html">
<NOEMBED>
  No Java 2 SDK, Standard Edition v 1.4 support for APPLETT!
</NOEMBED>
</EMBED>
```

This is identical to the EMBED tag example outlined above.

You can use the combined OBJECT-EMBED-JavaScript tag to activate Java Plug-in in the right browser on the right platform. This combined tag is complicated and it should be used only if your HTML page is browsed by users in a heterogeneous environment.

## Summary

This document describes the OBJECT tag and EMBED tag styles used by Java Plug-in. It focuses on the conversion from an APPLETT tag to the OBJECT and EMBED tags. Currently, HTML 4.0 suggests that the OBJECT tag is the best way to insert Java applets and JavaBeans components into a HTML page. This document will be updated in the near future should there be a need to convert the OBJECT tag to the Java Plug-in tag style. Information disclosed in this document is intended to assist ISVs for writing HTML migration tools and to assist webmasters with Java Plug-in migration. The tag style described in this document is subject to change in the future.

Note that the use of Java Plug-in is not limited to the tag styles described in this document. In fact, webmasters are encouraged to modify the tag style or mix the tag with JavaScript to fit their needs. As long as the described OBJECT tag is used in IE and EMBED tag is used in Navigator, there should be no problems running Java Plug-in. Currently, there are several conversion templates shipped with the Java Plug-in HTML converter. Webmasters may find one template better than others for their needs, and are encouraged to modify these templates themselves if necessary.

---

# Using the HTML Converter to Convert Applet Tags for Java Plug-in

---

The Java Plug-in HTML Converter is a utility for converting an HTML page (file) containing applets to a format for Java Plug-in. The conversion process is as follows:

First, HTML that is not part of an applet is transferred from the source file to a temporary file. Then, when an <APPLET> tag is reached, the converter parses the applet to the first </APPLET> tag (not contained in quotes) and merges the applet data with the template. (See [Details about templates](#) in the section called [More About HTML Converter](#) for more information.) If this completes without error, the original HTML file is moved to the backup folder and the temporary file is then renamed as the original file's name.

The converter effectively converts the files in place. Thus, once the converter runs, files are setup for Java Plug-in.

For detailed information about installing and using the HTML Converter, see [More About HTML Converter](#).

---

# Proxy Configuration

---

This section covers the following topics:

- [Introduction](#)
- [How Java Plug-in Obtains Proxy Information](#)
- [Direct Connection](#)
- [Manual Proxy Configuration](#)
- [Automatic Proxy Configuration](#)

## Introduction

For enterprise customers it is important to be able to set up secure computing environments within their companies, and proxy configuration is an essential part of doing that. Proxy configuration acts as a security barrier; it ensures that the proxy server monitors all traffic between the Internet and the intranet. This is normally an integral part of security enforcement in corporate firewalls within intranets. Enterprise customers who want to use Java Plug-in to deploy applets on their intranet web pages may also set up proxy support. This support is required for Java Plug-in to work in an intranet environment and can be set up through the Java Plug-in Control Panel.

The Control Panel provides three [proxy options](#):

- "Use browser settings"
- Manual configuration through the Protocol-Address-Port table
- "Automatic proxy configuration URL"

If "Use browser settings" is selected, then proxy information is entered entirely through the browser. For Internet Explorer, go to Tools>Internet Options ... and select the Connections tab and then LAN Settings ... ; for Netscape, go to Edit>Preferences ... and select Advanced under Category and then Proxies. How this works and the three types of connections that can be set up through the browser—Direct, Manual, and Automatic—are described in the following sections.

If you select manual configuration in the Control Panel, then you must enter in the table for each protocol the address and port for the proxy server. Note that you may select to exclude some hosts from requiring proxy servers by listing them in the field labeled "No proxy host".

If you select "Automatic proxy configuration URL", then you must enter the URL for the location of the JavaScript called `FindProxyForURL(URL url)` that returns the proxy server to be used for a URL. Support for this script is the same as described below under [Automatic Proxy Configuration](#).

# How Java Plug-in Obtains Proxy Information From the Browser

Because browsers on different platforms store proxy information differently, there is no generic mechanism to obtain proxy information. Here's how Java Plug-in obtains proxy information for three different browser-platform combinations:

**Internet Explorer on Win32:** IE stores proxy information in the same set of keys in the windows registry. Java Plug-in obtains this information directly.

**Netscape Navigator on Win32:** Navigator 4 stores proxy information in the user preference file on the local machine. Java Plug-in reads and parses the user preference file to obtain the Navigator 4 proxy information. Netscape 6 has an API for obtaining proxy information. `findProxyForURL(URL)` returns proxy configuration information for the URL passed to it.

**Netscape Navigator on Solaris and Linux:** Navigator stores proxy information in a file in the local machine. Java Plug-in reads and parses this file to obtain the proxy information. For Netscape 6 the process is the same as described in the previous section.

Java Plug-in obtains proxy information at startup time. If you change the proxy setting after Java Plug-in has started, you may force Java Plug-in to reload the proxy information from the browser through the [p option in the Java Console](#).

## Direct Connection

Direct connection does not use a proxy. For certain situations, such as when mobile users connect to the company through a modem, direct connection to the intranet environment is required, and proxies should not be used in these cases.

## Manual Proxy Configuration

Both Internet Explorer and Netscape Navigator support manual proxy configuration. Users can specify the proxy server and port for each protocol. Users can also specify one proxy server and port for all protocols. To minimize the workload of the proxy server, some sites might bypass the proxy server completely when a machine is connecting to another machine inside the intranet environment. To do this, network administrators and users can specify the proxy server bypass list in the browser's setting.

**Internet Explorer:** Java Plug-in recognizes and supports the proxy server and port setting associated with the protocol. IE supports various syntaxes in the proxy server bypass list, as follows:

- IP address/hostname only
- IP address/hostname with wildcard
- IP address/hostname with protocol

For example, if you specify "121.141.23.5;\*.eng;http://\*.com" in the proxy server bypass



list, then the browser bypasses the proxy whenever one of the following occurs:

- "121.141.23.5" is requested
- the URL hostname ends with ".eng"
- the URL protocol is http and the URL hostname ends with ".com"

Currently Java Plug-in supports the first two syntaxes in the proxy server bypass list in IE. IE also supports bypassing the proxy server for local (intranet) addresses without using the proxy server bypass list. Java Plug-in supports this option by bypassing the proxy server if the URL hostname is plain; i.e., the hostname contains no dot (.).

**Netscape Navigator:** Java Plug-in recognizes and supports the proxy server and port setting associated with the protocol. For example, if you specify ".eng, .sun.com" in the proxy server bypass list in Navigator, it bypasses the proxy whenever the URL hostname ends with ".eng" or ".sun.com". Java Plug-in fully supports this syntax in the proxy server bypass list in Navigator.

For more information about manual proxy configuration in your browser, consult the user guide for your browser.

## Automatic Proxy Configuration

Automatic proxy configuration is supported in the browser by setting a particular URL that contains a JavaScript file with .pac or .js extension. This file contains a function called `FindProxyForURL` that contains the logic to determine which proxy server to use when the browser receives a connection request. This function is written by the system administrator for the particular intranet environment. When the browser starts up, it recognizes the URL of the JavaScript file and downloads the file to the local machine using direct connection. Then whenever it needs to make a new connection, the browser executes the JavaScript function `FindProxyForURL` in the file to obtain the proxy information to set up the connection.

**Internet Explorer:** During startup, Java Plug-in downloads the JavaScript file to the local machine using direct connection. Then whenever it needs to make a new connection, it executes the `FindProxyForURL` function to obtain the proxy information using the JavaScript engine in IE.

**Netscape Navigator:** During startup, Java Plug-in downloads the JavaScript file to the local machine using direct connection. Then whenever it needs to make a new connection, it executes the `FindProxyForURL` function to obtain the proxy information by using the JavaScript engine in Navigator.

There are a number of JavaScript *helper* functions which can be called from the JavaScript function `FindProxyForURL`. Java Plug-in provides its own implementation of these functions to completely emulate the automatic proxy configuration. Regarding the implementation of these helper function, note the following:

- Function `dnsResolve` always returns an empty string if the host is not an IP address
- Function `isResolvable` always returns false if the host is not an IP address.
- Function `isInNet` always returns false if the host is not an IP address.

Note that executing the function `FindProxyForURL` always returns proxy information as a string. Java Plug-in extracts the setting in the following way :

- If "DIRECT" is in the string, Java Plug-in assumes direct connection.
- If "PROXY" is in the string, it uses the first proxy setting for the connection.
- If "SOCKS" is in the string, it uses the SOCKS v4 for the connection.
- Otherwise, the proxy information in the string is incorrect. In this cases, Java Plug-in assumes direct connection.

For more information about automatic proxy configuration in your browser, consult the user guide for your browser.

---

# Protocol Support

---

This chapter includes the following topics:

- [HTTP, FTP, and GOPHER](#)
- [HTTPS](#)
- [Socks](#)
- [NTLM Authentication](#)

## HTTP, FTP, and GOPHER

Java Plug-in supports HTTP, FTP, and GOPHER protocols, including built-in proxy configuration support.

## HTTPS

### Introduction

HTTPS is supported in Java Plug-in through Java Secure Socket Extension (JSEE), which provides a Java implementation of SSL and HTTPS for the Java platform.

### Error handling support

When accessing an HTTPS server, errors may occur. Java Plug-in has hooked into JSSE to provide the following types of error handling:

- *Hostname mismatch*: If the HTTPS server host name does not match the name on the server certificate, a warning dialog will appear.
- *Untrusted server certificate*: If the server certificate can not be verified during the SSL handshaking, a warning dialog will appear.
- *Untrusted client certificate*: In case client authentication is required by the server and the client certificate cannot be verified, a warning dialog will be appear.
- *Server authentication*: If the client accesses a protected directory on the HTTPS server, the users will be prompted for a username and password. **Note:** Only basic authentication is currently supported.

## Potential issues with HTTPS through JSSE

Although support of HTTPS through JSSE eliminates many browser-specific problems, there are several issues that developers should be aware of:

- *Untrusted server certificate:* When SSL handshaking takes place in establishing an HTTPS connection, the server certificate is verified against the root CA store in J2SE. However, J2SE supports fewer root CA certificates than does the browser. As a result, you may have problems with untrusted server certificates.
- *Client authentication:* An HTTPS server may require client authentication, in which case a local client certificate is sent to the server for authentication. In JSSE, client certificates are stored in a separate file and are independent of the browser. In order for client authentication to work, developers must import client certificates into JSSE through the keytool. See the [JSSE documentation](#) for more information.
- *Level of error handling:* Java Plug-in currently handles the types of error listed in the previous section. However, if there are additional types of error that Java Plug-in doesn't recognize, the Java applet code may break.

## Socks

Java Plug-in currently supports SOCKS version 4.

**Note:** For HTTP/HTTPS, a SOCKS proxy server may be used with a web proxy server to add caching. The behavior, however, may differ from that observed when running a similar configuration in a browser without Java Plug-in.

## NTLM Authentication

Java Plug-in supports NTLM authentication protocol for HTTP/HTTPS. When attempting to access a server requiring NTLM authentication, the user will be presented with a dialog requesting two items (fields) of information: username and password. If the user's domain is the same as the domain of the server to be accessed, only username and password need be entered. But when the domain of the user is different from that of the server, then the domain of the server must be entered as well. It should be entered in the username field with a separating backslash (\) as follows:

```
<domain>\<username>
```

password is entered as it normally is in the password field.

---

# Cookie Support

---

This section covers the following topics:

- [Introduction](#)
- [How Java Plug-in supports cookies](#)
- [Cookie policy support in Java Plug-in](#)

## Introduction

Cookies are a way of storing data on the client side. They have been used extensively for personalization of portal sites, user preference tracking, and logging into web sites. For enterprise customers using cookies in their web sites, cookie support in Java Plug-in facilitates deployment of client-side Java.

Cookie support allows a Java component to pass a cookie back to a web server if that cookie originated from the web server. This provides the server with information about the state of the client. Java Plug-in provides bidirectional cookie support.

## How Java Plug-in Supports Cookies

Java Plug-in provides cookie support through the browser API. Because browsers on different platforms implement the browser's API differently, cookie support in Java Plug-in varies according to platform.

When a browser makes an HTTP/HTTPS request through a URL connection, it normally checks the cookie cache and policy to determine if a cookie should be sent along with the HTTP/HTTPS request header. If so, the browser will read the cookie from the cache and append the cookie as part of the HTTP/HTTPS request header.

When a browser processes the HTTP/HTTPS respond header through a URL connection, it will check the header to see if any cookies should be set. The browser also checks the cookie policy to determine if the action is allowed. If so, it will extract the cookie from the HTTP/HTTPS respond header and write it into the cookie cache.

When an HTTP/HTTPS request is made using Java Plug-in, Java Plug-in consults the browser to determine if a cookie should be sent along. If so, the HTTP/HTTPS request will contain the cookie as part of the header. Otherwise, the HTTP/HTTPS request will be sent with no cookie attached.

When a cookie needs to be set from the HTTP/HTTPS respond header, Java Plug-in uses the browser API to do so.

Cookie support in Java Plug-in is triggered transparently when an HTTP/HTTPS connection needs to be

made.

## Netscape 4 Limitation

When using Java Plug-in with Netscape 4, cookie support works only if the codebase is the same or a subdirectory of the document base. See examples in the table below:

| Document Base            | Codebase                | Will It Work? |
|--------------------------|-------------------------|---------------|
| http://host.com/my/      | http://host.com/my/     | Yes           |
| http://host.com/my/      | http://host.com/my/page | Yes           |
| http://host.com/my/page/ | http://host.com/my/     | No            |

To ensure that cookie support in Java Plug-in always works as expected, the following is recommended:

- Use Internet Explorer or Netscape 6
- If you must use Netscape 4, avoid having your web server set cookies in an HTTP/HTTPS connection with an applet.

The above recommendations apply to an intranet environment, where deployment of browsers and web servers is controllable.

## Cookie Policy Support in Java Plug-in

Java Plug-in supports all cookie policies that are supported in both Internet Explorer and Netscape Navigator. Cookie policy can be configured in both browsers (see your browser guide for details). There are various options, including the following:

- Always accept cookies
- Disable all cookie use
- Prompt/Warn before accepting a cookie
- Only accept cookies from a server if they get sent back to that server

When cookie policy is changed in the browser, it will take effect the next time an HTTP/HTTPS connection is made via Java Plug-in.

Java Plug-in does not provide cookie-caching support. Instead, it consults the browser every time an HTTP/HTTPS connection is made. Any change to a cookie in the browser is reflected immediately in Java Plug-in when a new HTTP/HTTPS connection is made.

# Applet Caching

This section covers the following topics:

- [Caching Option](#)
  - [cache\\_archive](#)
  - [cache\\_version](#)
  - [cache\\_archive\\_ex](#)
  - [Cache Update Algorithm](#)
- [Security](#)
- [Known Issues](#)

## Caching Option

Once an applet is cached, it no longer needs to be downloaded when referenced again. Thus performance is improved.

This release introduces an alternative form of applet caching, allowing an applet deployer to decide if an applet should be *sticky*, i.e., placed in a disk cache created and controlled by Java Plug-in which the browser cannot overwrite. The only time a sticky applet gets downloaded after caching is when it is updated on the server; otherwise the applet is always available for fast loading. Applets providing core business applications should be made sticky to improve startup performance.

This new feature is activated by including the new `cache_archive`, `cache_version`, and `cache_archive_ex` values in the OBJECT/EMBED tag as described below.

### Note

.jar files specified with the `archive` attribute also get cached in this sticky cache.

### cache\_archive

The `cache_archive` attribute contains a list of the files to be cached:

```
<PARAM NAME="cache_archive" VALUE="a.jar,b.jar,c.jar">
```

Like the `archive` attribute in the APPLET tag, the list of .jar files in the `cache_archive` attribute do not contain the full URL, but are always downloaded from the codebase.

### cache\_version

The `cache_version` is an optional attribute. If used, it contains a list of file versions to be cached:

```
<PARAM NAME="cache_version" VALUE="1.2.0.1, 2.1.1.2, 1.1.2.7">
```

Each version number is in the form `xxxx.xxxx.xxxx.xxxx`, where `x` is a hexadecimal number. Each version number corresponds to a respective .jar file in the `cache_archive`.

### cache\_archive\_ex

In order to allow pre-loading of .jar files, the HTML parameter `cache_archive_ex` can be used. This parameter allows you to specify whether the .jar file needs to be pre-loaded; optionally the version of the .jar file can also be specified. VALUE for `cache_archive_ex` has the following format:

```
VALUE="<jar_file_name>;<preload(optional)>;<jar_file_version(optional)>,<jar_file_name>;<preload(optional)>;<jar_file_version(optional)>, ..."
```

The optional tags `preload` and `jar_file_version` can appear after the `jar_file_name` in any order separated by the delimiter ";". " ; " separates multiple entries.

The following shows how these tags might be used in an HTML page:

```
<OBJECT . . . . >
<PARAM NAME="archive" VALUE="a.jar">
<PARAM NAME="cache_archive" VALUE="b.jar, c.jar, d.jar">
<PARAM NAME="cache_version" VALUE="0.0.0.1, 0.0.2A.1,
0.3D.22.FFFE">
<PARAM NAME="cache_archive_ex" VALUE="applet.jar;preload,
util.jar;preload;0.9.0.AC1, tools.jar;0.9.8.7F">
</OBJECT>
```

In the above example, `a.jar` is specified in `archive`, whereas `b.jar`, `c.jar` and `d.jar` are specified in `cache_archive`. The versions are also specified for `b.jar`, `c.jar`, and `d.jar` as `0.0.0.1`, `0.0.2A.1`, and `0.3D.22.FFFE`, respectively. In `cache_archive_ex`, `applet.jar` is specified to be pre-loaded. `util.jar` is also specified to be pre-loaded but along with the version. For `tools.jar`, only version is specified.

Java Plug-In doesn't compare the versions if they are not specified for all the `.jar` files specified in HTML parameter `cache_archive`. If `cache_archive` is used without `cache_version`, the `.jar` files specified in `cache_archive` are treated no differently than the `.jar` files specified in HTML parameter `archive`. Similar treatment is given to `.jar` files specified in `cache_archive_ex` when `preload` and version options are not provided.

Class files and resources will be searched in the following order from the `.jar` files specified by the HTML parameters

1. `cache_archive_ex`
2. `cache_archive`
3. `archive`

## Applet Caching Update Algorithm

By default, without the `cache_version` attribute, applet caching will be updated if:

- The `cache_archive` has not been cached before, or
- The "Last-Modified" value of the `cache_archive` on the web server is newer than the one stored locally in the applet cache, or
- The "Content-Length" of the `cache_archive` on the web server is different from the one stored locally in the applet cache.

However, in some situations, the "Last-Modified" value returned from the web server through HTTP/HTTPS may not reflect the actual version of the applets. For example, if the web server crashes and all the files are restored, the `cache_archive` may have a different modification date on the server. Even if the `cache_archive` has not been updated, it will still force all the Java Plug-in clients to download the `cache_archive` again.

To strongly enforce the version update, it is recommended that the applet deployer use the `cache_version` attribute.

If `cache_version` is used, applet caching will be updated if the `cache_version` for the `cache_archive` is larger than the one stored locally in the applet cache. Note that the version number is used for triggering an update; there is no actual version number attached to the `.jar` file on the web server. In fact, unless version is used to trigger an update, it is possible the applet on the web server could be updated without the applet in `cache_archive`.

Using `cache_version` eliminates the need to connect to the web server to obtain "Modification-Date" and "Content-Length" of the `cache_archive`. In most cases this will speed up performance.

## Security

Although sticky applets are cached locally, they will still conform to the security policy defined by their original codebase and signer.

## Known Issues

- Caching of the `.jar` files specified in the manifest's `Class-Path` variable using Java Plug-in's cache is currently not supported.
- The path specified in the `cache_archive` must be a relative URL to the applet's codebase. Full URLs are not supported in `cache_archive`.



- If you get the "java.io.IOException: Caching not supported for ..." exception, this is because Java Plug-in was unable to get the expiration and last-modification dates for a given JAR file from the web server. If the Java Plug-in cannot get this information, there is no point in caching since the JAR file will be downloaded every time it is used. Even with this exception, however, the applet should work fine. This exception is merely informative.

---

# Using the Java Plug-in Control Panel to Set Plug-in Behavior/Options

---

This section covers the following topics:

- [Overview](#)
- [Starting the Java Plug-in Control Panel](#)
- [Saving Options](#)
- [Control Panel Help](#)
- [Setting Control Panel Options](#)
  - [Basic Panel](#)
  - [Advanced Panel](#)
  - [Browser Panel](#)
  - [Proxies Panel](#)
  - [Cache Panel](#)
  - [Certificates Panel](#)
  - [About Panel](#)

## Overview

The Java™ Plug-in Control Panel enables you to change the default settings used by the Java Plug-in at startup. All applets running in an active instance of Java Plug-in use these settings.

## Starting the Java Plug-in Control Panel

Start the Java Plug-in Control Panel as follows:

**On Windows:** From the Windows Control Panel, double-click the Java coffee-cup icon labeled Java Plug-in to launch the Java Plug-in Control Panel.

**On Solaris and Linux:** You can run the Control Panel by launching the ControlPanel executable file. In the Java 2 SDK, this file is located at:

```
<SDK installation directory>/jre/bin/ControlPanel
```

It is also located at:

```
<SDK installation directory>/bin/ControlPanel
```

For example, if your Java 2 SDK is installed at `/usr/j2se`, launch the Control Panel with this command:

```
/usr/j2se/jre/bin/ControlPanel
```

In a Java 2 Runtime Environment installation, the file is located at:

```
<JRE installation directory>/bin/ControlPanel
```

You can also use Netscape to visit the Control Panel applet page, which was installed inside the JRE directory as the file `ControlPanel.html`. In the Java 2 SDK this file is located at:

```
<SDK installation directory>/jre/ControlPanel.html
```

In the JRE:

```
<JRE installation directory>/ControlPanel.html
```

## Saving Options

When you have completed your changes to the Control Panel options, click **Apply** to save the changes. Click **Reset** to cancel your changes and reload the last values that were entered and applied. Note that this is not the same as the set of default values that were originally set for Java Plug-in when it was installed.

## Control Panel Help

Help for Java Plug-in is available from any panel in the Control Panel. Help comes up in its own window in response to pushing the **Help** button and looks like this:



# Setting Control Panel Options

There are eight sub-panels (tabs) in the Java Control Panel. From six of them you can set various options. The panels are labeled:

- [Basic](#)
- [Advanced](#)
- [Browser](#)
- [Proxies](#)
- [Cache](#)
- [Certificates](#)
- [Update Panel](#)
- [About](#)

Each is described separately below.

## Note

The settings shown for check boxes and radial buttons in each panel below, with the exception of the Browser panel, are the default (installation) settings. The check boxes for the Browser panel are determined according to user selection during installation.

## Basic Panel

The Basic panel looks like this:



Use the Basic panel to set the following options:

## Show Java Console

Displays the Java Console while running applets. The console displays messages printed by `System.out` and `System.err` objects. It is useful for debugging problems.

## Hide console

The console is running but hidden. This is the default setting (checked).

## Do not start console

The console is not started.

## Show Java in System Tray (Windows only)

The **Show Java in System Tray** option indicates to the user that a Java VM is running; it provides control over the Java Plug-in Control Panel and Java Console; and it provides information about the Java release. When enabled and Java Plug-in is started, the coffee-cup logo displays in the system tray. When Java Plug-in is shutdown, the icon is removed from the system tray. This option is enabled by default (checked).

### Note

If **Show Java in System Tray** is checked, the icon will show up in the system tray even if **Do not start console** is selected.

When Java Plug-in is first started and the mouse is pointed at the icon, a balloon tooltip will appear saying "Java 2 Standard Edition" along with a clickable link to <http://java.sun.com/getjava>. Thereafter, until Java Plug-in is shut down, when the mouse is pointed at the icon, a text box will say "Java 2 Standard Edition."

When the system tray icon is left-double-clicked, the Java Plug-in Control Panel will appear. If it is already open, a second Java Plug-in Control Panel will appear. When the system tray icon is right-clicked, a popup menu will appear with following menu items:

- Open Control Panel
- Open/Hide Console
- About Java Technology
- Hide Icon

**Open Control Panel** opens the Java Plug-in Control Panel. If it is already open, a second Java Plug-in Control Panel will appear.

**Open/Hide Console: Open Console** opens the Java Console if it is closed; **Hide Console** hides the Java Console if it is open.

**About Java Technology** brings up the **About - Java** box.

**Hide Icon** removes the icon from the system tray until the user re-enables it and Java Plug-in is restarted (i.e., the browser is closed, then reopened on a page with an applet).

### Note:

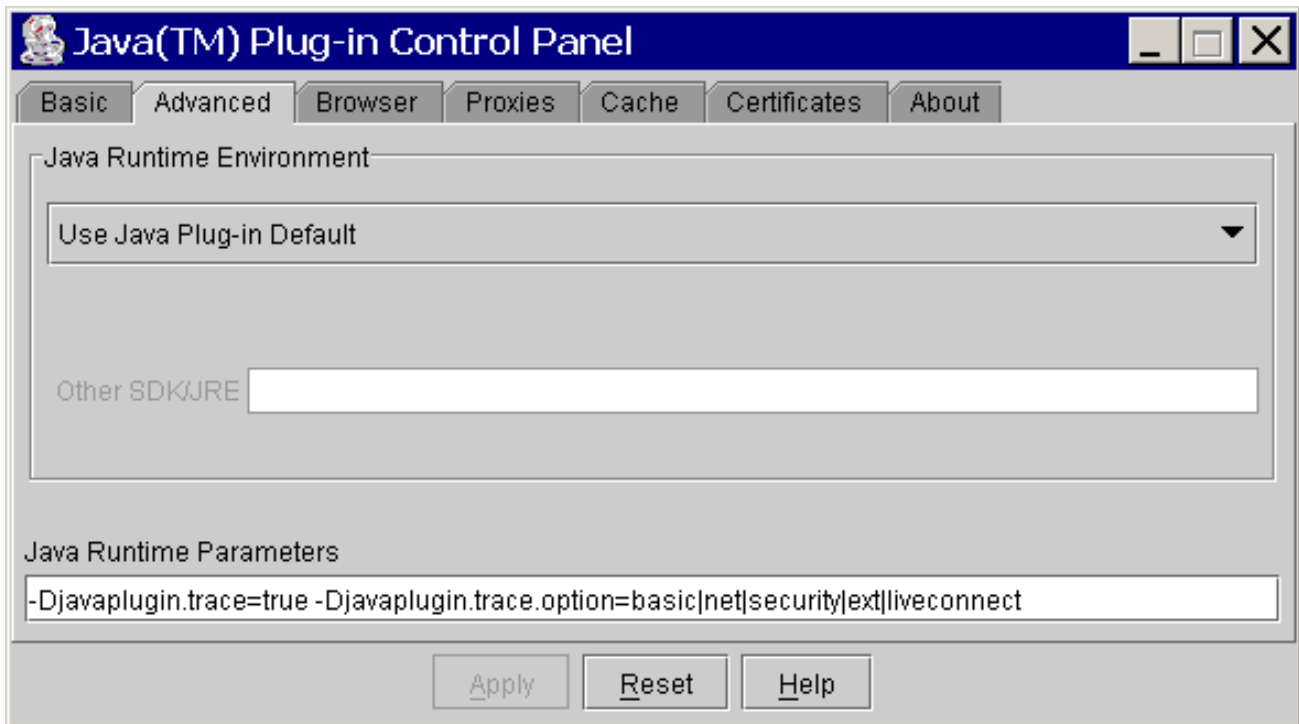
To re-enable the icon, start the Java Plug-in Control Panel, check **Show Java in System Tray**, and press **Apply**.

## Show Exception Dialog Box

Show an exception dialog box when exceptions occur. The default setting is hide the exception dialog box (unchecked).

# Advanced Panel

The Advanced panel looks like this:



Use the Advanced panel to set the following options:

## Java Runtime Environment

Enables Java Plug-in to run with any Java 2 JRE or SDK, Standard Edition v 1.4 installed on your machine. Java Plug-in 1.4 is delivered with a default JRE. However, you can override the default JRE and use an older or newer version in the 1.4 family. The Control Panel automatically detects all versions of the Java 2 SDK or JRE installed on your machine, and it displays in a list box all versions which you can use. The first item in the list will always be the Java Plug-in default; the last item will always say Other. If you choose Other, you must specify the path to the Java 2 JRE or SDK, Standard Edition v 1.4 that you wish to use. Only advanced users should change this option.

### Note

Only advanced users should change this option. Changing the default JRE is not recommended.

## Java Runtime Parameters

You can override the Java Plug-in default startup parameters by specifying custom options in the Java Runtime Parameters field. The syntax is the same as used with parameters to the java command line invocation. See the Java 2 Standard Edition (J2SE) documentation for a full list of command line options. The URL below is subject to change:

`http://java.sun.com/j2se/1.4/docs/tooldocs/<platform>/java.html`

where <platform> is one of the operating systems: solaris, linux, windows.

Below are some examples of Java runtime parameters

### *Enabling and disabling assertion support*

To enable assertion support, the following system property must be specified in the Java Runtime

Parameters:

```
-D[ enableassertions | ea ][:<package name>"..." | : <class name> ]
```

To disable assertion in the Java Plug-in, specify the following in the Java Runtime Parameters:

```
-D[ disableassertions | da ][:<package name>"..." | : <class name> ]
```

See [Assertion Facility](#) for more details on [enabling/disabling assertions](#).

Assertion is disabled in Java Plug-in code by default. Since the effect of assertion is determined during Java Plug-in startup, changing assertion settings in the Java Plug-in Control Panel will require a browser restart in order for the new settings to take effect.

Because Java code in Java Plug-in also has built-in assertion, it is possible to enable the assertion in Java Plug-in code through the following:

```
-D[ enableassertions | ea ]:sun.plugin
```

### ***Tracing and logging support***

Tracing is a facility to redirect any output in the Java Console to a trace file (`.plugin<version>.trace`).

```
-Djavaplugin.trace=true  
-Djavaplugin.trace.option=basic|net|security|ext|liveconnect
```

If you do not want to use the default trace file name:

```
-Djavaplugin.trace.filename=<tracefilename>
```

Similar to tracing, logging is a facility to redirect any output in the Java Console to a log file (`.plugin<version>.log`) using the Java Logging API. Logging can be turned on by enabling the property `javaplugin.logging`.

