Java Platform, Standard Edition

Deployment Guide

Release 9

E61695-03

September 2017

Java Platform, Standard Edition Deployment Guide, Release 9

E61695-03

Contents

Preface

This guide provides information about building, packaging, and deploying your Java and JavaFX applications.

Audience

This document is intended for application developers who create Java or JavaFX applications and want to deploy them to users on remote systems.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

See Oracle JDK 9 Documentation for other JDK 9 guides.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Part I

Deployment Basics

The topic in this part provides general information about the deployment process and the tools that are available for deploying your Java and JavaFX applications.

1

Getting Started

This topic describes the basics of deploying your Java and JavaFX applications. This topic contains the following sections:

Basic Steps

Have an application ready to publish? Follow these steps for basic deployment:

1. Decide how you want users to access and run your application.

   Applications can be deployed on a user's desktop, embedded in a web page, or launched from a browser.

> **Note:**
>
> Although available and supported in JDK 9, the Applet API and the Java Plug-in are marked as deprecated in preparation for removal in a future release. Alternatives for applets and embedded JavaFX applications include Java Web Start and self-contained applications.

2. Create the application package.

   The application package consists of the following items:

   - JAR files needed to run your application. If your application is embedded in a web page or is launched from a browser, the JAR files must be signed with a valid signing certificate.
   - Deployment descriptor or JNLP file for applications that are embedded in a web page or are launched from a browser
   - JRE for self-contained applications. The Java packaging tools use the `jlink` tool to create a custom runtime, which reduces the size of the package.

3. Set up the web page, if your application is embedded in a web page or is launched from a browser.

   The web page needs either HTML elements or JavaScript code to run an application embedded in the page. JavaScript code is needed to launch an application from the browser using Java Web Start. The Java packaging tools generate an HTML file with JavaScript code for both types of execution, which you can copy into your web page.

4. Copy the package to the location from which you want users to access it.

   A web server is typically used for applications embedded in a web page or launched from a browser. Desktop and self-contained applications can be delivered directly to users or made available through an app store.

Choose the Execution Environment

Java and JavaFX applications can be run in multiple execution environments. How you want users to access your application determines how you deploy it. The following options are available:

- Launch as a native application

  Users can install a self-contained application and launch it from a menu or desktop shortcut.
- Launch as a desktop application

  Users can start the application from the command line using the Java launcher, or by double-clicking the JAR file for the application.
- Launch from a browser

  Users can download and start the application by clicking a link in the browser.
- View in a web page

  The application starts when the web page is loaded.

> **Note:**
>
> Although available and supported in JDK 9, the Applet API and the Java Plug-in are marked as deprecated in preparation for removal in a future release. Alternatives for applets and embedded JavaFX applications include Java Web Start and self-contained applications.

Each environment has advantages and disadvantages.

Create the Package

By default, the Java packaging tools generate the following collection of files needed to run the application:

- An application JAR file (or multiple JAR files for large applications)

  Applications that are embedded in a web page or launched from a browser must be signed with a valid signing certificate. The packaging tools or the `jar` command can be used to sign the JAR file.
- A JNLP file with a deployment descriptor

  A deployment descriptor is an XML file that describes application components, platform requirements, and launch rules.

- An HTML file containing JavaScript code to embed or launch applications from the web page

Applications can also be packaged as platform-specific, self-contained applications. Self-contained applications include all application resources, the JRE, and a launcher. These applications provide the same install and launch experience as native applications for the operating system.

Self-contained applications can be distributed as zip files or as installable packages: EXE or MSI for Windows; DMG, PKG, or mac.appStore for macOS; or RPM or DEB for Linux.

Self-contained applications provide the following benefits:

- They resemble native applications for the target platform.

  Users install the application with an installer that is familiar to them, and launch it in the usual way.
- They offer no-hassle compatibility.

  The version of JRE that is used by the application is controlled by the application developer.
- They are easily deployed on fresh systems with no requirement for the JRE to be installed.
- Deployment occurs with no need for admin permissions when using ZIP or user-level installers.

Self-Contained Application Packaging describes how to generate a self-contained application package.

Packaging Tools

Three different tools are available for packaging your application:

- NetBeans IDE
- Ant Tasks
- Java Packager Command-Line Tool

NetBeans IDE

If you use Netbeans IDE, then much of the work is done for you. Open **Project Properties** to specify preferred dimensions for your JavaFX application scene. Build the project with **Clean and Build**. The application package is generated in the `dist` folder. To test your application, open this folder and double-click the HTML, JNLP, or JAR file, depending on your execution environment.

To package a self-contained application, customize the `build.xml` script in the NetBeans IDE. See Basic Build.

Ant Tasks

If you are using another IDE, then you can add Java packaging as a post-build step, using Ant tasks that are included in the JDK. Example 1-1 shows an Ant package task for the JavaFX example Colorful Circles. Download the ColorfulCircles.zip file for the complete Colorful Circles example.

When you add the `nativeBundles="all"` attribute into the `<fx:deploy>` Ant task, all possible packages are created: a standalone application package, one or more self-contained application packages for the platform on which you are running, and a web deployment package. Installable packages are created based on the third-party software that is available at packaging time. For example, if you have both Inno Setup and WiX on Windows, then you get three packages: a folder with the application, an .exe installer file, and an .msi installer file. A simple Ant task with the `nativeBundles` attribute is shown in Example 1-1.

If your application will be embedded in a web page or launched from a browser, include the required JAR manifest attributes in the `<fx:jar>` task and include an `<fx:signjar>` task.

Note:

The `<fx:signjar>` task for the Java Packager tool is deprecated in JDK 9 in preparation for removal in a future release. It also does not work with multi-release JAR files. Use the standard Ant `signjar` task instead.

Example 1-1 Ant Task to Produce All Packages for the ColorfulCircles Application

```
<taskdef resource="com/sun/javafx/tools/ant/antlib.xml"
        uri="javafx:com.sun.javafx.tools.ant"
        classpath="${JAVA_HOME}/lib/ant-javafx.jar"/>

<fx:jar destfile="dist-web/ColorfulCircles.jar">
    <fx:application mainClass="colorfulcircles.ColorfulCircles"/>
    <fileset dir="build/classes/">
        <include name="**"/>
    </fileset>
</fx:jar>

<fx:deploy width="800" height="600" outdir="dist-web"
        outfile="ColorfulCircles" nativeBundles="all">
    <fx:info title="Colorful Circles"/>
    <fx:application name="Colorful Circles example"
            mainClass="colorfulcircles.ColorfulCircles"/>
    <fx:resources>
        <fx:fileset dir="dist-web" includes="ColorfulCircles.jar"/>
    </fx:resources>
</fx:deploy>
```

Java Packager Command-Line Tool

If you cannot use Ant or prefer command-line tools, use the Java Packager tool that comes with the JDK. The Java Packager tool has several commands for packaging applications, see javapackager in the *Java Platform, Standard Edition Tools Reference*.

For a quick test build, you can use the `javapackager -makeall` command. This command compiles source code and combines the `javapackager -createjar` and `javapackager -deploy` commands with simplified options, as shown in the following example.

```
javapackager -makeall -appclass colorfulcircles.ColorfulCircles
    -name "Colorful Circles" -width 800 -height 600
```

Note:

The `-makeall` command for the Java Packager tool is deprecated in JDK 9 in preparation for removal in a future release.

As a command intended only to help to build simple projects quickly, the `-makeall` command supports a limited set of options to customize the command behavior. The `-makeall` command makes the following assumptions about input and output files:

- Source and other resource files must be in a directory named `src` under the main project directory.
- The resulting package is always generated to a directory named `dist`, and file names all start with the dist prefix.
- By default, the `-makeall` command tries to build a self-contained application package. If this is not possible, the JAR, HTML, and JNLP files are generated so you can deploy to other execution modes.

Note:

For JavaFX applications that are embedded in a web page, stage width and height must always be specified.

When your application is ready to go live, use the `-createjar` and `-deploy` commands instead of the `-makeall` command. The `-createjar` and `-deploy` commands have considerably more options. You can create a self-contained application package with the `-deploy` command plus the `-native` option, for example:

```
javapackager -deploy -native -outdir packages -outfile ColorfulCircles
    -srcdir dist -srcfiles ColorfulCircles.jar
    -appclass colorfulcircles.ColorfulCircles
```

Tip:

Ant tasks provide more flexibility of options than the Java Packager tool.

Create the Web Page

If your application is embedded in a web page or launched from a browser, you need to set up the web page that provides users with access to your application. The Java Plug-in is used to run an application embedded in the web page. Java Web Start is used to run an application that is launched from the browser.

Note:

Although available and supported in JDK 9, the Applet API and the Java Plug-in are marked as deprecated in preparation for removal in a future release. Alternatives for applets and embedded JavaFX applications include Java Web Start and self-contained applications.

You can use an `<applet>` or `<object>` element or JavaScript code for applications that are embedded in a web page and use the Java Plug-In to run. Use JavaScript code to create the link or button that calls Java Web Start to launch an application from the browser. The Java packaging tools generate JavaScript code for both types of execution, which you can copy into your web page.

The HTML page generated by the packaging tools is a simple test page for your application. It includes sample JavaScript code to launch and embed your application, which you can copy to your own web page. To avoid manual copying, consider using HTML templates for application packaging to insert the JavaScript code into an existing web page.

Distribute Your Application

When you have your application package and any web pages that you are using, copy them to the appropriate location to make your application available to users.

- If your application is embedded in a web page or launched from a browser, copy your package and the web page to the web server from which they will be loaded.
- If your application is a desktop application, copy the application to the location from which users will download it. Self-contained applications provide installable packages and the required JRE, which makes it easier for users to install and run your application.

Beyond the Basics

This topic provides the minimum information needed to deploy a simple application, which makes use of the default processing that is provided by the packaging tools. More advanced applications could have additional requirements, for example:

- Include a custom splash screen that is shown when your application is loaded.
- Include a custom progress bar that is shown while your application is loading.
- Use JavaScript code to communicate between your application and the web page in which it is embedded.
- Deploy Swing and SWT Applications with Embedded JavaFX Content
- Minimize the number of security dialogs and the warnings contained within the dialogs to help ensure users that it is safe to run your application.

The Deployment trail in the Java Tutorials also provides information to help you deploy applications embedded in a web page or launched from a browser.

Part II

Packaging

The topics in this part describe the packaging process and the packaging tools that are available.

- Self-Contained Application Packaging
- JavaFX Ant Tasks

2

Self-Contained Application Packaging

This topic describes how to generate the package for a self-contained application. A self-contained application contains your Java or JavaFX application and the JRE needed to run the application. Self-contained packages can be created for distribution to systems running Linux, macOS, and Windows.
This topic includes the following sections:

Introduction

The Java packaging tools provide built-in support for several formats of self-contained application packages. The basic package is a single folder on your hard drive that includes all application resources and the JRE. The package can be redistributed as is, or you can build an installable package (for example, EXE or DMG format.)

From the standpoint of process, producing a self-contained application package is similar to producing a basic application package, with the following differences:

- Self-contained application packages must be explicitly requested by passing the `native` argument to the `<fx:deploy>` Ant task or `javapackager -deploy` command.
- Self-contained application packages must be built on the operating system on which it is intended to run. Prerequisite tools must be available to build a package in a specific format.
- Self-contained application packages can only be built using JDK 7 Update 6 or later. The Java Packager for JDK 9 packages applications with a JDK 9 runtime image. To package a JDK 8 or JDK 7 JRE with your application, use the JDK 8 Java Packager.

While it is easy to create a basic self-contained application package, tailoring it to achieve the best user experience for a particular distribution method usually requires some effort and a deeper understanding of the topic.

Benefits and Drawbacks of Self-Contained Application Packages

Deciding whether the use of self-contained application packages is the best way to deploy your application depends on your requirements.

Self-contained application packages provide the following benefits:

- Users install the application with an installer that is familiar to them and launch it in the usual way.
- You control the version of the JRE used by the application.
- Applications can be deployed on fresh systems with no requirement for the JRE to be installed.
- Deployment occurs with no need for admin permissions when using ZIP or user-level installers.
- File associations can be registered for the application.
- Support for secondary launchers enables a suite of applications to be bundled in a single self-contained application package.

Self-contained application packages have the following drawbacks:

- "Download and run" user experience

  Unlike web deployment, the user experience is not about "launch the application from the web." It is more one of "download, install, and run" process, in which the user might need to go through additional steps to get the application launched. For example, the user might have to accept a browser or operating system security dialog, or find and launch the application installer from the download folder.
- Larger download size

  In general, the size of self-contained application packages is larger than the size of a standalone application, because a private copy of the JRE is included.
- Package per target platform

  Self-contained application packages are platform-specific and can only be produced for the same system on which you build. To deliver self-contained application packages on Windows, Linux, and macOS, you must build your project on all three platforms.
- Application updates are the responsibility of developer

  Web-deployed Java applications automatically download application updates from the web as soon as they are available. The Java Autoupdate mechanism takes care of updating the JRE to the latest secure version several times every year. Self-contained applications do not have built-in support for automatic updates.

Basics

Each self-contained application package includes the following items:

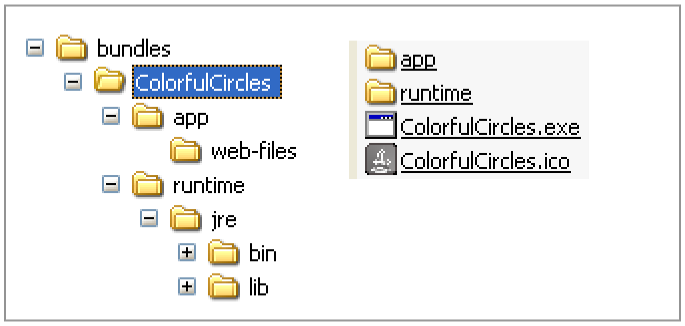
- Application code in a set of JAR files, plus any other application resources (data files, native libraries)
- Copy of the JRE, to be used by this application only
- Native launcher for the application, multiple launchers for a single package are supported
- Metadata, such as icons

Multiple package formats are possible. Built-in support is provided for several types of packages. You can also assemble your own packages by post-processing a self-contained application packaged as a folder, for example, if you want to distribute your application as a ZIP file.

Self-Contained Application Structure

The basic form of a self-contained application package is a single folder on your hard drive, such as the example in Figure 2-1. When any of the packages are installed, the result is a folder with the same content.

Figure 2-1 Example of a Self-Contained Application Package



The internal structure of a self-contained application folder is platform-specific, and might change in the future. However, the following items apply to all platforms and are not likely to change:

- The application package is included as a folder, preserving the application directory structure.
- A copy of the JRE is included as another folder and the JRE directory structure is preserved.

Because directory structure is preserved, the application can load external resources using paths relative to the application JAR or `java.home` system property.

Note:

The Java packaging tools use the `jlink` tool to generate a custom JRE for the application. If you need something that is not included by default, then you can copy it in as a post-processing step. For installable packages, you can do this from the `config` script that is executed after populating the self-contained application folder. See Customizing the Package Using Drop-In Resources.

Basic Build

The easiest way to produce a self-contained application is to modify the deployment task. To request the creation of all types of self-contained application packages for the platform on which you are running, add `nativeBundles="all"` to the `<fx:deploy>` task, as shown in the following example.

```
<fx:deploy width="${javafx.run.width}" height="${javafx.run.height}"
           nativeBundles="all"
```

```
        outdir="${basedir}/${dist.dir}" outfile="${application.title}">
    <fx:application name="${application.title}" mainClass="${javafx.main.class}"/>
    <fx:resources>
        <fx:fileset dir="${basedir}/${dist.dir}" includes="*.jar"/>
    </fx:resources>
    <fx:info title="${application.title}" vendor="${application.vendor}"/>
</fx:deploy>
```

You can also specify the exact package format that you want to produce. Use the value `image` to produce a basic package, `exe` to request an EXE installer, `dmg` to request a DMG installer, and so on. For the full list of attribute values, see the `nativeBundles` attribute in the <fx:deploy> entry in the Ant Task Reference.

You can also produce native packages using the Java Packager tool. You can request specific formats using the `-native` option with the `-deploy` command. See the `javapackager` command reference in *Java Platform, Standard Edition Tools Reference*.

Example 2-1 shows the use of the `-native` option with the `-deploy` command, used to generate all applicable self-contained application packages for the BrickBreaker application. The `-deploy` command requires a JAR file as input, so it assumes that `dist/BrickBreaker.jar` has already been built:

Example 2-1 Java Packager Command to Generate Self-Contained Application Packages

```
javapackager -deploy -native -outdir packages -outfile BrickBreaker
    -srcdir dist -srcfiles BrickBreaker.jar -appclass brickbreaker.Main
    -name "BrickBreaker" -title "BrickBreaker demo"
```

Customizing the Package Using Drop-In Resources

The packaging tools use several built-in resources to produce a package, such as the application icon or configuration files. One way to customize the resulting package is to substitute a built-in resource with your customized version.

The following actions are needed:

- Prepare Custom Resources
- Substitute a Built-In Resource

Prepare Custom Resources

To get more insight into what resources are being used, enable verbose mode by adding the `verbose="true"` attribute to <fx:deploy>, or pass the `-v` option to the `javapackager -deploy` command.

Verbose mode includes the following actions:

- The following items are printed:
  – List of configuration resources that are used for the package that you are generating
  – Role of each resource
  – Expected custom resource name

- A copy of the configuration files and resources used to create the self contained package are saved to a temporary folder. You can use these files as a starting point for customization.

The following example shows sample output in verbose mode, with the important parts in bold:

```
Using base JDK at: /Library/Java/JavaVirtualMachines/jdk1.7.0_06.jdk
  Using default package resource [Bundle config file] (add
    package/macosx/Info.plist to the class path to customize)
  Using default package resource [icon] (add package/macosx/DemoApp.icns
      to the class path to customize)
Creating app bundle: /tmp/test/TestPackage/bundles/DemoApp.app
Config files are saved to /var/folders/rd/vg2ywnnx3qj081sc5pn9_
    vqr0000gn/T/build7039970456896502625.fxbundler/macosx. Use them
    to customize package.
```

Now you can grab a copy of the configuration files and tune them to your needs. For example, you can take the configuration file `Info.plist` and add localized package names.

Note:

It is recommended that you disable verbose mode after you are done customizing, or add a custom cleanup action to remove sample configuration files.

Substitute a Built-In Resource

Packaging tools look for customized resources on the class path before reverting to built-in resources. The Java Packager has "." (the current working directory) added to the class path by default. Therefore, to replace the application icon, copy your custom icon to `./package/macosx/DemoApp.icns` in the directory from which `javapackager` is run (typically, the root project directory).

The class path for Java Ant tasks is defined when task definitions are loaded. You must add an additional path to the lookup before the path `ant-javafx.jar`.

Example 2-2 shows how to add "." to the custom resource path.

After you provide a customized resource, verbose build output reports that the resource is used. For example, if you added a custom icon to an application, then the verbose output reports the addition, shown in Example 2-3.

Example 2-2 Enabling Resource Customization for JavaFX Ant Tasks

```
<fx:bundleArgument arg="dropinResourcesRoot" value="."/>
```

Example 2-3 Verbose Output After Adding a Customized Icon Resource

```
Using base JDK at: /Library/Java/JavaVirtualMachines/jdk1.7.0_06.jdk
  Using default package resource [Bundle config file] (add
      package/macosx/Info.plist to the class path to customize)
Using custom package resource [icon] (loaded from
    package/macosx/DemoApp.icns on class path)
Creating app bundle: /tmp/test/TestPackage/bundles/DemoApp.app
```

Customization Options

Many of the existing JavaFX Ant elements are used to customize self-contained application packages. Different sets of parameters are needed for different packages, and the same element might have different roles. Table 2-1 introduces some of the customization options and relevant attributes. See JavaFX Ant Helper Parameter Reference for a complete description of the elements and their attributes.

Table 2-1 Customization Options with Ant Elements and Attributes

Element	Desciption
`<fx:application>`	Application descriptor. Use this to set application attributes, such as the name and version.
`<fx:preferences>`	Deployment preferences for the application. Use this to set installation options, such as requesting a shortcut or an entry in the system application menu.
`<fx:fileset>`	Extension of the standard Ant `FileSet` type. Use this to identify the types of resources provided.
`<fx:info>`	Application description for users. Use this to define the application information shown in system dialog boxes, such as the title and vendor of the application.
`<fx:argument>`	Arguments to pass to the application when it is started.
`<fx:association>`	Types of files to associate with the application.
`<fx:jvmarg>`	JVM arguments to be passed to JVM and used to run the application, for example, large heap size.
`<fx:jvmUserArg>`	User-changeable JVM arguments to be passed to JVM and used to run the application.
`<fx:property>`	Properties to be set in the JVM running the application.
`<fx:runtime>`	Customization of the Java runtime generated for the application. Use this to add modules to the runtime, specify the location of modules, and include command-line tools.

Platform-Specific Customization for Basic Packages

Creating and customizing the basic form of self-contained application packages is a fairly straightforward process, but note the following points:

- Different icon types are needed for different platforms.

  For example, on Windows, the `.ico` format is expected, on Linux, the fomat is `.png`, and on macOS the format is `.icns`. No icon is embedded into the launcher on Linux, instead the `.desktop` file references the icon.

- For JavaFX applications, add the icon to the application stage to ensure that the icon is set in the runtime. For example, add the following code to the `start()` method of your JavaFX application:

  ```
  stage.getIcons().add(new
       Image(this.getClass().getResourceAsStream("app.png")));
  ```

- Sign files in the output folder if you plan to distribute the application.

  For example, on Windows, the launcher executable can be signed using `signtool.exe`.

macOS

The resulting package on macOS is an "application bundle".

Several configuration parameters are placed in the `Info.plist` file in the application bundle and must conform to the following rules:

- Application ID (or main class name if ID is not specified) is used as `CFBundleIdentifier`.
- Application version is used as `CFBundleShortVersionString`.

Gatekeeper in macOS prevents execution of untrusted code by default, regardless of whether this code is implemented in Objective-C or Java.

The user can manually enable the application to run, but this is not a perfect user experience. To get optimal user experience, obtain a Developer ID Certificate from Apple. The Mac bundler uses the certificate to sign the `.app` folder. If your local user information differs from the name of the certificate, you might need to set the bundle argument `mac.signing-key-user-name`, as shown in Example 2-4. See Developer ID and Gatekeeper at the Apple Developer site.

Example 2-4 Example using mac.signing-key-user-name

```
// Using javapackager tool
 javapackager ... -Bmac.signing-key-user-name="Jane Appleseed"

// Using Ant tasks
   <fx:deploy>
      //...
<fx:bundleArgument arg="mac.signing-key-user-name" value="Jane Appleseed"/>
      //...
    </fx:deploy>
```

Passing Arguments to a Self-Contained Application

Arguments can be passed to a self-contained application when the application is started from the command line. You can also define a set of arguments to pass to the application if no arguments are provided. To define default arguments, use the `-argument` option with the `javapackager deploy` command or the <fx:argument> element in an Ant task when the application package is created. Arguments entered from the command line override the default arguments. If the application is started from the launcher icon, the default arguments are used.

Associating Files with a Self-Contained Application

The installer for a self-contained application can be set up to register file associations for the application. The <fx:association> element is used in an Ant task to identify the files that can be handled by the application. File associations are based on either the file extension or MIME type.

The following example associates the application with files that have the MIME type `application/x-vnd.MyAppFile`.

```
<fx:info title="Association example">
  <fx:association mimetype="application/x-vnd.MyAppFile" description="Sample Test
Files">
  </fx:association>
</fx:info>
```

Supporting Multiple Entry Points

The package for self-contained applications can be built to support a suite of products with more than one entry point. Each entry point can have its own shortcut or icon. The `mainClass` attribute for the <fx:application> element identifies the primary entry point. Use the <fx:secondaryLauncher> element with the <fx:deploy> task to define each secondary entry point.

> **Note:**
>
> Multiple entry points are supported only for Windows and Linux applications.

The following example defines entry points for the `TestSuite` application for Windows.

```
<fx:deploy outdir="test/apps" nativeBundles="image">
    <fx:application name="TestSuite Sample"
                    mainClass="samples.TestSuite"/>

    <fx:info title="Test Suite"/>

    <fx:secondaryLauncher
        mainClass="samples.TestSuite"
        name="Suite Applications"/>
        shortcut="true"/>

    <fx:secondaryLauncher name="Editor">
        <fx:bundleArgument arg="icon" value="../resources/editor.ico"/>
    </fx:secondaryLauncher>

    <fx:secondaryLauncher name="Spreadsheet">
        <fx:bundleArgument arg="icon" value="../resources/spreadsheet.ico"/>
    </fx:secondaryLauncher>
</fx:deploy>
```

Customization of the JRE

The Java packaging tools use the `jlink` tool to generate a runtime for the self-contained application. Add command-line tools and additional modules as needed.

By default, command-line tools such as `java.exe` are removed from the JRE that is bundled with self-contained application packages. To keep these tools in the generated JRE, set the `strip-native-commands` attribute of the `<fx:runtime>` element to `false`.

To minimize the size of the JRE, the `jlink` tool is used to generate a custom runtime that contains only the packages needed to run the application. If additional modules are needed, use the `<fx:add-modules>` element to add them to the runtime. To add multiple modules, use a single `<fx:add-modules>` element with a comma-separated list of modules, or use a separate `<fx:add-modules>` element for each module.

The following example includes command-line tools and adds modules from `jdk.packager.services` and `javafx.controls`.

```
<fx:runtime strip-native-commands="false">
  <fx:add-modules value="jdk.packager.services,javafx.controls"/>
</fx:runtime>
```

Packaging for Modular Applications

Use the Java Packager tool to package modular applications as well as non-modular applications.

Modular applications can be packaged as self-contained applications. They cannot be packaged as Java Web Start applications. To identify the main module of a modular application, set the `module` attribute of the `<fx:secondaryLauncher>` element.

The following example identifies the main module for an application named `HelloWorldModular`.

```
<fx:secondaryLauncher name="HelloWorldModular"
    module="hello.world"
    mainClass="com.sample.app.HelloWorld">
</fx:secondaryLauncher>
```

Installable Packages

A self-contained application can be wrapped into a platform-specific installable package to simplify distribution. Java packaging tools provide built-in support for several formats of installable packages, depending on the availability of third-party tools.

Tuning the user experience for the installation process is specific to the particular installer technology, as described in other sections in this chapter. However, you must decide what type of installer you need. The following considerations might help with your decision:

- **System-wide or per-user installation?**

  System-wide installation results in a package installed into a shared location and can be used by any user on the system. Admin permissions are typically required and additional steps are likely to be needed during the installation process, such as an OS prompt to approve elevating installer permissions.

Per-user installation copies the package into a private user directory and does not require admin permissions. This type of installation enables you to show as few dialogs as possible and run the program even if the user is not eligible for admin privileges.

Note that whenever a user- or system-level installable package is requested, the build procedure itself does not require admin permissions.

- **Do you need a click-through license?**

  Some installable packages support showing license text before initiating the installation. The installation process starts only after the user accepts the license.

- **What menu and desktop integration is needed?**

  The user should be able to launch your application easily. Therefore, having a desktop shortcut or adding the application to the list of applications in the menu is required.

Note that the current implementation contains many simplifying assumptions. For example, installers never ask the user to choose the location in which to install the package. Developers also have limited control of the installation location, and can only specify system-wide or per-user installation.

If the default assumptions do not meet your needs, advanced customizations are available by tuning the configuration file templates (see Customizing the Package Using Drop-In Resources) or packaging a basic self-contained application and then wrapping it into an installable package on your own.

Types of Installable Packages

Create installable packages for self-contained applications based on the target operating system and the packaging tools available.

The following table shows the supported installable-package formats and the tools needed to create them:

Table 2-2 Installable Package Formats and Tool Prerequisites

Package format	Installation Location (Default mode in bold)	Click-Through License	Prerequisites
EXE	**Per user**: `%LOCALAPPDATA%` System: `%ProgramFiles%`	Yes (option)	• Windows • Inno Setup 5 or later
MSI	Per user: `%LOCALAPPDATA%` **System**: `%ProgramFiles%`	No special support	• Windows • WiX 3.0 or later
DMG	Per user: user's desktop folder **System**: /Applications	Yes (option)	• macOS
PKG	Per user: user's desktop folder **System**: /Applications	Yes (option)	• macOS
RPM	Per user: unsupported System: /opt	No special support	• Linux • RPMBuild

Table 2-2 (Cont.) Installable Package Formats and Tool Prerequisites

Package format	Installation Location (Default mode in bold)	Click-Through License	Prerequisites
DEB	Per user: unsupported System: /opt	No special support	• Linux • Debian packaging tools

EXE Package

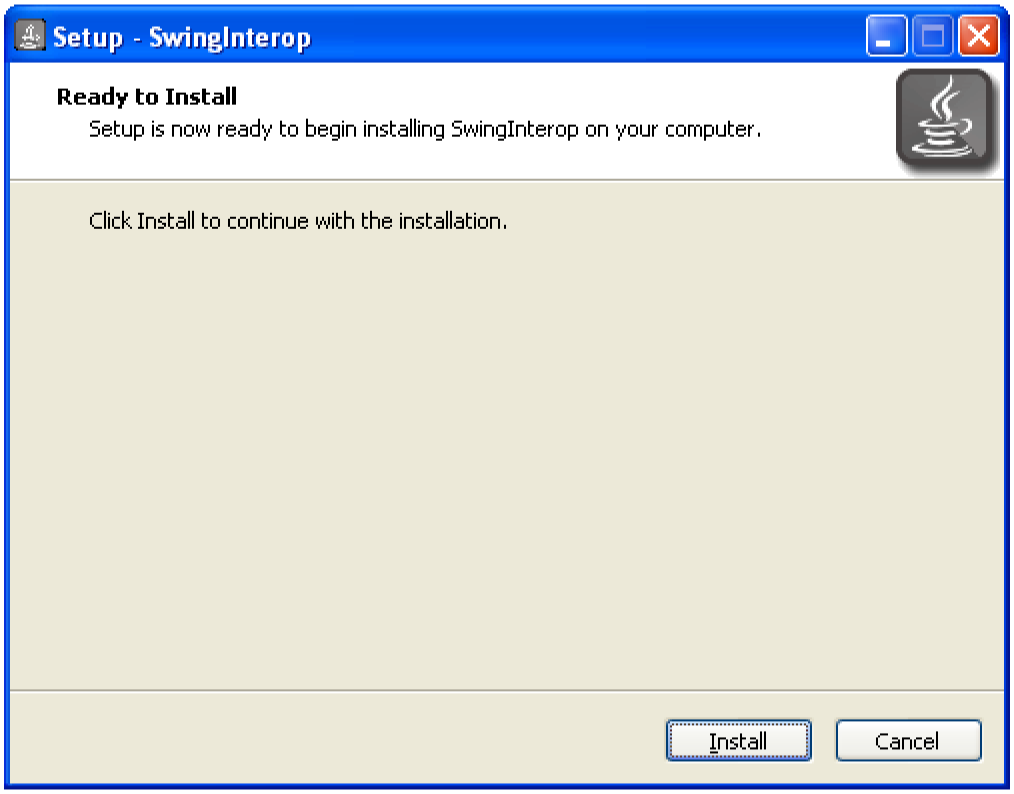
To generate an EXE package, you must have Inno Setup 5 or later installed and available on the `PATH`. To validate that it is available, try running `iscc.exe` from the command line where you launch the build or from your build script.

By default, the generated package has the following characteristics:

- Admin privileges not required
- Optimized to have a minimum number of dialogs
- Referenced from the programs menu or a desktop shortcut, or both
- Launches the application at the end of installation

Figure 2-2 shows a typical dialog box for a self-contained JavaFX application being installed on Windows.

Figure 2-2 Windows Installation Dialog for a Self-Contained JavaFX Application


Setup - SwingInterop

Ready to Install

Setup is now ready to begin installing SwingInterop on your computer.

Click Install to continue with the installation.

Customization tips:

- If you chose system-wide installation, then the user needs to have admin permissions, and the application is not launched at the end of installation.
- A click-through license is supported. An `.rtf` file is required.
- The image shown in the installation dialogs can be different from the application icon.

  The current version of Inno Setup assumes the image is a bitmap file with a maximum size of 55x58 pixels.
- For JavaFX applications, the icon can be added to the application stage to ensure that the icon is set in the runtime.
- The resulting `.exe` package can be signed.

  You need to get a certificate from a Trusted Certificate Authority (TSA), then use the Windows `signtool.exe` utility to sign the code.
- A Windows script file can be used to fine tune the self-contained application folder before it is wrapped into an `.exe` file, for example to sign the launcher executable.

The techniques for adding a custom image and providing a Windows script file are described in Customizing the Package Using Drop-In Resources. To add an icon to the application stage in a JavaFX application, see Platform-Specific Customization for Basic Packages.

Note:

While the resulting package is displayed in the list of installed applications, it does not use Windows Installer (MSI) technology and does not require the use of GUIDs. See the Inno Setup FAQ for details.