```
-Djavaplugin.logging=true
```

If you do not want to use the default log file name, enter:

```
-Djavaplugin.log.filename=<logfilefilename>
```

Furthermore, if you do not want to overwrite the trace and log files each session, you can set the property:

```
-Djavaplugin.outputfiles.overwrite=false.
```

If the property is set to `false`, then trace and log files will be uniquely named for each session. If the default trace and log file names are used, then the files would be named as follows

```
.plugin<username><date hash code>.trace  
.plugin<username><date hash code>.log
```

Tracing and logging set through the Control Panel will take effect when the Plug-in is launched, but changes made through the Control Panel while a Plug-in is running will have no effect until a restart.

For more information about tracing and logging, see the chapter called [Tracing and Logging](#).

### ***Debugging applets in Java Plug-in***

The following options are used when debugging applets in the Java Plug-in. For more information on this topic see the [Debugging Support](#) in the [Java Plug-in Developer Guide](#).

```
-Djava.compiler=NONE
```

```
-Xnoagent
-Xdebug
-Xrunjdw:transport=dt_shmem,address=<connect-address>,server=y,suspend=n
```

The `<connect-address>` can be any string (example: 2502) which is used by the Java Debugger (jdb) later to connect to the JVM

### ***Default connection timeout***

When a connection is made by an applet to a server and the server doesn't respond properly, the applet may appear to hang and may also cause the browser to hang because, since by default there is no network connection timeout.

To avoid this problem, Java Plug-in 1.4 has added a default network timeout value (2 minutes) for all HTTP connections. You can override this setting in the Java Runtime Parameters:

```
-Dsun.net.client.defaultConnectTimeout=<value in
milliseconds>
```

Another networking property that you can set is `sun.net.client.defaultReadTimeout`.

```
-Dsun.net.client.defaultReadTimeout=<value in milliseconds>
```

## **Note**

Java Plug-in does not set `sun.net.client.defaultReadTimeout` by default. If you want to set it, do so through the Java Runtime Parameters as shown above.

### **Networking properties description:**

```
sun.net.client.defaultConnectTimeout
sun.net.client.defaultReadTimeout
```

These properties specify, respectively, the default connect and read timeout values for the protocol handlers used by `java.net.URLConnection`. The default values set by the protocol handlers is `-1`, which means there is no timeout set.

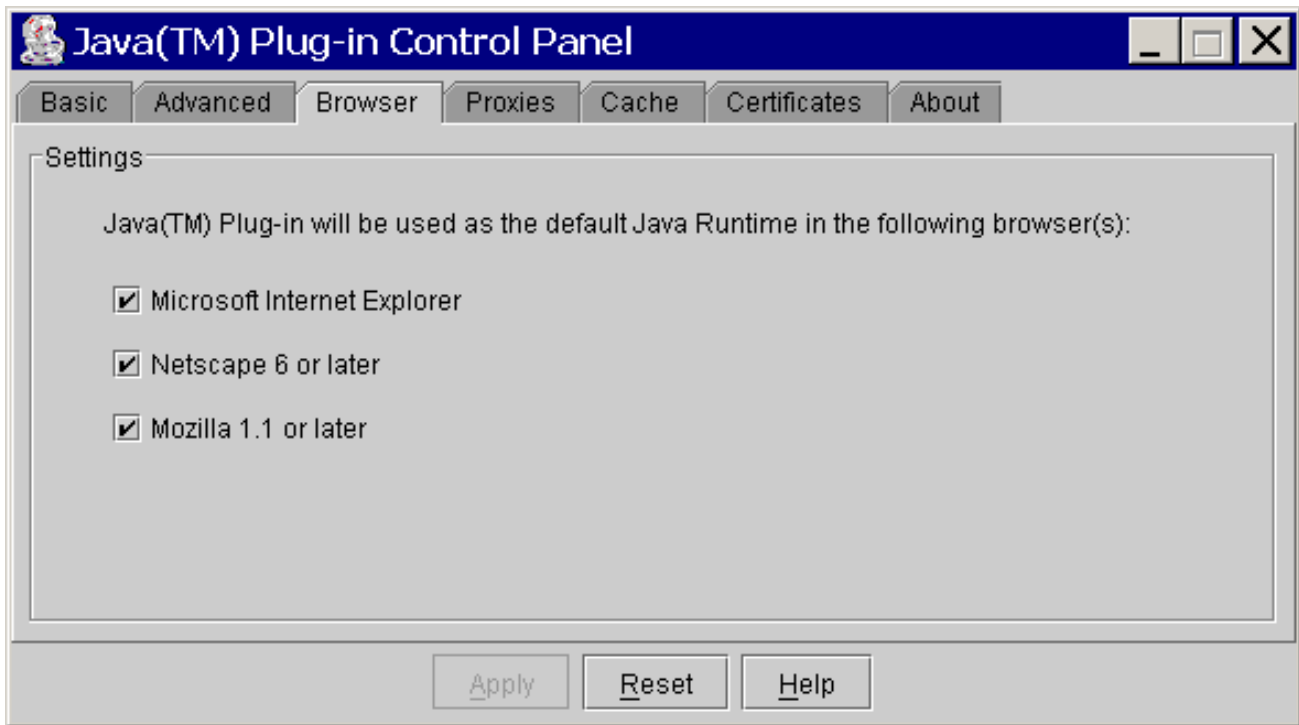
`sun.net.client.defaultConnectTimeout` specifies the timeout (in milliseconds) to establish the connection to the host. For example, for http connections it is the timeout when establishing the connection to the http server. For ftp connections it is the timeout when establishing the connection to ftp servers.

`sun.net.client.defaultReadTimeout` specifies the timeout (in milliseconds) when reading from an input stream when a connection is established to a resource.

For the official description of these properties, see [Networking Properties](#).

## **Browser Panel**

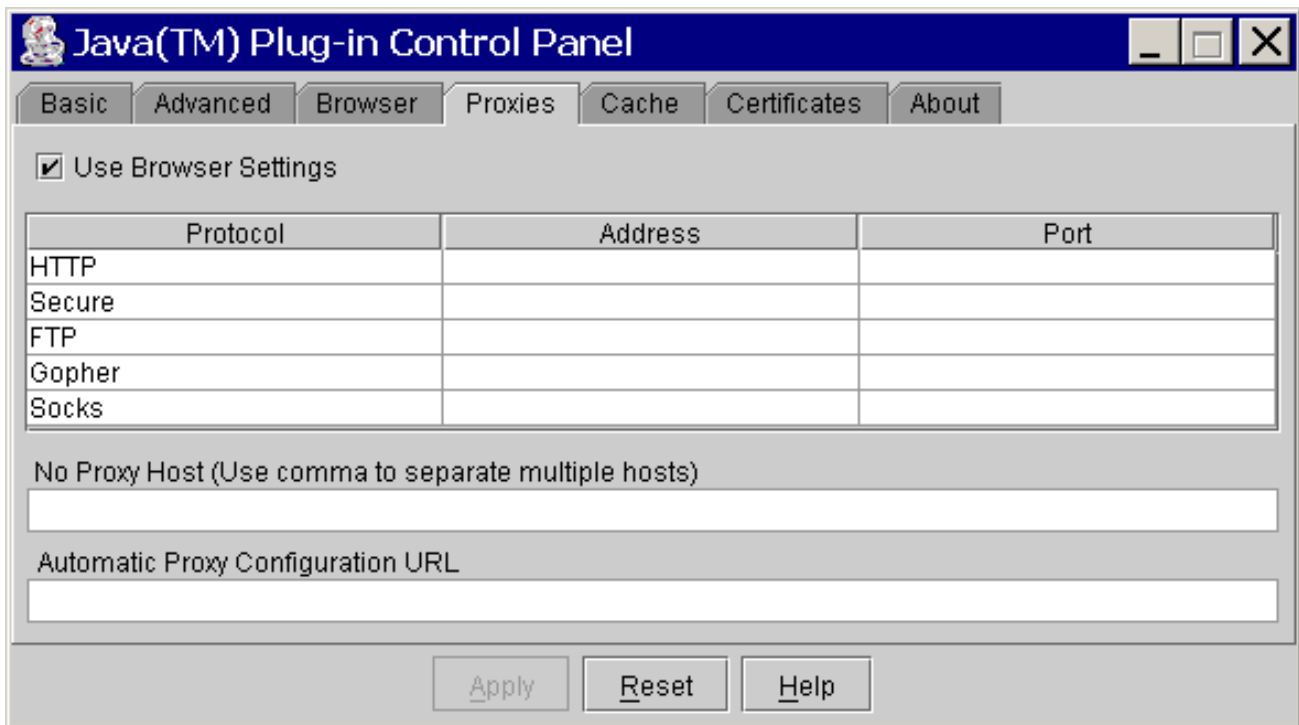




This panel relates only to Microsoft Windows installations; it does not appear in other installations. Check any browser for which you want Java Plug-in to be the default Java runtime, instead of the internal JVM of that browser. This is to enable APPLETTAG support in Internet Explorer and Netscape 6 via Java Plug-in.

## Proxies Panel

The Proxies panel looks like this:



Use the Proxies panel to use the browser default settings or to override the proxy address and port for the different protocols.

### Use browser settings

Check this to use the browser default proxy settings. This is the default setting (checked).

## Proxy information table

You can override the default settings by unchecking the "Use browser settings" check box, then completing the proxy information table beneath the check box. You can enter the proxy address and port for each of the supported protocols: HTTP, Secure (HTTPS), FTP, Gopher, and Socks.

## No proxy host

This is a host or list of hosts for which no proxy/proxies are to be used. No proxy host is usually used for an internal host in an intranet environment.

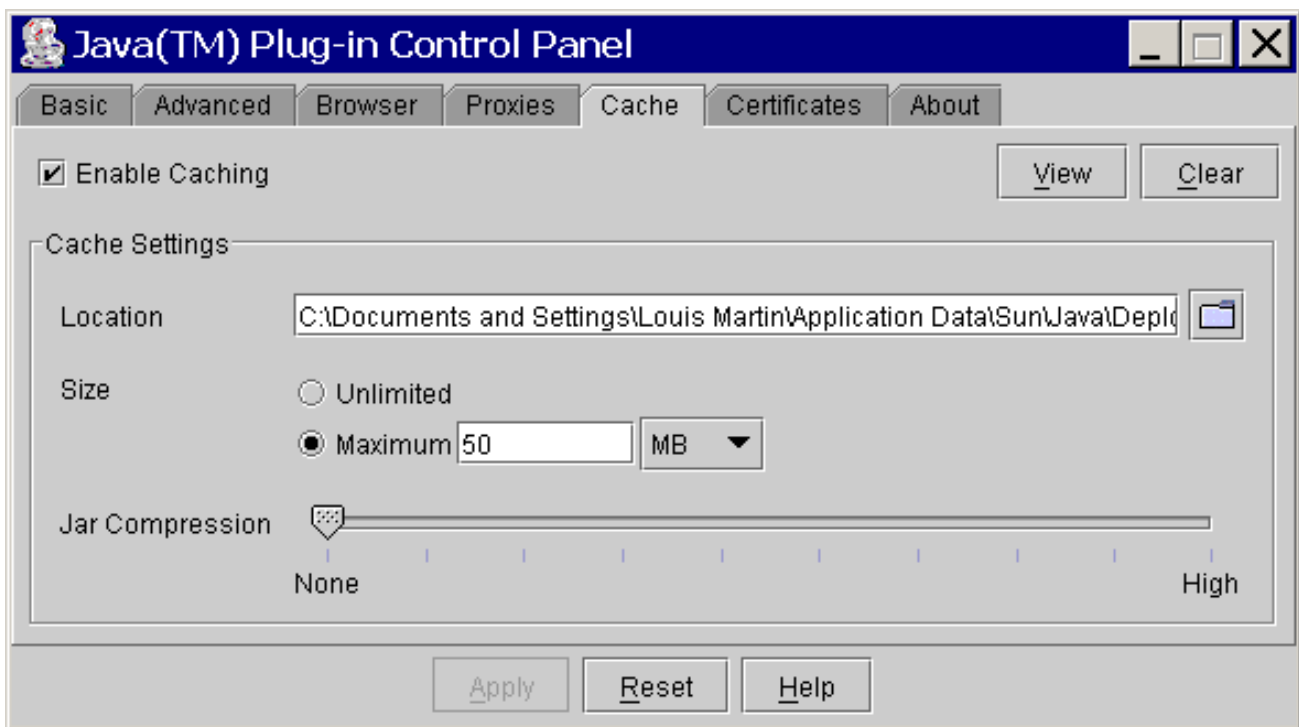
## Automatic proxy configuration URL

This is the URL for the JavaScript file (.js or .pac extension) that contains the `FindProxyForURL` function. `FindProxyForURL` has the logic to determine the proxy server to use for a connection request.

For additional details about proxy configuration, see the chapter called [Proxy Configuration](#).

## Cache Panel

The Cache panel looks like this:



### Note

The cache referred to here is the *sticky* cache; i.e., the disk cache created and controlled by Java Plug-in which the browser cannot overwrite. See [Applet Caching](#).

## Enable Caching

Check this to enable caching. This is the default setting (checked). With applet caching enabled, performance is improved because once an applet is cached it no longer needs to be downloaded when referenced again.

The Java Plug-in caches files of the following types downloaded via HTTP/HTTPS:

- . jar (jar file)

- .zip (zip file)
- .class (java class file)
- .au (audio file)
- .wav (audio file)
- .jpg (image file)
- .gif (image file)

## View files in Cache

Press this to view the cached files. Another dialog (Java Plug-in Cache Viewer) will pop up and display the cached files. The Cache Viewer displays the following information about the files in cache: Name, Type, Size, Expire Date, Last Modified, Version, and URL. In the Cache Viewer you can also selectively delete files in the cache. This is an alternative to the Clear Cache option described below, which deletes all files in the cache.

## Clear Cache

Press this to clear all files in the cache. You will be prompted (*Erase all files in ... \_cache?*) before the files are removed.

## Location

You can use this to specify the location of the cache. The default location of the cache is `<user home>/ . jpi_cache`, where `<user home>` is the value of the system property `user . home`. Its value depends on the OS.

## Size

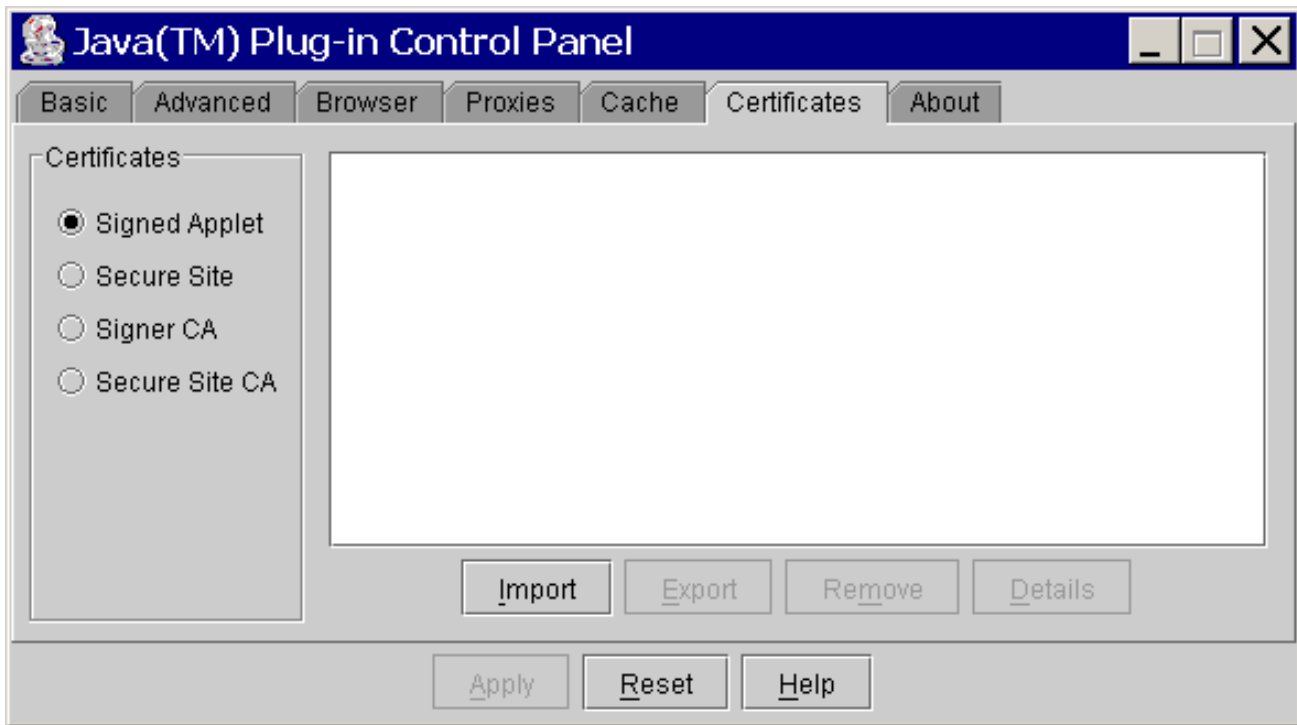
You can check Unlimited to make the cache unlimited in size; or you can set the Maximum size of the cache. If the cache size exceeds the specified limit, the oldest files cached will be removed until the cache size is within the limit.

## Compression

You can set the compression of the JAR cache files between None and High. While you will save memory by specifying higher compression, performance will be degraded; best performance will be achieved with no compression.

## Certificates Panel

The Certificates panel looks like this:



Four types of certificates may be selected:

- [Signed applet](#)
- [Secure site](#)
- [Signer CA](#)
- [Secure site CA](#)

### Signed applet

These are certificates for signed applets that are trusted by the user. The certificates that appear in the signed applet list are read from the certificate file `jplicerts<version>` located in the `<user home>/ . java` directory.

### Secure site

These are certificates for secure sites. The certificates that appear in the Secure site list are read from the certificate file `jphttpscerts<version>` located in the `<user home>/ . java` directory.

### Signer CA

These are certificates of Certificate Authorities (CAs) for signed applets; they are the ones who issue the certificates to the signers of signed applets. The certificates that appear in the Signer CA list are read from the certificate file `cacerts`, located in the `<jre>/lib/security` directory.

### Secure site CA

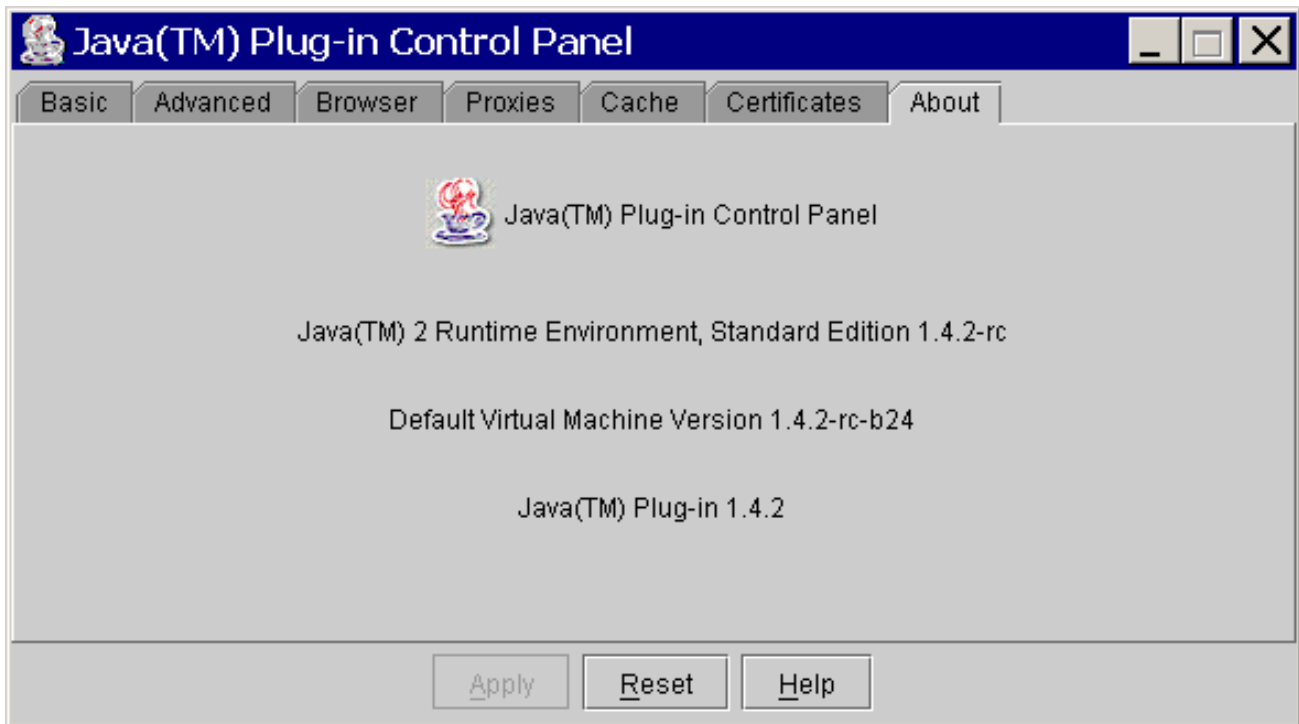
These are certificates of Certificate Authorities (CAs) for secure sites; they are the ones who issue the certificates for secure sites. The certificates that appear in the Secure site CA list are read from the certificate file `jssecacerts`, located in the `<jre>/lib/security` directory.

For **Signed applet** and **Secure site** certificates, there are four options: **Import**, **Export**, **Remove**, and **Detail**. The user can import, export, remove and view the details of a certificate.

For **Signer CA** and **Secure site CA**, there is only one option: **Detail**. The user can only view the details of a certificate.

# About Panel

The About panel looks like this:



---

# Installation for Conventional Applets (Microsoft Window Only)

---

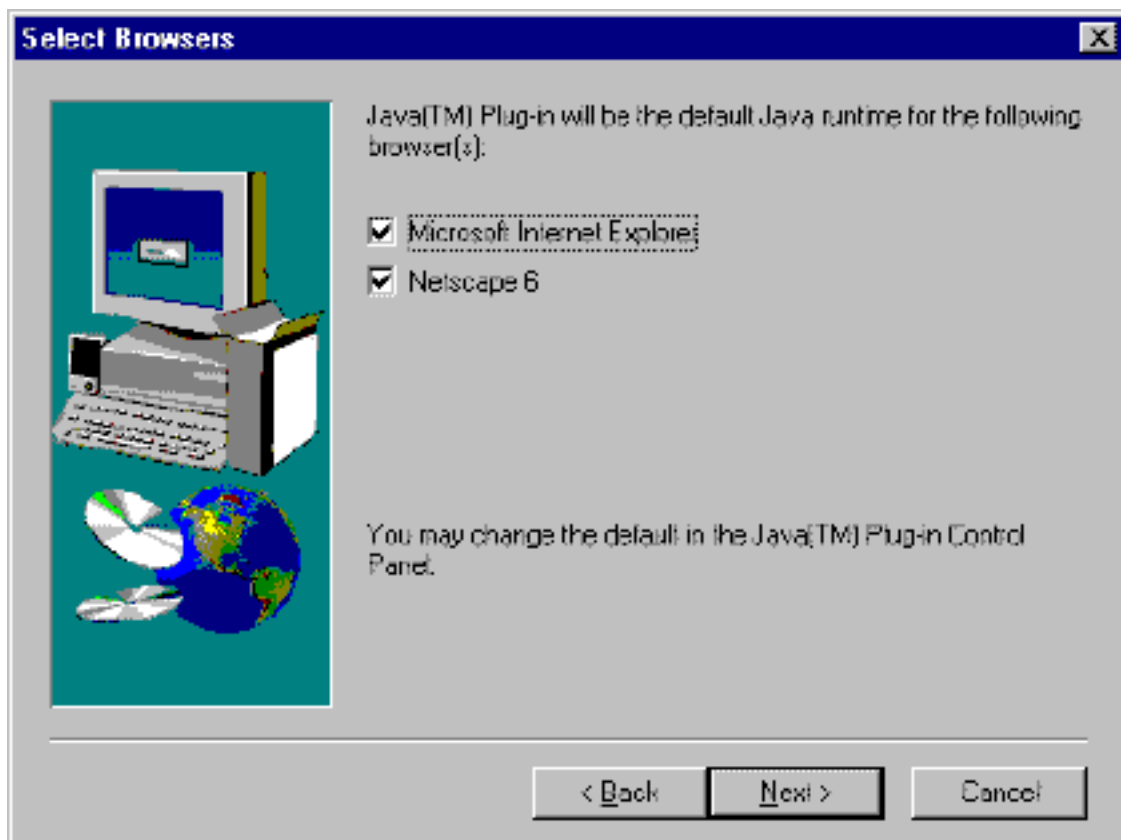
This section covers the following topics:

- [Enabling APPLETTAG Support](#)
- [Silent Installation Options](#)

## Enabling APPLETTAG Support

Developers should be aware that end users of Java Plug-in can choose to enable or disable APPLETTAG support during installation or at any time after installation.

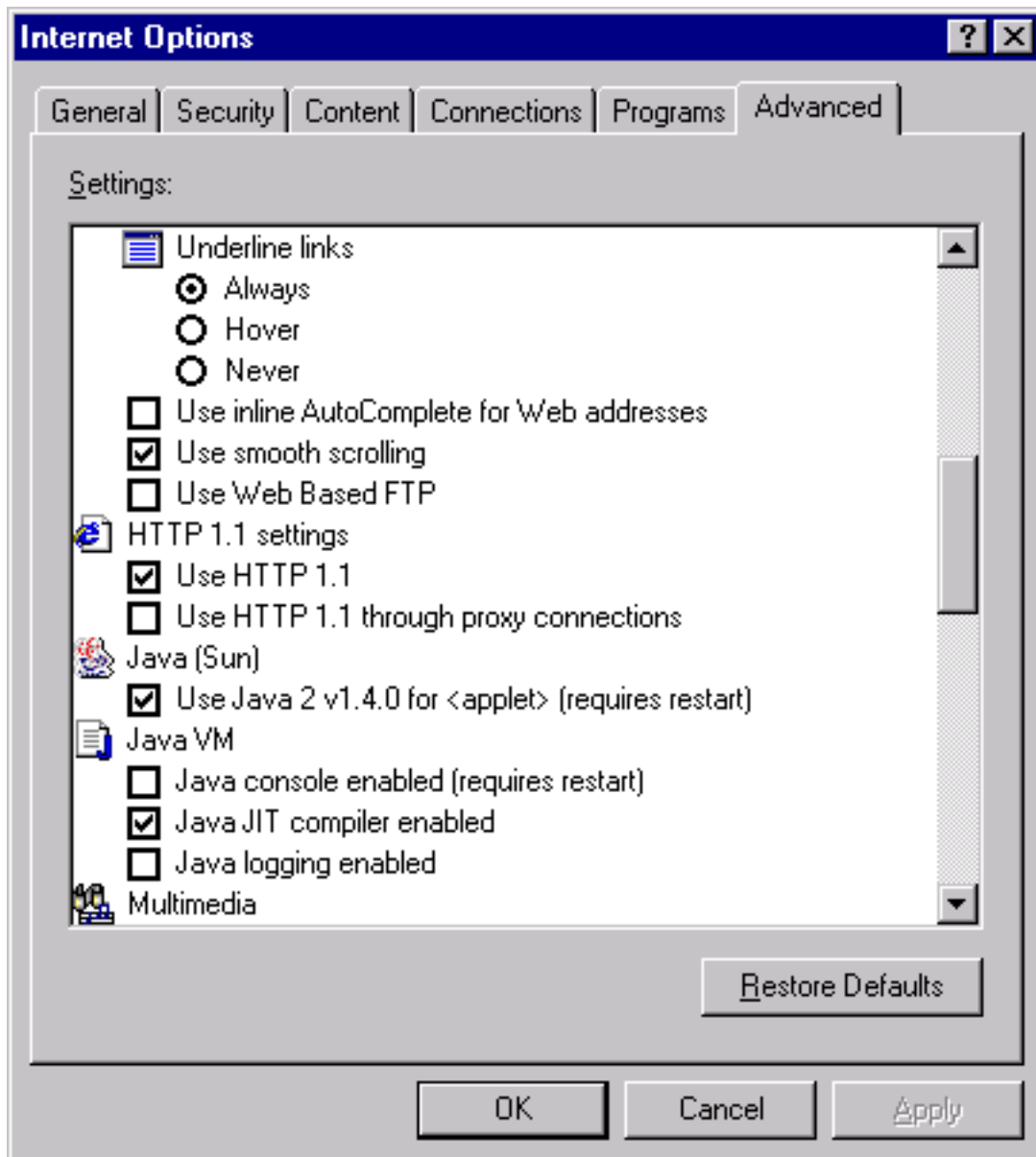
As part of the installation process, the user will see the following panel:



This panel allows the user to select JRE/Java Plug-in to be the default runtime environment for handling APPLETTAG tags in Microsoft Internet Explorer browsers and/or Netscape 6 browsers. The installer will use the Windows registry to determine whether the user's machine has these browsers installed and will check by default the box corresponding to any installed browser(s).

The user can enable or disable APPLET tag support after installation through the [Browser panel](#) of the [Java Plug-in Control Panel](#).

On Microsoft Internet Explorer browsers, users also can use the Advanced tab in the Internet Options window to enable/disable Java Plug-in's APPLET tag support. (Tool>Internet Options ... and select Advanced tab.)



## Silent Installation Options

Java Plug-in supports silent installation as an alternative to the standard installation wizard. Silent installation might be used, for example, for large-scale deployment. For general information on silent installation of Java Plug-in, see [Silent Installation](#).

Unlike the installation using the wizard, silent installation will *not* enable APPLET tag support by default for any browsers on the end user's machine. To enable APPLET tag support during silent installation, you must use one or both of the following command-line options:

**-iexplorer**

If this flag is passed to the installer, APPLETTAG support will be enabled for Microsoft Internet Explorer browsers.

**-netscape6**

If this flag is passed to the installer, APPLETTAG support will be enabled for Netscape 6 browsers.



---

# Intranet With OBJECT / EMBED Tag

---

This section includes the following topics:

- [Introduction](#)
- [Deploying Java Plug-in with Internet Explorer](#)
- [Deploying Java Plug-in with Navigator](#)

## Introduction

The intranet is a more controlled environment than the internet, and you can decide whether you want to deploy your applets via the APPLET, OBJECT, or EMBED tags. This chapter deals with deployment using the OBJECT or EMBED tags.

Java Plug-in can be deployed in an intranet environment so that users can download and install it without leaving the firewall. Deployment details for Internet Explorer and Netscape Navigator are described below.

### Note

You need to understand the Java Plug-in tagging structure to understand the discussion below. The tagging structure is explained in [Using OBJECT, EMBED and APPLET Tags in Java Plug-in.](#))

## Deploying Java Plug-in with Internet Explorer (IE)

To deploy Java Plug-in in intranet environments with IE, you need to download and store the Java Plug-in binary file on a web server. Then you need to modify the codebase attribute inside the <OBJECT> tag in the HTML file to point to the binary. For example, if you want to use the Java Plug-in binary file in `http://my_company.com/plugin/`, then you should specify the codebase attribute as