MSI Package

MSI packages are generated using the Windows Installer XML (WiX) toolset (also known as WiX). WiX 3.8 or later is required, and it must be available on the `PATH`. To validate, try running `candle /?` from the command line where you launch the build or from your build script.

By default, a generated MSI package has the following characteristics:

- Optimized for deployment using enterprise deployment tools
- Installs to a system-wide location
- No click-through UI, only a progress dialog is shown
- Referenced from the programs menu or a desktop shortcut, or both
- Removes all files in the installation folder, even if they were created outside of the installation process. (WiX 3.5 or later is required.)
- Tries to use the application identifier as `UpgradeCode`.

  If the application identifier is not a valid GUID, then a random GUID for `UpgradeCode` is generated.

- Randomly generates `ProductCode`.

To use a fixed Product code or add a custom UI to the MSI package, customize the WiX template file used by Java Packager as described in Customizing the Package Using Drop-In Resources. For a custom UI, also see WiX documentation.

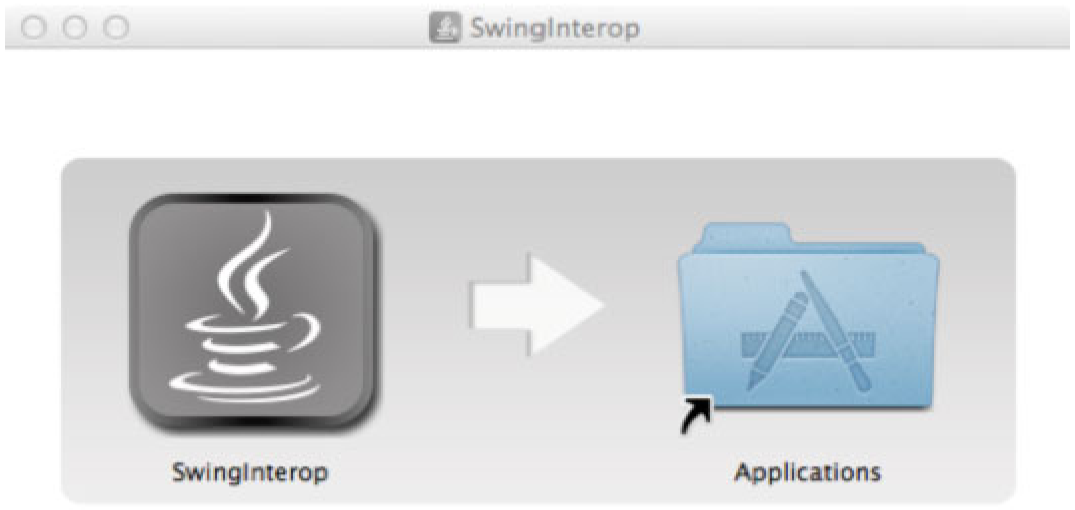
If you plan to distribute your MSI package on the network, sign it for the best user experience.

You can also fine tune the self-contained application folder before it is wrapped into the `.msi` file, for example, to sign the launcher executable.

DMG Package

By default, a DMG package provides a simple drag-and-drop installation experience. Figure 2-3 shows an example of the default behavior during installation.

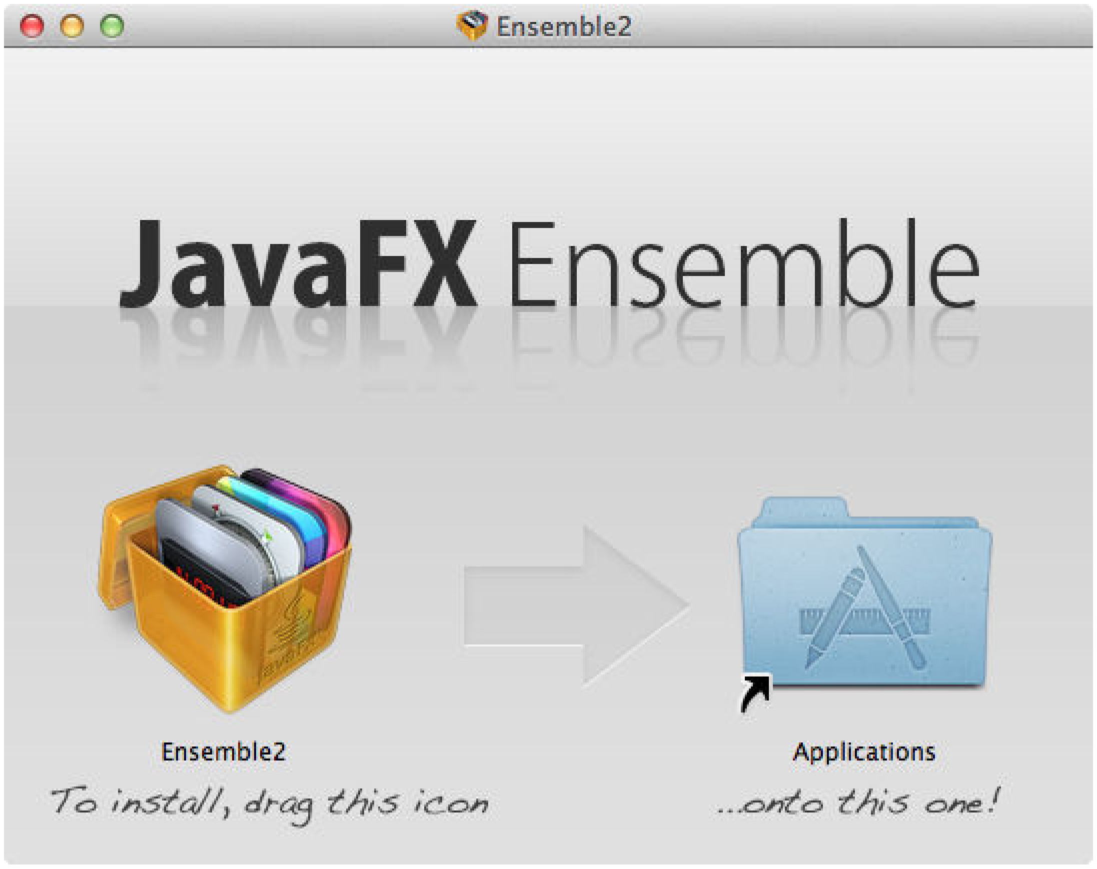
Figure 2-3 Example of Default Installer for macOS



To customize the appearance of the installation window, you can provide a custom background image.

If the background image has different dimensions or you need to position the icons differently, then you must also customize the DMG setup script that is used to modify sizes and positions of elements in the install view. See Customizing the Package Using Drop-In Resources.

Figure 2-4 Example of Customized Appearance of Installable Package for macOS



To fine tune the self-contained application folder before it is wrapped, provide your own bash script to be executed after the application folder is populated. You can use the script for such actions as adding localization files to the package. Figure 2-4 shows an example of a "tuned" application installer.

To create a Gatekeeper-friendly package, the application in the DMG package must be signed. It is not necessary to sign the DMG file itself. The Mac bundler handles the signing of your application. If your local user information differs from the name of the certificate, you might need to set the bundle argument `mac.signing-key-user-name`, as shown in Example 2-4.

To sign the application manually, you can use a technique described in macOS to provide a configuration script that is executed after the application bundle is populated. For the sample `DemoApp`, the configuration script is located at `package/macosx/DemoApp-post-image.sh` and has the content shown in the following example.

```
echo "Signing application bundle"
#Move to the folder containing application bundle
cd ../images/dmg.image
#do sign
codesign -s "Developer ID Application" *.app
echo "Done with signing"
```

The DMG installer also supports a click-though license provided in text format. If use of rich text format is desired, then prepare the `license.plist` file externally and add it to the package using the technique described in Customizing the Package Using Drop-In Resources.

No third party tools are needed to create a DMG package.

Linux Packages

Producing install packages for Linux assumes that the native tools needed to build install packages are installed. For RPM packages, this typically means the RPMBuild package and its dependencies. For DEB packages, `dpkg-deb` and dependencies are needed.

No admin permissions are needed to build the package.

By default the resulting package has the following characteristics:

- Installs the application to the `/opt` directory
- Adds a shortcut to the application menu
- Does not have any UI for installation, which is normal behavior for Linux packages

Customization tips:

- To place the application into a specific category in the application menu, use the `category` attribute of <fx:info>.

  See Desktop Menu Specification, and your window manager docs for the list of category names.
- The icon is expected to be a .png file
- Advanced customization is possible by tuning the build template files using techniques described in Customizing the Package Using Drop-In Resources..

  See the DEB/RPM packaging guides for more information about available options.

Working Through a Deployment Scenario

Consider a scenario where you have a JavaFX application with the following characteristics:

- Uses several third-party libraries
- One of the third-party libraries uses JNI and loads a platform-specific native library using `System.loadLibrary()`
- Needs a large 1Gb heap

You want to package this application as a self-contained application that does not need admin permissions to install.

It is assumed that your application works fine as a standalone application, that the main JAR file is built in the `dist` folder (using <fx:jar>) and that third-party libraries are copied to the `dist/lib` directory.

One way to assemble a self-contained application package is shown in Example 2-5, and consists of the following actions:

- Include all application JAR files.

- Add native libraries applicable to current platform as resources of type data.

  Ensure that the fileset base directory is set to the folder containing the library. This ensures that the libraries are copied to the top-level application folder.
- Request a user-level installation with `<fx:preferences install="false"/>`

Note that the top-level application folder is added to the library search path, and therefore `System.loadLibrary()` can be used.

Example 2-5 shows an example <fx:deploy> task.

Example 2-5 Example <fx:deploy> Task

```
<fx:deploy nativeBundles="all" width="600" height="400"
           outdir="${basedir}/dist" outfile="NativeLibDemo">
    <fx:application name="NativeLib Demo" mainClass="${javafx.main.class}"/>

    <fx:resources>
        <!-- include application jars -->
        <fx:fileset dir="dist" includes="*.jar"/>
        <fx:fileset dir="dist" includes="lib/*.jar"/>

        <!-- native libs for self-contained application -->
        <!-- assume they are stored as
                 native/windows/x86/JNativeHook.dll
                 native/linux/x86_64/libJNativeHook.so
                 .... -->
        <!-- ensure libraries are included as top level elements
                to get them on java.library.path -->
        <fx:fileset dir="${basedir}/native/${os.name}/${os.arch}"
                    type="data">
            <include name="*.dll"/>
            <include name="*.jnilib"/>
            <include name="*.so"/>
        </fx:fileset>
    </fx:resources>

    <!-- Custom JVM setup for application -->
    <fx:platform>
        <fx:jvmarg value="-Xmx1024m"/>
        <fx:jvmarg value="-verbose:jni"/>
        <property name="my.property" value="something"/>
    </fx:platform>

    <!-- request user level installation -->
    <fx:preferences install="false"/>
</fx:deploy>
```

3

JavaFX Ant Tasks

This chapter shows how to use Ant to package Java and JavaFX applications. JavaFX Ant tasks and the Java Packager tool are the recommended ways to package your applications.

This chapter contains the following topics:

- Requirements to Run JavaFX Ant Tasks
- JavaFX Ant Elements
- Using JavaFX Ant Tasks
- Ant Script Examples

See also the following two Ant Task Reference sections:

- JavaFX Ant Task Reference
- JavaFX Ant Helper Parameter Reference

Requirements to Run JavaFX Ant Tasks

The `ant-javafx.jar` file is required to use these tasks. It is located in the following locations:

- In JDK 7 Update 6 or later, it is located in `*jdk_home*/lib`
- In a standalone JavaFX installation, it is located in `*javafx-sdk-home*/lib`

JavaFX Ant Elements

There are two categories of Ant elements: JavaFX Ant tasks and Ant helper parameters.

JavaFX Ant Tasks

JavaFX Ant tasks perform the following tasks:

- Creating JAR files that can be double-clicked
- Creating an HTML page and deployment descriptor for Web Start applications or applications embedded in a web page
- Digitally signing an application, when necessary
- Converting CSS files to binary format
- Assembling self-contained application packages

Elements are described in JavaFX Ant Task Reference.

Ant Helper Parameters

Ant helper parameters are used by the JavaFX Ant tasks. The helper parameters are described in JavaFX Ant Helper Parameter Reference.

Using JavaFX Ant Tasks

To use the JavaFX Ant tasks in your Ant script, you must load their definitions. An example is shown in the `build.xml` file in Example 3-1.

Notes about Example 3-1:

- Ensure that you declare the fx: namespace, shown in bold in Example 3-1, because short names for some of JavaFX tasks are the same as those used for some system tasks.
- The current directory (".") is added to the classpath to simplify customization using drop-in resources. See Customizing the Package Using Drop-In Resources.

After JavaFX Ant task definitions are loaded, the `javafx.ant.version` property can be used to check the version of Ant tasks APIs. Use the following list for version numbers:

- Version 1.0: shipped in the JavaFX 2.0 SDK
- Version 1.1: shipped in the JavaFX 2.1 SDK
- Version 1.2: shipped in the JavaFX 2.2 SDK and JDK 7 Update 6

Example 3-1 Load JavaFX Ant Task Definitions

```
<project name="JavaFXSample" default="default" basedir="."
         xmlns:fx="javafx:com.sun.javafx.tools.ant">
    <target name="default">
        <taskdef resource="com/sun/javafx/tools/ant/antlib.xml"
                uri="javafx:com.sun.javafx.tools.ant"
                classpath=".:${JAVA_HOME}/lib/ant-javafx.jar"/>
    </target>
</project>
```

Ant Script Examples

This section covers the following topics:

- Deploying the JavaFX Hello World Example
- Deploying the JavaFX Hello World Example as a Self-Contained Application
- Deploying a JavaFX Application with External JAR Files
- Overriding JVM Options for Self-Contained Applications

Deploying the JavaFX Hello World Example

Follow these steps to deploy the JavaFX Hello World example as a JAR file with an Ant script:

1. Create a directory to contain the example application. These steps use the directory `C:\example`.
2. Save Example 3-2 as `C:\example\src\HelloWorld.java`.

3. Save Example 3-3 as `C:\example\build.xml`.
4. In the file `build.xml`, specify the location of the JDK installed in your computer by changing the value of the `JAVA_HOME` property. Change the highlighted text to the full path of your JDK:

   ```
   <property name="JAVA_HOME" value="C:\\Java\\jdk-9"/>
   ```

5. At a command-line prompt, change directory to `C:\example`. Run the following command to compile, build, and deploy the JavaFX HelloWorld example:

   ```
   ant
   ```

6. To run the example, at a command-line prompt, change directory to `C:\example\dist` and run the following command:

   ```
   java -jar HelloWorld.jar
   ```

Example 3-2   HelloWorld.java

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloWorld extends Application {

    @Override
    public void start(Stage primaryStage) {
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);

        Scene scene = new Scene(root, 300, 250);

        primaryStage.setTitle("Hello World!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
 public static void main(String[] args) {
        launch(args);
    }
}
```

Example 3-3   Ant Script to Deploy JavaFX Hello World Example

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

<project name="JavaFX Hello World Example" default="default" basedir="."
  xmlns:fx="javafx:com.sun.javafx.tools.ant">
```

```
<property name="JAVA_HOME" value="C:\\Java\\jdk-9"/>
<property name="build.src.dir" value="src"/>
<property name="build.classes.dir" value="classes"/>
<property name="build.dist.dir" value="dist"/>

<target name="default" depends="clean,compile">

  <taskdef resource="com/sun/javafx/tools/ant/antlib.xml"
    uri="javafx:com.sun.javafx.tools.ant"
    classpath="${JAVA_HOME}/lib/ant-javafx.jar"/>

    <fx:application id="HelloWorldID"
      name="JavaFXHelloWorldApp"
      mainClass="HelloWorld"/>

    <fx:resources id="appRes">
      <fx:fileset dir="${build.dist.dir}" includes="HelloWorld.jar"/>
    </fx:resources>

    <fx:jar destfile="${build.dist.dir}/HelloWorld.jar">
      <fx:application refid="HelloWorldID"/>
      <fx:resources refid="appRes"/>
      <fileset dir="${build.classes.dir}"/>
    </fx:jar>

    <fx:deploy width="300" height="250"
      outdir="." embedJNLP="true"
      outfile="helloworld">

      <fx:application refId="HelloWorldID"/>

      <fx:resources refid="appRes"/>

      <fx:info title="JavaFX Hello World Application"
        vendor="Oracle Corporation"/>

    </fx:deploy>

</target>

<target name="clean">
  <mkdir dir="${build.classes.dir}"/>
  <mkdir dir="${build.dist.dir}"/>

  <delete>
    <fileset dir="${build.classes.dir}" includes="**/*"/>
    <fileset dir="${build.dist.dir}" includes="**/*"/>
  </delete>

</target>

<target name="compile" depends="clean">

  <javac includeantruntime="false"
    srcdir="${build.src.dir}"
    destdir="${build.classes.dir}"
    fork="yes"
    executable="${JAVA_HOME}/bin/javac"
    source="9"
    debug="on">
  </javac>
  </target>

</project>
```

Deploying the JavaFX Hello World Example as a Self-Contained Application

To deploy the JavaFX Hello World example as a self-contained application, add the attribute `nativeBundles="all"` to the element `<fx:deploy>` in the `build.xml` script:

```
<fx:deploy width="300" height="250"
  outdir="." embedJNLP="true"
  outfile="helloworld"
  nativeBundles="all">
```

Compile, build, and deploy the JavaFX Hello World example as described in the previous section. The Ant script creates all applicable self-contained application packages and stores them in the directory `C:\example\bundles\JavaFXHelloWorldApp`. (The name `JavaFXHelloWorldApp` is the value of the `name` attribute of the `<fx:application>` element.) If you are deploying the JavaFX Hello World example with Windows, for example, the Ant script creates an application named `C:\example\bundles\JavaFXHelloWorldApp\JavaFXHelloWorldApp.exe` that you can run by double-clicking it in a file browser.