```
"http://my_company.com/plugin/jre-1_4-win.exe#Version=1,4,0,0"
```

(For Internet deployment the above might use a .cab file rather than .exe. The .cab file points to an .exe and provides a download progress bar.)

Note the version number in the above URL. For Java Plug-in 1.4, the version number is 1,4,0,0. This version number changes for each release, and you must change the version number inside the

<OBJECT> tag if you want to specify the newest release of Java Plug-in.

The file `jre-1_4-win.exe#Version=1,4,0,0` is the same binary that can be download directly from the Java Software web site. It is digitally signed with a VeriSign digital signature signed by Sun Microsystems, Inc. If you are using the Java Plug-in HTML Converter, you can simply change the conversion template file to specify the latest Plug-in version (the `CabFileLocation` variable) before the conversion. For information about modifying the conversion template, consult the README file in the Java Plug-in HTML Converter. Note that `CabFileLocation` can be the location of the `.cab` file or the `.exe`.

Once you have implemented these changes, when a machine that does not have Java Plug-in installed browses an HTML page with IE, IE will look into your predefined location (`http://my_company.com/plugin` in the example) and download Java Plug-in. Notice that there is no need to download Java Plug-in over the firewall if the web server is inside the intranet.

## **Note: Recommendations**

The following is recommended for long-term reliability:

- For the `codebase` attribute in the OBJECT tag, it is recommended that you use the `.exe` version, rather than the `.cab` version, on `java.sun.com`. That `.cab` version on `java.sun.com` may point to an updated version that you do not want to use for your deployment.
- It is also recommend that you create your own download site for the versions you need so that you do not rely on `java.sun.com`. Downloads on `java.sun.com` frequently change.

## **Deploying Java Plug-in with Navigator**

For Netscape 6 you must use the APPLET tag.

To deploy Java Plug-in in intranet environments with Navigator 4, you need to download and store the Java Plug-in binary file on one of your web servers. Then you need to set up a Java Plug-in Download page and modify the `pluginspage` attribute in the EMBED tag to refer to this page. For example, if you have set up the Java Plug-in Download page at "`http://my_company.com/plugin/`" and the page is called `plugin-install.html`, you can specify the `pluginspage` as "`http://my_company.com/plugin/plugin-install.html`". The Download page should have options to download different versions of Java Plug-in, such as for Windows and Solaris.

If you are using the Java Plug-in HTML Converter, you can simply change the conversion template file to specify your download page (the `NSFileLocation` variable) before the conversion. For information about modifying the conversion template, see [Details About Templates](#) in the chapter called [More About HTML Converter](#) or consult the Help file in the HTML Converter itself (Help>Help).

Once you have implemented these changes, when a machine that does not have Java Plug-in installed browses an HTML page with Navigator, users will see a missing-plugin icon on the HTML page. When the user clicks on this picture, Navigator directs the user to your predefined Java Plug-in Download Page

([http://my\\_company.com/plugin/plugin-install.html](http://my_company.com/plugin/plugin-install.html) in this example). Notice that there is no need to download Java Plug-in over the firewall if the web server is inside the intranet.

## Deploying Java Plug-in with Unix/Linux with Navigator

There are two ways that Java Plug-in might be installed in the Intranet environment with Unix or Linux:

- It might be installed on a per-user-account basis on individual machines, as it is installed in the Window environment.
- It might be installed by a system administrator on a shared NSF drive for shared installation.

In either case, for Netscape 4 the environmental variable `NPX_PLUGIN_PATH` must be set to point to the Java Plug-in. When Java Plug-in is installed on the local machine, then the user needs to set this variable to point to the Java Plug-in in the JRE on the local machine. For a shared installation, the system administrator needs to determine the shared location of the Java Plug-in in the shared JRE on the NSF drive and set the user profile for every machine to point to this via `NPX_PLUGIN_PATH`.

---

# Internet Deployment

---

## Deploying Java Plug-in with Internet Explorer

With Internet Explorer it is recommended that you use the `APPLET` tag for internet deployment. However, you could use the `OBJECT` tag in certain cases; e.g., if you want users to use a particular version of Java Plug-in. In the case that the you chose to use the `OBJECT` tag, the following is recommended.

### Recommendations

The following is recommended for long-term reliability:

- For the `codebase` attribute in the `OBJECT` tag, it is recommended that you use the `.exe` version, rather than the `.cab` version, on `java.sun.com`. That `.cab` version on `java.sun.com` may point to an updated version that you do not want to use for your deployment.
- It is also recommend that you create your own download site for the versions you need so that you do not rely on `java.sun.com`. Downloads on `java.sun.com` frequently change.

These are the same recommendations as for the intranet environment.

## Deploying Java Plug-in with Netscape

With Netscape it is recommended that you use the `APPLET` tag for internet deployment.

---

# Silent Installation

---

This section includes the following topics:

- [Introduction](#)
- [Running InstallShield in Silent Mode: Installation](#)
- [Running InstallShield in Silent Mode: Uninstalling](#)
- [Creating a Log File](#)

## Notes

1. This chapter applies to 32-bit Windows and silent installation of the Java Runtime Environment (JRE).
2. It is intended for:
  - system administrators who want to deploy Java Plug-in and the JRE on multiple PCs in the Intranet environment without user interaction;
  - companies with products that require the JRE, which they can silently install with their product.

## Introduction

Java™ 2 Platform Runtime Environment (JRE) installations are built using InstallShield Developer 7, which is based on Microsoft Window Installer. This product contains built-in support for silent or unattended installations. This document tells how to silently install the JRE. Refer to <http://support.installshield.com/> for complete information about silent installations.

## Running InstallShield in Silent Mode: Installation

Here is the command line for installing in silent mode:

```
<jre>.exe /s /v"/qn [ADDLOCAL=jrecore[,extra][,other_US] |  
ALL] [IEXPLORER=1] [NETSCAPE6=1] [MOZILLA=1]  
[INSTALLDIR=<drive>:\<install_path>]  
[PRIVATE=1][REBOOT=Suppress]"
```

where

<jre>.exe is the single executable installer for the Java Runtime Environment (JRE); ADDLOCAL, if

used, is either `jrecore[ ,extra][ ,other_US]` or `ALL`; `IEXPLORER=1`, if used, indicates that the Plug-in should be registered with the Internet Explorer browser; `NETSCAPE6=1`, if used, indicates that the Plug-in should be registered with Netscape 6 or later browsers; `MOZILLA=1` indicates that the Plug-in should be registered with Mozilla 1.1 and later browsers; `INSTALLDIR`, if used, specifies the drive and path of the installation; `PRIVATE=1`, if used, specifies that the jre is not to be registered in the Windows registry; and `REBOOT=Suppress`, if used, indicates that if locked files are encountered the computer should not be rebooted.

If `ADDLOCAL=jrecore[ ,extra][ ,other_US]` is used,

`jrecore` indicates the core of the JRE will be installed;

`extra` (optional) indicates additional Fonts, Colors, and Soundbank will be installed;

`other` (optional) indicates locale-specific `.jar` files will be installed.

If `ADDLOCAL=ALL` is used, then all the features will be installed.

If `ADDLOCAL` is not used, then only the recommended features will be installed: `jrecore` will be installed; `extra` will not be installed; `other` will be installed only if I10n support, other than English, is installed.

If `INSTALLDIR` is not specified, a private installation will go into `C:\Program Files\java\j2re1.4.2` (default location). Since a public jre might get installed there later, it is highly recommended that you do not install a private installation into the default location. I.e., if you set `PRIVATE=1`, set `INSTALLDIR` to some other location than the default.

## **Note**

The command is case sensitive and there must be no spaces in the features listed with `ADDLOCAL`.

## **Examples**

Suppose the JRE installer is `j2re-1_4_2-bin-b18-windows-i586-05_mar_2003.exe`; you want to install the JRE core and additional Fonts, Colors and Soundbank; and you want to register the Plug-in with Netscape 7 and Mozilla 1.3. Then the command would be:

```
j2re-1_4_2-bin-b18-windows-i586-05_mar_2003.exe /s /v"/qn  
ADDLOCAL=jrecore,extra NETSCAPE=1 MOZILLA=1"
```

Or suppose you want all features to be installed; you only want to register the Plug-in with Internet Explorer; and you want a private installation on the D drive at `java\jre`. Then the command line could be simply:

```
j2re-1_4_2-bin-b18-windows-i586-05_mar_2003.exe /s /v"/qn  
ADDLOCAL=ALL IEXPLORER=1 INSTALLDIR=D:\java\jre PRIVATE=1"
```

You may find it convenient to use the `"start /w"` command, as that will cause MS-DOS to wait until the install is complete. For example:

```
start /w j2re-1_4_2-bin-b18-windows-i586-05_mar_2003.exe /s
/v"/qn ADDLOCAL=ALL IEXPLORER=1 INSTALLDIR=D:\java\jre
PRIVATE=1"
```

## Running InstallShield in Silent Mode: Uninstalling

This is the command line for uninstalling in silent mode:

```
<jre>.exe /s /v"/qn [REBOOT=Suppress]" /x
```

where <jre>.exe is the single executable installer for the JRE and REBOOT=Suppress, if used, indicates that if locked files are encountered the computer should not be rebooted.

### Example

```
j2re-1_4_2-bin-b18-windows-i586-05_mar_2003.exe /s /v"/qn
REBOOT=Suppress" /x
```

## Creating a Log File

If you want to create a log file describing the installation, add /L C:\<path>setup.log to the command. Note that this comes within the quote marks (""). For example:

```
j2re-1_4_2-bin-b18-windows-i586-05_mar_2003.exe /s/v"/qn
ADDLOCAL=ALL /L C:\<path>setup.log"
```

This will cause the log to be written to the setup.log file.

To verify if a silent installation succeeded, scroll to the end of the log file.

---

# Jar Indexing

---

In the case that an applet uses multiple `.jar` files, more efficient downloading can be achieved by indexing the `.jar` files. By indexing, only required `.jar` files are downloaded. For more information on Jar Indexing, see [Java Archive \(JAR\) Features](#).



---

# JavaServer Pages

---

This chapter includes the following topics:

- [JSP Java Plug-in Action Element: <jsp:plugin>](#)
- [Deploying Java Plug-in through JSP](#)

JavaServer Pages (JSP) is a technology for generating dynamic web-page content from a web/application server. JSP provides support for Java Plug-in HTML elements. This document tells how to use the JSP Plug-in action element for deploying applets with Java Plug-in.

## JSP Java Plug-in Action Element: <jsp:plugin>

The <jsp:plugin> action element enables a JSP page author to generate HTML for downloading the Java Plug-in, if it is not installed, and executing an applet. Specifically the action generates the OBJECT element for Internet Explorer or the EMBED element for Netscape Navigator.

When the JSP is executed, the <jsp:plugin> action element is replaced by either an <OBJECT> or <EMBED> HTML tag, depending on the browser.

For example, an applet may be specified as follows in JSP:

```
<jsp:plugin type=applet code="Molecule" codebase="/html" >
<jsp:params>
  <jsp:param name="molecule" value="molecules/benzene.mol"/>
</jsp:params>
<jsp:fallback>
  <p> Unable to start Plug-in. </p>
</jsp:fallback>
</jsp:plugin>
```

In the above example, the attributes of the <jsp:plugin> element provide configuration data for the applet itself. The <jsp:param> elements specify the parameters to the applet. The <jsp:fallback> element specifies content for the browser if Java Plug-in fails to start.

Depending on the User Agent requesting the JSP page, the web server will generate either an EMBED or OBJECT tag.

For example, if Netscape Navigator is the User Agent requesting the JSP page, the following EMBED tag will be produced by the server:

```
<EMBED type="application/x-java-applet"
      code="Molecule" codebase="/html" molecule="molecules/benzene.mol">
<NOEMBED>
<p> Unable to start plug-in </p>
</NOEMBED>
```

## **Note**

The above works for Netscape 4, as it supports the EMBED tag, but it does not work for Netscape 6.x, as it currently does not support the EMBED tag.

For more information about the `<jsp:plugin>` element, see [Java Server Pages Specification](#).

## **Deploying Java Plug-in through JSP**

JSP provides an alternative way to deploy applets through Java Plug-in, without converting the HTML pages through the HTML Converter. Because of the dynamic nature of JSP, developers will be able to generate HTML pages on-the-fly that take advantage of Java Plug-in much more easily than before.

Although `<jsp:plugin>` tag support is part of the JSP specification, various web or application server vendors may provide different implementations and different levels of controls over this feature. For more information, contact your server vendor.

---

# Overview—Applet Security Basics

---

## Applet Security Basics

Below are the basic facts regarding applet security and Java Plug-in. More detail can be found in the next chapter, [How RSA Signed Applet Verification Works in Java Plug-in](#).

- All unsigned applets are run under the standard applet security model.
- If `usePolicy` IS NOT DEFINED in the `java.policy` file, then a signed applet has the `AllPermission` permission if:  
Java Plug-in can verify the signers, and the user, when prompted, agrees to granting the `AllPermission` permission.
- If `usePolicy` IS DEFINED, then a signed applet has only the permissions defined in `java.policy` and no prompting occurs.

Moreover, note that Java Plug-in now handles certificate management; i.e., the certificate verification task is no longer passed off to the browser.

---

# How RSA Signed Applet Verification Works in Java Plug-in

---

This section covers the following topics:

- [Introduction](#)
- [Support for RSA Verification](#)
- [Support for Dynamic Trust Management](#)
- [usePolicy\\_Permission](#)

## Introduction

This page provides an overview of how RSA signed applet support is implemented in Java Plug-in. For an overview of how to use RSA signed applets, see [How to Deploy RSA-Signed Applets in Java Plug-in](#).

## Support for RSA Verification

So that Java Plug-in can verify RSA signatures in a browser-independent way, Java Plug-in includes a Cryptographic Service Provider (CSP). The CSP supports the "MD2withRSA", "MD5withRSA", and "SHA1withRSA" digital signature algorithms. It is automatically registered with the Java Cryptographic Architecture framework as part of the static initializer of the `PluginClassLoader`.

## Support for Dynamic Trust Management

Java 2 SDK, Standard Edition v 1.3 introduced fine-grained access control based on "codesource" and "protection domain," as described below:

Every class that is loaded from a JAR file has a codesource, which encapsulates two pieces of information:

1. The location (URL) where the class came from;
2. The list of principals who signed it (its certificates).

Each signer principal in the codesource is represented by its X.509 public-key certificate and supporting certificate chain.

In addition, every class that is loaded by a classloader belongs to one and only one protection domain,

based on its codesource (i.e., based on where the class came from and who signed it). Every protection domain has a set of permissions associated with it, based on the configured security policy. This means that a protection domain encapsulates two things:

1. A codesource;
2. The set of permissions granted to it.

A certificate chain is a list of hierarchically ordered public-key certificates, starting at the signer's public-key certificate and ending at the certificate of a Root Certification Authority ("Root CA"). The public key of one certificate in the chain is used to verify the signature on the previous certificate in the chain. The Root CA certificate is self-signed. The assumption is that the Root CA is trusted because it is well known and widely published.

The `PluginClassLoader` checks the configured security policy to determine which permissions to grant to a given codesource. The codesource and the set of permissions granted to it then form a protection domain. This behaviour is common to all secure classloaders (i.e., instances of `java.security.SecureClassLoader`). (Note: `sun.plugin.security.PluginClassLoader` extends `sun.applet.AppletClassLoader`, which is a subclass of `java.net.URLClassLoader`, which in turn extends `java.security.SecureClassLoader`.)

The `PluginClassLoader` does additional work: If the applet is signed, and the permissions granted to it do not include the special [usePolicy](#) permission, the `PluginClassLoader` extracts the signers (and their supporting certificate chains) from the applet's codesource and attempts to verify them.

If Plug-in can verify the certificate chain all the way up to its Root CA certificate, it checks if that Root CA certificate is contained in the database of trusted Root CA certificates. If so, Plug-in will display the certificate chain of the authenticated signer and ask the user whether or not to grant `AllPermission` to code signed by that principal. Java code that is assigned the `AllPermission` permission is treated the same as system code, meaning it has all the privileges that system code has. The user can then choose whether or not to grant `AllPermission` to code signed by that principal, and whether such permission should be granted to any code signed by that principal for all subsequent sessions or for the current session only.

(For the case of unsigned applets, or signed applets whose permissions include the `usePolicy`, see [Overview—Applet Security Basics](#).)

## usePolicy Permission

A permission named `usePolicy` (introduced with version 1.3) allows system administrators to turn off the `PluginClassLoader` behaviour. If the `usePolicy` permission is granted to the codesource by the configured security policy, no user prompting will take place; only the permissions specified in the security policy will be granted to the codesource.

---

# How to Sign Applets Using RSA-Signed Certificates

---

This chapter covers the following topics:

- [Introduction](#)
- [Signing Tools](#)
- [Getting RSA Certificates](#)
  - [Getting Certificates With Jarsigner](#)
  - [Getting Certificates With Netscape Signing Tool](#)
- [Bundling Java Applets as JAR Files](#)
- [Signing Java Applets](#)
  - [Signing applet using jarsigner](#)
  - [Signing applet using Netscape signing tool](#)
- [Converting Old Netscape-Signed Applets](#)
- [Microsoft Authenticode](#)
- [Common Problems](#)

## Introduction

RSA-signed applets are supported to make deployment of signed applets easier. However, signing applets through RSA is still difficult for most novice applet developers and prevents them from taking full advantage of this Java Plug-in feature. This document provides step-by-step instructions for signing applets using RSA certificates, allowing novice applet developers to sign their applets without having to wade through the many complex security issues involved.

To sign an applet, several things are required:

1. Signing tools.
2. An RSA keypair and a certificate chain for the public keys.
3. The applet and all its class files, bundled as JAR files.

## Signing Tools

For RSA signing of applets, two types of signing tools are currently supported in Java Plug-in:

1. *Jarsigner*—a tool that is shipped as part of the Java 2 SDK. Command is `jarsigner ...`
2. *Netscape Signing Tool*—a tool that is provided by Netscape for signing applets in Navigator/Communicator. The latest version of the signing tool may be download from



```
MxFjAUBgNVBAsTDUphdmEgU29mdHdhcmUxEzARBgNVBAMTC1N0YW5sZXk
gSG8wgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBALTgU8PovA4y59eb
oPjY65BwCSc/zPqtOZKJlaW4WP+UhmebE+T2Mho7P5zXjGf7elo3tV5uI
3vzgGfnhgpf73EoMow8EJhly4w/YsXKqeJEqqvNogzAD+qUv7Ld6dLOv0
CO5qvpmBAO6mfaI1XAgx/4xU/6009jVQe0TgIoocB5AgMBAAGgADANBgk
qhkiG9w0BAQQFAAQBGAQMwLrkiFkiUYtd4ykhBtPWSwW/IKkgyfIuNMML
dF1DH8neSnXf3ZLI32f2yXvs7u3/xn6chnTXh4HYCJoGYOAbB3WNbAoQR
i6u6TLL0vgv9pMNUo6v1qB0xly1faizjimVYBwLhOenkA3Bw7S8UIVfdv
84c09dFUGcr/Pfrl3GtQ==
```

-----END NEW CERTIFICATE REQUEST-----

3. The CA (e.g., VeriSign/Thawte) will send you a certificate reply (chain) by email. Copy the chain and store it in a file. Use "keytool -import" to import the chain into your keystore. E.g.,

```
C:\>C:\jdk1.3\bin\keytool -import -alias MyCert -file VSSStanleyNew.cer
```

4. Your RSA certificate and its supporting chain have been validated and imported into your keystore. You are now ready to use jarsigner to sign your JAR file.

## Note

You must use the same alias name for all the above steps—or no alias name, in which case the alias name defaults to "mykey".

## Getting Certificates With Netscape Signing Tool

Most CAs (e.g., VeriSign/Thawte) support Netscape Signing Tool. To use the Netscape Signing Tool to sign applets using RSA certificates, obtain the [Netscape Object Signing](#) certificate from Verisign or the [Netscape Object Signing](#) certificate from Thawte—or similar certificates from other CAs. During the process of enrollment, you will be asked for personal/company information, since the CA will need to verify your identity before issuing a certificate. This process may take from several hours to several days.

Once the RSA certificate is issued, it usually consists of three files:

- cert7.db
- key3.db
- secmod.db

Depending on the CA, the certificate may be issued and stored on a floppy diskette, or it may be stored directly in the security modules of Netscape Navigator/Communicator. Once you have the certificate, you are ready to use the Netscape Signing Tool to sign your JAR file.

## Bundling Java Applets as JAR Files

To use Jarsigner to sign applets with RSA certificates, the applets must be bundled as JAR files. The Jar tool (command `jar . . .`), which comes with the Java 2 SDK, can be used for that purpose. E.g.,

```
C:\>C:\jdk1.3\bin\jar cvf C:\TestApplet.jar .
```

```
added manifest
```

```
adding: TestApplet.class (in = 94208) (out= 20103)(deflated 78%)
```

```
adding: TestHelper.class (in = 16384) (out= 779)(deflated 95%)
```

This example creates a JAR file `C:\TestApplet.jar`, and it contains all the files under the current directory and its



sub-directories.

After the JAR file is created, you should verify its content using the jar tool again, e.g.,

```
C:>C:\jdk1.3\bin\jar tvf TestApplet.jar
  0 Mon Mar 06 18:02:54 PST 2000 META-INF/
  68 Mon Mar 06 18:02:54 PST 2000 META-INF/MANIFEST.MF
94208 Wed Mar 10 11:48:52 PST 2000 TestApplet.class
16384 Wed Mar 10 11:48:52 PST 2000 TestHelper.class
```

This ensures that the class files are stored with the proper path within the JAR file.

To sign an applet with an RSA certificate using the Netscape Signing Tool, the applet must be placed in a directory, e.g., C:\signdir. The Netscape Signing Tool will bundle it as JAR file after the signing process.

## Signing Java Applets

Once you have the RSA certificates, the signing tool and the applet's JAR files, you are ready to sign the applets.

### Signing applets using jarsigner

To sign applets using jarsigner, follow these steps:

1. Use jarsigner to sign the JAR file, using the RSA credentials in your keystore that were generated in the previous steps. Make sure the same alias name is specified. E.g.,

```
C:>>C:\jdk1.3\bin\jarsigner C:\TestApplet.jar MyCert
Enter Passphrase for keystore: *****
```

2. Use "jarsigner -verify -verbose -certs" to verify the jar files

```
C:>C:\jdk1.3\bin\jarsigner -verify -verbose
-certs d:\TestApplet.jar
```

```

      245 Wed Mar 10 11:48:52 PST 2000 META-INF/manifest.mf
      187 Wed Mar 10 11:48:52 PST 2000 META-INF/MYCERT.SF
      968 Wed Mar 10 11:48:52 PST 2000 META-INF/MYCERT.RSA
smk    943 Wed Mar 10 11:48:52 PST 2000 TestApplet.class
smk    163 Wed Mar 10 11:48:52 PST 2000 TestHelper.class
```

```
X.509, CN=XXXXXXXX YY, OU=Java Software,
      O=Sun Microsystems, L=Cupertino,
      ST=CA, C=US (mycert)
X.509, CN=Sun Microsystems, OU=Java Plug-in QA,
      O=Sun Microsystems, L=Cupertino, ST=CA, C=US
X.509, EmailAddress=server-certs@thawte.com,
      CN=Thawte Server CA, OU=Certification
      Services Division, O=Thawte Consulting cc,
      L=Cape Town, ST=Western Cape, C=ZA
```

s = signature was verified

m = entry is listed in manifest



Uptime Group Plc. Class 3 CA  
GTE CyberTrust Root CA  
Uptime Group Plc. Class 4 CA  
- -----

Certificates that can be used to sign objects  
have '\*'s to their left.

2. Create an empty directory. E.g.,  
mkdir signdir
3. Put all the applet class files into it.
4. Use "signtool -Z" to sign the applet. E.g.,

```
C:\signtool13>signtool -k "Sun Microsystems, Inc.'s VeriSign, Inc. ID"
-d a:\cert -Z c:\TestApplet.jar c:\signdir
using certificate directory: a:\cert
Generating c:\signdir\META-INF\manifest.mf file..
--> TestApplet.class
adding c:\signdir\TestApplet.class to c:\TestApplet.jar...
      (deflated 57%)
--> TestHelper.class
adding c:\signdir\TestHelper.class to c:\TestApplet.jar...
      (deflated 43%)
Generating zigbert.sf file..
adding c:\signdir\META-INF\manifest.mf to c:\TestApplet.jar...
      (deflated 44%)
adding c:\signdir\META-INF\zigbert.sf to c:\TestApplet.jar...
      (deflated 46%)
adding c:\signdir\META-INF\zigbert.rsa to c:\TestApplet.jar...
      (deflated 40%)
tree "c:\signdir" signed successfully
```

5. Use "signtool -w" to verify the archive. E.g.,

```
C:\signtool13>signtool -w c:\TestApplet.jar -d a:\cert
using certificate directory: a:\cert
```

Signer information:

```
nickname: Sun Microsystems, Inc.'s VeriSign, Inc. ID
subject name: C=US, ST=CA, L=Palo Alto, OU=Java Software,
              CN=Sun Microsystems, OU=Digital ID Class 3 - Netscape
Object Signing, OU="www.verisign.com/repository/RPA Incorp.
                  by Ref.,LIAB.LTD(c)99", OU=VeriSign Trust Network,
                  O="VeriSign, Inc."
issuer name: CN=VeriSign Class 3 CA - Commercial Content/Software
              Publisher, OU="www.verisign.com/repository/RPA Incorp.
                  by Ref.,LIAB.LTD(c)98", OU=VeriSign Trust Network,
```

O="VeriSign, Inc."

Your applet has been signed properly. You are now ready to deploy your RSA signed applet. See [How to Deploy RSA Signed Applets](#) for deployment information.

## Converting Old Netscape-Signed Applets

Existing RSA signed applets designed for Netscape may use Netscape-specific security APIs. These Netscape-specific APIs are not supported in Java Plug-in. Instead, the Plug-in supports the standard Java security APIs in both Netscape Navigator and Internet Explorer.

To migrate Netscape-signed applets using the Netscape security APIs to run in Java Plug-in:

1. Comment or remove all `netscape.security.*` related statements from the Java applet.
2. Compile and archive the applet as a JAR file.
3. Re-sign the JAR file using Object Signing.

This ensures that an RSA signed applet will run in both Netscape Navigator and Internet Explorer with Java Plug-in.

## Microsoft Authenticode

Authenticode is a proprietary signing technology used in Microsoft Internet Explorer on Win32 for supporting signed applets in IE's JVM. Authenticode is not supported in Java Plug-in. Instead, the Java Plug-in supports use of RSA signed applets in both IE and Netscape.

## Common Problems

- If the JAR file is not signed properly, if the RSA certificate has expired, or if the RSA certificate is a self-generated, self-signed certificate, Java Plug-in may fail silently and not pop up the security dialog. The applet will be treated as unsigned.
- The Netscape Signing Tool is very particular about JAR file format. In Netscape Signing Tool, it expects the MANIFEST file to be at the end of the JAR file, whereas Jarsigner puts it at the beginning. The standard does not mandate where the MANIFEST file should be in the JAR file. Therefore, if you create a JAR file using Jar tool, the Netscape Signing Tool may complain about "Invalid Jar File Format". On the other hand, Jarsigner is not picky; it can verify a JAR file regardless of whether its MANIFEST file is located at the beginning or the end. To avoid this problem when using Netscape Signing Tool, you should both generate and sign the JAR file through the Netscape Signing Tool.

---

# How to Deploy RSA-Signed Applets in Java Plug-in

---

This section covers the following topics:

- [How to deploy RSA Signed Applets](#)
- [Certificate Management](#)
- [Disabling RSA signed applet support](#)

## How to Deploy RSA Signed Applets

To deploy RSA signed applets:

1. Reference the JAR from the HTML page using `archive`, `cache_archive`, or `cache_archive_ex` format. See [Applet Caching](#).
2. Place the JAR file and HTML page on the web server.

When users of Java Plug-in encounter an RSA signed applet, the Plug-in will verify whether:

1. the applet is correctly signed
2. the RSA certificate chain and root CA are valid

If both verify positive, the Plug-in will pop-up a security dialog telling the user and providing four options:

1. *Grant always*: If selected, the applet will be granted the `AllPermission` permission. Any applet signed with the same certificate will be trusted automatically in the future, and no security dialog will pop up when the certificate is encountered again. This option selection can be changed from the Java Plug-in Control Panel.
2. *Grant this session*: If selected, the applet will be granted the `AllPermission` permission. Any applet signed with the same certificate will be trusted automatically within the same browser session.
3. *Deny*: If selected, the applet will be treated as untrusted.
4. *View Issuer*: If selected, the user can examine the attributes of each certificate in the certificate chain in the JAR file.

Once the user selects the options from the security dialog, the applet will be run in the corresponding security context. Note that all options are selected on the fly; no preconfiguration is required.

# Certificate Management

The Java Plug-in Control Panel provides a [Certificates Panel](#) for managing RSA signed applets. This panel contains a list of certificates that received "*Grant always*" permission when the Java Plug-in security dialog (pop-up) ran. Users can remove any certificate from the list, and if an applet signed by a removed certificate is encountered again, a security dialog pop-up will appear asking for permission. Users can also export and view certificates through the control Panel.

## Disabling RSA Signed Applet Support

RSA signed applets can be entirely disabled in Java Plug-in by specifying the `usePolicy` permission in the policy file. If the `usePolicy` permission is among the permissions granted to the given codesource (by the configured security policy), user prompting will not take place, and only permissions specified in the security policy will be granted to the codesource. By default, RSA signed applets are enabled in the Java Plug-in.

---

# Debugging Support

---

This section covers the following topics:

- [How to Debug Applets in Java Plug-in](#)
- [Java Plug-in Console](#)
- [Java Plug-in Trace File](#)
- [javaplugin.trace property](#)
- [java.security.debug property](#)
- [Documentation](#)
- [Isolating Bugs](#)
- [Submitting Bug Reports](#)
- [Submitting Feature Requests](#)
- [Java Plug-in Feedback Alias](#)

Debugging Java applets in Java Plug-in has not been simple in the past, partly because the applets use many services and facilities from the Java 2 Runtime Environment, Java Plug-in, and the browser itself. If the applet does not work, the developer needs to spend time diagnosing the problem.

The purpose of this document is to simplify the debugging process. It provides techniques and suggestions for developing applets in Java Plug-in and describes some common mistakes in applet development.

## How to Debug Applets in Java Plug-in

In order to debug applets, you must have the appropriate version of the Java 2 SDK, Standard Edition, installed on your machine. Also make sure to compile your `.java` files with `-g` option with `javac`. To begin debugging your applet:

1. Start the Java Plug-in Control Panel. On the Advanced tab, specify the following parameters in the Java Runtime Parameters:

```
-Djava.compiler=NONE
-Xnoagent
-Xdebug
-Xrunjdwp:transport=dt_shmem,address=
    <connection-address>,server=y,suspend=n
```

The `<connection-address>` could be any string which is used by the java debugger later to connect

to the JVM. For example,

```
-Djava.compiler=NONE  
-Xnoagent  
-Xdebug  
-Xrunjdwpt:transport=dt_shmem,address=2502,server=y,suspend=n
```

See [JPDA Connection and Invocation](#) for the details on the possible runtime parameters for debugging.

2. On the *Advanced* tab in the Java Plug-in Control Panel, select "JDK <version> in <jdk-path>" for the Java Runtime Environment, where <version> is the Java Plug-in version and <jdk-path> is the path to the Java 2 SDK installation. For example, "JDK1.4 in C:\jdk1.4".
3. Start Internet Explorer or Netscape Navigator and load the page which contains the applet to be debugged. Make sure the applet code has been compiled with `-g` option with `javac`.
4. Run the command `jdb -attach <connection address>` in a DOS command prompt. <connection address> is the name mentioned in the step 1. For example, if <connection address> is 2502, you will run the command as

```
jdb -attach 2502
```

To learn more about the Java Debugger (jdb), see [The Java Debugger](#).

5. Once the jdb has attached to the VM, you can set up breakpoints in the applet.
6. When the applet in the browser reaches the breakpoint, it will stop executing, and you will see the debugger waiting for your input to continue debugging.

When debugging applets in Java Plug-in, make sure that only one instance of the browser is being used for debugging using the same connection address at the same time. Otherwise, it will result in a conflict, since the Java Runtime for each instance of the browser will try to gain exclusive access to the connection address. To debug applets in both Internet Explorer and Netscape Navigator, run either Internet Explorer or Netscape Navigator with Java Plug-in—but not both at the same time.

Debugging applets in Java Plug-in with Active Desktop is discouraged because an instance of Internet Explorer will always be running in the desktop process during the lifetime of the user session.

You can use other Java 2 debuggers, like Inprise's JBuilder or Symantac's VisualCafe, instead of jdb. To use these debuggers, you will need to change the project option in these IDEs to *attach* Java Plug-in in the browser process on the same machine or remote machine. Different Java Runtime Parameters may also be required in the Java Plug-in Control Panel. For more information, consult your Java 2 debugger or IDE manuals.

## Java Plug-in Console

One of the most powerful tools in Java Plug-in is the [Java Plug-in Console](#). It is a simple console window for redirecting all the `System.out` and `System.err` messages. The console window is disabled by



default; it can be enabled from the Java Plug-in Control Panel or the task bar. If the console is enabled, you will see the console window appear when Java Plug-in is used in the browser.

## Java Plug-in Trace File

Similar to the Java Plug-in Console, this is a file that records all the `System.out` and `System.err` messages. The trace file is disabled by default but is automatically enabled when the Java Plug-in Console is enabled. The trace file is normally located in `user.dir`, and the file is called `.plugin<version>.trace`. For example, in Windows NT this file is located in `C:\WINNT\Profiles\\.plugin<version>.trace`.

## javaplugin.trace property

This property controls whether Java Plug-in prints its trace messages during execution. This is useful to applet developers to determine what is occurring within Java Plug-in. The possible values are `true` or `false`. By default this property is `false`. To enable this property, set `-Djavaplugin.trace=true` in the Java Runtime Parameters field in the Advance Panel of the Java Plug-in Control Panel.

## java.security.debug property

This property controls whether the security system of the Java 2 Runtime Environment prints its trace messages during execution. This is useful when a security exception is thrown in an applet or when a signed applet is not working. The following options are supported:

- `access` — print all `checkPermission` results
- `jar` — print jar verification information
- `policy` — print policy information
- `scl` — print permissions `SecureClassLoader` assigns

The following options can be used with `access`:

- `stack` — include stack trace
- `domain` — dumps all domains in context
- `failure` — before throwing exception, dump the stack and domain that didn't have permission

For example, to print all `checkPermission` results and dump all domains in context, set `-Djava.security.debug=access:stack` in the Java Runtime Parameters field in the Advanced Panel of the Java Plug-in Control Panel.

# Documentation

Java Plug-in provides a rich set of documentation to help developers use the various features of Java Plug-in. The documentation includes a [FAQ](#), which includes some of the most frequently asked questions by developers. Make sure you read and understand these documents before applet development, as it may save you hundreds of hours in debugging.

## Isolating Bugs

Although Java Plug-in provides the Java 2 Runtime Environment within Internet Explorer and Netscape Navigator, most of the facilities are provided by the Java 2 Runtime itself, rather than by Java Plug-in. Therefore, if a problem occurs in Java Plug-in, it may be either a problem in Java Plug-in, the Java 2 Runtime itself or a user error. It is extremely important to determine where bugs originate, as it will affect the speed of bug evaluation and fixing. Here are some suggestions for isolating bugs:

1. Run the applets in both Internet Explorer and Netscape Navigator through Java Plug-in.
2. Run the applets in `appletviewer`. Java Plug-in is mainly derived from `appletviewer` and has inherited problems from `appletviewer` as well. This step should be performed only if the applet doesn't require specific browser facilities that Java Plug-in provides, like HTTPS or RSA signing.
3. If the applet fails in `appletviewer`, it is likely the problem is in the Java 2 Runtime Environment—and not in Java Plug-in.
4. If the applet fails in only one of the browsers, IE or Netscape, it is likely a Java Plug-in problem.
5. If the applet fails in both browsers but not `appletviewer`, it could be either a Java Plug-in problem or user error. Please examine the applet code to see if it makes any assumptions about the execution environment. For example, in `appletviewer` the current directory is set to the current directory in the shell when `appletviewer` is launched, whereas the current directory in Java Plug-in may be set to the browser's directory. Therefore, loading resources from the current directory may work in `appletviewer` but not in Java Plug-in.
6. Try to reproduce the problem on other machines or platforms. In some cases, the root of the problem may be in the machine configuration, e.g., an improper DNS setup.
7. If you have identified the problems in the Java 2 Runtime Environment or Java Plug-in, please follow the instructions in the next section to submit a bug report to the appropriate product categories.

## Submitting Bug Reports

To submit a bug report, go to the Java Development Connection's [BugParade](#). Before submitting a bug, search the BugParade to determine if the bug has already been reported. In some cases, a workaround may also have been suggested. If the bug is not already reported, submit a new bug report to the Java Plug-in team. In the bug report, include the following information:

- Complete description of the problem and step-by-step instructions for reproducing it;

- Error messages captured by Java Plug-in Console or trace file;
- Proxy configuration information, e.g., auto proxy configuration with proxy configuration file attached;
- Browser and platform information, e.g., Navigator 4.7 on Win 2000;
- A test case demonstrating the problem;
- Specify whether the problem occurs in other browsers and appletviewer;
- Specify any workaround available;
- Specify personal information—your name and email address—so you may be contacted if additional information is required.

## Submitting Feature Requests

To submit a feature request, do so through the [Report A Bug or Request a Feature](#) page. In the feature request, please make sure the following information is included:

- Complete description of the requested feature;
- How this feature will improve the quality of your product or Java Plug-in in general.

## Java Plug-in Feedback Alias

The purpose of the Java Plug-in Feedback alias, [java-plugin-feedback@sun.com](mailto:java-plugin-feedback@sun.com), is for customers to provide feedback on product features and the product in general. This alias is not intended for bug report submission. To submit a bug report, please follow the instructions given above.

---

# Java Plug-in Console

---

This chapter includes the following topics:

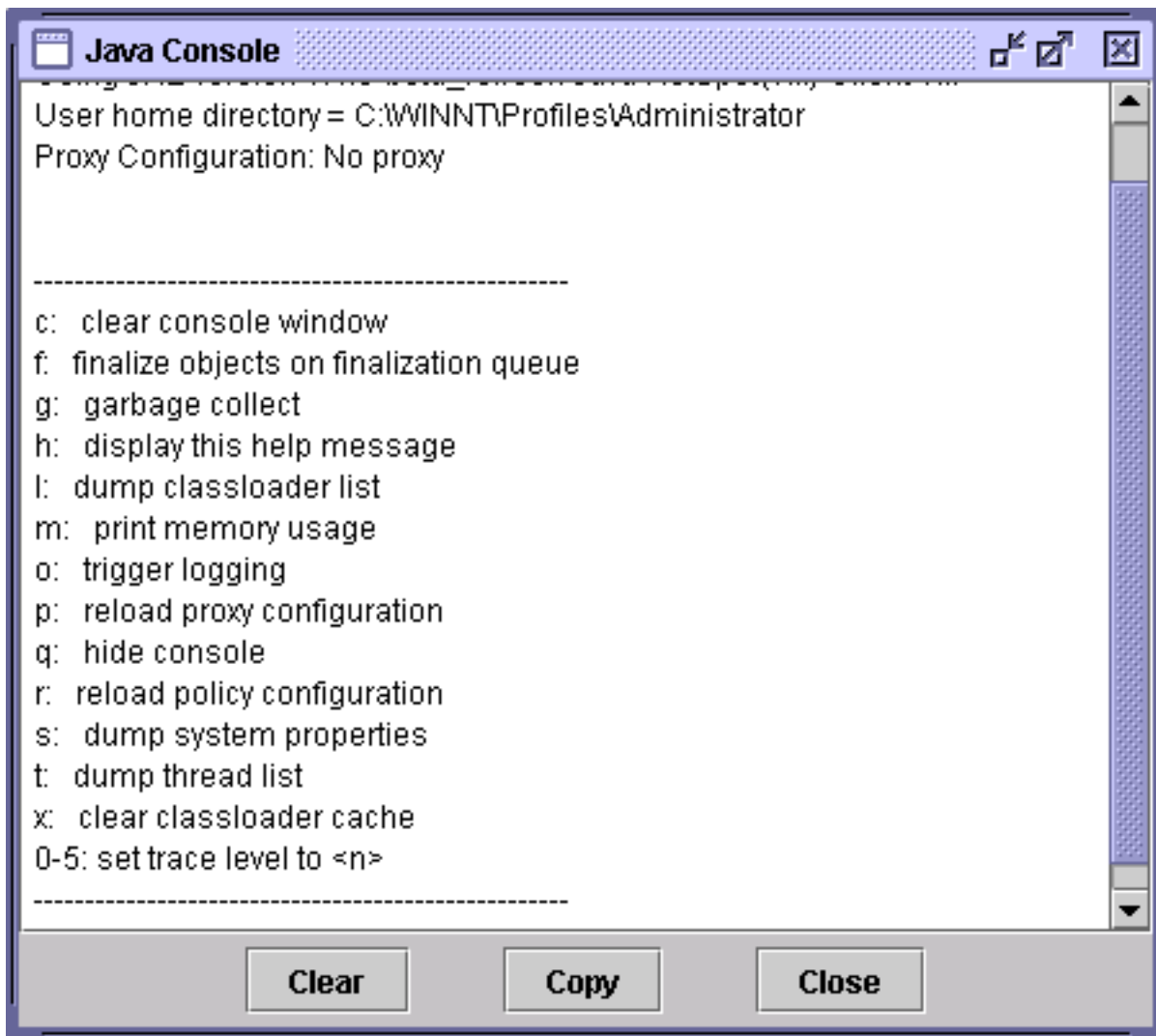
- [What is Java Console?](#)
- [Debugging Options](#)
- [Task-Bar Integration](#)
- [Startup Behavior](#)

## What is Java Console?

Java Console is a simple air for debugging that redirects any `System.out` and `System.err` to the console window.

## Debugging Options

Java Console provides various options as shown below to make applet debugging easier.



An action/option is selected by typing its letter/number while the Java Console window has focus.

| Key | Description  |
|-----|--|
| c:  | Clears the Java Console window.  |
| f:  | Triggers finalization on the objects in the finalization queue and then displays memory information. <i>Memory</i> refers to the current heap size used by the JRE. <i>Free</i> is the available memory that is free in the heap. The percent ( <i>xx%</i> ) is the free memory as a percent of the total heap size. |
| g:  | Triggers garbage collection and displays memory information as described above.  |
| h:  | Displays help message, which is being described here.  |

|      |  |
|------|--|
| l:   | <p>Displays a list of the cached <code>ClassLoader</code> objects in the Java Plug-in. (These are runtime objects cached in semiconductor memory, not on disk, and should not be confused with the cached JAR files mentioned in <a href="#">Applet Caching</a>.) Classes are cached to avoid having to load them again when returning to previously-visited pages. When a page is visited the first time, a <code>ClassLoader</code> object will be created and all of the classes that are downloaded will be cached in that object. These objects are created and cached according to their codebase. To identify a <code>ClassLoader</code> object, the "classloader list" displays the codebase for that object. Additional information displayed with a <code>ClassLoader</code> object includes <code>zombie</code>, <code>cache</code> and <code>info</code>. <code>zombie = true</code> indicates that a <code>ClassLoader</code> object is not being used (i.e., the applet is not currently loaded on the page). <code>cache = true</code> indicates that the applet should be cached, while <code>false</code> indicates that the applet will be destroyed when the page is left. <code>info</code> is a value used for debugging.</p> |
| m:   | <p>Displays heap memory usage as described above.</p>  |
| o:   | <p>Triggers logging, which directs output from the Java Plug-in Console to a log file.</p>   |
| p:   | <p>Reloads the proxy configuration.</p>  |
| q:   | <p>Causes the Java Console to disappear from the main screen.</p>  |
| r:   | <p>Reloads the policy configuration.</p>   |
| s:   | <p>Prints out the system properties. This is mostly for debugging.</p>   |
| t:   | <p>Prints out all the existing thread groups. The first group shown is <code>Group main</code>. <code>ac</code> stands for <i>active count</i>; it is the total number of active threads in a thread group and its child thread groups. <code>agc</code> stands for <i>active group count</i>; it is the number of active child thread groups of a thread group. <code>pri</code> stands for <i>priority</i>; it is the priority of a thread group. Following <code>Group main</code>, other thread groups will be shown as <code>Group &lt;name&gt;</code>, where <i>name</i> is the URL associated with an applet. Individual listings of threads will show the thread name, the thread priority, <code>alive</code> if the thread is alive or <code>destroyed</code> if the thread is in the process of being destroyed, and <code>daemon</code> if the thread is a daemon thread.</p>  |
| x:   | <p>This removes (destroys) all <code>ClassLoader</code> objects in the cache.</p> <p>Modified jar files will be downloaded from the server when a page with an applet is refreshed or revisited if you first do this: type "x" in the Java Console to clear the Classloader cache.</p>   |
| 0-5: | <p>This sets the trace-level options as described in the next section, <a href="#">Tracing and Logging</a>.</p>  |

---

## **Task-Bar Integration**

When Java Plug-in is running an icon is now displayed—in the taskbar for Windows or on the desktop for Solaris. When clicked, the Java Console opens. This allows the user to open and close the Java Console any number of times within the same browser session.

## **Startup Behavior**

Java Console may be shown, hidden, or not started at startup time, as configured through the Java Plug-in Control Panel in the Basic tab.

---

# Tracing and Logging

---

This section includes the following topics:

- [Tracing](#)
- [Logging](#)
- [Other Options](#)

## Tracing

Tracing is a facility to redirect any output in the JavaConsole to a trace file.

Tracing can be turned on by enabling the property `javaplugin.trace`. However, it turns on all tracing facilities inside Java Plug-in. To enable more fine-grained tracing, `javaplugin.trace.option` may be used. You can set trace-level options (0-5) in the [Java Console](#), shown in the previous chapter, with the following meanings:

- 0 — off
- 1 — basic
- 2 — network and basic
- 3 — security, network and basic
- 4 — extension, security, network and basic
- 5 — LiveConnect, extension, security, network and basic

This enables tracing on the fly.

Another way to set fine-grained tracing is through the [Java Plug-in Control Panel](#). For instance, to enable tracing for everthing (option 5 above), enter the following in the "Java Run Time Parameters" textfield:

```
-Djavaplugin.trace=true  
-Djavaplugin.trace.option=basic|net|security|ext|liveconnect
```

Tracing set through the Control Panel will take effect when the Plug-in is launched, but changes made through the Control Panel while a Plug-in is running will have no effect until restart.

## Logging

Similar to tracing, logging is a facility to redirect any output in the Java Console to a log file using the Java Logging API. Logging can be turned on by enabling the property `javaplugin.logging`:



```
-D javaplugin.logging = true
```

# Other Options

## File Names

By default the trace and log file names are, respectively:

```
.plugin<version>.trace and .plugin<version>.log
```

However, you can give them different names by setting the properties:

```
javaplugin.trace.filename and javaplugin.log.filename
```

in the Java Plug-in Control Panel.

## File Location

The default location (directory) of Java Plug-in related files is:

- /usr/home on Unix/Linux
- \Winnt\Profiles\*<user>* on Window

However, the location can be set with the environment variable `USER_JPI_PROFILE`.

## Overwrite Option

Furthermore, if you do not want to overwrite trace and log files each session, you can set the property `javaplugin.outputfiles.overwrite=false`. If the property is set `false`, then trace and log files will be uniquely named for each session. E.g., if default trace and log file names are used as shown above, then date information would be included as follows:

```
.plugin<username><date hash code>.trace and  
.plugin<username><date hash code>.log
```

---

# Multi-Version Support

---

This chapter includes the following topics:

- [Unique CLSID](#)
- [Unique MIME Type and dll](#)
- [Unique Java Plug-in Registry Key](#)
- [Unique Java Plug-in Property File](#)
- [Unique Java Plug-in Trace and Log Files](#)
- [Unique Java Plug-in Control Panel](#)
- [Unique Registry Keys for JRE/JDK](#)

So that multiple JRE versions may be deployed in the same environment, every new or patch release of Java Plug-in uniquely identifies registry keys, CLSID, MIME type and other resources.

**Notes:** (1) For an explanation of product version numbers, see the [note](#) in the section called "Using OBJECT, EMBED and APPLET Tags in Java Plug-in." In this section, a mix of version 1.3 and 1.4 examples is given. (2) While the examples below are for Windows, multi-version support also applies to Solaris and Linux.

## Unique CLSID

There is a unique CLSID for every new/patch version of Java Plug-in. (CLSID is used in the OBJECT tag with Internet Explorer running on Windows.) If you want a particular version of Java Plug-in to be used, then specify this unique CLSID in the OBJECT tag. The CLSID is stored in the registry in:

```
HKEY_CLASSES_ROOT\CLSID\
```

In general the CLSID looks as follows:

```
CAFEEFAC-<major version>-<minor version>-<patch version>-ABCDEFEDCBA
```

where *major version*, *minor version* and *patch version* are all 4-digit hexadecimal numbers for the Java Plug-in/JRE release.

### Examples:

For example, the CLSID for Java Plug-in 1.4 is:

```
CAFEEFAC-0014-0000-0000-ABCDEFEDCBA
```

and you would find this in the registry:

```
HKEY_CLASSES_ROOT\CLSID\CAFEEFAC-0014-0000-0000-ABCDEFEDCBA
```

The CLSID for 1.3.0\_03 is:

```
CAFEEFAC-0013-0000-0003-ABCDEFEDCBA
```

and you would find this in the registry:

```
HKEY_CLASSES_ROOT\CLSID\CAFEEFAC-0013-0000-0003-ABCDEFEDCBA
```

Note that CLSID is also stored in:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Code Store  
Database\Distribution Units\
```

## Unique MIME Type and dll

There is a unique MIME type and NP\* .dll for every new/patch version of Java Plug-in. (MIME type is used in the EMBED tag with Netscape and as a PARM in the OBJECT tag for Internet Explorer. NP\* .dll is unique to Netscape.) If you want to use a particular version of Java Plug-in with Netscape, specify the unique MIME type in the EMBED tag.

Note that this MIME type will be supported by an NP\* .dll with a unique name for every new/patch version. Thus, installing different version of Java Plug-in does not overwrite other .dll files in Navigator's Plugins directory.

In general MIME type looks like this:

```
application/x-java-applet;jpi-version=<version>
```

where *version* includes the major, minor and patch version numbers.

### Examples:

The MIME type for the 1.4 release is:

```
application/x-java-applet;jpi-version=1.4
```

The MIME type for the 1.3.0\_03 release is:

```
application/x-java-applet;jpi-version=1.3.0_03
```

MIME type is supported in the file

```
NPJPI<modified version number>.dll
```

### Example:

For Java Plug-in 1.4.0\_01, the unique MIME type would be:

```
application/x-java-applet;jpi-version=1.4.0_01
```

and it would be supported in the file:

```
NPJPI140_01.dll
```

in Navigator's Plugins directory.

## Note

Prior to release 1.4.0, the supporting .dll file was of the form NPJava<*modified version number*>.dll; e.g., release 1.3.0\_03 would be supported by NPJava130\_03.dll.

## Unique Java Plug-in Registry Key

There is a unique Java Plug-in registry key for every new/patch version of Java Plug-in. The main Java Plug-in registry key is:

```
HKEY_LOCAL_MACHINE\Software\JavaSoft\Java Plug-in\<version number>
```

where the *<version number>* will include the major, minor and the patch version numbers.

### Example:

For Java Plug-in 1.4 you will find the following in your registry:

```
HKEY_LOCAL_MACHINE\Software\JavaSoft\Java Plug-in\1.4.0
```

## Unique Java Plug-in Property File

As in previous versions, the property file continues to be located in *<user.home>\.java*. However, the filename will be unique in every new/patch release and have the following form:

```
<user.home>\.java\properties_<modified version number>
```

where the *<modified version number>* will include the major, minor, and patch version numbers; e.g.,

```
C:\WINNT\Profiles\stanleyh\.java\properties140_xx
```

## Unique Java Plug-in Trace and Log Files

A trace file has been created automatically in *<user.home>* when Java Console is enabled. Its purpose is to capture in a single file the same messages that are displayed in Java Console. But in order to avoid conflict, this file is now uniquely named in every new/patch release. There is also a uniquely named log file. These files have the form:

```
<user.home>\.plugin<modified version number>.trace  
<user.home>\.plugin<modified version number>.log
```

where the *<modified version number>* includes the major, minor and the patch version numbers; e.g., `.plugin140_xx.trace`

## Unique Java Plug-in Control Panel

There is a unique file name for the Java Plug-in Control Panel dll in every new/patch release. From the Windows Control Panel, this allows the user to launch the Control Panel for a particular version of Java Plug-in. The filename for the dll is of the form:

```
pluginctl<modified version number>.cpl
```

where the *<modified version number>* will include the major, minor and the patch version numbers; e.g., `pluginctl130_01.cpl`

Note below that the user can choose which version of the Java Plug-in Control Panel to launch.





## Unique Registry Keys for JRE/JDK

There are unique registry keys for every new/patch release of the JRE/JDK. Prior to release 1.3 of Java Plug-in, installing multiple versions of Java Plug-in overwrote the JRE/JDK registry keys. These registry keys are located as follows:

```
HKEY_LOCAL_MACHINE\Software\JavaSoft\Java Runtime  
Environment\<version number>  
HKEY_LOCAL_MACHINE\Software\JavaSoft\Java Development  
Kit\<version number>
```

where the *<version number>* includes the major, minor and the patch version numbers; e.g., 1.3.0\_01

These keys allow Java Plug-in to locate the proper version of JRE/JDK.

---

# Java-to-Javascript Communication

---

This section includes the following topics:

- [Introduction](#)
- [JSObject](#)
  - [How JSObject Works](#)
  - [Degree of JSObject support in Java Plug-in](#)
  - [Enabling JSObject support in Java Plug-in](#)
- [Common DOM API](#)

## Introduction

Java Plug-in offers two way to access the DOM: Through `JSObject` and through the Common DOM API. Each method is described below, along with security considerations.

## JSObject

Java applets may need to perform Java-to-JavaScript communication to access the Document Object Model (DOM) or to call JavaScript functions on an HTML page. Browsers allow communication between Java and JavaScript through the Java wrapper class `netscape.javascript.JSObject`. For more information, see [Java Packages for LiveConnect](#).

Because of differences in DOM implementations between browsers, Java Plug-in provides different degrees of support for `JSObject` in Internet Explorer and Navigator. This document specifies how `JSObject` support works in different browser environments.

## How JSObject Works

`JSObject` provides an easy way to access the DOM of an HTML page. But because different browsers implement the DOM differently, using `JSObject` in a Java applet may yield different behaviors in Java Plug-in. For details about the DOM implementation in a particular browser, consult the developer guide for that browser.

In general, applets access `JSObject` as follows:

```
import netscape.javascript.*;
import java.applet.*;
import java.awt.*;
class MyApplet extends Applet {
    public void init() {
        JSObject win = JSObject.getWindow(this);
        JSObject doc = (JSObject) win.getMember("document");
        JSObject loc = (JSObject) doc.getMember("location");

        String s = (String) loc.getMember("href"); // document.location.href
        win.call("f", null); // Call f() in HTML page
    }
}
```

The starting point is the static method

```
public static JSObject getWindow(Applet a)
```

which returns a JSObject representing the Window that contains the given applet. Since this method takes only `java.applet.Applet` as parameter, JSObject can be accessed from an applet, but not from a bean unless the bean is also an applet.

Once the Window object is obtained, the applet can navigate the DOM of the HTML page using the following methods:

- `public Object call(String methodName, Object args[])`
- `public Object eval(String s)`
- `public Object getMember(String name)`
- `public Object getSlot(int index)`
- `public void removeMember(String name)`
- `public void setMember(String name, Object value)`
- `public void setSlot(int index, Object value)`
- `public String toString()`

We recommend using only `getWindow()`, `call()`, `eval()`, `setMember()` and `getMember()` in Java Plug-in. The implementations of `getSlot()`, `setSlot()`, `removeMember()` and `toString()` are browser-dependent; i.e., the result of execution may vary depending on the version and platform of the browser in which Java Plug-in is running.

For more information about using JSObject, please read the section called "Java to JavaScript Communication" on the [LiveConnect](#) page.

To compile Java code to take advantage of JSObject, you must have the package `netscape.javascript` in the CLASSPATH. Currently, Java Plug-in 1.3 ships `netscape.javascript` in a JAR file called `JAWS.JAR`. To compile an applet which uses JSObject, add `JAWS.JAR` in the CLASSPATH before compilation.

Note that although JSObject is supported in Java Plug-in, it is not supported in AppletViewer in the Java 2 platform. As a result, applets using JSObject may not run in AppletViewer or result in exceptions.

## Degree of JSObject support in Java Plug-in

### Internet Explorer:

By accessing the DOM through COM, Java Plug-in provides full support of JSObject in IE.

### Netscape Navigator:

By accessing the DOM through Netscape's Plug-in API, Java Plug-in provides limited support of JSObject in Navigator 4. In Navigator 4 the following JavaScript objects can be accessed through JSObject:

- Anchor
- Document
- Element
- Form
- Frame
- History
- Image
- Layer
- Link
- Location
- Navigator
- Option
- URL
- Window

No other JavaScript objects are supported with Netscape 4, and attempting to access them through JSObject will result in Java



exceptions. These restrictions do not apply to Netscape 6; it provides full support of `JObject`.

**Note:** Even though different browsers may support the same JavaScript object, support of an object's methods and properties may differ. Be sure to check the JavaScript developer guide for your browser.

## Enabling `JObject` Support in Java Plug-in

Due to security reasons, `JObject` support is not enabled in Java Plug-in by default. To enable `JObject` support in Java Plug-in, a new attribute called `MAYSCRIPT` needs to be present in the `EMBED/OBJECT` tag as follows:

### Original `APPLET` tag:

```
<APPLET code="XYZApp.class" codebase="html/" align="baseline"
  width="200" height="200" MAYSCRIPT>
<PARAM NAME="model" VALUE="models/HyaluronicAcid.xyz">
  No JDK 1.3 support for APPLELET!!
</APPLET>
```

### New `OBJECT` tag:

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
  width="200" height="200" align="baseline"
  codebase="http://java.sun.com/products/plugin/1.3/jinstall-13-win32.cab#Version=1,3,0,0">
  <PARAM NAME="code" VALUE="XYZApp.class">
  <PARAM NAME="codebase" VALUE="html/">
  <PARAM NAME="type" VALUE="application/x-java-applet;version=1.3">
  <PARAM NAME="MAYSCRIPT" VALUE="true">
  <PARAM NAME="model" VALUE="models/HyaluronicAcid.xyz">
    No JDK 1.3 support for APPLELET!!
</OBJECT>
```

### New `EMBED` tag:

```
<EMBED type="application/x-java-applet;version=1.3" width="200"
  height="200" align="baseline" code="XYZApp.class"
  codebase="html/" model="models/HyaluronicAcid.xyz" MAYSCRIPT=true
  pluginspage="http://java.sun.com/products/plugin/1.3/plugin-install.html">
<NOEMBED>
  No JDK 1.3 support for APPLELET!!
</NOEMBED>
</EMBED>
```

If `MAYSCRIPT` is specified as false, or if `MAYSCRIPT` is absent, `JObject` is disabled. For more information about the `MAYSCRIPT` attribute in the `EMBED/OBJECT` tag, see [Using `OBJECT`, `EMBED` and `APPLET` Tags in Java Plug-in](#).

## Common DOM API

### Overview

This section discusses the [Common DOM API](#) in Java Plug-in for 1.4. It provides a standard API for accessing the DOM in the browser for browsers from different browser vendors running on various platforms.

## Note

Full support is provided with Internet Explorer 6 and Netscape 6; support is limited with other browsers.

This API is built upon the Document Object Model (DOM) Level 2 recommendation. (See <http://www.w3.org/DOM/> for various w3c recommendations.) The Document Object Model (DOM) is a set of interfaces defined using Interface Definition Language (IDL) and including Java Language bindings.

The Common DOM API includes the following w3c-defined interface packages:

- `org.w3c.dom.*`
- `org.w3c.dom.css.*`
- `org.w3c.dom.events.*`
- `org.w3c.dom.html.*`
- `org.w3c.dom.stylesheets.*`
- `org.w3c.dom.views.*`

## DOMService

The Common DOM classes allow an application to access the underlying DOM of the browser through the APIs in the `org.w3c.dom` and `org.w3c.dom.html` packages.

Each DOM represents the underlying representation of a single XML/HTML document within the browser, and each browser may display more than one XML/HTML document in multiple browser frames/windows. Thus, when an application requests to access the DOM through the Common DOM classes, it is important to return the DOM associated with the application. In order to obtain the proper DOM associated with the application, a Java object will be passed to `DOMService.getService()`. `DOMService` will return the proper `DOMService` implementations associated with the object, or an exception will be thrown. Ordinarily the Java object is an applet or JavaBeans component. However, this specification doesn't preclude other Java object types from being used, if the `DOMServiceProvider` can handle it.

To perform an action on the DOM, an object that implements `DOMAction` will need to be passed to `DOMService.invokeAndWait()` or `DOMService.invokeLater()`. `DOMAction.run()` will then be executed on the DOM access dispatch thread.

Here is a simple example of obtaining the title of the Document object:

```
DOMService service = null;

try
{
    service = DOMService.getService(MyApplet);
    String title = (String) service.invokeAndWait(new DOMAction()
        {
            public Object run(DOMAccessor accessor)
            {
                HTMLDocument doc = (HTMLDocument)
                accessor.getDocument(MyApplet);
                return doc.getTitle();
            }
        });
}
catch (DOMUnsupportedException e1)
{
}
catch (DOMAccessException e2)
{
}
```

## DOMService Plugability

Since there may be more than one `DOMServiceProvider`, it is important to allow third parties to plug their `DOMServiceProvider` implementations into the Common DOM classes. To achieve that, a new Java property, `com.sun.browser.dom.DOMServiceProvider`, is defined.

If this property is defined, it should contain a list of class names of the `DOMServiceProvider` implementations, each separated by a character "|".

When `DOMService.getService()` is called, `DOMServiceProvider` implementations specified by the `com.sun.browser.dom.DOMServiceProvider` will be called one-by-one to determine if the provider can determine the DOM association of the object, according to the order that is specified in the property. In case two `DOMServiceProvider` implementations may be able to handle the same object, the provider that is specified first in the property will be used.

## Thread Safety

Because the DOM of each browser is implemented differently, DOM access is not expected to be thread safe. Accessing implementation of DOM objects in this specification must be restricted on the DOM access dispatch thread only, so thread safety can be ensured. To accomplish that, code accessing the DOM objects must be scoped within the `DOMAction.run()` block. To invoke the action, either `DOMService.invokeAndWait()` or `DOMService.invokeLater()` should be used, so that `DOMAction.run()` will be executed in the DOM access dispatch thread.

Although implementations of DOM objects should not be called outside the `DOMAction.run()` block, the application may cache these DOM objects around as instance member of a class, or pass these DOM objects between threads. However, caching the DOM objects as static members of any object is prohibited, since static members tend to stay around much longer than the lifecycle of the underlying DOM object.

The only object in the Common DOM classes that can be called from any thread is `DOMService`. Access to other objects in the Common DOM classes is restricted within the `DOMAction.run()` block; otherwise, an exception will be thrown.

## Security

The browser DOM provides access to every service in the browser, so it is important to ensure that security is checked properly during each call into the DOM. When `DOMService.invokeAndWait()` or `DOMService.invokeLater()` is called, the security context of the caller will be captured. Later, when the corresponding `DOMAction.run()` is executed on the DOM access dispatch thread, the caller's security context will be passed to the browser DOM when implementations of the DOM objects are called. `DOMAccessException` will be thrown if the original caller doesn't have the required privileges to access the DOM.

Security policy of DOM access in each browser is different. As a result, even if a Java application is signed and fully trusted, accessing the DOM objects in the browser may still result in `DOMAccessException`.

## DOM Object Lifetime

Implementation of the DOM objects represents the real underlying objects in the browser DOM. As the XML/HTML document is changed on-the-fly, DOM objects in Java may no longer be valid. Accessing an invalid DOM object will result in `org.w3c.dom.DOMException`, according to the W3C DOM Level 2 Specification. Developers who write code to access DOM objects in `DOMAction.run()` should not assume the validity of the DOM objects at any given moment.

## Conformance Requirements

Third parties implementing this specification must provide implementations of the Common DOM classes in the `com.sun.browser.dom` package. Implementations of the W3C DOM Level 2 APIs must also conform to the W3C DOM Level 2 Specification. However, because each browser may implement a different subset of the W3C DOM APIs, third parties are not required to provide implementations of all `org.w3c.dom` and `org.w3c.dom.html` classes.

## Package com.sun.browser.dom

This section defines the API of the com.sun.browser.dom package.

### public abstract class DOMService

The DOMService defines a service API that enables applications to access a document object representing the underlying DOM of the browser window that embeds the applications.

```
public abstract class DOMService
{
    /**
     * Returns new instance of a DOMService. The
    implementation
     * of the DOMService returns depending on the setting
    of the
     * javax.browser.dom.DOMServiceProvider property or,
    if the
     * property is not set, a platform-specific default.
     *
     * Throws DOMUnsupportedException if the DOMService is
    not
     * available to the obj.
     *
     * @param obj Object to leverage the DOMService
     */
    public static DOMService getService(Object obj) throws
    DOMUnsupportedException;

    /**
     * An empty constructor is provided. Implementations
    of this
     * abstract class must provide a protected no-argument
    constructor
     * in order for the static getService() method to work
    correctly.
     * Application programmers should not be able to
    directly
     * construct implementation subclasses of this
    abstract subclass.
     */
    protected DOMService();

    /**
     * Causes action.run() to be executed synchronously on
    the
     * DOM action dispatching thread. This call will block
    until all
     * pending DOM actions have been processed and (then)
     * action.run() returns. This method should be used
    when an
     * application thread needs to access the browser's
    DOM.
     *
     * It should not be called from the
    DOMActionDispatchThread.
     *
     * Note that if the DOMAction.run() method throws an
    uncaught
     * exception (on the DOM action dispatching thread),
```

```

it's caught
    * and re-thrown as an DOMAccessException on the
caller's thread.
    *
    * @param action DOMAction.
    */
    public abstract Object invokeAndWait(DOMAction action)
throws DOMAccessException;

    /**
on the
    * Causes action.run() to be executed asynchronously
be used
    * DOM action dispatching thread. This method should
browser's
    * when an application thread needs to access the
    * DOM. It should not be called from the
DOMActionDispatchThread.
    *
    * Note that if the DOMAction.run() method throws an
uncaught
    * exception (on the DOM action dispatching thread),
it will not be
    * caught and re-thrown on the caller's thread.
    *
    * @param action DOMAction.
    */
    public abstract void invokeLater(DOMAction action);
}

```

Note that `Object obj` in the `getService(Object obj)` method above is the applet or the bean.

### **public abstract class DOMServiceProvider**

Implementation instances of the public abstract class `DOMServiceProvider` enable access to the underlying browser `DOM` for a given Java object. Instances of `DOMServiceProvider` should not be obtained by the application directly.

```

public abstract class DOMServiceProvider
{
    /**
should
    * An empty constructor is provided. Implementations
DOMService
    * provide a protected constructor so that the
class.
    * can instantiate instances of the implementation
directly
    * Application programmers should not be able to
abstract subclass.
    * construct implementation subclasses of this
obtain a
    * The only way an application should be able to
    * reference to a DOMServiceProvider implementation
    * instance is by using the appropriate methods of the
    * DOMService.
    */
protected DOMServiceProvider();

    /**
    * Returns true if the DOMService can determine the

```

```

association
    * between the obj and the underlying DOM in the
browser.
    */
    public abstract boolean canHandle(Object obj);

    /**
    * Returns the Document object of the DOM.
    */
    public abstract org.w3c.dom.Document
getDocument(Object obj) throws DOMUnsupportedException;

    /**
    * Returns the DOMImplementation object of the DOM.
    */
    public abstract org.w3c.dom.DOMImplementation
getDOMImplementation();
}

```

Note that `Object obj` in the `canHandle(Object obj)` and `getDocument(Object obj)` methods above is the applet or the bean.

### **public interface DOMAccessor**

`DOMAccessor` represents the interface that can be used within `DOMAction.run()` to access the entry point of the browser DOM.

```

public interface DOMAccessor
{
    /**
    * Returns the Document object of the DOM.
    */
    public org.w3c.dom.Document getDocument(Object obj)
throws org.w3c.dom.DOMException;

    /**
    * Returns a DOMImplementation object.
    */
    public org.w3c.dom.DOMImplementation
getDOMImplementation();
}

```

Note that `Object obj` in the `getDocument(Object obj)` method above is the applet or the bean.

### **public interface DOMAction**

`DOMAction` encapsulates all the actions of the applications that will be performed on the browser's DOM.

```

public interface DOMAction
{
    /**
    * When an object implementing interface DOMAction is
passed
    * to DOMService.invokeAndWait() or
DOMService.invokeLater(),
    * the run method is called in the DOM access dispatch
thread.
    *
    * accessor is used for the DOMAction to access the
entry point of
    * the browser's DOM, if necessary.

```

```

*
* @param accessor DOMAccessor
*/
public Object run(DOMAccessor accessor);
}

```

## **public class DOMUnsupportedException**

This exception is thrown from `DOMService.getService()` and `DOMServiceProvider.getDocument()` if the association between the Java object and the DOM cannot be found.

```

public class DOMUnsupportedException extends Exception
{
    /**
     * Constructs a new DOMUnsupportedException with no
    detail message.
     */
    public DOMUnsupportedException();

    /**
     * Constructs a new DOMUnsupportedException with the
    given detail message.
     *
     * @param msg Detail message.
     */
    public DOMUnsupportedException(String msg);

    /**
     * Constructs a new DOMUnsupportedException with the
    given exception as a root cause.
     *
     * @param e Exception.
     */
    public DOMUnsupportedException(Exception e);

    /**
     * Constructs a new DOMUnsupportedException with the
    given exception as a root cause and the given
     * detail message.
     *
     * @param e Exception.
     * @param msg Detail message.
     */
    public DOMUnsupportedException(Exception e, String
msg);

    /**
     * Returns the detail message of the error or null if
    there is no detail message.
     */
    public String getMessage();

    /**
     * Returns the root cause of the error or null if
    there is none.
     */
    public Exception getException();
}

```

## public class DOMAccessException

This exception is thrown from `DOMService.accessAndWait()` if any DOM objects throws any exception when it is accessed within `DOMAction.run()`.

```
public class DOMAccessException extends Exception
{
    /**
     * Constructs a new DOMAccessException with no detail
    message.
     */
    public DOMAccessException();

    /**
     * Constructs a new DOMAccessException with the given
    detail message.
     *
     * @param msg Detail message.
     */
    public DOMAccessException(String msg);

    /**
     * Constructs a new DOMAccessException with the given
    exception as a root clause.
     *
     * @param e Exception.
     */
    public DOMAccessException(Exception e);

    /**
     * Constructs a new DOMAccessException with the given
    exception as a root clause and the given          * detail
    message.
     *
     * @param e Exception.
     * @param msg Detail message.
     */
    public DOMAccessException(Exception e, String msg);

    /**
     * Returns the detail message of the error or null if
    there is no detail message.
     */
    public String getMessage();

    /**
     * Returns the root cause of the error or null if
    there is none.
     */
    public Exception getException();
}
```



---

# JavaScript to Java Communication (Scripting)

---

This section includes the following topics:

- [How to Script Applets](#)
- [Using Scripts to Invoke a Method](#)

## How to Script Applets

Scripting for applets is supported in Internet Explorer and Netscape 6; it is not supported in Netscape 4. It is assumed in this chapter that you understand the special HTML tags required for Java Plug-in. For information about these special tags, refer to [Using OBJECT, EMBED, and APPLET Tags in Java Plug-in](#). You also should be familiar with JavaScript.

With scripting you have the capability to call applet methods from within an HTML page. In addition to invoking methods, you can use scripts to:

- Get and set properties in applets
- Run Java objects that have been built for scripting

## Using Scripts to Invoke a Method

You often want to use scripts to invoke methods on an applet. For example, you might have an HTML button that, when clicked, starts an animation sequence. You do this through a combination of HTML tags and scripting in the HTML file, plus the actual code in the applet itself.

You need to include the following in your applet's HTML page:

- Within the OBJECT tag, an ID parameter that specifies the name of the applet
- Tags specifying a scripting language and scripting method associated with a particular action
- A SCRIPT tag for the scripting method that the action will invoke

These tags are explained in the following sections.

### Specify the Applet

You must designate an ID parameter within the OBJECT tag for your HTML page. Recall that the OBJECT tag includes such parameters as `classid`, `width`, `height`, and so on.

The ID parameter is the symbolic name of the applet. Once you establish a symbolic name for an applet through the ID parameter, you can reuse this name later in the scripts to refer to this applet.

For example, suppose you have an applet called Fractal. You add the ID parameter to the OBJECT tag and set ID to the symbolic name of the applet. You might set the tag as follows:

```
ID="Fractal"
```

Now, you can use the name Fractal within scripts to refer to the Fractal applet.

Using the same Fractal applet example, your HTML page would begin with a FORM tag, followed by an OBJECT tag, that together might look as follows:

```
<form name="Form1">
<OBJECT ID="Fractal" WIDTH=500 HEIGHT=120
CLASSID="CLSID:8AD9C840-044E-11d1-B3E9-00805F499D93"
<PARAM NAME="code" value="CLSFractal.class">
<PARAM NAME="codebase" value="1.0.2">
<PARAM NAME="level" value="5">
...
</OBJECT>
```

## Associate the Action to the Script

The HTML page defines components that are intended to invoke actions triggered by the user. You use the INPUT tag to define these components. You specify the TYPE of the component, such as button, its NAME, and VALUE. To have the button or other component actually invoke the intended action, you need to add tags that specify:

- What the user does to trigger the action, such as onClick for when the user clicks on the button
- The name of the script method that the HTML page will invoke when the specified trigger action occurs
- The language in which the script method is written

For example, suppose your HTML page creates a button that, when clicked, starts a particular animation sequence. Your HTML tag creates the button and gives that button a name and a value (label).

To do this you want to add two tags. One tag indicates that on a certain action, such as onclick, a corresponding script method should be called. You might have the tag onClick="method name". The method name is a script method within the same HTML page.

Thus, you might have the following in your HTML page:

```
<input type="button" name="Button1" value="Start"
onClick="startJSFractal" language="JavaScript">
```

This INPUT tag creates a button, names the button "Button1", and gives it the value "Start" (the label

that appears on the button). It also specifies the scripting method that will be called when a user clicks the button, and the scripting method's language. In this example, the scripting method is `startJSFractal`, and the scripting language is JavaScript. When the user clicks this button, the HTML page branches to the script method `startJSFractal`, which is written in JavaScript.

## The Script Tag and Method

You must include a `SCRIPT` tag for the method (function) that the `onClick` tag specifies. The `SCRIPT` tag must have the same name as the name used in the `onClick` tag. It also has a parameter that specifies the script language. More importantly, the script method calls the Java applet method. It identifies the method by using the name of the applet as specified by the `ID` tag, followed by the actual method name as implemented in the applet code.

For example, the same HTML page might have the following `SCRIPT` tag:

```
<SCRIPT language="JavaScript">
function startJSFractal() {
    document.Form1.Fractal.startFractal()
}
</SCRIPT>
```

In this example, the `SCRIPT` tag begins by specifying that the scripting language is JavaScript. This is followed by the JavaScript `function` statement, which starts the definition of a scripting method. The `function` statement supplies a label or name for the scripting method, calling it `startJSFractal`. This name must match the method name given for the input component's action parameter.

For this example, both the `onClick` parameter and the `function` statement specify the identical scripting method. The scripting method `startJSFractal` merely calls the actual method, `startFractal()`, implemented in the applet code. It qualifies the method name by using the document form name, then the applet name (OBJECT ID), then the method name itself, as follows:

```
document.Form1.Fractal.startFractal()
```

---

# Deploying Java Extensions

---

**Note:** *Java extensions* are also referred to as *optional packages*, *standard extensions*, or simply *extensions*.

This section covers the following topics:

- [Overview](#)
- [Manifest of the applet JAR file](#)
- [Manifest of each extension JAR file](#)
- [Manifest of the `Implementation-URL` JAR file](#)
  - [Raw installation](#)
  - [Java installer](#)
  - [Native installer](#)
- [Security](#)
- [Detailed Instructions](#)
- [Known Limitations and Other Notes](#)

Java applets may use Java extensions to provide extra functionality to users. Java Plug-in enables applets to trigger installation of various [Java extensions](#) (e.g., JavaHelp, Java 3D, Java Media Framework ...) in the Java 2 Runtime Environment. This document describes the basic steps for deploying *installed Java extensions* (versus *bundled Java extensions*) with Java Plug-in. For details about how Java Extensions work, see the [The Java Extensions Mechanism](#).

## Overview

An applet that uses extensions is packaged as a signed JAR file including manifest. When an applet is downloaded and run with Java Plug-in, Java Plug-in checks the manifest of the applet JAR file. The manifest will contain a list of all extensions that the applet requires. An extension consists of one or more JAR files to be installed into the `<jre>/lib/ext` directory.

In general, for each extension the applet manifest will list name, vendor, and version information of the extension JARs; it will also list URLs from which the JARs, or an installer for them, may be obtained if the JARs are not already installed in `<jre>/lib/ext` or are out of date. A URL may directly specify one of the extension JARs, or it may specify an installer, native or Java, that will install the extension JARs. The rules for deciding that an update is required are described in [Optional Package Versioning](#).

To use Java Plug-in for deploying Java Extensions, information about the extensions must be specified in three different manifest files:

1. [Manifest of the applet JAR file](#)
2. [Manifest of each extension JAR](#)
3. [Manifest of the `Implementation-URL` JAR file](#)

Each of these types of manifest files is described in detail below.

## Manifest of the applet JAR file

To deploy Java extensions with an applet, the applet must be packaged as a JAR file. Moreover, the manifest file of the applet JAR must define the list of extensions it requires and specify the URLs from which the extensions can be downloaded, along with other information about the extensions, according to the [Optional Package Versioning](#). For example, below is the manifest file for two extensions:

```
Extension-List: RectangleArea RectanglePerimeter
RectangleArea-Extension-Name: com.mycompany.RectangleArea
RectangleArea-Specification-Version: 1.2
RectangleArea-Implementation-Version: 1.2
```

```
RectangleArea-Implementation-Vendor-Id: com.mycompany
RectangleArea-Implementation-URL: http://mycompany.com/RectangleArea.jar
RectanglePerimeter-Extension-Name: com.mycompany.RectanglePerimeter
RectanglePerimeter-Specification-Version: 1.2
RectanglePerimeter-Implementation-Version: 1.2
RectanglePerimeter-Implementation-Vendor-Id: com.mycompany
RectanglePerimeter-Implementation-URL: http://mycompany.com/RectanglePerimeter.jar
```

In this example, two extensions are deployed with the applet—`RectangleArea` and `RectanglePerimeter`. Each has a single JAR file. If they have not been installed or if updated versions are needed, the proper versions will be downloaded from the `Implementation-URL` specifications. Notice that an `Implementation-URL` must point to a JAR file that:

- is the desired extension JAR itself (for raw installation) or
- contains a Java installer that will install the extension or
- contains a native installer that will install the extension.

This will be explained in detail in the section below called [The manifest of the Implementation-URL JAR file](#).

## Extension-List names and attribute prefixes

There are two basic scenarios here: An extension may have a single JAR file, or it may have multiple JAR files. `Extension-List` names and attribute prefixes are discussed below for these two scenarios:

### Extension with single JAR file

For an extension with a single JAR file (as in the example above), the name in the `Extension-List`, and the prefix of the related manifest attributes, should be the name of the extension JAR file.

### Extension with multiple JAR files

Some extensions consist of multiple JAR files. For example, the Java 3D extension consists of the following JAR files: `j3daudio.jar`, `j3dcore.jar`, `j3dutils.jar`, and `vecmath.jar`. There are two scenarios that need to be considered: (1) The JARs are installed by a native or Java installer or (2) no installer is used (i.e., raw installation of the extension JARs).

If a **native or Java installer** is used to install an extension, then only one of the JAR file names should be used in the `Extension-List`, and only one set of attributes, using that name as the prefix, should appear. Usually an extension has a main JAR file; if so, you should use its name in the `Extension-List` and as the prefix for the related manifest attributes. If there is no main JAR file, you can use the name of any JAR file in the optional package.

Here is an example of the applet manifest for the Java 3D extension. `j3dcore.jar` is the main JAR file.

```
Extension-List: j3dcore
j3dcore-Extension-Name: javax.media.j3d
j3dcore-Specification-Version: 1.2
j3dcore-Specification-Vendor: Sun Microsystems, Inc
j3dcore-Implementation-Version: 1.2.1_03
j3dcore-Implementation-Vendor-Id: com.sun
j3dcore-Implementation-URL: http://<myserver>/native/java3d-win.jar
```

For a **raw installation** with multiple JAR files, the story is different: You must treat each JAR file as though it were a separate extension and list each according to its name in the `Extension-List`. Each one listed then must have its own set of manifest attributes, where the prefix for an attribute set is the name of the related JAR file.

## Note on JAR Extension Identification:

Note that Java Plug-in checks four manifest attributes of an installed extension JAR file:

- `Extension-Name`
- `Specification-Version`
- `Implementation-Version`
- `Implementation-Vendor-Id`

`Extension-Name` and `Implementation-Vendor-Id` must match exactly the values specified in the applet manifest file.

- See [Optional Package Versioning](#) for how Java Plug-in evaluates the version attributes of an installed extension to decide if a newer extension needs to be downloaded.
- See [Appendix 6: Sun-Supported Specification-Version and Implementation-Version Formats](#) for rules about the format of `Specification-Version` and `Implementation-Version`.

## Manifest of each extension JAR file

Here we are talking about the JAR files that Plug-in can obtain from the URLs specified by `Implementation-URL`. The URL-obtainable extension JARs may be directly obtained (raw installation) or they may be obtained via a Java or native installer. In either case they are installed into `<jre>/lib/ext`.

The extensions that the applet requires are listed in the applet manifest. This allows Plug-in to examine the JAR files present in the `<jre>/lib/ext` directory when an applet is launched and to decide if it needs to install missing or out-of-date extensions.

In general, the manifest of an extension JAR obtained via an `Implementation-URL` needs to include various name, version, and vendor information. Thus, when such an extension JAR is installed, it will be possible in the future for Java Plug-in to compare this information to the information about an extension that an applet requests; and Plug-in will be able to determine if an extension needs to be installed/updated. Prior to any applet ever requesting an extension, it is more than likely that no extension is installed in `<jre>/lib/ext`, or that no or incomplete manifest information is present in the installed extension JAR.

For an extension with a single JAR file, the JAR file must be signed and include a manifest file with the following attributes:

- `Extension-Name`
- `Specification-Vendor`
- `Specification-Version`
- `Implementation-Vendor-Id`
- `Implementation-Vendor`
- `Implementation-Version`

## Example

```
Extension-Name: javax.help
Specification-Vendor: Sun Microsystems, Inc
Specification-Version: 1.0
Implementation-Vendor-Id: com.sun
Implementation-Vendor: Sun Microsystems, Inc
Implementation-Version: 1.1.3
```

If an extension consists of more than one JAR file and the extension is installed with a native/Java installer, then only the JAR file whose name is listed in the `Extension-List` of the applet manifest needs to have extension information (i.e., `Extension-Name`, `Specification-Version`, etc.). If no installer is used, then all JAR files must include extension information.

See [Optional Package Versioning](#) for more information about these attributes.

# Manifest of the Implementation-URL JAR file

This is the JAR file which the applet refers to with the `Implementation-URL` attribute in its manifest. It is the URL from which the extension can be obtained if no extension is installed in `<jre>/lib/ext`, or an extension is installed but it is out of date.

If the `Implementation-URL` JAR is a native or Java installer, this is indicated in the manifest via two special attributes: `Main-Class` indicates a Java installer; `Extension-Installation` indicates native installer. Note that if no installer is indicated, then the `Implementation-URL` JAR file is simply the extension JAR file itself.

As implied from the above, there are three ways that extensions can be installed by Java Plug-in:

- [By raw installation](#)
- [With a Java installer](#)
- [With a native installer](#)

Each method is discussed below:

## Raw installation

With raw installation of an extension, each extension JAR is installed by Java Plug-in into the `<jre>/lib/ext` directory without an installer (Java or native); i.e., Java Plug-in is the "installer" for each JAR. If an extension has a single JAR file, then the URL of that JAR is shown as the `Implementation-URL` in the applet JAR manifest; and Java Plug-in knows it is a raw extension because the manifest of the extension JAR file includes neither `Main-Class` nor `Extension-Installation` attribute.

Suppose we have an extension called `javax.mediax` with a single JAR, `mediax.jar`. Then the applet and extension JAR might be as shown below:

### Example: Applet JAR manifest

```
Extension-List: mediax
mediax-Extension-Name: javax.mediax
mediax-Specification-Version: 1.1
mediax-Implementation-Version: 1.1.2
mediax-Implementation-Vendor-Id: com.sun
mediax-Implementation-URL:
http://java.sun.com/products/plugin/extensions/examples/media/mediax.jar
```

### Example: Extension JAR manifest

```
Extension-Name: javax.mediax
Specification-Vendor: Sun Microsystems, Inc
Specification-Version: 1.1
Implementation-Vendor-Id: com.sun
Implementation-Vendor: Sun Microsystems, Inc
Implementation-Version: 1.1.2
```

Now suppose we have another version, `javax.mediax-2`, that has two JARs: `mediax_core.jar` and `mediax_codec.jar`. Then we must treat the two JAR files as though they were separate extensions and list each in the applet JAR manifest.

### Example: Applet JAR manifest

```
Extension-List: mediax_core mediax_codec
mediax_core-Extension-Name: javax.mediax_core
mediax_core-Specification-Version: 1.1
mediax_core-Implementation-Version: 1.1.2
mediax_core-Implementation-Vendor-Id: com.sun
mediax_core-Implementation-URL:
http://java.sun.com/products/plugin/extensions/examples/media/mediax_core.jar
mediax_codec-Extension-Name: javax.mediax_codec
mediax_codec-Specification-Version: 1.1
mediax_codec-Implementation-Version: 1.1.2
mediax_codec-Implementation-Vendor-Id: com.sun
mediax_codec-Implementation-URL:
```

[http://java.sun.com/products/plugin/extensions/examples/media/mediax\\_codec.jar](http://java.sun.com/products/plugin/extensions/examples/media/mediax_codec.jar)

### **Example: Extension JAR manifests**

```
Extension-Name: javax.mediax_core
Specification-Vendor: Sun Microsystems, Inc
Specification-Version: 1.1
Implementation-Vendor-Id: com.sun
Implementation-Vendor: Sun Microsystems, Inc
Implementation-Version: 1.1.2
Extension-Name: javax.mediax_codec
Specification-Vendor: Sun Microsystems, Inc
Specification-Version: 1.1
Implementation-Vendor-Id: com.sun
Implementation-Vendor: Sun Microsystems, Inc
Implementation-Version: 1.1.2
```

## **Java Installer**

An extension can be installed through a Java installer. The Java installer must be bundled as a JAR file, and the resulting JAR file must be specified as `Implementation-URL` in the applet JAR manifest file. During installation the JAR file will be downloaded and verified, and the `Main-Class` of the Java installer inside the JAR file will be executed to start the installer. It is the job of the Java installer to copy the extension JAR files, normally bundled with the installer, into the right location of the Java 2 Runtime (i.e., `<jre>/lib/ext`).

Though we are now dealing with an application JAR file, the attributes in its manifest should be the same as those shown for the extension JAR whose name is listed in the `Extension-List` of the applet manifest—with the addition of the `Main-Class` attribute.

### **Example: Java Installer JAR manifest**

```
Extension-Name: javax.help
Specification-Vendor: Sun Microsystems, Inc
Specification-Version: 1.1
Implementation-Vendor-Id: com.sun
Implementation-Vendor: Sun Microsystems, Inc
Implementation-Version: 1.1.3
Main-Class: com.sun.javahelp.installer
```

In this case, because `Main-Class` is present in the manifest, the JAR will be treated as a Java Installer, and `Main-class` will be invoked. It is the job of the Java installer to copy the extensions JAR files into the `<jre>/lib/ext` directory. Note that each extension JAR file must contain proper versioning information.

## **Native Installer**

An extension can also be installed through a native installer. The native installer must be bundled as a JAR file, and the resulting JAR file must be specified as the `Implementation-URL` in the applet JAR manifest file. During installation the JAR file will be downloaded and verified, and the native installer will be started. It is the job of the native installer to copy the extension JAR files, normally bundled with the installer, into the right location of the Java 2 Runtime (i.e., `<jre>/lib/ext`).

Though we are now dealing with an application JAR file, the attributes in its manifest should be the same as those shown for the extension JAR whose name is listed in the `Extension-List` of the applet manifest—with the addition of the `Extension-Installation` attribute.

### **Example: Native Installer Jar Manifest**

```
Extension-Name: javax.media.jmf
Specification-Vendor: Sun Microsystems, Inc
Specification-Version: 2.1
Implementation-Vendor-Id: com.sun
Implementation-Vendor: Sun Microsystems, Inc
Implementation-Version: 2.1.1
Extension-Installation: jmf-2_1_1-win.exe
```

In this case, because `Extension-Installation` is present in the manifest, the JAR will be treated as a native installer; and the installer itself will be launched. It is the job of the native installer to copy the Java extensions into the `<jre>/lib/ext` directory. Note that each



Java extension JAR file must contain proper versioning information.

## Security

When an installed extension needs to be update, the extension will be downloaded and verified to ensure that it is correctly signed. If it is valid, the Plug-in will pop-up a security dialog providing three options:

1. *Grant always*: If selected, the `Implementation-URL` JAR will be granted the `AllPermission` permission. Any applet or extension signed with the same certificate will be trusted automatically in the future, and no security dialog will pop up when this certificate is encountered again. This decision can be changed from the Java Plug-in Control Panel.
2. *Grant this session*: If selected, the `Implementation-URL` JAR will be granted the `AllPermission` permission. Any applet or extension signed with the same certificate will be trusted automatically within the same browser session.
3. *Deny*: If selected, the installation is cancelled.

Once the user selects the options from the security dialog, the extensions installation will be executed in the corresponding security context. The applet will not be started until the extensions are properly installed.

Because Java extensions are downloaded and installed into the Java 2 Runtime `<jre>/lib/ext` directory, each must be signed. Once the extensions are installed, they will have the permissions granted to Java extensions through the policy file.

## Detailed Instructions

Follow these steps to set up extensions for use with Java Plug-in:

### I. Create/obtain the extensions that your applet needs.

Each extension will consist of one or more JAR files, each of which must include a manifest file with version information as described above in [Manifest of each extension JAR file](#), and each must be signed. ([See exception to this with multiple JARs installed with native/Java installer.](#))

To create a JAR file from any set of files, use this command:

```
% jar cmf my_manifest my_jar input_files
```

For more information about the `jar` tool, see the [Tools and Utilities](#) documentation for your platform.

To sign the JAR file is going to take some trouble. In outline form, this is what you can do:

1. Use the `keytool -genkey` option to generate a key pair.
2. Use the `keytool -certreq` to generate a certificate request for a Certificate Authority (CA), such as [VeriSign](#) and [Thawte](#). Email the request to the CA. After the CA has confirmed your identity, it will respond with a certificate chain via email. Copy the certificate chain to a file.
3. You can then use the `keytool -import` option to import the chain to the keystore.
4. You can now use the `jarsigner` tool to sign the JAR and the `-verify` option to check that it is signed.

For more information about `keytool` and `jarsigner`, see the [Tools and Utilities](#) documentation for your platform.

More information on this topic, along with examples, is given in the chapter called [How to Sign Applets Using RSA-Signed Certificates](#). Although that chapter discusses how to sign an applet JAR file, the process is identical to signing an extension JAR file.

### II. Create/obtain the Implementation-URL JAR files

If no installer is to be used:

The extension JAR files described in step I are the `Implementation-URL` JARs.

If an installer is to be used:

1. Create/obtain the installer.
2. Create the manifest for the JAR of the installer and any bundled extensions that need to go into it.
  - For a Java installer include the `Main-Class` attribute in the manifest;
  - for a native installer include the `Extension-Installation` attribute.

3. JAR the installer, the manifest, and any bundled extensions that need to be included and sign the JAR. (The steps for JARing and signing are the same as described in step 1 above.)

## Example

Suppose we have an applet that requires Sun's Java Advanced Imaging as an installed extension. You can download this here:

```
http://java.sun.com/products/java-media/jai/downloads/download.html
```

Suppose you select the "Windows JRE Install" version. The following file will be downloaded:

```
jai-1_1_1_01-lib-windows-i586-jre.exe
```

This installer bundles the following JAR files, which it will install into the `<jre>/lib/ext` directory:

- `jai_codec.jar`
- `jai_core.jar`
- `mllibwrapper_jai.core`

You need to create the manifest for a JAR file that contains the `.exe` installer above, and you need to sign the JAR file.

The manifest would look like this:

```
Extension-Name: javax.media.jai
Specification-Vendor: Sun Microsystems, Inc
Specification-Version: 1.1
Implementation-Vendor-Id: com.sun
Implementation-Vendor: Sun Microsystems, Inc
Implementation-Version: 1.1.1_01
Extension-Installation: jai-1_1_1-01-windows-i586-jre.exe
```

Now JAR up the installer as `jai_win.jar`, together with the manifest file. You don't need to include the extension JAR files, as they are bundled with the `.exe` installer in this case. Be sure to include the `.jar` extension in the JAR file name.

Now sign `jai_win.jar`.

## III. Create the applet JAR

1. Create a manifest file for the applet. Below is a manifest file for the `jai` example using a native installer:

```
Extension-List: jai_core
jai_core-Extension-Name: javax.media.jai
jai_core-Specification-Version: 1.1
jai_core-Implementation-Version: 1.1.1_01
jai_core-Implementation-Vendor-Id: com.sun
jai_core-Implementation-URL: http://myserver.com/jai_win.jar
```

Some optional packages come packaged in different JAR files for different operating systems. If you want your applet to work on different OSs, you can use the `$(os-name)$` construction in the `Implementation-URL` manifest attribute. The `$(os-name)$` will translate to the target OS that the applet is being run on—i.e., SunOS, Linux, Windows-98, Windows-NT, Windows-2000, Windows-Me.