You can customize how your Ant script creates self-contained applications. See Self-Contained Application Packaging.

Deploying a JavaFX Application with External JAR Files

The JavaFX Ensemble8 sample application requires Apache Lucene. Example 3-4 shows how to deploy Ensemble8 as a self-contained application and include the Apache Lucene JAR files in it.

The following lines in the Ant script copy resources contained in the `src` directory to the directory that contains the compiled Java class files. The resources are copied after the Ant script compiles the sample application:

```
<copy todir="${build.classes.dir}">
  <fileset dir="src/app/resources"/>
  <fileset dir="src/generated/resources"/>
  <fileset dir="src/samples/resources"/>
</copy>
```

The following lines from the Ant script include the Apache Lucerne JAR files (which are contained in the `lib` directory):

```
<fx:resources id="appRes">
  <fx:fileset dir="${build.dist.dir}"
    includes="ensemble8.jar"/>
  <fx:fileset dir="lib"/>
  <fx:fileset dir="${build.classes.dir}"/>
</fx:resources>

<fx:jar destfile="${build.dist.dir}/ensemble8.jar">
  <fx:application refid="ensemble8"/>
  <fx:resources refid="appRes"/>
</fx:jar>
```

Example 3-4   Ant Script to Deploy Ensemble8 Sample Application

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

<project name="Ensemble8 JavaFX Demo Application" default="default" basedir="."
  xmlns:fx="javafx:com.sun.javafx.tools.ant">

  <property name="JAVA_HOME" value="C:\\Java\\jdk-9"/>

  <path id="CLASSPATH">
    <pathelement location="lib/lucene-core-3.2.0.jar"/>
    <pathelement location="lib/lucene-grouping-3.2.0.jar"/>
    <pathelement path="classes"/>
  </path>

  <property name="build.src.dir" value="src"/>
  <property name="build.classes.dir" value="classes"/>
  <property name="build.dist.dir" value="dist"/>

  <target name="default" depends="clean,compile">

    <taskdef resource="com/sun/javafx/tools/ant/antlib.xml"
      uri="javafx:com.sun.javafx.tools.ant"
      classpath="${JAVA_HOME}/lib/ant-javafx.jar"/>

      <fx:application id="ensemble8"
        name="Ensemble8"
        mainClass="ensemble.EnsembleApp"/>

      <fx:resources id="appRes">
        <fx:fileset dir="${build.dist.dir}" includes="ensemble8.jar"/>
        <fx:fileset dir="lib"/>
        <fx:fileset dir="${build.classes.dir}"/>
      </fx:resources>

      <fx:jar destfile="${build.dist.dir}/ensemble8.jar">
        <fx:application refid="ensemble8"/>
        <fx:resources refid="appRes"/>
      </fx:jar>

      <fx:deploy outdir="." embedJNLP="true"
        outfile="ensemble8"
        nativeBundles="all">

        <fx:application refId="ensemble8"/>

        <fx:resources refid="appRes"/>

        <fx:info title="Ensemble8 JavaFX Demo Application"
          vendor="Oracle Corporation"/>

      </fx:deploy>

  </target>

  <target name="clean">
    <mkdir dir="${build.classes.dir}"/>
    <mkdir dir="${build.dist.dir}"/>

    <delete>
      <fileset dir="${build.classes.dir}" includes="**/*"/>
      <fileset dir="${build.dist.dir}" includes="**/*"/>
    </delete>

  </target>

  <target name="compile" depends="clean">

    <javac includeantruntime="false"
      srcdir="${build.src.dir}"
      destdir="${build.classes.dir}"
      fork="yes"
      executable="${JAVA_HOME}/bin/javac"
      source="9"
      debug="on"
      classpathref="CLASSPATH">
    </javac>

    <!-- Copy resources to build.classes.dir -->

      <copy todir="${build.classes.dir}">
        <fileset dir="src/app/resources"/>
        <fileset dir="src/generated/resources"/>
        <fileset dir="src/samples/resources"/>
      </copy>

  </target>

</project>
```

Overriding JVM Options for Self-Contained Applications

You can override JVM options in your self-contained applications by specifying them in a preferences node, then setting the name of this node in the `app.preferences.id` system property. The following example overrides the `-Xms` and `-Xmx` JVM options specified in <fx:jvmuserarg> elements. To verify that these options have been overridden, the application displays the initial and maximum sizes of the memory allocation pool (based on the values of the `-Xms` and `-Xmx` options).

1. Create a directory to contain the example application. These steps use the directory `C:\memexample`.
2. Save Example 3-5 as `C:\memexample\src\MemoryExamplePreferences.java`.

   The following statement retrieves the `java.util.prefs.Preferences` node that stores the values of the options `-Xms` and `-Xmx` in your computer:

   ```
   prefs =
   Preferences.userRoot().node(System.getProperty("app.preferences.id")).node("JVMUs
   erOptions");
   ```

   The name of this node is `app.preferences.id/JVMUserOptions`. The value of the `app.preferences.id` property is the value of the `id` attribute of the <fx:application> element. In this example, the value of `app.preferences.id` is `MemoryTestAppID`. When you run a self-contained application, the launcher program sets the system property automatically. However, if you run the class directly, you must specify the value of `app.preferences.id` as a system property. For example, if you run this class from the command line, you would specify the value of `app.preferences.id` as follows:

```
java -cp classes -Dapp.preferences.id=MemoryTestAppID
```

Note that in this example, the Ant build script runs the `MemoryTestPreferences` class for you and sets the value of the `app.preferences.id` property.

For example, if you are using Microsoft Windows, then this class creates a preferences node named `Computer\HKEY_CURRENT_USER\Software\JavaSoft\Prefs\MemoryTestAppID\JVMUserOptions` in the Windows registry.

3. Save Example 3-6 as `C:\memexample\src\MemoryExample.java`.
4. Save Example 3-7 as `C:\memexample\build.xml`.

   This Ant script compiles and runs the `MemoryTestPreferences` class, and sets the value of the `app.preferences.id` property:

   ```
           <java fork="true" jvm="${env.JAVA_HOME}\bin\java"
                 classname="MemoryTestPreferences" classpath="${build.classes.dir}">
               <sysproperty key="app.preferences.id" value="MemoryTestAppID"/>
           </java>
   ```

5. In the file `build.xml`, specify the location of the JDK installed in your computer by changing the value of the `JAVA_HOME` property. Change the highlighted text to the full path of your JDK:

   ```
   <property name="JAVA_HOME" value="C:\\Java\\jdk-9"/>
   ```

6. At a command-line prompt, change directory to `C:\memexample`. Run the following command to compile, build, and deploy this example:

   ```
   ant
   ```

7. To run the Memory Test application, run one of the applications contained in `C:\memexample\bundles\bundles/JavaFXMemoryTestApp`.

   In addition to the initial and maximum sizes of the memory allocation pool, the example displays the JVM arguments that it uses:

   ```
   -Djava.library.path=C:\memexample\bundles\JavaFXMemoryTestApp\app\
   -Dapp.preferences.id=MemoryTestAppID
   -Xms2048m
   -Xmx2048m
   ```

   Note that the Ant build script uses <fx:jvmuserarg> elements to specify the initial and maximum sizes of the memory allocation pool as 31m and 64m, respectively. However, the Ant script in this example runs another program that stores the preferred values of 2048m and 2048m in the `Preferences` node before the application is run, so those values are used instead.

Example 3-5 MemoryExamplePreferences.java

```
import java.util.prefs.Preferences;

public class MemoryTestPreferences {
  private Preferences prefs;

  public void setPreferences() {

    // This will define a node in which the preferences can be stored
    prefs =
Preferences.userRoot().node(System.getProperty("app.preferences.id")).node("JVMUserOp
tions");

    // now set the values
```

```
    prefs.put("-Xmx", "2048m");
    prefs.put("-Xms", "2048m");

  }

  public static void main(String[] args) {
    MemoryTestPreferences myPrefs = new MemoryTestPreferences();
    myPrefs.setPreferences();
  }
}
```

Example 3-6 MemoryExample.java

```
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.geometry.HPos;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ListView;
import javafx.scene.control.TextField;
import javafx.scene.layout.ColumnConstraints;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Priority;
import javafx.scene.layout.Region;
import javafx.stage.Stage;

import java.lang.management.ManagementFactory;
import java.util.prefs.BackingStoreException;
import java.util.prefs.Preferences;

public class MemoryTest extends Application {

    @Override
    public void start(Stage primaryStage) {
        String startMemory = Long.toString(Runtime.getRuntime().totalMemory());
        String maxMemory = Long.toString(Runtime.getRuntime().maxMemory());

        System.out.println("Start memory: " + startMemory);

        System.out.println("Max memory: " + maxMemory);

        final Label startMemoryLabel = new Label("Start memory: ");
        TextField startMemoryTextField = new TextField(startMemory);
        startMemoryTextField.setPromptText(startMemory);

        Label maxMemoryLabel = new Label("Max memory: ");
        final TextField maxMemoryTextField = new TextField(maxMemory);
        maxMemoryTextField.setPromptText(maxMemory);

        Label jvmArgumentsLabel = new Label("JVM Arguments");
        ListView<String> jvmArguments = new 
ListView<>(FXCollections.observableArrayList(ManagementFactory.getRuntimeMXBean().get
InputArguments()));
        jvmArguments.setPrefSize(450, 150);

        Button btn = new Button();
        btn.setText("Update Preferences");
        btn.setOnAction(event -> {
            Preferences prefs = 
Preferences.userRoot().node(System.getProperty("app.preferences.id")).node("JVMUserOp
tions");
            String start = startMemoryTextField.getText();
            if (start == null || start.isEmpty()) {
                prefs.remove("-Xms");
            } else {
                prefs.put("-Xms", start);
            }

            String max = maxMemoryTextField.getText();
            if (max == null || max.isEmpty()) {
                prefs.remove("-Xmx");
            } else {
                prefs.put("-Xmx", max);
            }

            try {
                prefs.flush();
            } catch (BackingStoreException e) {
                e.printStackTrace();
            }
        });


        GridPane grid = new GridPane();
        grid.setAlignment(Pos.CENTER);
        grid.setHgap(10);
        grid.setVgap(10);
        grid.setPadding(new Insets(25, 25, 25, 25));
        grid.getColumnConstraints().setAll(
                new ColumnConstraints(Region.USE_PREF_SIZE, 
Region.USE_COMPUTED_SIZE, Region.USE_PREF_SIZE, Priority.NEVER, HPos.RIGHT, false),
                new ColumnConstraints(Region.USE_PREF_SIZE, 
Region.USE_COMPUTED_SIZE, Integer.MAX_VALUE, Priority.ALWAYS, HPos.LEFT, true)
        );

        grid.addRow(0, startMemoryLabel, startMemoryTextField);
        grid.addRow(1, maxMemoryLabel, maxMemoryTextField);
        grid.addRow(2, jvmArgumentsLabel, jvmArguments);
        grid.add(btn, 1, 3);
        grid.setMinSize(Region.USE_PREF_SIZE, Region.USE_PREF_SIZE);


        Scene scene = new Scene(grid);

        primaryStage.setTitle("Memory test");
        primaryStage.sizeToScene();
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Example 3-7 Ant Script for Memory Test Example

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

```
<project name="JavaFX Hello World Example" default="default" basedir="."
         xmlns:fx="javafx:com.sun.javafx.tools.ant">

    <property environment="env"/>
    <property name="env.JAVA_HOME" value="C:\\Java\\jdk-9"/>
    <property name="build.src.dir" value="src"/>
    <property name="build.classes.dir" value="classes"/>
    <property name="build.dist.dir" value="dist"/>

    <target name="default" depends="clean,compile">

        <taskdef resource="com/sun/javafx/tools/ant/antlib.xml"
                 uri="javafx:com.sun.javafx.tools.ant"
                 classpath="${env.JAVA_HOME}/lib/ant-javafx.jar"/>

      <fx:application id="MemoryTestAppID"
                        name="JavaFXMemoryTestApp"
                        mainClass="MemoryTest"/>

        <fx:resources id="appRes">
            <fx:fileset dir="${build.dist.dir}" includes="MemoryTest.jar"/>
        </fx:resources>

        <fx:jar destfile="${build.dist.dir}/MemoryTest.jar">
        <fx:application refid="MemoryTestAppID"/>
            <fx:resources refid="appRes"/>
            <fileset dir="${build.classes.dir}"/>
        </fx:jar>

        <fx:deploy width="300" height="250"
                   outdir="." embedJNLP="true"
                   outfile="memorytest"
                   nativeBundles="image">

            <fx:platform>
                <fx:jvmuserarg name="-Xms" value="31m"/>
                <fx:jvmuserarg name="-Xmx" value="64m"/>
            </fx:platform>

        <fx:application refId="MemoryTestAppID"/>

            <fx:resources refid="appRes"/>

            <fx:info title="JavaFX Hello World Application"
                     vendor="Oracle Corporation"/>

        </fx:deploy>

    </target>

    <target name="clean">
        <mkdir dir="${build.classes.dir}"/>
        <mkdir dir="${build.dist.dir}"/>

        <delete>
            <fileset dir="${build.classes.dir}" includes="**/*"/>
            <fileset dir="${build.dist.dir}" includes="**/*"/>
        </delete>

    </target>
    <target name="compile" depends="clean">

        <javac includeantruntime="false"
               srcdir="${build.src.dir}"
               destdir="${build.classes.dir}"
               fork="yes"
               executable="${env.JAVA_HOME}/bin/javac"
               source="9"
               debug="on">
        </javac>

        <!-- Set preferences -->

        <java fork="true" jvm="${env.JAVA_HOME}\bin\java"
              classname="MemoryTestPreferences" classpath="${build.classes.dir}">
            <sysproperty key="app.preferences.id" value="MemoryTestAppID"/>
        </java>
    </target>

    <target name="jar" depends="compile">
        <jar destfile="dist/MemoryTest.jar"
             basedir="classes"/>
    </target>

</project>
```

The `UserJvmOptionsService` API offers an alternative method for setting JVM options for self-contained applications. This API can be called from the application to get the current settings and to update the settings for the next time that the application is started.

JavaFX Ant Task Reference

The following items comprise the main JavaFX Ant tasks:

- <fx:csstobin>

  Converts CSS files to binary format for faster processing.

- <fx:deploy>

  Assembles the application package for redistribution. By default, the deploy task will generate the base application package, but it can also generate self-contained application packages if requested.

- <fx:jar>

  Creates one or more application JAR files.

- <fx:signjar>

  Provides the application with a digital signature.

Note:

The `<fx:signjar>` task for the Java Packager tool is deprecated in JDK 9 in preparation for removal in a future release. It also does not work with multi-release JAR files. Use the standard Ant `signjar` task instead.

Items are in alphabetical order.

<fx:csstobin>

Description

Converts a set of CSS files into binary form (BSS).

Parent Elements

None.

Parameters

Attribute	Description	Type	Required?
`outdir`	Name of the directory in which output files are generated.	String	Yes

Parameters Accepted as Nested Elements

- <fx:fileset>

<fx:csstobin> Task Usage Examples

Example 1 Convert CSS Files to Binary
This example converts all CSS files in the output tree to binary form.

```
<fx:csstobin outdir="build/classes">
    <fileset dir="build/classes" includes="**/*.css"/>
</fx:csstobin>
```

<fx:deploy>

Description

Generates a package for both web deployment and standalone applications. The package includes a set of JAR files, a JNLP file, and an HTML file.