```
optpkg-Implementation-URL: http://.../optpkg-$(os-name)$ .jar
```

2. JAR up the `*.class` files, and any other supporting files the applet needs, together with the applet's manifest file, and sign the JAR. (The procedure for JARing your files and signing the JAR is the same as discussed previously.) Be sure to include the `.jar` extension in the JAR file name.

## IV. Generate the HTML to launch the applet

Create the HTML page for the applet. You can do this manually or you can use the `HtmlConverter` that comes with the SDK. It is recommended that you use the `HtmlConverter`. But if you want to do it manually, see [Using OBJECT, EMBED and APPLET Tags in Java Plug-in](#) for information on how to do it. Note that the applet JAR file should go in the `archive` attribute.

Suppose your applet is called `JaiApplet`, the JAR file you created for it is called `JaiApplet.jar`, and the main class is `JaiApplet.class`.

Assume the original applet looks like this:

```
<html>
```

```

<head>
<title>JaiApplet</title>
</head>
<body>
<APPLET code="JaiApplet.class" archive="JaiApplet.jar" align="absmiddle"
WIDTH="400"HEIGHT="300"> </APPLET>
</body>
</html>

```

Then if we use the HtmlConverter to convert it for dynamic versioning, we will have this:

```

<html>
<head>
<title>JaiApplet</title>
</head>
<body>
<!--"CONVERTED_APPLET"-->
<!-- HTML CONVERTER -->
<OBJECT
  classid = "clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
  codebase =
"http://java.sun.com/products/plugin/autodl/jinstall-1_4-windows-i586.cab#Version=1,4,0,0"
  WIDTH = "400" HEIGHT = "300" ALIGN = "absmiddle" >
  <PARAM NAME = CODE VALUE = "JaiApplet.class" >
  <PARAM NAME = ARCHIVE VALUE = "JaiApplet.jar" >
  <PARAM NAME = "type" VALUE = "application/x-java-applet;version=1.4">
  <PARAM NAME = "scriptable" VALUE = "false">
  <COMMENT>
  <EMBED
    type = "application/x-java-applet;version=1.4"
    CODE = "JaiApplet.class"
    ARCHIVE = "JaiApplet.jar"
    WIDTH = "400"
    HEIGHT = "300"
    ALIGN = "absmiddle"
    scriptable = false
    pluginspage = "http://java.sun.com/products/plugin/index.html#download">
  <NOEMBED>
  </NOEMBED>
  </EMBED>
  </COMMENT>
</OBJECT>
<!--
<APPLET CODE = "JaiApplet.class" ARCHIVE = "JaiApplet.jar" WIDTH = "400" HEIGHT =
"300" ALIGN = "absmiddle">
</APPLET>
-->
<!--"END_CONVERTED_APPLET"-->
</body>
</html>

```

When you run the applet Java Plug-in will display a Java Security Warning if the extension is not already installed, informing you the applet requires installation of extension `javax.media.jai` from `http://myserver.com/jai_win.jar`. If you grant permission to install the extension, the installer will install the JAR files in the `<jre>/lib/ext` directory. Once the installation is complete your applet will run.

See [Appendix 5: Complete Example—Deploying Java Media Framework as Java Extension](#) for a complete, working example showing how to deploy the Java Media Framework as a Java Extension.

# Known Limitations and Other Notes

- If an `Implementation-URL` JAR file is not signed properly, Java Plug-in will fail silently.
- For any extension be sure that a newer version of the extension contains at least the same set of JAR file names as the older version. Otherwise, installing a newer extensions may not overwrite all the older extensions JARs, and there will be a mix of different versions of an extension in `<jre>/lib/ext`. The results will be unpredictable.
- If Java Installer is used, make sure the program does not exit the `Main-class` until the installation is done. In some cases, Java Installer may create an AWT window and switch control to a different thread and return immediately from the `Main-class`. Returning control from the `Main-class` will force the applets to be loaded and started immediately, even if the Java Installer is still in the process of installation. This will cause the applet to fail to load because the extension is not installed yet.

---

# Applet Persistence API

---

Beginning with Java2 SKD, Standard Edition v 1.4, two new methods have been added to interface `AppletContext.java` to allow applet persistence across browser sessions. These methods are:

- `setStream(String key, InputStream stream)`
- `getStream(String key)`

These new methods enable the applet developer to stream data and objects from one browser session so that they can be reused in subsequent browser sessions. This provides applet persistence and makes it unnecessary to use static objects in applets for this same purpose.

`setStream(key, stream)` maps a key to a stream. `getStream(key)` returns the stream mapped to the key. If the `AppletContext` already has a stream mapped to the key when `setStream(key, stream)` is invoked, the mapping is updated.

See the [API specification](#) for more information

---

# Special Applet Attributes

---

There are six special attributes that can be used for customizing the applet window during downloading of an applet. (Note: The same applies to a JavaBeans component.) This section discusses:

- The [default appearance](#) of the applet window (no special attributes specified)
- The six special attributes
  - [image](#)
  - [progressbar](#)
  - [boxmessage](#)
  - [boxbgcolor](#)
  - [boxfgcolor](#)
  - [progresscolor](#)
- The order of [attribute precedence](#)
- The [failure scenario](#), i.e., what happens when an applet does not download or run cleanly

## Default Appearance

When none of these tags are used, the **default appearance** of the applet window is as follows:

The coffee cup logo is placed in the upper left corner. Its dimensions are 24x24 pixels and it is located 6 pixels from the top and left sides of the applet window. If the entire size of the applet is less than 36x36 pixels, then it will not be displayed. The status bar of the browser will display "Loading Java Applet . . ." when the mouse points at the applet window.

## Special Attributes

### image

The `image` attribute allows you to replace the default coffee cup logo with a custom graphic. The format with the standard `APPLET` element is:

```
<APPLET ...>
<PARAM name="image" value="my_image.gif">
</APPLET>
```

See [Using OBJECT, EMBED and APPLET Tags in Java Plug-in](#) for how this would be mapped to the

OBJECT or the EMBED tags.

If a custom graphic is specified, it should be the same size as the area of the applet window. If these sizes do not match, the graphic will be placed in the upper left corner of the area specified for applet. If it is larger than the applet window, part of it will get chopped off. If it is smaller than the applet window, gray, or whatever color is specified for `boxbgcolor`, will appear around it.

The image can be either a GIF or JPEG, and it should reside in the same directory where other resources for the applet reside; i.e., if the applet uses the `codebase` attribute, then this image should be in the `codebase` directory.

**Note:** The image file should not be in a packaged jar file with other applet resources, since the image needs to be displayed while downloading resources.

The status bar of the browser will display "Loading Java Applet . . ." when the mouse is pointed at the applet window.

## progressbar

With the `progressbar` attribute you can request display of a progress bar instead of the default graphic. The progress bar will horizontally fill the applet window up to 6 pixels to the right and left. The text "Loading Java Applet . . ." will appear above the progress bar and be center-aligned with it. By default the progress bar is purple. The format for turning on the progress bar with the standard `APPLET` element is:

```
<APPLET . . .>  
<PARAM name="progressbar" value="true">  
</APPLET>
```

The status bar of the browser will display "Loading Java Applet . . ." when the mouse is pointed at the applet window.

## boxmessage

With the `boxmessage` attribute you can customize the text that is displayed. If you use this attribute, the text will be used with the progress bar and with the status bar of the browser. The format for using this with the standard `APPLET` element is:

```
<APPLET . . .>  
<PARAM name="boxmessage" value="<your custom message goes  
here">  
</APPLET>
```

# boxbgcolor, boxfgcolor, progresscolor

These attributes can be used to customize colors in the applet window.

By default the applet window background color is gray. The attribute **boxbgcolor** can be used to specify a different background color. The format for use with the standard APPLET element is:

```
<APPLET ...>
<PARAM name="boxbgcolor" value="<value>">
</APPLET>
```

where <value> may be:

- any `Color` from `java.awt.Color`;
- `r, g, b` where `r`, `g`, and `b` are integers in the range of 0–255 that would render an opaque standard RGB (sRGB) color in the `Color` constructor `Color(int r, int g, int b)`;
- standard HTML colors: silver, green, maroon, purple, navy, teal, and olive; or
- hexadecimal color format

**Examples of each item above:** `value="cyan"`, `value="111,222,145"`, `value="silver"`, `value="#33FF33"`

By default the applet window foreground color is black. (The applet window foreground color is used for messages that appear in the window and for the border of the progress bar.) The attribute **boxfgcolor** can be used to specify a different foreground color. The color values are the same as described above. The format for use with the standard APPLET element is:

```
<APPLET ...>
<PARAM name="boxfgcolor" value="<value>">
</APPLET>
```

By default the progress bar fill color is purple. The attribute **progresscolor** can be used to specify a different fill color. The color values are the same as described above. The format for use with the standard APPLET element is:

```
<APPLET ...>
<PARAM name="progresscolor" value="<value>">
</APPLET>
```

## Attribute Precedence

The order of precedence for these attributes is as follows:

- 1) If no parameters are provided, the [default appearance](#) described above is used.
- 2) If a custom graphic is specified via the `image` attribute, then the custom graphic will appear as described above under [image](#).
- 3) If a progress bar is requested via the `progressbar` attribute, then the progress bar is displayed with



the default text and color as described above under [progressbar](#).

4) If custom text is provided via the `boxmessage` attribute, then the progress bar is displayed with the custom text as described above under [boxmessage](#).

With any combination of `image`, `progressbar`, and `boxmessage` attributes, custom colors can always be provided via the `boxbgcolor`, `boxfgcolor`, and `progresscolor` attributes.

## Failure Scenario

If an applet fails to load, the applet window will display error information. The applet window will have a single pixel border using the foreground color, and a small "broken" graphic will occupy the upper left corner. The user can right-mouse click over the applet window to get a popup menu with options to:

- open the Java Console, or
- open a window with information about the Java VM and Java Plug-in.

---

# Netscape 6

---

This section describes the primary features of the Java runtime in Netscape 6, which is enabled by Java Plug-in. It includes the following topics:

- [APPLET, EMBED AND OBJECT Tag Support](#)
- [Java-JavaScript Bi-directional Communication](#)
- [RSA Signed Applet Verification](#)
- [Display of Java Console](#)
- [Enable/Disable Java](#)
- [Applet Lifecycle Change](#)
- [Proxy and Cookie Support](#)
- [HTTPS](#)
- [Automatic Download](#)
- [Backward Compatibility Issue](#)

## APPLET Tag Support

Applets are loaded by Java Plug-in if the conventional APPLET tag is used.

### **Note:**

Currently the OBJECT element does not work with Netscape 6 with Java Plug-in. You must use the APPLET or EMBED element with Netscape 6.

## Java-JavaScript Bi-directional Communication

JavaScript can access the methods of applets, and applets can access the Document Object Model (DOM) through JavaScript.

See [Java-to-JavaScript Communication](#) and [JavaScript-to-Java Communication](#). Be sure to read the sections on security.

## **RSA Signed Applet Verification**

RSA signed applet verification is supported.

## **Display of Java Console**

You may display the Java Console through the Netscape 6 browser menu: Tasks>Tools>Java Console.

## **Enable/Disable Java**

You may enable/disable Java through the Netscape 6 browser menu: Edit>Preferences>Advanced. Note that to take effect, the browser must be restarted.

## **Applet Lifecycle Change**

Whenever a page is visited, the `init()` and `start()` methods of the applet are called; and whenever the page is left, the `stop()` and `destroy()` methods may be called.

## **Proxy and Cookie Support**

Java Plug-in previously handled proxy and cookie support alone. In Netscape 6 this support is moved to the browser.

## **HTTPS**

HTTPS is supported through Java Secure Socket Extension (JSSE) in J2SE.

## **Automatic Download**

Via its XPInstall mechanism, Netscape 6 will support automatic download of Java Plug-in (JRE) if it is not present.

## **Backward Compatibility Issue**

Although Sun has tried to ensure backward compatibility as much as possible between Java 2 and the Netscape VM, it may not be 100%. Some applets may run as is; other may only need recompilation; others, however, may need to be ported to Java 2.

---

# Frequently Asked Questions (FAQ)

---

## [Basic Information FAQ](#)

Questions and answers for people unfamiliar with Java Plug-in Software. Includes information about availability, versioning and supported features.

## [Developer Information FAQ](#)

Information for system administrators and people developing applets or webpages that use Java Plug-in Software. Includes information about the HTML Converter.

## [Troubleshooting FAQ](#)

Answers to common problems using Java Plug-in Software. Includes installation, debugging, and security information.

## [About Applet Tag Support in Java Plug-in](#)

Answers questions about support for conventional applet tage support in Java Plug-in.

---

# Basic Information

---

This section covers the following topics:

- [Basic Information](#)
- [Supported Features](#)
- [Version Information](#)

## Basic Information

### **Q: What is Java Plug-in Software?**

**A:** Java Plug-in Software is a software product from Sun Microsystems, Inc., that allows enterprise web managers to direct Java applets and JavaBeans™ components on their intranet web pages to run using Sun's Java Runtime Environment (JRE). With the 1.4 release of Java Plug-in, it also allows the conventional APPLET tag to direct Java applets to use Sun's JRE. This is especially useful in browser, such as Internet Explorer 6, that do not support Java.

### **Q: For whom is Java Plug-in Software intended?**

**A:** Java Plug-in Software is designed for enterprise customers who want to deploy Java 2 SDK, Standard Edition based applets on their intranet web pages, and support Microsoft Windows-, Solaris-, and Linux-based browsers in their enterprise. With the 1.4 release, it is also designed for the consumer who wants to run Java applets on a web page.

### **Q: Is there documentation available for Java Plug-in Software?**

**A:** To assist in your deployment of Java Plug-in Software, Sun has made available a variety of technical documentation. Documentation regarding the HTML specifications, using Java Plug-in Software in intranet environments, how proxy configuration works, how to script applets, and much more are available from the [Java Plug-in Software documentation page](#). For the consumer-oriented product that uses conventional APPLET tag support, you may also want to visit the [Get Java Technology](#) page and see the documentation available there.

### **Q: What features does Java Plug-in Software offer?**

**A:** Java Plug-in Software delivers several key capabilities to for users of Internet Explorer or Netscape's Navigator:

- Full Java 2 SDK, Standard Edition
- Full Java Compatible™ support
- Future-ready architecture
- Free public download and easy install

- Free Java Plug-in Software HTML Converter

**Q: What is the advantage of downloading and using Java Plug-in Software rather than the browser's default Java™ virtual machine?**

**A:** Using Java Plug-in Software allows enterprises and consumers to:

1. Take full advantage of Java 2 SDK, Standard Edition functionality such as JavaBeans, JNI™, and RMI today.
2. Develop and deploy 100% Pure Java<sup>SM</sup> applets on Internet Explorer and Netscape Navigator browsers, and be assured that they will run reliably and consistently in both browsers.
3. Be assured that they will receive support for the latest releases of Sun's Java 2 platform --including the high-performance Java HotSpot™ virtual machine--in Internet Explorer and Navigator as soon as Sun releases them.
4. Run applets in web browsers that no longer support Java.

**Q: Is Java Plug-in Software included in the Java 2 Software Development Kit (SDK), Standard Edition?**

**A:** Yes.

**Q: Does Java Plug-in Software work on platforms other than Microsoft Windows or Solaris (for example, Mac OS, AIX, Linux, HP-UX, etc.) versions of IE or Navigator?**

**A:** Sun has made Java Plug-in Software available for porting to all operating system providers. For information about support for Java Plug-in Software on other operating systems, contact the operating system provider.

**Q: What information should I provide when reporting a bug to <http://java.sun.com/cgi-bin/bugreport.cgi>**

**A:** When reporting bugs against Java Plug-in Software, always include the following information:

1. Operating system, including version number
2. Web browser, including version number
3. Complete output of the Java™ Console window
4. Network configuration information—proxy, special intranet environment, etc.
5. Complete description of the failure/problem encountered. Include the steps required to reproduce the problem.

## Supported Features

**Q: Does Java Plug-in Software offer support for all the features in Java 2 SDK, Standard Edition?**

**A:** Yes. Java Plug-in Software uses the same JRE that users can download from Sun's web site today (see <http://java.sun.com/j2se/>).

**Q: Is Java Plug-in Software fully compliant with the JCK (Java Compatibility Kit) test suite for**

## **Java 2 SDK, Standard Edition?**

**A:** Java Plug-in Software is fully compliant with the JCK test suite for Java 2 SDK, Standard Edition.

## **Q: What browsers and operating systems does Java Plug-in support?**

**A:** See [Supported Operating Systems and Browsers](#).

## **Q: Does Java Plug-in Software support scripting?**

**A:** Java Plug-in Software supports scripting of applets in Internet Explorer and Netscape 6. However, scripting is not available when using Java Plug-in Software in Navigator 4.

## **Q: Does Java Plug-in Software support signed applets?**

**A:** Yes, Java Plug-in Software supports standard Java 2 SDK, Standard Edition signed JARs and applets signed using Netscape signing tools. See the [How to Sign Applets Using RSA-Signed Certificates](#) for more details.

## **Q: Does Java Plug-in Software automatically recognize and use the proxy server configuration that the browser was using?**

**A:** It can. There are in fact three options beginning with the 1.4 release. See [Proxy Configuration](#) for details.

## **Q: What protocol does Java Plug-in Software support over the proxy?**

**A:** HTTP, HTTPS, FTP, Gopher, and SOCKS are supported.

## **Q: Does Java Plug-in Software support SSL?**

**A:** Yes, SSL is supported through Java Secure Socket Extension (JSSE).

## **Q: Is there a way to pre-load Java Plug-in Software during the browser startup?**

**A:** Internet Explorer and Netscape Navigator do not have mechanisms to support pre-loading Java Plug-in Software during startup. You can work around this limitation by providing your own startup page that launches a dummy applet using Java Plug-in Software.

## **Q: Can I set up Java Plug-in Software to download from an intranet web server (behind a firewall) rather than from Sun's web site?**

**A:** Yes. For more information, see [Intranet With OBJECT/EMBED Tag](#) under Part II: Deployment Schemes.

## **Q: How can I open the Java Plug-in Control Panel on Microsoft Windows?**

**A:** On the Start menu, choose Start>Settings>Control Panel, then double-click on the steaming Java cup icon labeled *Java Plug-in*.

# Version Information

**Q: How can I tell what version of Java Plug-in Software has been installed?**

**A:** There are a couple of ways:

- Check the version number in C:\Windows\Downloaded Program Files\Java Runtime Environment or C:\WINNT\Downloaded Program Files\
- Open Netscape Navigator. Go to the Help>About Plugins. Check the MIME types listed under Java Plug-in. If the following exist:

application/x-java-applet;version=1.4, you are using Java Plug-in Software 1.4

application/x-java-applet;version=1.3, you are using Java Plug-in Software 1.3

application/x-java-applet;version=1.2, you are using Java Plug-in Software 1.2

application/x-java-applet;version=1.1.2, you are using Java Plug-in Software 1.1.2

application/x-java-applet;version=1.1.1, you are using Java Plug-in Software 1.1.1

application/x-java-applet;version=1.1, you are using Java Plug-in Software 1.1

Note: You may see more than one if you have multiple versions of Java Plug-in installed.

**Q: Can different versions of Java Plug-in Software co-exist on the same system?**

**A:** Yes, it can be done through multi-version support. See [Supporting Multiple Versions of JRE/Java Plug-in](#).



---

# Developer Information

---

This section includes the following topics:

- [Information for Web-Page Authors](#)
- [Information for Applet Developers](#)
- [Information for System Administrators](#)

## Information for Web Page Authors

**Q: As a web page author, how do I use Java Plug-in Software?**

**A:** To utilize all of the features and capabilities of Java 2 SDK, Standard Edition, web page authors must modify the page's HTML to specify the use of Sun's JRE via Java Plug-in. Sun provides a specification—[Using OBJECT, EMBED and APPLETTags in Java Plug-in](#)—to guide web page authors on how to make these changes. In addition, Sun provides the Java Plug-in HTML Converter, free of charge; it will automatically make the changes to the HTML of a selected web page or set of web pages.

**Q: For Windows, the above specification mentions JRE releases that can be autdownloaded via .cab files. Where do I find a complete list of such JRE releases?**

**A:** See [Autodownload Files \(Windows Only\)](#).

**Q: How can I get the the Java Plug-in Software HTML Converter?**

**A:** The Java Plug-in HTML Converter is bundled with the J2SE SDK. It is free.

**Q: How do I run the HTML Converter?**

**A:** See [More About the HTML Converter](#).

**Q: What capabilities do the supplementary templates provide with the Java Plug-in HTML Converter?**

**A:** The HTML Converter provides both a default template (`default.tpl`) and three supplementary templates. The supplementary templates allow web-page authors to more explicitly target browsers and platforms in their environment when modifying pages with the HTML Converter:

1. **default.tpl:** The default template used by the Java Plug-in HTML Converter. The converted page can be used in [supported versions of Internet Explorer and Navigator on supported operating systems](#) to invoke the Java Plug-in.
2. **extend.tpl:** The converted page can be used with any supported browser or platform. Java Plug-in will be invoked on all [supported operating systems and browsers](#). If operating system or browser is not supported by Java Plug-in, then the browser's default Java runtime will be used.

3. **ieonly.tpl**: The converted page can be used to invoke Java Plug-in in [supported versions of Internet Explorer on supported operating systems](#).
4. **nsonly.tpl**: The converted page can be used to invoke Java Plug-in in [supported versions of Navigator on supported operating systems](#).

**Q: How do I specify a JAR file as part of an OBJECT or EMBED tag?**

**A:** You can specify one or more JAR files by defining an `archive`, `cache_archive`, or `cache_archive_ex` parameter in the OBJECT or EMBED tag.

The `archive` parameter with the OBJECT tag this looks like:

```
<PARAM NAME="archive" VALUE="demo.jar,fred.jar">
```

For more information on the `cache_archive` and `cache_archive_ex` parameters, see [Applet Caching](#).

With the EMBED tag this looks like:

```
<EMBED ... archive="demo.jar,fred.jar" ... >
```

**Q: Does Java Plug-in Software support multiple JAR files in the archive attribute in the APPLET tag? If so, why can't I get this to work?**

**A:** The `archive` attribute is supported in both the EMBED and OBJECT tags. The most common mistake is to put the JAR files in the wrong order. For example, if you use the Swing set with Java Plug-in and specify `archive="Myjar.jar,swing.jar,..."`, Java Plug-in will fail to load the applet because when `Myjar.jar` is loaded and Java Plug-in tries to initialize the applet, `swing.jar` is not yet loaded. The JAR files in the `archive` should be in the order of dependency; since `Myjar.jar` depends on other JARs, it should be put at the end of the list. The other common mistake is to put spaces or paths with the JAR file lists.

## Information for Applet Developers

**Q: Do developers need to modify their applets in order to support Java Plug-in?**

**A:** No. Any Java 2 SDK, Standard Edition, 100% Pure Java applets should run unmodified using the Java Plug-in.

**Q: What is the applet lifecycle in Java Plug-in Software?**

**A:** When an applet is encountered on an HTML page, the applet will be initialized and started. When the HTML page is closed, or the back button is pushed, the applet will be stopped and destroyed immediately.

When the same HTML page is encountered again, the applet will be initialized and started again.

**Q: Does Java Plug-in Software support Drag and Drop between applets and the native environment? If so, why can't I get it to work?**

**A:** Yes, Java Plug-in Software does support Drag and Drop. You must make sure to grant the applet the correct socket permissions using [policytool](#) to use this feature. Please see the [SocketPermission class](#) documentation for more information.

**Q: How can I speed up applet download time?**

**A:** There are several ways:

- JAR files—bundle your applets as JAR files.
- JAR indexing—optimize the class loading process. See the [JAR File Specification](#) for details.
- Applet caching—cache your applets in a permanent cache (see [Applet Caching](#)).

**Q: How do I prevent the warning banner from covering my GUI state?**

**A:** You should use the `getInsets()` method to find the size of your frame's decorative border. This includes the warning banner. For example, if you create a `Frame` with size 100x100, you might find it has `insets [top=42, left=5, bottom=5, right=6]` giving you a drawable area of 89x53. You need to position your work within the drawable area.

If you need to create a drawable area of a particular size:

1. create and show the `Frame`;
2. use `getInsets` to find the insets' sizes;
3. figure out the desired frame size by adding your desired size to the insets;
4. then use `frame.setSize()` to set your frame to that size.

**Q: Why does `InetAddress.getLocalHost().getHostName()` return `localhost`?**

**A:** This is a deliberate security feature in the Java 2 platform. Untrusted applets will not be given the real host name. Trusted applets (such as signed applets) will be given the real host name.

## Information for System Administrators

**Q: I would like to change the Java Plug-in setting in thousands of machines through system management tools. What should I do?**

**A:** Java Plug-in stores user specific settings under `<user.home>/ .java/properties<version>`. Therefore, if administrators would like to change the Java Plug-in setting globally, they may use system management tools to update the file in all machines.

**Q: Does Java Plug-in Software replace Microsoft's or Netscape's Java runtime with Sun's JRE?**

**A:** No. Java Plug-in Software does not replace the browser's underlying virtual machine if it has one. (Netscape 6 and later versions of Internet Explorer 6 do not have a Java virtual machine.) Rather, Java Plug-in Software simply enables web-page authors to specify that Sun's JRE be used instead of the default Java runtime.

**Q: What happens on browsers other than IE or Navigator, or on non-supported platforms?**

**A:** The default conversion template provided with the Java Plug-in HTML Converter is designed so that an applet will not be rendered in browsers other than Internet Explorer and Netscape Navigator on unsupported platforms. However, the Java Plug-in HTML Converter provides additional templates allowing web-page authors to specify that on non-supported platforms applets will be rendered using the original <APPLET> tag with the browser's default Java runtime.

**Q: We are trying to deploy Java Plug-in Software within our intranet environment. The chapter called [Using OBJECT, EMBED and APPLETTags in Java Plug-in](#) and other documents say we should set up an installation page within our intranet for Netscape users to install Java Plug-in. What does this page do and how should we set it up?**

**A:** The purpose of this installation page is to act as an entry point for Netscape users to install Java Plug-in Software. If a user encounters an HTML page that requires Java Plug-in, the user will get redirected to this installation page according to the `pluginspage` attribute in the `EMBED` tag. At that point, the users should be able to download and install the correct version of Java Plug-in Software for their platform. Thus, the installation page should have links to download the Java Plug-in binary.

An simple example page would look like:

```
<HTML>
<HEAD>
<TITLE>Java Plug-in Software Download Page</TITLE>
</HEAD>
<BODY>
<P><A HREF="ftp://myhost.com/public/jrel4-win32.exe">
    Java Plug-in Software for Microsoft Windows</A>
</P>
<P><A HREF="ftp://myhost.com/public/plugin-14-solaris.bin">
    Java Plug-in Software for Solaris</A>
</P>
</BODY>
</HTML>
```

Depending on how your webserver is configured, you may want to consider using CGI scripting instead of FTP for the download. Contact your webmaster for more information.

**Q: Why is the experience of downloading and installing Netscape Navigator different from Internet Explorer?**

**A:** The releases of Netscape Navigator supported by Java Plug-in Software do not provide mechanisms that allow for the automatic download and installation of Java Plug-in, as in Internet Explorer. The first time Netscape Navigator comes across a web page that is enabled for Java Plug-in (the *activated page*), it redirects the user to another web page to download and install Java Plug-in Software on the user's system. The user must then return to the activated page to render the applet using Java Plug-in Software. From that point on, the browser automatically invokes Java Plug-in whenever it encounters an activated page.

---

# Troubleshooting

---

This section includes the following topics:

- [General Troubleshooting Issues](#)
- [Troubleshooting Installation Issues](#)
- [Troubleshooting Security Issues](#)

## General Troubleshooting Issues

**Q. Why do I get a `javax.net.ssl.SSLException` (or hang or disconnect) when accessing an applet from an HTTPS site.**

A. In some SSL/TLS servers you will encounter this problem if a client message is received in a format it doesn't understand or with a protocol version number that it doesn't support. The problem is on the server side. There may be several aspects of the SSL/TLS protocol that are not implemented correctly. If the server only speaks SSLv3, when a client sends a TLSv1 (aka SSLv3.1) hello the server is supposed to respond with a SSLv3 server hello (aka SSLv3.0). But the server is not doing so; hence, you get the exception (`SSLException`).

In Java Plug-in 1.3.x the browser's implementation of SSL was used. Netscape 4.x and Internet Explorer provide only an SSLv3.0 implementation. The problem will not be seen because in this version of Plug-in only SSLv3.0 is used.

In Java Plug-in 1.4.0 the JSSE implementation of TLS/SSL was used. By default, JSSE enables the TLSv1, SSLv3, and SSLv2Hello protocols. In this version of Plug-in TLSv1 will be used, and this problem may be seen on servers with incorrect protocol implementations.

Below are some ways to work around this problem. Turn off the TLSv1 protocol and use only SSLv3.

1. In the Java Plug-in Control Panel (**Advanced** tab) specify:

```
-Dhttps.protocols="SSLv3,SSLv2Hello"
```

2. Set the system property:

```
System.setProperty("https.protocols", "SSLv3");
```

3. If you have access to the socket, you can do this:

```
socket.setEnabledProtocols("SSLv3");
```

In Java Plug-in 1.4.1 the SSLv3 and SSLv2Hello protocols are used by default. Because most browsers use SSLv3 by default and most web servers support it—and to avoid seeing the above problem—the change was made to this version of the Plug-in. Users that need to use TLSv1 should set the

https.protocols settings.

**Q: Is there a way, other than restarting the browser, to force the JVM to check the server for modified JAR files for a page with an applet?**

**A:** Modified jar files will be downloaded from the server when a page with an applet is refreshed or revisited if you first do this: type "x" in the Java Console to clear the Classloader cache.

**Q: When trying to play a game on `http://games.yahoo.com` I get a `ControlAccessException`. What is the problem and is there a workaround?**

**A:** The problem is that the game applet needs permission to connect to one or more servers, and it is being denied permission for security reasons. The workaround is this: Add the following to your `java.policy` file:

```
grant codeBase "http://download.yahoo.com/games/clients/" {
    permission java.net.SocketPermission "*", "connect";};
```

`java.policy` is located in <JRE installation directory>/Java/j2re1.4.0/lib/security/.

**Q: How do I get Netscape to find my plugin when I've downloaded and installed the J2SE, which includes Java Plug-in?**

**A:** Set the `NPX_PLUGIN_PATH` environment variable to the location of the Java Plug-in (the directory in which the `javaplugin.so` file is located):

```
NPX_PLUGIN_PATH=$JAVAHOME/jre/plugin/sparc/ns4 for Netscape 4
NPX_PLUGIN_PATH=$JAVAHOME/jre/plugin/sparc/ns6 for Netscape 6
```

**Q: I can't get Java Plug-in software to install in an intranet environment when I place it on our Netscape Enterprise 3.0 SuiteSpot web server. Why not?**

**A:** We have reports that the Netscape Enterprise 3.0 SuiteSpot Webserver is unable, at least in some circumstances, to serve up `.exe` files. One apparent workaround has been to configure the HTML so that the Java Plug-in product is installed by a Visigenic Orb Gatekeeper, which also functions as a web server.

**Q: I'm having trouble debugging with Java Plug-in Software. Do you have any tips?**

**A:** In some circumstances, Java Plug-in Software will use a different debug connection address than expected. This occurs when Java Plug-in Software is loaded into the `Explorer.exe` process running in one of the following configurations:

- Windows 95/NT running Active Desktop with IE4 or later
- Windows 98/ME/2000

Java Plug-in Software is loaded into the `Explorer.exe` process when an HTML page containing the `OBJECT` tag is viewed in the following ways:

- via preview in an Explorer folder, if the desktop settings allow web content to be viewed in folders (in other words, if the View->As Web Page menu option is checked)

- via Internet Explorer, if the Microsoft Internet settings for "Launch browser windows in a separate process" or "Browse in a new process" are not set
- via a web page shown on the Active Desktop

This can cause problems when debugging Java applets, since no two processes should use the same debug connection address. (See [How to Debug Applets in Java Plug-in](#) in [Debugging Support](#) regarding setting of the connection address.). If the Explorer.exe process has already claimed the debug connection address, and the Netscape.exe or Iexplorer.exe tries to use it, debugging problems may result.

Java Plug-in Software avoids this complication when loaded into the Explorer.exe process by prepending the debug connection address specified in the Control Panel with the string Explorer.

For example, if the default connection address set at the time of Java Plug-in Software installation is 2502, when running under Explorer.exe it is actually set to Explorer:2502.

When running JDB from the Java 2 SDK you should specify

```
jdb -attach Explorer:2502
```

to attach to the JVM loaded into the Explorer.exe process.

**Q: My applet is no longer scriptable in Internet Explorer with Java Plug-in. Why?**

**A:** With Java Plug-in Software 1.3 a `scriptable` tag must be included and given a value of "true" in order for an applet to be scriptable. See the [Using OBJECT, EMBED and APPLETTags in Java Plug-in](#) for more information.

**Q: Why am I having problems using some standard extensions/optional packages?**

**A:** Only extensions installed in the `<jre>\lib\ext` directory will be added to the classpath. Extensions installed in directories pointed to by the `java.ext.dirs` system property will not be added.

**Q: Java Plug-in used to work with my Navigator 4.0.x browser. But when I upgraded to Navigator 4.5 and re-installed Java Plug-in, it does not work with Navigator 4.5. Why?**

**A:** It has been reported that Navigator 4.5 may not install the user profile properly during installation. As a result, Java Plug-in Software may not read the correct user profile setting. To make sure the user profile setting is correct, check the following:

- HKEY\_LOCAL\_MACHINE\Software\Netscape\Netscape Navigator\Users\CurrentUser
- HKEY\_LOCAL\_MACHINE\Software\Netscape\Netscape Navigator\Users\<UserName>\DirRoot

Make sure that these two registry keys exist and `<DirRoot>` points to an existing user profile. If any of the registry keys are missing or incomplete, use the User Profile Manager tool to recreate your profile.

**Q: We are trying to use Java 3D with Java Plug-in but it doesn't work at all. Why?**

**A:** Java 3D comes with various packages. Installing it incorrectly may inadvertently disable Java Plug-in or cause it to fail. Follow these general instructions for using Java 3D and Java Plug-in Software:

1. Install Java Plug-in first.
2. Install Java 3D in a new directory. Do not install it over the existing Java 2 SDK, Standard Edition/JRE. See [this page](#) for installation instructions.
3. Install a version of the SDK/JRE that is appropriate for Java 3D if one is not already installed on your computer.
4. In the Java Plug-in Control Panel, select the appropriate version of the SDK/JRE..

The Java 3D demos should now run within Java Plug-in.

**Q: Some web/proxy servers require users to login for authentication. When I used the browser to access this server with Java Plug-in, two login dialog boxes appeared. Why?**

**A:** Normally Java Plug-in will download the applets using its own connection. If the web/proxy server requires login, the browser will first encounter the request and bring up a login dialog box. After the HTML page is downloaded, Java Plug-in will try to download the class or jar files for the applet. However, since Java Plug-in has no access to the login information that the browser previously obtained, it will bring up its own login dialog box.

**Q: When I tried to deploy Java Plug-in in the intranet and put the binaries on the internal web server, IE doesn't download and install Java Plug-in Software when it encounters the converted page. What's going on?**

**A:** You may want to check that the CODEBASE in the OBJECT tag actually has the correct URL for Java Plug-in. Also, turning off execute privileges on the directory in which you put the Java Plug-in Software executable may help.

**Q: I am experiencing problems getting an applet to render using Java Plug-in Software. What is the cause of this?**

**A:** While this may be due to a variety of circumstances unique to your operating environment, a frequent cause of this problem is a security exception.

**Problem:** Your network does not support DNS (Domain Name Service). In order to perform certain security checks, the applet `SecurityManager` needs to be able to find the IP address from which your applet was downloaded. If DNS is not available, these security checks may fail.

**Workaround:** When visiting the target web page, specify an IP address rather than a hostname in the URL. For example, use "http://123.45.35.128/fred.html".

**Q: I changed my browser setting while Java Plug-in Software was running, but it still uses the old settings after the change. Why?**

**A:** The browser settings are read in by Java Plug-in when it is started. These settings are valid throughout the lifetime of the browser session. To make Java Plug-in read in the new settings, restart your browser. If you are running Active Desktop with Java Plug-in, you need to restart the computer.

**Q: When I loaded my applet, it said "noninit" or "applet not initialized" in the browser's status**



## **bar. How can I identify the cause of the problem?**

**A:** Follow these steps:

1. Look at the error message in the Java Console.
2. If you are accessing the applet through the network, make sure the proxy info shown in the Java Console is correct.
3. Make sure all the class/JAR files are in the right directory.
4. Make sure the converted HTML page is correct.
5. Try the unconverted page with AppletViewer on the same machine. If it works, check 3, 4, and 5 again.

**Q: When I try to use the `AppletClassLoader` with Java Plug-in Software, it crashes with a null pointer exception in IE4 but works fine in Netscape. Why does this happen? Is there a way to make it work correctly in IE4?**

**A:** Java Plug-in Software in IE4 tries to load `<YourAppletName>BeanInfo.class` even if your applet is not a bean. There is a bug in Java 2 SDK, Standard Edition v 1.3 in the `AppletClassLoader` that occurs when trying to load nonexistent classes. To prevent this problem, create an empty `<YourAppletName>BeanInfo.class`.

**Q: Why does Java Plug-in Software sometimes crash Internet Explorer but not Netscape Navigator?**

**A:** In certain circumstances, bad HTML will cause the browser to crash. One example is the absence of an `</XMP>` tag. Please make sure your HTML is correct.

Another possibility is that the plug-in was disabled using the Control Panel before you accessed a Plug-in-enabled page. Please check your settings to make sure the Plug-in is enabled.

**Q: My applet used to create a top-level frame that would remain visible through page switches. When I upgraded to Java Plug-in this no longer occurs. Why?**

**A:** In Java Plug-in Software, applets are stopped and destroyed during page switches. All the visible components should be destroyed as well. There has been an enhancement in Java 2 SDK that makes sure that all of the resources of the applet are properly released, including the top-level frame.

## **Troubleshooting Installation Issues**

**Q: I am trying to install the Java Plug-in on a network drive. However, it doesn't install. Why?**

**A:** If your network drive is protected or read-only, you will not be able to install the Java Plug-in. Contact your system administrator for more details.

**Q: I am trying to install Java Plug-in. However, whenever the install program tries to install the Java Plug-in for Netscape Navigator, it displays an error. My Navigator is on a network drive. Is there a way to fix this?**

**A:** If your Navigator is installed on a network drive, you may not have permission to install the plugin

DLL on the Navigator Plugins directory. Contact your system administrator for more details.

**Q: I'm having trouble installing the Java Plug-in on my Microsoft Windows machine. I see the error: "An application error has occurred and an application error log is being generated. Exception: access violation ..." What might be the problem?**

**A:** The Microsoft Window installation (using Installshield's installer) may not work if you have Quarterdeck's Cleansweep product running in the background.

**Q: When I reboot after I uninstalling Java Plug-in software, I get a dialog box warning that a .cpl file has been removed. What is this?**

**A:** If the Java Plug-in Control Panel is open while you are uninstalling the plug-in, `pluginctl.cpl` will not be removed as it is locked when the Control Panel folder is open.

**Q: When I uninstall Java Plug-in, the Microsoft Window Plugin for Netscape seems to stay on the machine. How can I remove that?**

**A:** Delete the `NPJPI<modified version number>.dll` from the Netscape Plugins directory. For example, for Java Plug-in 1.4.0, delete `NPJPI140.dll` from the Plugins directory.

## Troubleshooting Security Issues

**Q: I have some security-related issues in my applet. How do I debug it?**

**A:** You can enable the `java.security.debug` property to enable trace messages from the security system. Please see [java.security.debug Property](#) in the chapter called Debugging Support for more information.

**Q: When downloading applets from the Internet, an `AccessControlException` is thrown. However, when the applet is located on the intranet, it works. Why?**

**A:** To prevent DNS spoofing, a security check in Java Plug-in requires the host name to be resolved into an IP address before any connection is made from the applet. However, the side-effect of this security check is to require the intranet DNS server to be able to resolve any external host name from the Internet. For some customers, this is not the way their DNS servers are setup within the enterprise, and it causes Java Plug-in to fail. To resolve this issue, there are several options:

1. Modify the HTML page of the applet, so the codebase and document base of the applet contains IP addresses instead of host names. This will avoid Java Plug-in performing a DNS lookup for the host name.
2. Set `trustProxy` to true in Java Plug-in. By setting this property, Java Plug-in will trust the proxy to perform a proper DNS lookup and return information to Java Plug-in from trusted hosts. For intranet customers whose proxy servers are setup internally and can be trusted, this property may be used. This property may be specified as `-DtrustProxy=true` in the Java Plug-in Control Panel.

Note that this property should be set by administrator, as setting this property improperly with an untrusted proxy server may expose the client machines to DNS spoofing. This property is similar

to the security.lower\_java\_network\_security\_by\_trusting\_proxies property supported by Netscape Navigator's JVM.

**Q: I keep getting a `ClassNotFoundException` exception when my webpage is loaded with HTTPS in Netscape Navigator. Why?**

**A:** This is caused by an applet specifying a nonexistent `.jar` or `.class` file in the `EMBED` tag. Due to limitation of what information can be returned via HTTPS in Navigator, the web server's "File Not Found" HTML page is returned instead of the appropriate status. This HTML page is treated as a `.class` file and that causes the exception.

**Q: I tried to run an RSA signed applet with Java Plug-in, but it is being treated as an untrusted applet. What is the problem?**

**A:** There are several possibilities:

- Your applet is not signed properly. Use the Netscape's `signtool` or Sun's `jarsigner` to verify it.
- Your RSA certificate and its certificate chain may have expired.

Please see our [How to Sign Applets Using RSA-Signed Certificates](#) for details.

**Q: Why do I get a yellow banner across my applet frame when using Java Plug-in?**

**A:** When an applet creates a free-standing Frame, Java Plug-in adds a yellow warning banner so users will know they are dealing with an untrusted applet window.

**Q: Can I disable the yellow warning banner on frames?**

**A:** The yellow warning banner is an important security feature. It cannot be disabled by untrusted applets.

If you use a signed applet, where the signing key is trusted by the end user, then the warning banner will not be shown.

**Q: How do I prevent the warning banner from covering my GUI state?**

**A:** See the same question in the [Developer FAQ](#).

**Q: Why does `InetAddress.getLocalHost().getHostName()` return "localhost"?**

**A:** See the same question in the [Developer FAQ](#).

---

# About APPLET Tag Support in Java™ Plug-in

---

## Note

For more information about compatibility issues between the Microsoft VM and Sun's, see [Appendix 4: Microsoft VM and Java 2 Applet Compatibility Issues](#).

### **Q: Why should I encourage people to use the Java 2 Platform?**

**A:** The Java 2 Platform is richer than its predecessors in functionality, performance, and security options. As a developer, you want the users of your software to have the most feature-complete Java platform possible so that the full range of APIs and libraries are available for your applications. If your customers are using the Java 2 Platform, you can write more powerful, robust, and full-featured applications and applets. And with the Java 2 Platform the performance of your code will be better too.

By using the latest Java Plug-in with APPLET tag support, you can help to ensure that clients running your applets will have the latest Java 2 Platform.

### **Q: Can I continue to use OBJECT tags to call my applets?**

**A:** Yes. Even though the Java Plug-in has been enhanced to support APPLET tags, it is fully backward compatible with existing web pages that use OBJECT tags to launch applets. Moreover, the JRE/Java Plug-in continues to ship with the HTML Converter for those developers who want to invoke their applets using OBJECT rather than APPLET tags.

### **Q: Are there compatibility issues with using the Java 2 Platform?**

**A:** In the large majority of cases, there are no compatibility issues to be concerned about when moving from JDK 1.1 to the Java 2 Platform. The documentation for a given Java 2 Platform release describes all known incompatibilities, most of which are in the category of "corner cases." For example, known compatibility issues in version 1.4 of the Java 2 Platform are described at <http://java.sun.com/j2se/1.4/compatibility.html>.

Some old class files generated by early bytecode compilers do not strictly conform to the proper class-file format, and the Java 2 Runtime Environment may not load such classes.

In addition to compatibility issues between different releases of the Java platform, there is the question of incompatibilities between Sun's Java runtime environment and other Java runtime environments; e.g., the one included with some Internet Explorer browsers. Sun has made a great effort to minimize such incompatibilities. Among the steps that Sun has taken in this JRE/Java Plug-in are the following:

- The classloader sharing policy in Java Plug-in has been changed to minimize incompatible behavior in applets that depend explicitly on the classloader sharing policy of the Microsoft VM.
- The image and sound resource lookup sequence in Java Plug-in has been changed to give resources specified with the `HTML archive` parameter priority over those on the codebase URL. This change was made to minimize incompatible behavior for applets that rely explicitly on the resource lookup sequence employed by the Microsoft VM.
- The implementation of `AppletgetParameter()` has been modified to strip off leading and trailing whitespace from parameters before they are returned to the applet. This is the behavior of the Microsoft VM in some Internet Explorer browsers, and this change was made to eliminate possible incompatible behavior in applets that rely explicitly on that behavior.

**Q: What security model does Java Plug-in use?**

**A:** JRE/Java Plug-in is a product in the Java 2 Platform family, and it uses the new Java 2 Platform's security model. This security model is more flexible and configurable than the security model used by JDK 1.1. For more information, see the Java 2 Platform's security documentation at <http://java.sun.com/j2se/1.4/docs/guide/security/index.html>.

Note that applets that rely explicitly on proprietary technology in the security model of the Microsoft VM may not work with Java Plug-in.

**Q: My applet fails with a `ClassFormatError`. What's wrong?**

**A:** This problem may be caused by bytecode generated by old compilers. Such bytecode may not conform to the virtual machine specification, which is more strictly enforced by recent releases of the JRE.

**Q: Why doesn't Java Plug-in load my applet's `.cab` file?**

**A:** The Java Plug-in supports applet packaging in `.jar` files. It does not support the Microsoft proprietary `.cab` file format. Therefore any applets packaged in Microsoft's `.cab` file format will not load in JRE/Java Plug-in.

**Q: Will my Authenticode-signed applet work with Sun's JRE/Java Plug-in?**

**A:** No. The Sun JRE/Java Plug-in does not support Microsoft proprietary technology such as Authenticode signing and `.cab` file formats. Signed applets that rely on Microsoft's Authenticode technology will not load in JRE/Java Plug-in.

**Q: My applet uses J/Direct, AFC, and WFC, etc. Will it work in Sun's JRE/Java Plug-in?**

**A:** No. Libraries for these and other proprietary Microsoft technologies are not included with JRE/Java Plug-in. Applets that rely on these Microsoft proprietary technologies will not work properly when run on JRE/Java Plug-in.

**Q: Why does my applet throw a `NullPointerException` from the AWT Dispatch Event thread?**

**A:** The events that occur during applet startup and shutdown may differ between the Microsoft and Sun implementation of the Java platform. For example, the logic in an applet may rely on the applet being

visible when `Applet.start()/Applet.stop()` is called. That condition may be true when the applet is run on Microsoft's implementation but may not be true on Sun's implementation.

Applets that rely on specific events that occur during startup and shutdown on Microsoft's implementation of the Java platform may not work properly with JRE/Java Plug-in. The most common symptom of this type of problem is a `NullPointerException` from the AWT Dispatch Event thread.

**Q: Why does my applet throw a `ClassCastException` from the AWT Dispatch Event thread?**

**A:** The number of containers between the applet and the encompassing frame are different in the Microsoft and Sun implementations. Therefore, an applet that relies on the frame being at some particular level of containment in the Microsoft VM, without navigating the entire AWT hierarchical component tree, is likely to fail when run on Sun's JRE/Java Plug-in. The most common symptom of this problem is a `ClassCastException` from the AWT Dispatch Event thread.

**Q: Does Java Plug-in support Java-JavaScript communication?**

**A:** Yes, Java Plug-in supports basic, bidirectional Java-JavaScript communication. The following, however, is a known incompatibility.

In the Microsoft implementation, applet methods and properties exposed in JavaScript are exactly the same as the methods and fields in the applet object. In Java Plug-in, an applet's methods and properties are exposed in JavaScript through `JavaBeans™` introspection, which treats the applet's fields in a different manner than the Microsoft VM. Therefore, JavaScript accessing fields in an applet object may not work the same when run on JRE/Java Plug-in.

---

# More About HTML Converter

---

This section includes the following topics:

- [Running the GUI Version of the HTML Converter](#)
  - [Choosing files within folders to convert](#)
  - [Choosing a backup folder](#)
  - [Generating a log file](#)
  - [Choosing a conversion template](#)
  - [Choosing version scheme](#)
  - [Converting](#)
  - [Exit or convert more files](#)
  - [Details about templates](#)
- [Running the converter from the command line](#)

## Notes:

1. Backup all files before converting them with this tool.
2. Cancelling a conversion will not rollback the changes.
3. Comments within the APPLET tag are ignored.

## Running the GUI version of the HTML Converter

The HTML Converter is contained in the SDK, not the JRE. To run the converter, go to the `lib` subdirectory of your SDK installation directory. For example, if you installed the SDK on Windows on the C drive, then `cd` to

```
C:\sdk1.4.2\lib\
```

The converter (`htmlconverter.jar`) is contained in that directory.

To launch the converter type:

```
C:\sdk1.4.2\lib>..\bin\java -jar htmlconverter.jar -gui
```

Launching the converter on UNIX/Linux is analogous.

## Windows Alternative

To launch the converter using Window Explorer, navigate to the following directory:

```
C:\j2sdk1.4.2\bin
```

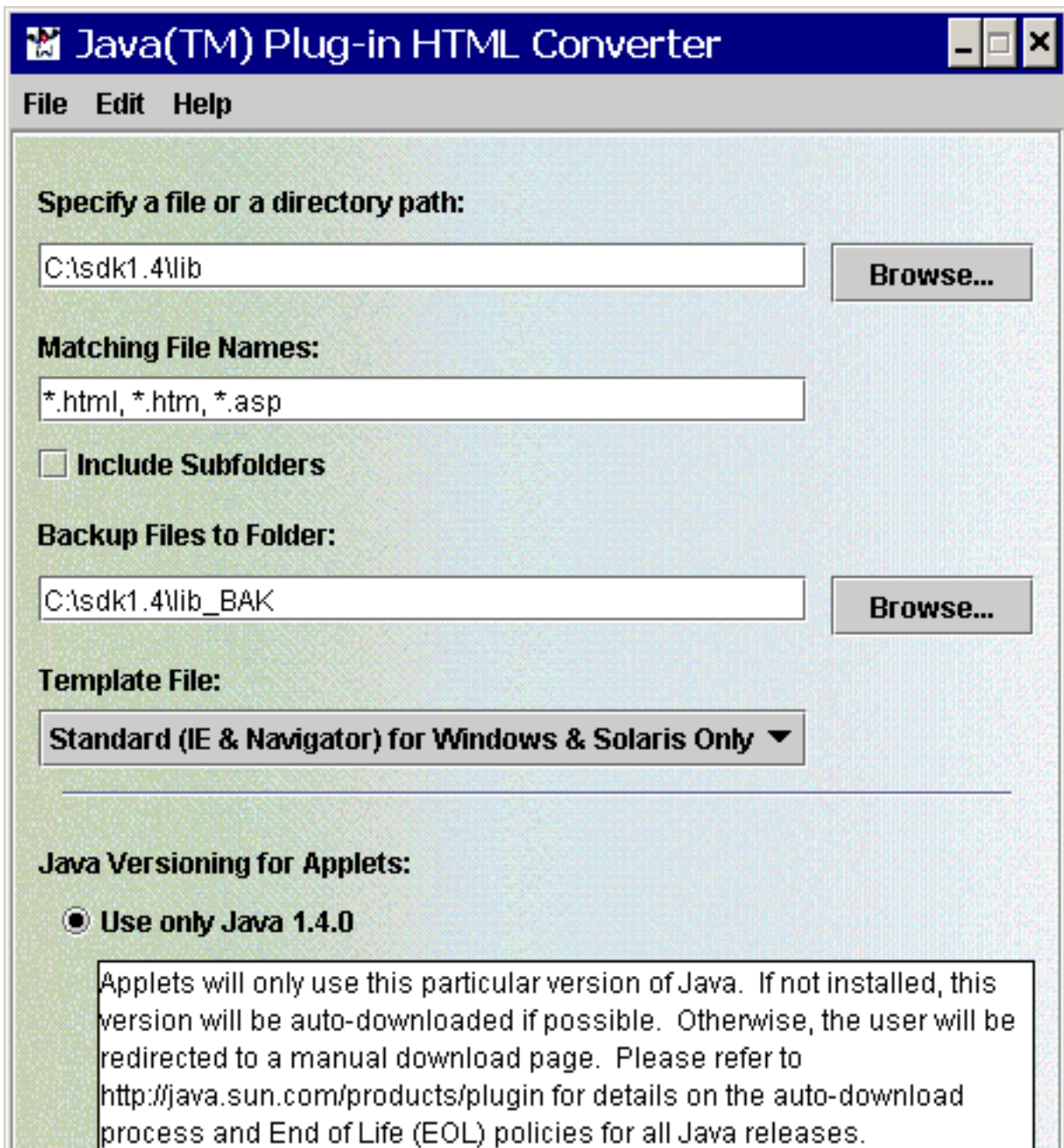
(or to `<sdk_location>\bin` if it is not located in the directory shown above) then double-click on the `HtmlConverter.exe` application.

## Unix/Linux Alternative

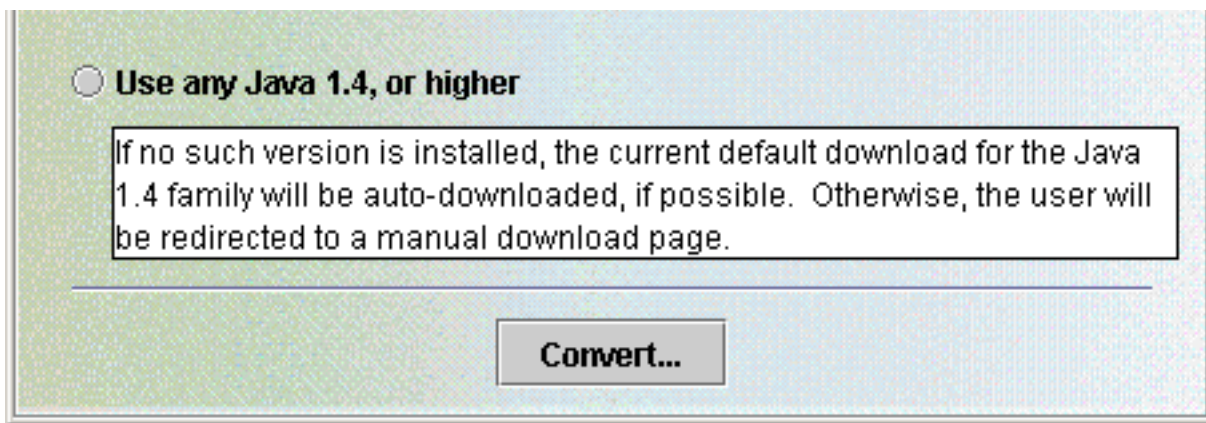
cd to `<sdk_location>/bin`—for example `/home/jones/j2sdk1.4.2/bin`—then enter:

```
HtmlConverter -gui
```

The HTML Converter window will appear:







## Choosing files within folders to convert:

To convert a *single file within a folder*, you may specify the path to the file and include the file name; you may also browse to the file and select it. To convert *all files within a folder*, you may type in the path to the folder, or choose the browse button to select a folder from a dialog. Once you have chosen a path, you may supply any number of file specifiers in "Matching File Names". Each specifier must be separated by a comma. You may use \* as a wildcard. Finally, if you would like all files in nested folders matching the file name to be converted, select the checkbox "Include Subfolders".

## Choosing a backup folder:

### Windows

The default backup folder path is the source path with an "\_BAK" appended to the name; e.g., if the source path is `c : /html`, then the backup path would be `c : /html_BAK`. The backup path may be changed by entering a path in the field labeled "Backup Files to Folder:", or by browsing for a folder.

### Unix (Solaris):

The default backup folder path is the source path with an "\_BAK" appended to the name; e.g., if the source path is `/home/user1/html`, then the backup path would be `/home/user1/html_BAK`. The backup path may be changed by typing a path in the field labeled "Backup Files to Folder:", or by browsing for a folder.

## Generating a log file:

If you would like a log file to be generated, go to the Advanced Options screen (Edit>Options) and check "Generate Log File". Enter a path in the text field, including the name of the log file; or use the browse button to set the path (including log file). The log file may be an existing one or a new one. The log file contains basic information related to the conversion process.

## Choosing a conversion template:

A default template will be used if none is chosen. This template will produce converted html files that will work with IE and Netscape. If you would like to use a different template, you may choose it from the menu on the main screen. If you choose "Other Template ..." from the menu, you will be allowed to choose a file to be used as the template. If you choose a file, be sure that it is a template.

## Choosing a version scheme

**Note:** The *specified version* mentioned below is the version of the JRE you use to launch the converter; e.g., 1 . 4 . 2. The first two numbers in the version indicate the *family*; e.g., 1 . 4 . 3\_02 is in the 1 . 4 family. For an explanation of product version numbers, see [Note on Version Numbers](#) in the section called [Using OBJECT, EMBED and APPLETTags in Java Plug-in](#).

There are two choices here:

1. You can choose to have conversion tags that will require the *specified version* of Java Plug-in to run your applets. If the specified version is not installed, then the client will be asked if he/she wants to install and download it.
2. You can choose conversion tags that will allow any installed version of Java Plug-in in the *family*, equal to or higher than the *specified version*, to run your applets. If there is no installed version of Java Plug-in, or no installed version in the family equal to or higher than the *specified version*, then the client will be asked if he/she wants to install and run the latest version of Java Plug-in in the family.

## Converting:

Click the "Convert..." button to begin the conversion process. A dialog will show the files being processed, the number of files processed, the number of applets found, and number of errors.

## Exit or Convert More Files:

When the conversion is complete, the button in the process dialog will change from "Cancel" to "Done". You may choose "Done" to close the dialog. You can then exit the Java Plug-in HTML Converter program, or select another set of files to convert.

## Details about templates:

The template file is the basis behind converting applets. It is simply a text file containing tags that represent parts of the original applet. By adding/removing/moving the tags in a template file, you can alter the output of the converted file.

### *Supported Tags:*

|                    |  |
|--------------------|--|
| \$OriginalApplet\$ | This tag is substituted with the complete text of the original applet. |
|--------------------|--|

|                      |   |
|----------------------|---|
| \$AppletAttributes\$ | This tag is substituted with all of the applets attributes (code, codebase, width, height, etc.).                     |
| \$ObjectAttributes\$ | This tag is substituted with all the attributes required by the object tag.   |
| \$EmbedAttributes\$  | This tag is substituted with all the attributes required by the embed tag.  |
| \$AppletParams\$     | This tag is substituted with all the applet's <param . . . > tags   |
| \$ObjectParams\$     | This tag is substituted with all the <param. . . > tags required by the object tag.                                   |
| \$EmbedParams\$      | This tag is substituted with all the <param. . . > tags required by the embed tag in the form name=value              |
| \$AlternateHTML\$    | This tag is substituted with the text in the no-support-for-applets area of the original applet                       |
| \$CabFileLocation\$  | This is the URL of the cab file that should be used in each template that targets IE.                                 |
| \$NSFileLocation\$   | This is the URL of the Netscape plugin to be used in each template that targets Netscape.                             |
| \$SmartUpdate\$      | This is the URL of the Netscape SmartUpdate to be used in each template that targets Netscape Navigator 4.0 or later. |
| \$MimeType\$         | This is the MIME type of the Java object.   |

Below are four templates that come with the HTML Converter. You can make up others and put them in the template folder to use them.

**default.tpl**— the default template for the converter. The converted page can be used in IE and Navigator on Windows to invoke Java Plug-in. This template can also be used with Netscape on Unix (Solaris)

```
<!-- HTML CONVERTER -->
<OBJECT classid="clsid:E19F9330-3110-11d4-991C-005004D3B3DB"
$ObjectAttributes$ codebase="$CabFileLocation$" >
$ObjectParams$
<PARAM NAME="type" VALUE="$MimeType$" >
<PARAM NAME="scriptable" VALUE="false" >
$AppletParams$
<COMMENT>
<EMBED type="$MimeType$" $EmbedAttributes$
$EmbedParams$ scriptable=false
```

```

pluginspage="$NSFileLocation$"><NOEMBED>
</COMMENT>
$AlternateHTML$
</NOEMBED></EMBED>
</OBJECT>

<!--
$ORIGINALAPPLET$
-->

```

**ieonly.tpl** — the converted page can be used to invoke Java Plug-in in IE on Windows only.

```

<!-- HTML CONVERTER -->
<OBJECT classid="clsid:E19F9330-3110-11d4-991C-005004D3B3DB"
$ObjectAttributes$ codebase="$CabFileLocation$" >
$ObjectParams$
<PARAM NAME="type" VALUE="$MimeType$" >
<PARAM NAME="scriptable" VALUE="false" >
$AppletParams$
$AlternateHTML$
</OBJECT>

<!--
$ORIGINALAPPLET$
-->

```

**nsonly.tpl** — the converted page can be used to invoke Java Plug-in in Navigator on Windows and Solaris.

```

<!-- HTML CONVERTER -->
<EMBED type="$MimeType$" $EmbedAttributes$
$EmbedParams$ scriptable=false
pluginspage="$NSFileLocation$"><NOEMBED>
$AlternateHTML$
</NOEMBED></EMBED>

<!--
$ORIGINALAPPLET$
-->

```

**extend.tpl** — the converted page can be used in any browser and any platform. If the browser is IE or Navigator on Windows, or Navigator on Solaris, Java(TM) Plug-in will be invoked. Otherwise, the browser's default JVM is used.

```

<!-- HTML CONVERTER -->
<SCRIPT LANGUAGE="JavaScript"><!--
var _info = navigator.userAgent; var _ns = false; var _ns6 = false;

```

```

var _ie = (_info.indexOf("MSIE") > 0 && _info.indexOf("Win") > 0 &&
_info.indexOf("Windows 3.1") < 0);
//--></SCRIPT>
<COMMENT><SCRIPT LANGUAGE="JavaScript1.1"><!--
var _ns = (navigator.appName.indexOf("Netscape") >= 0 &&
((_info.indexOf("Win") > 0 && _info.indexOf("Win16") < 0 &&
java.lang.System.getProperty("os.version").indexOf("3.5") < 0) ||
_info.indexOf("Sun") > 0));
var _ns6 = ((_ns == true) && (_info.indexOf("Mozilla/5") >= 0));
//--></SCRIPT></COMMENT>

<SCRIPT LANGUAGE="JavaScript"><!--
if (_ie == true) document.writeln('<OBJECT
classid="clsid:E19F9330-3110-11d4-991C-005004D3B3DB"
$ObjectAttributes$
codebase="$CabFileLocation$" ><NOEMBED><XMP>');
else if (_ns == true && _ns6 == false) document.writeln('<EMBED
type="$MimeType$" $EmbedAttributes$
$EmbedParams$ scriptable=false
pluginspage="$NSFileLocation$" ><NOEMBED><XMP>');
//--></SCRIPT>
<APPLET $AppletAttributes$></XMP>
$ObjectParams$
<PARAM NAME="type" VALUE="$MimeType$" >
<PARAM NAME="scriptable" VALUE="false" >
$AppletParams$
$AlternateHTML$
</APPLET>
</NOEMBED></EMBED></OBJECT>

<!--
$ORIGINALAPPLET$
-->

```

## Running the converter from the command line:

### Format:

```

java -jar htmlconverter.jar [-options1 value1 [-option2 value2 [...]]]
[-simulate] [filespecs]

```

If only "java -jar htmlconverter.jar -gui" is specified (only -gui option with no filespecs), the GUI version of the converter will be launched. Otherwise, the GUI will be suppressed.

filespecs:

space-delimited list of file specifications which may include wildcard (\*), e.g. \*.html, file\*.html).

-simulate:

Set to preview a conversion without actually doing the conversion. Use this option if you are unsure about a conversion. You will be shown detail information about the conversion had it been done.

| <b>Options:</b> | <b>Description</b>   |
|-----------------|--|
| -source         | Path to files; e.g., c:\htmldocs in Windows, /home/user1/htmldocs in Unix. Default is <userdir><br>If the path is relative, it is assumed to be relative to the directory from which the HTMLConverter was launched. |
| -dest           | Path to converter file location. Default: <usrdir>   |
| -backup         | Path to the directory where you want backup files to be stored. Default: <source>_BAK<br>If the path is relative, it is assumed to be relative to the directory from which the HTMLConverter was launched.           |
| -f              | Force overwriting of backup files.   |
| -subdirs        | Sets whether files in subdirectories should be converted or not. Default: false  |
| -template       | Name of template file to use for conversion. Default: Standard (IE & Navigator) for Windows & Solaris Only.<br><br><b>Note:</b> Use the default if you are unsure.   |
| -log            | Path and filename for the log. If not provided, no log file is written.  |
| -progress       | Set to display standard out progress during conversion. Default: true  |
| -latest         | Use the latest JRE supporting the MIME type.   |
| -gui            | Display the graphical user interface for the converter.  |

---

# Appendix 4: Microsoft VM and Java 2 Applet Compatibility Issues

---

This appendix documents the known applet compatibility issues between the Microsoft VM (Virtual Machine) and Sun's Java 2 VM.

## **ClassFormatError**

This problem is caused by bytecode generated from old JDK 1.0.2 or 1.1 compilers, which generate bytecode that does not conform to the Java VM Specification. Since the verifiers in recent J2SE releases are much stricter about bad class format, `ClassFormatError` is thrown by the VM when these bad class files are loaded.

To allow some applets with bad class file format to run in Java 2, a bytecode transformer is put into the Java Plug-in to transform some of the bad class files to good ones. Currently only bad class files with the following `ClassFormatError` may be transformed:

- Local variable name has bad constant pool index
- Extra bytes at the end of the class file
- Code segment has wrong length
- Illegal field/method name
- Illegal field/method modifiers
- Invalid `start_pc/length` in local var table

Unfortunately, the following `ClassFormatError` is not fixable by the bytecode transformer, so any class file that has any of these known problems will not run under Java 2:

- Illegal use of nonvirtual function call
- Arguments can't fit into locals
- Unsorted lookup switch
- Truncated class file

## **Security exception with `sun.audio`**

The `sun.audio` package was accessible by applets in JDK 1.1. However, the applet sandbox was closed up in Java 2, so that applets that try to access any class libraries in the `sun.audio` package will result in a `SecurityException`.

To provide maximum applet compatibility, the applet sandbox in Java Plug-in has been opened up to allow applets to access the `sun.audio` package again.

## Unable to find resources from `AppletContext.getAudioClip()` and `AppletContext.getImage()`

The resource-lookup sequence in `AppletContext.getImage()` and `AppletContext.getAudioClip()` is different in the Microsoft and Sun implementations.

In the Microsoft VM, resources are looked up in the following sequence:

1. Archives specified through HTML `archive` or `codebase` parameters
2. `codebase` URL

In Sun's implementation, resources are looked up according to `codebase` URL.

As a result, some applets that rely on the resources-lookup sequence in the Microsoft VM may not load resources properly in Java 2.

To provide maximum applet compatibility, the resources-lookup sequence in Java Plug-in has been changed to the following:

1. Archives specified through HTML `archive` parameters
2. `codebase` URL

## ClassLoader sharing policy

`ClassLoader` sharing policy is different in the Microsoft and Sun implementations.

In the Microsoft VM, the `ClassLoader` object is shared between applets if and only if all of the following are true:

- `codebase` values are the same
- `archive` values are the same
- `codebase` values are the same

In Sun's implementation, a `ClassLoader` object is shared between applets if and only if:

- `codebase` values are the same

Some applets relying on the `ClassLoader` sharing policy in the Microsoft VM may not run properly in Java 2 because of potential class definition conflicts.

To provide maximum applet compatibility, the `ClassLoader` sharing policy in Java Plug-in has been changed to the following requirements:

- `codebase` values are the same and
- `cache_archive` values are the same and
- `java_archive` values are the same and
- `archive` values are the same



## Security model—Java 2 Versus Microsoft

Java 2 has a new security model providing much more capability and flexibility than JDK 1.1 provides. The Microsoft VM security model is based on JDK 1.1 and its own propriety technologies.

This issue is not fixable, so applets that rely on the Microsoft's security model will not run properly in Java 2.

## Applet packaging

Applet packaging in Java 2 and JDK 1.1 is through the `.jar` file. However, Microsoft's VM supports applet packaging through `.jar` file and its own propriety `.cab` (cabinet) file.

This issue is not fixable, so applets packaged in Microsoft's `.cab` file format will not load in Java 2.

## Java language specification strictness (`null` versus zero-length `String`)

In JDK 1.1, the Java language specification was loose in dealing with `null` and zero-length strings in the class libraries. Some APIs may treat `null` as a zero-length `String`, while some other APIs may treat `null` as it is. In Java 2, the Java language specification has been tightened up to specify what the exact behavior should be.

This issue is not fixable, so applets that rely on the APIs to treat `null` as a zero-length `String` may result in an exception in Java 2.

## Applet signing—RSA versus Authenticode

In Java 2, applet signing is supported through RSA and the `.jar` file, while Microsoft supports applet signing through its own proprietary Authenticode and `.cab` technologies.

This issue is not fixable, so signed applets that rely on Microsoft's Authenticode and `.cab` technologies may not load properly in Java 2.

## Implementation changes in AWT class libraries

In JDK 1.1, AWT class libraries were used by developers as thread-safe libraries; i.e., applets may perform lots of actions through the AWT in multiple threads and assume the class libraries will take care of synchronization issues. In Java 2, the AWT class libraries implementation has been changed to guarantee thread safety only when it is called by the AWT event-dispatch thread.

This issue is not fixable, so applets in Java 2 that make calls to AWT class libraries without awareness of the thread-safety issue may result in deadlock or race conditions.

## Java/JavaScript communication

In the Microsoft implementation, applet methods and properties exposed in JavaScript in an HTML page are exactly the same as the methods/fields of the applet object. In Java Plug-in, applet methods and properties exposed in JavaScript in HTML are obtained via JavaBeans Introspection, which analyzes methods and properties through naming convention in the applet object. The side effect is that applet fields are treated differently.

This problem will be fixed in future release of Java Plug-in. In the meantime, JavaScript accessing fields in the applet object may not work properly in Java 2.

## Microsoft proprietary Java class libraries

Microsoft has provided lots of proprietary class libraries in its VM implementation, including J/Direct, AFC, WFC, etc.

This issue is not fixable, so applets that rely on any of the Microsoft proprietary Java class libraries will not work properly in Java 2.

## Whitespace characters in string returned from `Applet.getParameter()`

In the Microsoft implementation, whitespace characters are stripped off before the string is returned to the applets in `Applet.getParameter()`. However, the Sun implementation returns the string as it is specified in the HTML parameters. As a result, some JDK 1.1 applets refuse to run in Java 2 because the applet's logic doesn't take the whitespace into account.

To provide maximum applet compatibility, the implementation of `Applet.getParameter()` in Java Plug-in has been changed to strip off whitespace characters in the return value.

## Design changes in `java.util.Hashtable.hashCode()`

In JDK 1.1, `Hashtable.hashCode()` was implemented based on object identity, so that each `Hashtable` object returns its unique value when `hashCode()` is called. In Java 2, implementation of `Hashtable.hashCode()` was changed to be value-based as part of the Java Collection Framework. `Hashtable` object returns its hashcode value based on the objects it contains.

This change breaks some JDK 1.1 applets if they add `Hashtable` object into itself, as this breaks the fundamental design of the Collection Framework and causes a `StackOverflowError`. It breaks the logic in the code of some applets that rely on `Hashtable.hashCode()` to return a constant value from the same `Hashtable` object.

To provide maximum applet compatibility, the implementation of `Hashtable.hashCode()` is changed to check for this special case and avoid stack overflow.

## Accessing frame from applet

To track mouse events or for other reasons, an applet may sometimes try to access to its frame. In the Microsoft and Sun implementations, the number of containers between the frame and the applet is different.

An applets that relies on the location of its frame being at a particular level of containment in the Microsoft VM (without navigating the entire AWT hierarchical component tree) is likely to fail in Java 2. The most common symptom is `ClassCastException` from the AWT Dispatch Event Thread.

This issue is not fixable; thus an applets affected by this issue may not run properly in Java 2.

## MAYSCRIPT

In Netscape Navigator and Java Plug-in, an applet accessing JavaScript is required to specify the `MAYSCRIPT` parameter in the applet tag. However, the Microsoft implementation doesn't honor this parameter and allows applets to access JavaScript under all conditions. Since most Internet applets target the Microsoft VM, `MAYSCRIPT` is not specified in these applets.

To provide maximum applet compatibility, the `MAYSCRIPT` check has been removed from Java Plug-in.

## HTTP User-Agent

In the Microsoft and Sun implementations, different HTTP User-Agent strings are passed to the server when an HTTP connection is requested. Since most web sites target the Microsoft VM instead of Sun's, these web sites do not recognize Sun's HTTP User-Agent, and this may result in failure.

As a result, HTTP User-Agent used in Java Plug-in has been changed to be similar to the one in the browser; thus most web servers will recognize requests made from applets running in Java Plug-in.

## Events during applet startup and shutdown

In the Microsoft and Sun implementations, the events occurring during applet startup and shutdown may not be exactly the same. For example, the logic in the applet may rely on the applet being visible when `Applet.start()`/`Applet.stop()` is called, which may be true in Microsoft's implementation but not in Sun's.

Applets that rely on specific events during applet startup or shutdown in Microsoft's VM may not function properly in Java 2. The most common symptom is `NullPointerException` from the AWT Dispatch Event Thread.

This issue is not fixable.

## Java class libraries compatibility

There are lots of changes in the Java class libraries in Java 2. Some APIs are clarified, some are deprecated, and some other have been altered in implementation. These following have caused some of the applets run in Microsoft's VM to fail in Java 2:

### **java.awt.Graphics.drawString( )**

`drawString( )` treats `null` as an empty string in the Microsoft VM. In Java 2, `drawString( )` treats `null` as it is and throws `NullPointerException`.

### **java.awt.Graphics.drawImage( )**

`drawImage( )` ignores `null` image in the Microsoft VM. In Java 2, `drawImage( )` treats `null` as it is and throws `NullPointerException`.

### **java.awt.Color constructors**

In the Microsoft VM, passing over-bound/under-bound values in the `Color` constructor will cause the VM to print out a warning message in the console, but the values will be reset to max/min automatically. In Java 2, `Color` constructor checks for illegal values and throws `IllegalArgumentException`.

### **Thread.stop( ), Thread.suspend( ), Thread.resume( )**

In the Microsoft VM, stopping, suspending, or resuming a dead thread will cause the call to be ignored. In Java 2, calling these methods on a dead thread will result in `AccessControlException`.

All of these issues are fixable but have not yet been fixed. At this time, applets affected by these issues will result in exceptions and may not run properly in Java 2.

---

# Appendix 5: Complete Example—Deploying Java Media Framework as Java Extension

---

This section covers the following topics:

- [Introduction](#)
- [Creating the jar File to Be Installed](#)
- [Creating the Applet jar File](#)
- [Creating the HTML for the Applet](#)
- [Testing the Example](#)

## Introduction

This is a complete, working example showing how to deploy a single jar file from Java Media Framework (JMF) as a Java Extension. The example uses `SimplePlayerApplet.java` and `jmf.jar` to play an `.avi` media file. It uses the raw installation method for installing a repackaged and signed version of `jmf.jar` called `s_my_jmf.jar`. Normally there are other jar files that are installed with JMF, but for `SimplePlayerApplet.java` only the functionality in `jmf.jar` is required.

For simplicity sake, this example makes the following assumptions:

- You are working on a Microsoft Windows system.
- You have intalled the 1.4 SDK in the following location on your computer: `C:\j2sdk1.4.0`
- For signing purposes you have set up a keystore in the following directory: `C:\plugin\keystores`
- The keystore name is `thawte.p12`
- The `storepass` and the `keypass` are the same: `mypass`
- The keystore alias is "Sun Microsystems Inc.'s Thawte Consulting cc ID"

## Creating the jar File to Be Installed

In this case there is no installer. All you need to do is obtain the required jar file, `jmf.jar`, that needs to be downloaded and copied into `<jre_location>/lib/ext`. Plug-in, in conjunction with the extension mechanism in the JRE, handles the installation (downloading and copying of the file) for you.

You can get `jmf.jar` by downloading the cross-platform installation zip file `jmf-2_1_1a-alljava.zip` from <http://java.sun.com/products/java-media/jmf/2.1.1/download.html>. Along with other jar files, the zip file contains `jmf.jar`, which you can extract from the zip.

Once you have obtained `jmf.jar`, you will want to extract `jmf.jar` itself into some directory, say `C:\plugin\extensions\workspace1`. Here you will want to delete the `META-INF` directory, as the `manifest.mf` file contains signing information that you do not want.

Next you need to create your own manifest file for the new jar file to be based on `jmf.jar`. The manifest file that we create we call `jmf_manifest`. It will be provided as input to the `jar` tool. Here is what is used in this example:

```
Extension-Name: javax.media.s_my_jmf
Specification-Vendor: Sun Microsystems, Inc
Specification-Version: 2.1
Implementation-Vendor-Id: com.sun
Implementation-Vendor: Sun Microsystems, Inc
Implementation-Version: 2.1.1
```

First we will jar the files in `jmf.jar` and rename the result `my_jmf.jar`. Then we will sign the result and call it `s_my_jmf.jar`.

In order to jar the files in `workspace1` with our new manifest file `jmf_manifest`, we `cd` to the locations of `workspace1`, then we use the `jar` tool in the SDK as follows:

```
C:\plugin\extensions\workspace1>C:\j2sdk1.4.0\bin\jar cmf jmf_manifest my_jmf.jar
```

```
*.class codecLib com javax jmapps
```

Note that codecLib, com, javax, and jmapps are subdirectories that must be jar'd as well \*.class.

In this example we use the jarsigner tool to sign the new jar file.

Assuming that we have a Thawte keystore called thawte.p12 located in C:\plugin\keystores with the same password, mypass, for both storepass and keypass, and storetype is "pkcs12" and the keystore alias is "Sun Microsystems Inc.'s Thawte Consulting cc ID", then we can sign my\_jmf.jar as follows, creating a signed jar file called s\_my\_jmf.jar:

```
C:\plugin\extensions\workspace1>C:\j2sdk1.4.0\bin\jarsigner -keystore
C:\plugin\keystores\thawte.p12 -storepass mypass -keypass mypass -storetype "pkcs12"
-signedjar s_my_jmf.jar my_jmf.jar "Sun Microsystems Inc.'s Thawte Consulting cc ID"
```

We can verify the new signed jar file as follows:

```
C:\plugin\extensions\workspace1>C:\j2sdk1.4.0\bin\jarsigner -verify s_my_jmf.jar
```

We now have a signed jar file with the proper manifest.mf file for raw installation.

Next we need to create the applet jar file.

## Creating the Applet jar file

The applet consists of a single file, SimplePlayerApplet.class, that can be used to playback a media file. The source code for the applet can be viewed [here](#). What we need to do is create a manifest file called for the the applet, which we will call applet\_manifest, jar the applet with the manifest, then sign the result..

The applet applet\_manifest is as follows:

```
Extension-List: s_my_jmf
s_my_jmf-Extension-Name: javax.media.s_my_jmf
s_my_jmf-Specification-Version: 2.1
s_my_jmf-Implementation-Version: 2.1.1
s_my_jmf-Implementation-Vendor-Id: com.sun
s_my_jmf-Implementation-URL:
http://java.sun.com/products/plugin/extensions/examples/jmf/s_my_jmf.jar
```

Note that the above manifest says that the extension jar, s\_my\_jmf.jar, can be downloaded from the java.sun.com web server at http://java.sun.com/products/plugin/extensions/examples/jmf

If the SimplePlayerApplet.class and applet\_manifest are located in C:\plugin\workspace2, we can jar the applet with the manifest with the following command:

```
C:\plugin\extensions\workspace2>C:\j2sdk1.4.0\bin\jar cmf applet_manifest
my_SimplePlayerApplet.jar *.class
```

Again, we use jarsigner to sign the jar file:

```
C:\plugin\extensions\workspace2>C:\j2sdk1.4.0\bin\jarsigner -keystore
C:\plugin\keystores\thawte.p12 -storepass mypass -keypass mypass -storetype "pkcs12"
-signedjar s_my_SimplePlayerApplet.jar my_SimplePlayerApplet.jar "Sun Microsystems
Inc.'s Thawte Consulting cc ID"
```

and we verify it as follows:

```
C:\plugin\extensions\workspace2>C:\j2sdk1.4.0\bin\jarsigner -verify
s_my_SimplePlayerApplet.jar
```

We now have our signed applet jar file called s\_my\_SimplePlayerApplet.jar, whose manifest contains the correct information to trigger the installation of the required extension jar file, s\_my\_jmf.jar if no such file or an older version is found in <jre\_location>/lib/ext.

Next we need to create the HTML for the applet.

## Creating the HTML for the Applet

We have several choices. We can use the conventional `APPLET` tag and assume those who visit the page have Java Plug-in version 1.3.1\_01 or later installed on their systems. (To use Java Plug-in to launch an applet with the conventional applet tag requires 1.3.1\_01 or later.) We can also use the HTML Converter, located in the SDK in the `bin` directory (`<sdk_location>/bin/HtmlConverter.exe`) to convert the applet to various forms. Here we have chosen to do both: `SimplePlayerApplet-1.html` uses the conventional applet form; `SimplePlayerApplet-2.html` is a converted format for both the `OBJECT` and `EMBED` tags that assumes dynamic versioning (`clsid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"`) for the `OBJECT` tag and `type="application/x-java-applet;jpi-version=1.4"` for the `EMBED` tag for Plug-in 1.4.2).

The two forms are shown below:

### SimplePlayerApplet-1.html

```
<html>
<head>
<title>SimplePlayerApplet</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body bgcolor="#FFFFFF" text="#000000">
<applet code="SimplePlayerApplet.class" archive="s_my_SimplePlayerApplet.jar"
width=320 height=300>
<param name="file" value="0720crt1.avi">
</applet>
</body>
</html>
```

### SimplePlayerApplet-2.html

```
<html>
<head>
<title>SimplePlayerApplet</title>
</head>
<body bgcolor="#FFFFFF" text="#000000">
<!--"CONVERTED_APPLET"-->
<!-- HTML CONVERTER -->
<OBJECT
  classid = "clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
  codebase =
"http://java.sun.com/products/plugin/autodl/jinstall-1_4-windows-i586.cab#Version=1,4,0,0"
  WIDTH = 320 HEIGHT = 300 >
  <PARAM NAME = CODE VALUE = "SimplePlayerApplet.class" >
  <PARAM NAME = ARCHIVE VALUE = "s_my_SimplePlayerApplet.jar" >
  <PARAM NAME = "type" VALUE = "application/x-java-applet;version=1.4">
  <PARAM NAME = "scriptable" VALUE = "false">
  <PARAM NAME = "file" VALUE="0720crt1.avi">

<COMMENT>
<EMBED
  type = "application/x-java-applet;version=1.4"
  CODE = "SimplePlayerApplet.class"
  ARCHIVE = "s_my_SimplePlayerApplet.jar"
  WIDTH = 320
  HEIGHT = 300
  file = "0720crt1.avi"
  scriptable = false
  pluginspage = "http://java.sun.com/products/plugin/index.html#download">
</EMBED>

</NOEMBED>
</COMMENT>
</OBJECT>
```

```
<!--  
<APPLET CODE = "SimplePlayerApplet.class" ARCHIVE = "s_my_SimplePlayerApplet.jar"  
WIDTH = 320 HEIGHT = 300>  
<PARAM NAME = "file" VALUE="0720crt1.avi">  
  
</APPLET>  
-->  
  
<!--"END_CONVERTED_APPLET"-->  
  
</body>  
</html>
```

Note that the media file is 0720crt1.avi.

For this example the following files have been placed on the java.sun.com web server at <http://java.sun.com/products/plugin/extensions/examples/jmf>:

- 0720crt1.avi
- s\_my\_jmf.jar
- s\_mySimplePlayerApplet.jar
- SimplePlayerApplet-1.html
- SimplePlayerApplet-2.htm

## Testing the Example

You can test the setup by pushing either of the buttons below:

When you point your browser at the URL, the applet jar will first be downloaded and cached; and, if the extension has not already been installed, you will see a Java Security Warning dialog that says: "The applet requires the installation of optional package "javax.media.s\_my\_jmf" from

[http://java.sun.com/products/plugin/extensions/examples/jmf/s\\_my\\_jmf.jar](http://java.sun.com/products/plugin/extensions/examples/jmf/s_my_jmf.jar)'. You will have the options to **Grant this session**, **Deny**, or **Grant Always**. If you grant permission for installation, the extension will be installed in `<jre_location>/lib/ext` and the applet will run.



---

# Appendix 6: Sun-Supported Specification-Version and Implementation-Version Formats

---

Specification-Version and Implementation-Version follow these rules for Sun products, and third-party products must follow the same rules for Java Plug-in to make reliable decisions about whether an extension is up-to-date or not:

The Specification-Version string will be of the form:

`n1.n2[.n3]`

where `n1`, `n2`, and `n3` are integers, `n1.n2` is the *major version number*, and optional `n3` is the *minor version number* (also referred to as the *maintenance version number*).

The Implementation-Version will be of the same form initially but may:

- append with a preceding underscore("\_") `n4n5` to indicate a *patch version number* (also referred to as the *update version number*)
- or append with a hyphen("-") a milestone name (`ea`, `alpha`, `beta`, `rc` ...), which may also include a trailing integer number (`ea1`, `beta2`, `rc1` ...).

Both patch version number and milestone name may not be used together in the Implementation-Version string.

The general form is as follows:

`n1.n2[.n3][_<patch_number>|-<milestone_name>]`

Integers (`n1`, `n2`, `n3` ...), letters, dots, hyphens, and underscores, may be used in the version format as described above; no other characters ("`*`", "`+`" ...) are allowed.

Note that the Specification-Version and Implementation-Version numbers are in theory independent, though in practice they are often in sync with each other.

## Examples:

Specification-Version examples: `1.3`, `1.4`

Implementation-Version examples: `1.3.1`, `1.4.0_02`, `1.4.0-beta3`

While these are the rules, it is always a good idea when using a third-party extension JAR to examine its `MANIFEST.MF` file to see what actual values are in it. If it does not follow these rules, you may need to change your applet JAR `MANIFEST.MF` accordingly or alter the extension JAR.