Parent Elements

None.

Parameters

Attribute	Description	Type	Required?
`embeddedHeight`	If present, this value will be used for Javascript/HMTL code instead of width/height. Affects only embedded deployment mode. Use it if you want to specify a relative dimension for an embedded application.	String	No

Attribute	Description	Type	Required?
`embeddedWidth`	Same description as for `embeddedHeight`.	String	No
`embedjnlp`	If true, embed the JNLP descriptor into the web page. Reduces number of network connections to be made on startup and helps to improve startup time.	Boolean	No Default is `false`.
`extension`	Treat the files named in `srcfiles` as extensions. If present, only a portion of the deployment descriptor is generated, and the HTML file is not generated.	Boolean	No Default is `false`.
`height`	Height of the application scene, for embedding applications into a web page.	String	Yes
`includeDT`	If set to `true`, files related to the Deployment Toolkit will be copied to a `web-files` subdirectory of the directory specified in `outdir`. This setting is useful for offline development but is not advised for production.	Boolean	No Default is `false`.
`nativeBundles`	Values: • all • deb • dmg • exe • image • installer • jnlp • mac.appStore • msi • none • pkg • rpm Value `all` produces all applicable self-contained application packages for the platform on which the Ant tasks are run. Value `installer` produces only installable packages, not a disk image. Value `jnlp` produces only the `.jnlp` and `.html` files for a Java Web Start application. Value `none` produces no self-contained application packages. Other values produce a specific package installer.	String	No Default is `none`.

Attribute	Description	Type	Required?
`offlineAllowed`	If the value is `true`, the cached application can operate even if the client system is disconnected from the network.	Boolean	Default is `true`.
`outdir`	Name of the directory in which output files are generated.	String	Yes
`outfile`	Prefix of the output files, without the extension.	String	Yes
`placeholderref`	Placeholder in the web page where the application will be embedded. This is expected to be JavaScript DOM object.	String	Yes Either reference or ID of placeholder is required.
`placeholderid`	Used with callbacks. The ID of the placeholder in the web page where application will be embedded. The JavaScript function `document.getElementById()` is used to resolve it.	String	Yes Either the reference or the ID of the placeholder is required.
`signBundle`	Used for self-contained applications to request that the bundler sign the bundle that is generated. This attribute is ignored by bundlers that do not support signing. At the time of the 8u40 release of the JDK, only macOS bundlers support signing.	Boolean	Default depends on the bundler used.
`updatemode`	Indicates the preferences for when checks for application updates are performed for embedded and Web Start applications. A value of `always` means to always check for updates before launching the application. A value of `background` means to launch the application while checking for updates in the background.	String	No Default is `background`.
`width`	Width of the application scene, for embedding applications into a web page.	String	Yes

Parameters Accepted as Nested Elements

- <fx:platform>
- <fx:preferences>
- <fx:application>

- <fx:permissions>
- <fx:template>
- <fx:callbacks>
- <fx:info>
- <fx:resources>
- <fx:bundleArgument>
- <fx:secondaryLauncher>
- <fx:runtime>

<fx:deploy> Task Usage Examples

Example 1 Minimal <fx:deploy> Task

This is a simple example of an `<fx:deploy>` Ant task. It generates an HTML file and JNLP file into the web-dist directory and uses `Fish` as the prefix for the generated files.

```
<fx:deploy width="600" height="400"
        outdir="web-dist" outfile="Fish"
        offlineAllowed="false">
    <fx:info title="Sample application"/>
    <fx:application refid="myapp"/>
    <fx:resources refid="myresources"/>
</fx:deploy>
```

Example 2 <fx:deploy> Task for an Application with a Preloader

The following Ant task creates a distributable package for a simple application with a preloader. Details about the application and its resources are defined in the `<fx:application>` and `<resource>` elements in the task.

Note that the location of the output package is defined by the `outdir` attribute of the `<fx:deploy>` task. New files are generated using the `name` prefix specified in the `outfile` attribute. As a result of execution of this task, the following files are created in the web-dist folder:

- preloader.jar
- helloworld.jar
- App.jnlp
- App.html

Note:

By default, the deployment package uses auxiliary files from java.com to support web deployment. This is the preferred way, because it enables the application to always use the best way to deploy on the web. However, if you want to test your application in a closed network then you can include these files into your application package. To do this, pass `includeDT="true"` as an attribute in the `<fx:deploy>` Ant task.

```
<fx:deploy width="600" height="400"
        outdir="web-dist" outfile="App">
```

```
    <fx:info title="Sample application"/>
    <fx:application name="SampleApp"
            mainClass="testapp.MainApp"
            preloaderClass="testpreloader.Preloader">
        <fx:param name="testVariable" value="10"/>
    </fx:application>
    <fx:resources>
        <fx:fileset requiredFor="preloader" dir="dist">
            <include name="preloader.jar"/>
        </fx:fileset>
        <fx:fileset dir="dist">
            <include name="helloworld.jar"/>
        </fx:fileset>
    </fx:resources>
</fx:deploy>
```

Example 3 <fx:deploy> Task with Secondary Launchers

This example creates a package for a self-contained application that contains a secondary launcher that can be used to start the application in a memory-constrained environment. The main launcher does not set JVM options. The secondary launcher passes an option to limit the amount of memory used.

```
<fx:deploy outdir="../samples/test/ant" nativeBundles="image">
    <fx:application name="Secondary Launcher Sample"
                    mainClass="hello.Test"/>

    <fx:resources>
        <fx:fileset dir="../samples/test/resources" includes="mainApp.jar"/>
    </fx:resources>

    <fx:info title="Secondary Launcher Test"/>

    <fx:secondaryLauncher
        mainClass="hello.Test"
        name="Standard Launch"/>

    <fx:secondaryLauncher name="Memory Constrained">
        <fx:jvmarg="-xmx64m"/>
    </fx:secondaryLauncher>
</fx:deploy>
```

Example 4 <fx:deploy> Task for modular application with a custom runtime

```
<fx:deploy outdir="${bundles.dir}"
    outfile="MinesweeperFX"
    nativeBundles="all"
    verbose="true"

  <fx:runtime strip-native-commands="false">
    <fx:add-modules value="java.base"/>
    <fx:add-modules value="jdk.packager.services,javafx.controls"/>
    <fx:limit-modules value="java.sql"/>
    <fx:limit-modules value="jdk.packager.services,javafx.controls"/>
    <fx:module-path value="${java.home}/../images/jmods"/>
    <fx:module-path value="${build.dir/modules"/>
  </fx:runtime>

  <fx:application id="MinesweeperFX"
      name="MinesweeperFX"
```

```
        module="fx.minesweeper"
        mainClass="minesweeper.Minesweeper"
        version="1.0">
    </fx:application>

    <fx:secondaryLauncher name="Test2"
        module="hello.world"
        mainClass="com.greetings.HelloWorld">
    </fx:secondaryLauncher>
</fx:deploy>
```

<fx:jar>

Description

Packages an application into a JAR file. The set of files to be included is defined by nested `<fx:fileset>` parameters. The `<fx:jar>` task also embeds a JAR manifest into the JAR file.

In addition to creating a JAR archive, this task also:

- Embeds the JavaFX launcher for JavaFX applications, which detects the presence of JavaFX Runtime, sets up the environment, and executes the application.
- Creates a manifest in the JAR file.

The resulting JAR file supports launching by double-clicking.

Parent Elements

None.

Parameters

Attribute	Description	Type	Required?
`codebase`	Base location for all relative URLs specified in `href` attributes in the JNLP file.	String	No
`destfile`	Path to output JAR file (location and name)	String	Yes

Parameters Accepted as Nested Elements

- <fx:platform>
- <fx:fileset>
- <fx:application>
- <fx:info>
- <fx:resources>
- <fx:permissions>

<fx:jar> Usage Examples

See Example 3-3 and the following example.

Example 1 <fx:jar> Ant Task for a Simple Application

This example shows how to use the `<fx:jar>` Ant task to create the main application JAR file for a simple application without a custom preloader. The resulting JAR file performs the following two actions:

- Starts test.MyApplication with all resources needed on the classpath when launched as `java -jar application.jar` or by double-clicking the JAR file.
- Automatically detects the location of JavaFX Runtime and prompts the user to install it if it is not available, or reports if the platform is not supported.

```
<!-- Expect definition of JavaFX ant tasks is already imported -->

<fx:jar destfile="dist/application.jar">
    <!-- Details about application -->
    <fx:application name="Sample JavaFX application"
            mainClass="test.MyApplication"/>

    <!-- Define what auxilary resources are needed -->
    <fx:resources>
        <fx:fileset dir="dist" includes="lib/*.jar"/>
    </fx:resources>

    <!-- What to include into result jar file?
         Everything in the build tree -->
    <fileset dir="build/classes"/>

    <!-- Customize jar manifest (optional) -->
    <manifest>
        <attribute name="Implementation-Vendor" value="Samples Team"/>
        <attribute name="Implementation-Version" value="1.0"/>
    </manifest>
</fx:jar>
```

<fx:signjar>

Description

Note:

The `<fx:signjar>` task for the Java Packager tool is deprecated in JDK 9 in preparation for removal in a future release. It also does not work with multi-release JAR files. Use the standard Ant `signjar` task instead.

Digitally signs an application JAR file with a certificate.

Signs the JAR file as BLOB. In other words, instead of every entry being signed separately, the JAR file is signed as a single binary object.

Parent Elements

None.

Parameters

Attribute	Description	Type	Required?
`alias`	The alias for the key	String	Yes
`destdir`	Location of output file	String	Yes
`keypass`	Password for the private key	String	Yes
`keystore`	Keystore file name	File	Yes
`jar`	The JAR file to sign*	String	No Either this attribute or a nested `<fx:fileset>` element is required.
`storepass`	Password to check integrity of the keystore or unlock the keystore	String	Yes
`storetype`	Keystore type	String	No Default is `jks`.
`verbose`	Enable verbose output.	Boolean	No Default is `false`.

*Note that:

```
<fx:signjar jar="path/to/jar/folder/jarname" .../>
```

is simply a convenience syntax for the following:

```
<fx:signjar ...>
    <fileset dir="path/to/jar/folder" file="jarname"/>
</fx:signjar>
```

Parameters Accepted as Nested Elements

- <fx:fileset>

<fx:signjar> Usage Examples

Example 1 Sign JAR Files
The following snippet of Ant code shows how to sign JAR files using the new sign as BLOB technique.

```
<fx:signjar destdir="dist"
        keyStore="sampleKeystore.jks" storePass="****"
        alias="javafx" keyPass="****">
    <fileset dir='dist/*.jar'/>
</fx:signjar>
```

JavaFX Ant Helper Parameter Reference

Helper parameters are types that are used by the JavaFX tasks described in JavaFX Ant Task Reference. This reference page contains the following elements:

- <fx:add-modules>

- <fx:application>
- <fx:argument>
- <fx:association>
- <fx:bundleArgument>
- <fx:callback>
- <fx:callbacks>
- <fx:fileset>
- <fx:htmlParam>
- <fx:icon>
- <fx:info>
- <fx:jvmarg>
- <fx:limit-modules>
- <fx:module-path>
- <fx:param>
- <fx:permissions>
- <fx:platform>
- <fx:preferences>
- <fx:property>
- <fx:resources>
- <fx:runtime>
- <fx:secondaryLauncher>
- <fx:splash>
- <fx:template>

Items are in alphabetical order.

<fx:add-modules>

Description

List of modules to add to the runtime generated for a self-contained application. All modules can be added in a single instance of `<fx:add-modules>` using a comma-separated list, or an instance of `<fx:add-modules>` can be used for each module to add.

Parent Elements

- <fx:runtime>

Parameters

Attribute	Description	Type	Required?
`value`	One or more modules to include in the runtime. For more than one, separate the modules with a comma.	String	No

Parameters Accepted as Nested Elements

None.

<fx:add-modules> Usage Examples

Example 1 Include modules from `jdk.packager.services` and `javafx.controls` in the runtime

Use a single instance of `<fx:add-modules>`:

```
<fx:runtime>
  <fx:add-modules value="jdk.packager.services,javafx.controls"/>
</fx:runtime>
```

Use an instance of `<fx:add-modules>` for each module:

```
<fx:runtime>
  <fx:add-modules value="jdk.packager.services"/>
  <fx:add-modules value="javafx.controls"/>
</fx:runtime>
```

<fx:application>

Description

Basic application descriptor. It defines the main components and default set of parameters of the application.

Parent Elements

- <fx:deploy>

Parameters

Attribute	Description	Type	Required?
`daemon`	Indicates if the application is installed as a daemon or a service.	Boolean	No Default value is `false`.
`id`	Application ID that can be used to get a JavaScript reference to the application in HTML. The same ID can be used to refer to an application object in the Ant task (using `refid`).	String	No

Attribute	Description	Type	Required?
`mainClass`	Qualified name of the main application class, which should extend `javafx.application.Application`	String	Yes
`module`	Main module of a modular application. This is used as the default root module when constructing the application's initial module graph.	String	Yes, if bundling a modular application. Invalid if the application is not modular.
`name`	Short name of the application. For self-contained applications, also defines the name of the output package.	String	No Default value is derived from the main application class.
`preloaderClass`	Qualified name of the preloader class, which should extend `javafx.application.Preloader`	String	No Default is the preloader that is shipped with the JavaFX Runtime.
`refid*`	--	Reference	No
`toolkit`	Indicates your preference for the application to use a specific UI toolkit. Possible values: • fx • swing	String	No Default value is `fx`.
`version`	Version of the application being packaged.	String	No Default value is 1.0.

* If `refid` is used, then none of the other parameters can be specified.

Parameters Accepted as Nested Elements

- <fx:argument>
- <fx:htmlParam>
- <fx:param>

<fx:application> Usage Examples

Example 1 Application description for non-modular application

```
<fx:application id="HelloWorldID"
    name="JavaFXHelloWorldApp"
    mainClass="HelloWorld"/>
```

Example 2 Application description for modular application

```
<fx:application id="MinesweeperFX"
    name="MinesweeperFX"
    module="fx.minesweeper"
    mainClass="minesweeper.Minesweeper"/>
```

See Example 3-3 for an example of a complete ant script.

<fx:argument>

Description

An unnamed argument that is inserted in the `<fx:argument>` element in the deployment descriptor. Multiple arguments are added to the list of arguments in the same order as they are listed in the Ant script.

Parent Elements

- <fx:application>

Parameters

None.

Parameters Accepted as Nested Elements

None.

<fx:argument> Usage Examples

Example 1 Passing Various Unnamed Arguments

```
<fx:application name="Sample app"
        mainClass="test.MyApplication">
    <!-- unnamed arguments -->
    <fx:argument>Something</fx:argument>
    <!-- value with spaces that are generated at build time -->
    <fx:argument>JRE version: ${java.version}</fx:argument>
    <!-- example of value using a special character -->
    <fx:argument>true &amp; false</fx:argument>
</fx:application>
```

<fx:association>

Description

Associates file extensions or MIME types with a self-contained application. Multiple instances of this element are supported.

Parent Element

- <fx:info>

Parameters

Attribute	Description	Type	Required?
`description`	Description of the types of tiles that are associated with the application. If no description is provided, then *application-name* `File` is used.	String	No
`extension`	One or more file extensions to associate with the application. Separate values with a space, for example, `aaa bbb ccc`. Wild card symbols are not allowed.	String	Depends on the bundler[1]
`mimetype`	MIME type of the files to associate with the application. Wild card symbols are not allowed. Only one value is allowed on Windows, multiple values separated by a space can be provided for Linux or macOS.	String	Depends on the bundler[2]
`icon`	Name of the file that contains the icon for files associated with this application. For Windows the icon format must be `.ico`, for Linux the format must be `.png`, for macOS the format must be `.icns`. If no file name is provided, then the icon for the application is used.	String	No

1 Required on Windows. Either `extension` or `mimetype` must be provided on macOS.

2 Required on Linux. Either `extension` or `mimetype` must be provided on macOS.

Parameters Accepted as Nested Elements

None.

<fx:association> Usage Example

Example 1 Associate Files with a Self-Contained Application
This example associates a self-contained application on Windows with files that have the file extension `.aaa` or `.bbb`. and provides a description and icon.

```
<fx:info title="Association example">
  <fx:association extension="aaa bbb" description="MyApp Data Files"
      icon="MyApp.ico">
  </fx:association>
</fx:info>
```

<fx:bundleArgument>

Description

Specifies an argument for the bundler that is used to create self-contained applications. Each type of bundler has its own set of arguments.

Parent Elements

- <fx:deploy>

Parameters

Attribute	Description	Type	Required?
`arg`	Name of bundler argument	String	Yes
`value`	Value of bundler argument	String	Yes

Arguments for Self-Contained Application Bundlers

Each type of bundler (macOS, Linux, and Windows) has its own set of arguments.

General Bundler Arguments

dropinResourcesRoot
Directory in which to look for bundler-specific drop-in resources. For example, on macOS, to look in the current directory for the `Info.plist` file, use the following:

```
<fx:bundleArgument arg="dropinResourcesRoot" value="."/>
```

The file is then found in the current directory: `package/macosx/Info.plist`.

preferencesID
ID of a <fx:preferences> element to check for JVM options that the user can override.

macOS Application Bundler Arguments

icon
The path to the default icon to use for launchers and other assists. The file format is `.icns`.

mac.bundle-id-signing-prefix
Prefix that is applied to the signed binary when binaries that lack property list files (plists, which use the extension `.plist`) or existing signatures are found inside the bundles.

mac.category
Category for the application. The category must be in the list of categories found on the Apple Developer website.

mac.CFBundleIdentifier
Value stored in the `info.plist` file for `CFBundleIdentifier`. This value must be globally unique and contain only letters, numbers, dots, and dashes. Reverse DNS order is recommended, for example, `com.example.application.my-application`.

mac.CFBundleName
Name of the application as it appears on the Mac Menu Bar. A name of less than 16 characters is recommended. The default is the `name` attribute of the <fx:application> element.

mac.CFBundleVersion
Version number for the application, used internally. The value must be at least one integer and no more than three integers separated by periods (.) for example, 1.3 or 2.0.1. The value can be different than the value for the `version` attribute of the `<fx:application>` element. If the `version` attribute is specified with a valid value and the `mac.CFBundleVersion` argument is not specified, then the value for the `version` attribute is used. If neither value is specified, 100 is used as the version number.

mac.signing-key-developer-id-app
Name of the signing key used for Developer ID or Gatekeeper signing. If you imported a standard key from the Apple Developer Website, then that key is used by default. If no key can be identified, then the application is not signed.

macOS DMG (Disk Image) Bundler Arguments

licenseFile
Location of the End User License Agreement (EULA) to be presented or recorded by the bundler. The path is relative to the packaged application resources.

mac.CFBundleVersion
Version number for the application, used internally. The value must be at least one integer and no more than three integers separated by periods (.) for example, 1.3 or 2.0.1. The value can be different than the value for the `version` attribute of the `<fx:application>` element. If the `version` attribute is specified with a valid value and the `mac.CFBundleVersion` argument is not specified, then the value for the `version` attribute is used. If neither value is specified, 100 is used as the version number.

mac.dmg.simple
Flag that indicates if DMG customization steps that depend on executing AppleScript code are skipped. Set to `true` to skip the steps. When set to `true`, the disk window does not have a background image, and the icons are not moved into place. If the `systemWide` argument is also set to `true`, then a symbolic link to the root Applications folder is added to the DMG file. If the `systemWide` argument is set to `false`, then only the application is added to the DMG file, no link to the desktop is added.

macOS PKG Bundler Arguments

licenseFile
Location of the End User License Agreement (EULA) to be presented or recorded by the bundler. The path is relative to the packaged application resources.

mac.CFBundleVersion
Version number for the application, used internally. The value must be at least one integer and no more than three integers separated by periods (.) for example, 1.3 or 2.0.1. The value can be different than the value for the `version` attribute of the `<fx:application>` element. If the `version` attribute is specified with a valid value and the `mac.CFBundleVersion` argument is not specified, then the value for the `version` attribute is used. If neither value is specified, 100 is used as the version number.

mac.signing-key-developer-id-installer
Name of the signing key used for Developer ID or Gatekeeper signing. If you imported a standard key from the Apple Developer Website, then that key is used by default. If no key can be identified, then the application is not signed.

Mac App Store Bundler Arguments

mac.app-store-entitlements
Location of the file that contains the entitlements that the application operates under. The file must be in the format specified by Apple. The path to the file can be specified in absolute terms, or relative to your Ant file. If no entitlements are specified, then the application operates in a sandbox that is stricter than the typical applet sandbox, and access to network sockets and all files is prevented.

mac.CFBundleVersion
Version number for the application, used internally. The value must be at least one integer and no more than three integers separated by periods (.) for example, 1.3 or 2.0.1. The value can be different than the value for the `version` attribute of the `<fx:application>` element. If the `version` attribute is specified with a valid value and the `mac.CFBundleVersion` argument is not specified, then the value for the `version` attribute is used. If neither value is specified, 100 is used as the version number. If this version is an upgrade for an existing application, the value must be greater than the previous version number.

mac.signing-key-app
Name of the application signing key for the Mac App Store. If you imported a standard key from the Apple Developer Website, then that key is used by default. If no key can be identified, then the application is not signed.

mac.signing-key-pkg
Name of the installer signing key for the Mac App Store. If you imported a standard key from the Apple Developer Website, then that key is used by default. If no key can be identified, then the application is not signed.

Linux Bundler Arguments

icon
The path of the default icon to use for launchers and other assists. The file format is `.png`.

linux.bundleName
Name of the RPM or DEB package to create.

Windows EXE and MSI Bundler Arguments

icon
The path of the default icon to use for launchers and other assists. The file format is `.ico`.

licenseFile
Location of the End User License Agreement (EULA) to be presented or recorded by the bundler. The path is relative to the packaged application resources.

win.menuGroup
Menu group in which to install the application when the `menu` attribute of the <fx:preferences> element is `true`. This argument is ignored when `menu` is `false`.

<fx:bundleArgument> Usage Example

The following example specifies that the generated Windows Installer Package (MSI file) create a Start Menu shortcut in a menu group named **Sample Applications**. Note that you must specify that the bundler create an MSI file with the `nativeBundles` attribute of the <fx:deploy> element.

```
<fx:deploy outdir="." outfile="helloworld" nativeBundles="msi">
    <fx:platform basedir="${JAVA_HOME}"/>
    <fx:application refId="HelloWorldID"/>
    <fx:resources refid="appRes"/>
    <fx:info title="Hello World Example" vendor="Oracle Corporation"/>
    <fx:bundleArgument arg="win.menuGroup" value="Sample Applications"/>
</fx:deploy>
```

<fx:callback>

Description

Defines a JavaScript callback that can be used to customize user experience.

Parent Elements

- <fx:callbacks>

Parameters

Attribute	Description	Type	Required?
`name`	Name of the event for callback.	String	Yes
`refid*`	--	Reference	No

* If `refid` is used, then none of the other parameters can be specified.

Parameters Accepted as Nested Elements

<TEXT>

<fx:callback> Usage Examples

Example 1 A Callback Calling a JavaScript Function

In this example, a callback is used to create an HTML splash screen for an application embedded in a web page. When the event `onGetSplash` is triggered, the JavaScript function `customGetSplash` is executed.

```
<fx:callbacks>
    <fx:callback name="onGetSplash">customGetSplash</fx:callback>
</fx:callbacks>
```

Example 2 A Callback with JavaScript Inserted

In this example, the callback is defined with JavaScript code in the `<fx:callback>` element itself.

```
<fx:callbacks>
    <fx:callback name="onLoadHandler">
        function () {perfLog(0, "onLoad called");}
    </fx:callback>
</fx:callbacks>
```

Example 3 Multiple Callbacks

```
<fx:callbacks>
    <fx:callback name="onJavascriptReady">callAppFunction</fx:callback>
    <fx:callback name="onGetSplash">function(id) {}</fx:callback>
 </fx:callbacks>
```

<fx:callbacks>

Description

Collection of JavaScript callbacks to be used to customize the user experience.

Parent Elements

- <fx:deploy>

Parameters

Attribute	Description	Type	Required?
`refid*`	--	Reference	No

* If `refid` is used, then none of the other parameters can be specified.

Parameters Accepted as Nested Elements

- <fx:callback>

<fx:callbacks> Usage Examples

See the examples for <fx:callback>.

<fx:fileset>

Description

Extension of the standard Ant `FileSet` type, which provides the means to specify optional meta information about a selected set of files. This includes:

- Type of resource (see the `type` attribute)
- Operating system and architecture for which this resource is applicable
- When this resource is needed, which helps to optimize loading order

Depending on type, the resource might not be used by the enclosing task.

A fileset of type `"jar"` is expected to contain a set of JAR files to be added to the classpath.

Resource of type `"native"` is expected to be a JAR file with a set of native libraries. In most of cases, it makes sense to set the operating system and architecture for this resource too.

Resources of type `"jnlp"` are expected to contain JNLP files defining external JNLP extensions.

Filesets of type `"license"` can contain arbitrary files, but additional restrictions can be applied when they are actually used (for example, on Mac it has to be a plain text file, and on Windows it needs to be RTF).

Filesets of type `"data"` can contain arbitrary files.

Parent Elements

- <fx:jar>
- <fx:resources>

Parameters

Attribute	Description	Type	Required?
`arch` (used only when `<fx:fileset>` is nested under `<fx:resources>`	Specifies the architecture for which these resources should be considered.	String	No Default is `any`.
`dir`	Specifies the root directory that contains the files used in the task.	String	Yes
`excludes`	Specifies files in the `dir` directory that are excluded from the task. Use an asterisk (*) as a wild card to match multiple files, for example, `*test*`.	String	No

Attribute	Description	Type	Required?
`includes`	Specifies files in the `dir` directory that are included for the task. Use an asterisk (*) as a wild card to match multiple files, for example, `**/*.jar`.	String	No
`os` (used only when `<fx:fileset>` is nested under `<fx:resources>`	Specifies the operating systems for which these resources should be considered.	String	No Default is `any`.
`requiredFor` (used only when `<fx:fileset>` is nested under `<fx:resources>`	Defines when resources are needed (affects loading priority). Supported values are: • `preloader` - resources are needed to launch the preloader (first thing to be executed) • `startup` - resources are needed to launch the application. • `runtime` - resources are not critical to launch the application but may be needed later.	String	No Default is `startup`.
`type` (used only when `<fx:fileset>` is nested under `<fx:resources>`	Type of the resources in the set. Supported values are: • `auto` for autodetect • data • `jar` • `jnlp` • license • `native` for JAR files containing native libraries • `icon`	String	No Default is to guess based on extension.

* If `refid` is used, then none of the other parameters can be specified.

Parameters Accepted as Nested Elements

None (except standard Ant elements).

<fx:htmlParam>

Description

Parameter to be passed to the embedded or Web Start application from the HTML page. The value of the parameter can be calculated at runtime using JavaScript.

Parent Elements

- <fx:application>

Parameters

Attribute	Description	Type	Required?
`escape`	Defines how to interpret the value for the values that are passed—as string literal (true) or JavaScript variable (false).	Boolean	No Default is true, meaning value is treated as string literal.
`name`	Name of the parameter to be passed to the embedded or Web Start application from the HTML page.	String	Yes
`value`	Value of the parameter. Could also be the name of a JavaScript variable whose value is expected to be passed as parameter. For JavaScript variables, ensure `escape` is set to false.	String	Yes

Parameters Accepted as Nested Elements

None

<fx:htmlParam> Task Usage Examples

Example 1 Various Parameters Passed from HTML Page

```
<fx:application name="Sample app"
        mainClass="test.MyApplication">
    <!-- Parameters passed from HTML page. Only applicable
         for embeddeded and Web Start applications and unused when
         run in a standalone and self-contained context.  -->
    <!-- Parameter with name 'fixedParam', whose value is string
        '(new Date()).getTime()' -->
    <htmlParam name="fixedParam"
           value="(new Date()).getTime()"/>
    <!-- Parameter with name 'dynamicParam', whose value will be
         the timestamp of the moment when the application is added
         to the web page (value will be assigned the result
         of execution of JavaScript code) -->
    <htmlParam name="dynamicParam" escape="false"
            value="(new Date()).getTime()"/>
</fx:application>
```

<fx:icon>

Description

Passes an icon to the `<fx:deploy>` task, other than a splash screen image.

The icon specified in this element is used for Web Start and desktop applications.

Note that in JavaFX 2.2, only icons of type `default` are used for self-contained applications. For details on how to customize the icon for self-contained applications, see Customizing the Package Using Drop-In Resources.

Parent Elements

- <fx:info>

Parameters

Attribute	Description	Type	Required?
`depth`	Color depth of the image in bits-per-pixel. Common values are 8, 16, and 24.	String	No
`href`	Location of image. For self-contained applications, the supported graphic formats depend on the operating system: • macOS: `.icns` • Linux: `.png` • Windows: `.ico` Note that if you specify your own icon for a self-contained application, the values of the attributes `height`, `width`, and `depth` are not used. Ensure that the size and the color depth of your custom icons correspond to what your operating system expects. For Web Start applications, the supported graphic formats are `.gif`, `.jpg`, `.png`, and `.ico`.	String	Yes
`height`	Image height in pixels	String	No

Attribute	Description	Type	Required?
`kind`	Icon type. Supported values are: • `default` • `disabled` • `rollover` • `selected` • `shortcut` For Web Start applications, this attribute corresponds to the `kind` attribute in the `icon` element of a JNLP file. For desktop applications, if this attribute is not specified or set as `default`, then the icon specified is the icon of the launcher (`.exe` file for Windows, `.app` file for macOS), the icon in the taskbar or dock, and in some instances the icon for the installer package (such as a DMG package).	String	No Default value is `default`.
`width`	Image width in pixels	String	No

Parameters Accepted as Nested Elements

None.

<fx:icon> Usage Examples

Example 1 Use of <fx:icon>

```
<fx:info title="Sample application">
    <!-- icon to be used by default for anything but splash -->
    <fx:icon href="shortcut.ico" kind="shortcut"
            width="32" height="32" depth="8"/>
</fx:info>
```

<fx:info>

Description

Application description for users. These details are shown in the system dialog boxes, if they need to be shown.

Parent Elements

- <fx:deploy>

Parameters

Attribute	Description	Type	Required?
`category`	Application category. Creates a link to an application in a specified category. Semantics of the value depends on the format of the package. For example: • For a self-contained application on Linux, it is used to define the application menu category where the application is listed. • On Mac: Creates key in `info.plist` `<key>LSApplicationCategoryType</key>` `<string>unknown</string>` • On Windows creates a group, for instance, if you specify "My Music" it will create your app in `C:\ProgramData\Microsoft\Windows\Start Menu\Programs\My Music`	String	No
`copyright`	Short copyright statement	String	No
`description`	A short statement describing the application.	String	No
`license`	License type (for example, GPL). As of JavaFX 2.2, this attribute is used only for Linux bundles.	String	No
`title`	Title of the application	String	Yes
`vendor`	Provider of the application	String	Yes

Parameters Accepted as Nested Elements

- <fx:icon>
- <fx:splash>

<fx:info> Usage Examples

Example 1 <fx:info> Parameter Used in <fx:deploy> Task

```
<fx:info vendor="Uncle Joe" description="Test program"/>
```

<fx:jvmarg>

Description

The JVM argument to be set in the JVM, where the application is executed. Can be used multiple times. Note that you do not need to additionally escape values if they contain space characters.

Parent Elements

- <fx:platform>

Parameters

Attribute	Description	Type	Required?
`value`	Value of JVM argument.	String	Yes

Parameters Accepted as Nested Elements

None.

<fx:jvmarg> Usage Examples

See Example 2, <fx:platform> Parameter to Specify JVM Options.

<fx:jvmuserarg>

Description

The user overridable JVM argument to be set in the JVM, where the application is executed. Can be used multiple times. Note that you do not need to additionally escape values if they contain space characters.

Parent Elements

- <fx:platform>

Parameters

Attribute	Description	Type	Required?
`value`	Value of JVM argument.	String	Yes

Parameters Accepted as Nested Elements

None.

<fx:jvmuserarg> Usage Examples

See Example 2, <fx:platform> Parameter to Specify JVM Options.

<fx:limit-modules>

Description

Limit the set of observable modules to those in the transitive closure of the list provided plus the main module, if any, plus any further modules specified in the `<fx:add-modules>` element.

Parent Elements

- <fx:runtime>

Parameters

Attribute	Description	Type	Required?
`value`	Comma-separated list of modules used to limit the universe of observable modules.	String	No

Parameters Accepted as Nested Elements

None.

<fx:limit-modules> Usage Examples

Example 1 Limit observable modules to`jdk.packager.services` and `javafx.controls`

```
<fx:limit-modules value="jdk.packager.services,javafx.controls"/>
```

<fx:module-path>

Description

Path to the application modules to include in the generated runtime.

Parent Elements

- <fx:runtime>

Parameters

Attribute	Description	Type	Required?
`value`	List of module locations. On Linux and macOS, use a colon (:) to separate paths. On Windows, use a semicolon (;).	String	No

Parameters Accepted as Nested Elements

None.

<fx:module-path> Usage Examples

Example 1 On Linux, locate application modules in `${java.home}/../images/jmods` and `${build.dir}/modules`

```
<fx:runtime>
   <fx:module-path value="${java.home}/../images/jmods:${build.dir}/modules"/>
</fx:runtime>
```

<fx:param>

Description

Parameter to be passed to the application (embedded into application package).

This tag has no impact on standalone applications, including self-contained applications.

Parent Elements

- <fx:application>

Parameters

Attribute	Description	Type	Required?
`name`	Name of parameter	String	Yes
`value`	Value of parameter	String	Yes

Parameters Accepted as Nested Elements

None.

<fx:param> Task Usage Examples

Example 1 Passing Various Types of Parameters

```
<fx:application name="Sample app"
        mainClass="test.MyApplication">
    <!-- parameter with name 'simpleParam' and fixed string value-->
    <param name="simpleParam" value="something"/>
    <!-- parameter with name 'complexParam' with value generated
         at build time -->
    <param name="complexParam" value="Compiled by ${java.version}"/>
    <!-- parameter with name 'novalueParam' and no value -->
    <param name="novalueParam"/>
</fx:application>
```

<fx:permissions>

Description

Definition of security permissions needed by application. By default, the application runs in the sandbox. Requesting elevated permissions requires signing the application JAR files.

This option has no impact on standalone applications, including self-contained applications.

Parent Elements

- <fx:deploy>
- <fx:jar>

Parameters

Attribute	Description	Type	Required?
`elevated`	If set to false, the application runs in the sandbox.	Boolean	No Default is false.

Parameters Accepted as Nested Elements

None.

<fx:permissions> Usage Examples

Example 1 Run application with elevated permissions

```
<fx:permissions elevated="true"/>
```

<fx:platform>

Description

Defines application platform requirements.

Parent Elements

- <fx:deploy>
- <fx:jar>

Parameters

Attribute	Description	Type	Required?
`refid*`	--	Reference	No
`j2se`	Minimum version of JRE required by the application.	String	No Default is any JRE supporting JavaFX.

* If `refid` is used, then none of the other parameters can be specified.

Parameters Accepted as Nested Elements

- <fx:jvmarg>
- <fx:property>

<fx:platform> Usage Examples

Example 1 <fx:platform> Parameter to Specify Version
In this example, the application needs JRE version 9.0 or later.

```
<fx:platform j2se="9.0"/>
```

Example 2 <fx:platform> Parameter to Specify JVM Options
In this example, the application needs JRE version 9.0 or later and needs to run in a JVM launched with `"-Xmx400 -verbose:jni -Dpurpose="sample value"`.

```
<fx:platform j2se="9.0">
    <fx:jvmarg value="-Xmx400m"/>
    <fx:jvmarg value="-verbose:jni"/>
    <property name="purpose" value="sample value"/>
</fx:platform>
```

Example 3 <fx:platform> Parameter to Specify User Overridable JVM Options
In this example, -Xmx768m is passed as a default value for heap size. The user can override this value in a user configuration file.

```
            <fx:platform>
              <fx:jvmuserarg name="-Xmx" value="768m" />
            </fx:platform>
```

<fx:preferences>

Description

Deployment preferences for the application. Preferences can be expressed but may not necessarily be satisfied, for example in the following cases:

- The packager may ignore a preference if it is not supported for a particular execution mode.
- JRE may ignore it if it is not supported.
- The user may reject a request, for example if he is prompted whether a desktop shortcut can be created.

Parent Elements

- <fx:deploy>

Parameters

Attribute	Description	Type	Required?
`install`	Install `true` means that the application is installed for the system and `false` means the application is installed for the current user only. For self-contained applications, `true` indicates a developer preference that the application package should perform a system-wide installation. If `false`, then a package is generated for per-user installation. This value is ignored if the packager does not support different types of install packages for the requested package format.	Boolean	No For Web Start and embedded applications, default is `false`. For self-contained applications, default value is different for various package formats.
`installdirChooser`	If `true`, then a dialog is shown for the user to choose the directory in which the application is installed. If `false`, then the `install` attribute is used to determine where the application is installed.	Boolean	No Default is `false`.
`menu`	If `true`, then the application requests to add an entry to the system application menu.	Boolean	No Default is `false`.
`refid*`	--	Reference	No
`shortcut`	If `true`, then application requests a desktop shortcut to be created.	Boolean	No Default is `false`.

* If `refid` is used, then none of the other parameters can be specified.

Parameters Accepted as Nested Elements

None.

<fx:preferences> Usage Examples

Example 1 <fx:preferences> Parameter to Add a Desktop Shortcut
This example shows a request to create a desktop shortcut.

```
<fx:preferences id="p1" shortcut="true"/>
```

Example 2 <fx:preferences> Parameter to Mark as Installed
This example does the following:

- It requests creation of a web deployment descriptor that will add the application to the Applications Menu and mark it as installed (in other words, the application will be listed in Add/Remove programs.)
- If self-contained bundles are created, then they will be installed system-wide and will create an application entry in the Applications menu.

```
<fx:preferences shortcut="false" install="true" menu="true"/>
```

Example 3 Using a refid to the <fx:preferences> Parameter
This example uses a reference to the <fx:preferences> parameter in Example 1, <fx:preferences> Parameter to Add a Desktop Shortcut to create the shortcut.

```
<fx:resource refid="p1"/>
```

<fx:property>

Description

Optional element and can be used multiple times. Java property to be set in the JVM where the application is executed.

Parent Elements

- <fx:platform>

Parameters

Attribute	Description	Type	Required?
`name`	Name of property to be set.	String	Yes
`value`	Value of property to be set.	String	Yes

Parameters Accepted as Nested Elements

None.

<fx:resources>

Description

The collection of resources used by the application. Defined as a set of JavaFX FileSet filters. Could be reused using `id` or `refid`.

Parent Elements

- <fx:deploy>
- <fx:jar>

Parameters

Attribute	Description	Type	Required?
`id`	ID that can be referred from another element with a refid attribute.	String	No

Attribute	Description	Type	Required?
refid*	--	Reference	No

* If refid is used, then none of the other parameters can be specified.

Parameters Accepted as Nested Elements

- <fx:fileset>

<fx:resources> Usage Examples

Example 1 <fx:resources> Parameters Used with id and refid Attributes
In this example, both <fx:resources> elements define the collection, consisting of s.jar in the dist directory. The first <fx:resources> element uses an id attribute, and the second <fx:resources> element refers to the first with the refid attribute.

```
<fx:resources id="aaa">
    <fx:fileset dir="dist" includes="s.jar"/>
</fx:resources>
<fx:resources refid="aaa"/>
```

Example 2 Using <fx:resources> for Extension Descriptors
If you mix signed and unsigned JAR files, use an additional <fx:deploy> Ant task to generate an extension descriptor for each JAR file, and refer to the extension descriptors by treating them as resources in the main file, as shown in this example.

```
<!-- Prepare extension -->
<fx:deploy extension="true"
        outdir="dist" outfile="other">
    ...
<fx:deploy>

<!-- Use it in the main descriptor -->
<fx:deploy outdir="web-dist" ...>
    ...
    <fx:resources>
        <fx:fileset dir="dist" includes="other.jnlp"/>
            ...
    </fx:resources>
<fx:deploy>
```

Additional examples are available in Self-Contained Application Packaging.

<fx:runtime>

Description

Runtime generated for your self-contained application. The jlink tool is used to generate a runtime that contains only the packages that the application needs to run, and optionally, command-line tools such as java.exe.

The Java Packager for JDK 9 generates a JDK 9 runtime image. To package a JDK 8 or JDK 7 JRE with your application, use the JDK 8 Java Packager.

Parent Elements

- <fx:deploy>

Parameters

Attribute	Description	Type	Required?
`strip-native-commands`	If set to `true`, command-line tools such as `java.exe` are removed from the generated runtime. If set to `false`, command-line tools are included in the generated runtime.	Boolean	No Default is true.

Parameters Accepted as Nested Elements

- <fx:add-modules>
- <fx:limit-modules>
- <fx:module-path>

<fx:runtime> Usage Examples

Example 1 Generate a runtime that contains command line tools

```
<fx:runtime strip-native-commands="false"/>
```

Example 2 Include modules from `jdk.packager.services` and `javafx.controls` in the runtime

```
<fx:runtime>
  <fx:add-modules value="jdk.packager.services,javafx.controls"/>
</fx:runtime>
```

Example 3 Include modules from `jdk.packager.services` and provide the location of application modules

```
<fx:runtime>
  <fx:add-modules value="jdk.packager.services"/>
  <fx:module-path value="${java.home}/../images/jmods"/>
  <fx:module-path value="${build.dir}/modules"/>
</fx:runtime>
```

<fx:secondaryLauncher>

Description

Identifies a secondary entry point for a self-contained application and the main module for a modular application. This parameter is ignored for standalone applications, applications embedded in a web page, or applications launched from a browser.

This parameter is valid only for Windows and Linux applications.

Parent Elements

- <fx:deploy>

Parameters

Attribute	Description	Type	Required
`appDescription`	Brief description of the application.	String	No
`icon`	Path to the icon file for the application.	String	No
`mainClass`	Qualified name of the main application class.	String	No
`menu`	Flag that indicates if the application is added to the Start menu. Either the `menu` attribute or `shortcut` attribute must be set to `true`. If both attributes are set to `false`, then the application is added to the Start menu.	Boolean	No Default is `true`.
`module`	Main module of a modular application. This is used as the default root module when constructing the application's initial module graph.	String	Yes, if bundling a modular application. Invalid if the application is not modular
`name`	Name of the launcher that is used to start the application.	String	Yes
`shortcut`	Flag that indicates if a shortcut to the application is added to the desktop. Either the `menu` attribute or `shortcut` attribute must be set to `true`. If both attributes are set to `false`, then the application is added to the Start menu.	Boolean	No Default is `false`.
`title`	Title of the application.	String	No
`vendor`	Name of the vendor that provided the application.	String	No
`version`	Version of the application.	String	No

Parameters Accepted as Nested Elements

- <fx:argument>

- <fx:bundleArgument>, only the `icon` argument is valid when used with `<fx:secondaryLauncher>`
- <fx:jvmarg>
- <fx:jvmuserarg>
- <fx:property>

<fx:secondaryLauncher> Usage Example

Example 1 Deploy task for non-modular application with secondary launchers
See Example 3, <fx:deploy> Task with Secondary Launchers.

Example 2 Secondary launcher for modular application

```
<fx:secondaryLauncher name="HelloWorldModular"
    module="hello.world"
    mainClass="com.sample.app.HelloWorld">
</fx:secondaryLauncher>
```

<fx:splash>

Description

Passes the location of the image to be used as a splash screen. Currently custom splash images can only be passed to Web Start applications, and use of this parameter has no impact on standalone applications or applications embedded into web pages.

Parent Elements

- <fx:info>

Parameters

Attribute	Description	Type	Required?
`href`	Location of image. Supported graphic formats are JPEG, GIF, and PNG.	String	Yes
`mode`	Deployment mode. Supported values are: • any (but currently only functional in Web Start mode) • `webstart`	String	No Default value is `any`.

Parameters Accepted as Nested Elements

None.

<fx:splash> Usage Examples

Example 1 Use of <fx:splash>
In the following example, splash images of various types are passed.

```
<fx:info title="Sample application">
    <fx:splash href="http://my.site/custom.gif"/>
</fx:info>
```

<fx:template>

Description

Template to preprocess. A template is an HTML file that contains markers to be replaced with the JavaScript or HTML snippets that are required for web deployment. Using templates enables you to deploy your application directly into your own web pages. This simplifies the development process, especially when the application is tightly integrated with the page, for example when the web page uses JavaScript to communicate to the application.

Template markers have one of the following forms:

- `#`*XXX*`#`
- `#`*XXX*`(`*id*`)#`

id is the identifier of an application and *XXX* is one of following:

- `DT.SCRIPT.URL`

  Location of `dtjava.js` in the Deployment Toolkit. By default, the location is

  `https://java.com/js/dtjava.js`.
- `DT.SCRIPT.CODE`

  Script element to include `dtjava.js` of the Deployment Toolkit.
- `DT.EMBED.CODE.DYNAMIC`

  Code to embed the application into a given placeholder. It is expected that the code will be wrapped in the `function()` method.
- `DT.EMBED.CODE.ONLOAD`

  All the code needed to embed the application into a web page using the `onload` hook (except inclusion of `dtjava.js`).
- `DT.LAUNCH.CODE`

  Code needed to launch the application. It is expected that the code will be wrapped in the `function()` method.

A page with different applications can be processed multiple times, one per application. To avoid confusion, markers must use application IDs with an alphanumeric string and no spaces.

If the input and output files are the same then the template is processed in place.

Parent Elements

- <fx:deploy>

Parameters

Attribute	Description	Type	Required?
`file`	Input template file.	File	Yes

Attribute	Description	Type	Required?
`tofile`	Output file (after preprocessing).	File	No Default is the same as the input file.

Parameters Accepted as Nested Elements

None

<fx:template> Usage Examples

Example 1 <fx:template> Parameter Used in <fx:deploy> Task

This example shows a `<fx:template>` parameter in which both input and output files are specified.

```
<fx:template file="App_template.html" tofile="App.html"/>
```

Example 2 <fx:template> Parameter in Context

```
<fx:deploy placeholderId="ZZZ"
        width="600" height="400"
        outdir="dist-web" outfile="App1">
    <fx:application id="myApp" name="Demo"
            mainClass="fish.FishApplication"/>
    <fx:template file="src/templates/EmbedApp_template.html"
            tofile="dist-web/EmbedApp.html"/>
    <fx:resources>
        <fx:fileset requiredFor="startup" dir="dist" includes="*.jar"/>
    </fx:resources>
</fx:deploy>
```

Part III
Java Web Start Technology

The topics in this part describe Java Web Start technology and how to use it to deploy Java applications.

4

Overview of Java Web Start Technology

This chapter includes the following topics:

- Introduction to Java Web Start
- Using Java Web Start Software
- Setting Up the Web Server
- Installing the Java Web Start Protocol Handler

Introduction to Java Web Start

Java Web Start is an application-deployment technology that enables your users to launch full-featured applications with a single click from any web browser. Users can download and launch applications without going through complicated installation procedures.

With Java Web Start, your users launch applications by clicking a web page link. If the application is not present on their computer, Java Web Start automatically downloads all necessary files. It then caches the files on the computer so that the application is always ready to be relaunched anytime the user wants—either from an icon on the desktop or from the browser link. No matter which method is used to launch the application, the most current version of the application is always presented.

The technology underlying Java Web Start is the Java Network Launching Protocol & API (JNLP). This technology was developed through the Java Community Process (JCP). Java Web Start is the reference implementation (RI) for the JNLP specification. The JNLP technology defines, among other things, the JNLP file, which is a standard file format that describes how to launch an application. The JNLP specification is available at JSR 56: Java Network Launching Protocol and API.

Benefits of Java Web Start

From a technology standpoint, Java Web Start has a number of key benefits that make it an attractive platform to use for deploying applications.

The benefits include the following:

- Java Web Start is built exclusively to launch applications written to the Java Platform, Standard Edition. Thus, a single application can be made available on a web server and then deployed on a wide variety of platforms, including Windows 7+, Linux, and macOS.
- Java Web Start supports multiple revisions of the Java Platform, Standard Edition. Thus, an application can request a particular version of the platform it requires, such as Java SE 9. Several applications can run at the same time on different platform revisions without causing conflicts.
- Java Web Start enables applications to be launched independently of a web browser. This can be used for off-line operation of an application, where launching

the application through the browser is inconvenient or impossible. The application can also be launched through desktop shortcuts, making launching the web-deployed application similar to launching a native application.

- Java Web Start takes advantage of security features of the Java Platform. Sandbox applications are run in a protective environment with restricted access to local disk and network resources. Users must also agree to run the application the first time it is launched.
- Applications launched with Java Web Start are cached locally. Thus, an already-downloaded application is launched similar to a traditionally installed application.

Where to Find Java Web Start

Java Web Start is included in the Java Platform, Standard Edition development kit (JDK) and Java Runtime Environment (JRE), and includes the security features of the Java platform.

The JDK and JRE are available from the java.com website.

Using Java Web Start Software

Java Web Start allows you to launch Java-technology-based applications directly from the Web. An application can be launched in three different ways:

- From a web browser by clicking a link
- From desktop icons or the Start Menu
- From the Java Cache Viewer

Regardless of which way is used, Java Web Start will connect back to the web server each time an application is launched, to check whether an updated version of the application is available.

Launching from a Web Browser

Point your web browser to a page with a link to a JNLP application, and click that link.

A security dialog box will pop up with information about the origin of the application based on who digitally signed the code, and the level of access requested. The application will run only if you decide to trust the vendor.

That is really all there is to using Java Web Start, but how does it work? The HTML links that launch the applications are, in fact, standard HTML links. However, instead of pointing to another web page, they link to a special configuration file called a JNLP file. The web browser examines the file extension or the MIME type of the file, and sees that it belongs to Java Web Start. It then launches Java Web Start with the downloaded JNLP file as an argument. Java Web Start proceeds with downloading, caching, and running the application as directed by the JNLP file.

Launching from Desktop Icons and the Start Menu (Microsoft Windows and Linux Running GNOME 2.0+)

Java Web Start technology can automatically create shortcuts for your application on the desktop and in the Start Menu for web-deployed applications developed with Java technology. You can use the Java Control Panel to control the shortcut settings. Shortcuts can also be added by using the Java Web Start Cache Viewer, using the install shortcut menu item.

Using Java Web Start Software Behind a Proxy Server or Firewall

Java Web Start software must be configured with the correct proxy settings in order to launch applications from outside your firewall. Java Web Start software automatically tries to detect the proxy settings from the default browser on your system (Internet Explorer or Mozilla browsers on Microsoft Windows, and Mozilla browsers on the Solaris Operating Environment and Linux). Java Web Start technology supports most web proxy auto-configuration scripts. It can detect proxy settings in almost all environments.

You can also use the Network Settings Tab in the Java Control Panel to view or edit the proxy configuration.

Setting Up the Web Server

Applications can be deployed from any standard web server. Java Web Start leverages existing internet technology, such as the HTTP protocol and web servers, so existing infrastructure for deploying HTML-based content can be reused to deploy Java Technology-based applications using Java Web Start.

To deploy your application to client machines, ensure that all files that contain your application are accessible through a web server. This typically requires copying one or more JAR files and a JNLP file into the web server's directories. Enabling the website to support Java Web Start is similar to deploying HTML-based content. In addition, to use Java Web Start, the web server must be configured to support the `application/x-java-jnlp-file` MIME type.

Step 1 Configure the web server to use the Java Web Start MIME type.
Many web servers come with the Java Web Start MIME type configured by default. If your web server does not, configure it so that all files with the `.jnlp` file extension are set to the `application/x-java-jnlp-file` MIME type.
Most web browsers use the MIME type returned with the contents from the web server to determine how to handle the particular content. The server must return the `application/x-java-jnlp-file` MIME type for JNLP files in order for Java Web Start to be invoked.
Each web server has a specific way in which to add MIME types. For example, for the Apache web server, you add the following line to the `.mime.types` configuration file:
`application/x-java-jnlp-file JNLP`
Check the documentation for the specifics of your web server.

Step 2 Create a JNLP file for the application.
The easiest way to create this file is to modify an existing JNLP file with your requirements. A simple JNLP file is shown in the following example:

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="9.0+"
  codebase="http://example.com/demos/helloworld"
  href="HelloWorld.jnlp">
  <information>
    <title>HelloWorld</title>
    <description>HelloWorld demo application</description>
  </information>

  <resources>
    <j2se version="9"/>
    <jar href="HelloWorld.jar" size="47013" download="eager" />
  </resources>

  <application-desc main-class="HelloWorld"/>
</jnlp>
```

The syntax and format for the JNLP file is described in JNLP File Syntax.

Step 3 Make the application accessible on the web server.
Ensure that your application's JAR files and the JNLP file are accessible at the URLs listed in the JNLP file.

Step 4 Create the web page that launches the application.
Create the web page and include one of the following options for starting a Java Web Start application:

- Use a link to the JNLP file, as shown in the following examples:

  ```
  <a href="/some/path/HelloWorld.jnlp">Launch HelloWorld demo</a>

  <a href="https://docs.oracle.com/javase/tutorialJWS/samples/deployment/
  dynamictree_webstartJWSProject/dynamictree_webstart.jnlp">Launch DynamicTree
  demo</a>

  <a href="http://docs.oracle.com/javase/tutorialJWS/samples/deployment/
  dynamictree_webstartJWSProject/dynamictree_webstart.jnlp">Launch DynamicTree
  demo</a>
  ```

- Use JavaScript, as shown in the following example:

  ```
  <script src="https://www.java.com/js/deployJava.js"></script>
  <script>
      var url = "https://docs.oracle.com/javase/tutorialJWS/samples/deployment/
  dynamictree_webstartJWSProject/dynamictree_webstart.jnlp";
      deployJava.createWebStartLaunchButton(url, '1.6.0');
  </script>
  ```

- Use a link with the `jnlp:` schema, as shown in the following example:

  ```
  <a href="jnlp:https://docs.oracle.com/javase/tutorialJWS/samples/deployment/
  dynamictree_webstartJWSProject/dynamictree_webstart.jnlp">Launch DynamicTree
  demo</a>
  ```

- Copy the JavaScript code from the HTML file generated by the Java Packager tool.

  If you are using the Java Packager tool, see Create the Web Page.

Installing the Java Web Start Protocol Handler

Java Web Start includes a protocol handler to handle the custom URI schemes `jnlp:` and `jnlps:`. Use these schemes as a direct way to start Java Web Start applications.

The protocol handler is automatically installed on Windows and macOS systems. It must be manually installed on Linux systems.

- Installing the Protocol Handler for Chrome

- Installing the Protocol Handler in Firefox

Installing the Protocol Handler for Chrome

If you use the Chrome browser on Linux, manually install the protocol handler that enables you to start Java Web Start applications using the `jnlp` or `jnlps` protocol.

In Linux, the `xdg-open` command is used to open a file or URL in the user's preferred application. To install the protocol handler, configure `xdg-open` to use Java Web Start for URLs that include the `jnlp` or `jnlps` protocol:

1. Use a text editor to create a file named `javaws.desktop` in the `~/.local/share/applications` directory.
2. Include the statements shown in the following example.

   ```
   [Desktop Entry]
   Encoding=UTF-8
   Name=Java(TM) Web Launcher
   Exec=jre-home/bin/javaws %U
   Terminal=false
   Type=Application
   MimeType=x-scheme-handler/jnlp;x-scheme-handler/jnlps
   ```
3. Save the file.
4. Run the following commands:

   ```
   xdg-mime default javaws.desktop x-scheme-handler/jnlp
   xdg-mime default javaws.desktop x-scheme-handler/jnlps
   ```

After installing the protocol handler, Java Web Start is used to launch applications when the URL contains the `jnlp` or `jnlps` protocol. For example, the following URL starts the Dynamic Tree sample from the Java Tutorials:

```
jnlps://docs.oracle.com/javase/tutorialJWS/samples/deployment/
dynamictree_webstartJWSProject/dynamictree_webstart.jnlp
```

Installing the Protocol Handler in Firefox

If you use the Firefox browser on Linux, manually install the protocol handler that enables you to start Java Web Start applications using the `jnlp` or `jnlps` protocol.

To install the protocol handler, add properties to the Firefox configuration:

1. In the address bar in Firefox, enter `about:config`.

   If you see a page with a warning about voiding your warranty, click **I'll be careful, I promise!** to continue to the configuration page.
2. Add a Boolean property.
   a. Right-click on the page and select **New**, and then select **Boolean**.
   b. In the New Boolean Value window, enter the name `network.protocol-handler.expose.jnlp` and click **OK**.
   c. Select **false** for the value and click **OK**.

   The property is added to the configuration.
3. Repeat the previous step to add the property `network.protocol-handler.expose.jnlps` with the value **false**.

When you enter a URL or click a link that uses the `jnlp` or `jnlps` protocol, you are prompted to choose the application to use to open the file. Select **Java Web Start Launcher** or browse to the `javaws.exe` file in `*jre-home*/bin`.

5
Application Development Considerations

This chapter includes the following topics:

Introduction to Web Deployment

Developing applications for deployment with Java Web Start is generally the same as developing standalone applications for the Java Platform, Standard Edition. For instance, the entry point for the application is the standard `public static void main(String[] argv)`.

However, to support web deployment—automatic downloading and launching of an application—and to ensure that an application can run in a secure sandbox, there are some additional considerations:

- An application must be delivered as a set of signed JAR files. All entries in each JAR file must be signed.
- All application resources, such as files and images, must be stored in JAR files. The resources must be referenced using the `getResource` mechanism in the Java Platform, Standard Edition.
- If an application is written to run in a secure sandbox, it must follow these restrictions:
  - No access to a local disk.
  - All JAR files must be downloaded from the same host.
  - Network connections without user prompts are enabled only for the host from which the JAR files are downloaded. Connections to other hosts require approval from the user.
  - No security manager can be installed.
  - No native libraries can be used.
  - Limited access to system properties. The application has read/write access to all secure system properties defined in the JNLP file, as well as read-only access to the same set of properties that an applet has access to.

  Some of these restrictions can be overcome by the use of the JNLP API to access the file system and other system resources.

- An application is allowed to use the `System.exit` call.

Retrieving Resources from JAR Files

Java Web Start only transfers JAR files from the Web server to the client machine. It determines where to store the JAR files on the local machine. Thus, an application cannot use disk-relative references to resources such as images and configuration files.

All application resources must be retrieved from the JAR files specified in the `resources` section of the JNLP file, or retrieved explicitly using an HTTP request to the web server. Storing resources in JAR files is recommended, because they will be cached on the local machine by Java Web Start.

The following code example shows how to retrieve images from a JAR file:

```
// Get current classloader
ClassLoader cl = this.getClass().getClassLoader();
// Create icons
Icon saveIcon  = new ImageIcon(cl.getResource("images/save.gif"));
Icon cutIcon   = new ImageIcon(cl.getResource("images/cut.gif"));
...
```

The example assumes that the following entries exist in one of the JAR files for the application:

```
images/save.gif
images/cut.gif
```

Accessing the Client Using the JNLP API

The JNLP API can be used to access the client's file system and other resources. See the following topics for more information about using the JNLPI API to access the client:

- JNLP API Examples
- Accessing the Client Using JNLP API topic in the Deployment Trail of the Java Tutorials. The information for the sample applet also applies to Java Web Start applications.

Security and Code Signing

Java Web Start addresses the following security issues:

- Protecting users against malicious code (intentional and unintentional) that may affect local files
- Protecting enterprises against code that may attempt to access or destroy data on networks

Applications launched with Java Web Start are, by default, run in a restricted environment where they have limited access to local computing resources, such as storage devices and the local network.

An additional security feature supported by Java Web Start is digital code signing. If an application being invoked is delivered in one or more signed JAR files, then Java Web

Start will verify that the contents of the JAR file have not been modified since they were signed. If verification of a digital signature fails, then Java Web Start will not run the application, because it might have been compromised by a third party.

The support for code signing is important for both users and for application service providers. This service makes it possible for users to verify that an application comes from a trusted source. A signed application that is trusted by the user can also request additional system privileges, such as access to a local disk.

Java Web Start presents a dialog box that displays the application's origin, based on the signer's certificate, before the application is launched. This enables the user to make an informed decision about whether or not to grant additional privileges to the downloaded code.

By including the following settings in the JNLP file, an application can request full access to a client system if all its JAR files are signed:

```
<security>
   <all-permissions/>
</security>
```

The implementation of code signing in Java Web Start is based on the security API in the core Java Platform, Standard Edition.

Developers sign code for use with Java Web Start in the same way as for Java applets—by using the standard `jarsigner` tool from the Java Platform, Standard Edition. The `jarsigner` tool documentation provides examples of how to sign code and create test certificates, and it discusses other issues related to signing.

Signing JAR Files with a Test Certificate

For testing purposes, a self-signed certificate can be used to sign a JAR file. For production, use a code signing certificate issued by a trusted certificate authority.

These are the steps to sign a JAR file with a self-signed test certificate:

1. Ensure that you have an SDK `keytool` and `jarsigner` in your path. These tools are located in the SDK bin directory.
2. Create a new key in a new `keystore` as follows:

   ```
   keytool -genkey -keystore myKeystore -alias myself
   ```

   You are prompted for information about the new key, such as password and name. This creates the `myKeystore` file on the disk.
3. Create a self-signed test certificate as follows:

   ```
   keytool -selfcert -alias myself -keystore myKeystore
   ```

   This prompts for the password. Generating the certificate might take a few minutes.
4. Check to ensure that everything is OK. To list the contents of the keystore, use this command:

   ```
   keytool -list -keystore myKeystore
   ```

   The output should look similar to:

```
Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry:

myself, Tue Jan 23 19:29:32 PST 2001, keyEntry,
Certificate fingerprint (MD5):
C2:E9:BF:F9:D3:DF:4C:8F:3C:5F:22:9E:AF:0B:42:9D
```

5. Sign the JAR file with the test certificate as follows:

   ```
   jarsigner -keystore myKeystore test.jar myself
   ```

   Repeat this step with all of your JAR files.

   Note that a self-signed test certificate should only be used for internal testing, because it does not guarantee the identity of the user and therefore cannot be trusted. A trustworthy certificate can be obtained from a certificate authority, such as VeriSign or Thawte, and should be used when the application is put into production.

To run an application that is signed with a self-signed test certificate, do one of the following on the computer where the application will run:

- Add the location of the application to the Exception Site List, which is managed from the Java Control Panel.
- Import the test certificate into the User Signer CA store using the Java Control Panel or into the System Signer CA store using the `keytool` utility.

How to Encode JNLP Files

Encode JNLP files in any character encoding supported by the Java Platform, Standard Edition. See Supported Encodings in the *Java Platform, Standard Edition Internationalization Guide* for a list.

To encode a JNLP file, specify an encoding in the XML prolog of that file. For example, the following line indicates that the JNLP file will be encoded in UTF-16.

```
<?xml version="1.0" encoding="utf-16"?>
```

The XML prolog itself must be UTF-8-encoded.

Dynamic Download of HTTPS Certificates

Java Web Start dynamically imports certificates in a similar way as browsers do. To make this work, Java Web Start sets its own HTTPS handler, using the `java.protocol.handler.pkgs` system properties, to initialize defaults for `SSLSocketFactory` and `HostnameVerifier`. It sets the defaults with `HttpsURLConnection.setDefaultSSLSocketFactory` and `HttpsURLConnection.setDefaultHostnameVerifier`.

If your application uses those two methods, ensure that they are called after the Java Web Start HTTPS handler is initialized, otherwise your custom handler will be replaced by the Java Web Start default handler. You can ensure that your own customized `SSLSocketFactory` and `HostnameVerifier` are used by doing either of the following:

1. Install your own HTTPS handler, which will completely replace the Java Web Start HTTPS handler.

2. Call `HttpsURLConnection.setDefaultSSLSocketFactory` or `HttpsURLConnection.setDefaultHostnameVerifier` only after the first `HttpsURLConnection` object is created, which executes the Java Web Start HTTPS handler initialization code first.

6

Migrating Java Applets to Java Web Start and JNLP

Because browsers are reducing or eliminating support for the Java Plug-in, consider migrating existing Java applets to Java Web Start, which uses the Java Network Launching Protocol (JNLP) to download and run an application locally.

Although available and supported in JDK 9, the Applet API and the Java Plug-in are marked as deprecated in preparation for removal in a future release. Alternatives for applets and embedded JavaFX applications include Java Web Start and self-contained applications.

Migrating to Java Web Start provides the ability to launch the application independent of a web browser. When your user is offline or unable to access the browser, a desktop shortcut can launch the application, providing your user with the same experience as that of a native application.

This chapter includes the following topics:

Migrating an Existing Java Applet

Java Web Start technology has built-in support for applets. Your applet can run with Java Web Start technology without any recompilation of the applet. Just convert your applet HTML tags to a Java Network Launching Protocol (JNLP) file using the JNLP `applet-desc` element, for example:

```
<resources>
  <jar href="SwingSet2.jar"/>
</resources>
<applet-desc main-class="SwingSet2Applet" name="SwingSet" width="625" height="595">
  <param name="param1" value="value1"/>
  <param name="param2" value="value2"/>
</applet-desc>
```

Note:

Although available and supported in JDK 9, the Applet API is marked as deprecated in preparation for removal in a future release. Instead of applets, consider alternatives such as Java Web Start or self-contained applications.

Rewriting a Java Applet as a Java Web Start Application

The best way to migrate your applet is to rewrite it as a standalone Java application, and then deploy it with Java Web Start technology. Rewriting your applet and testing the resulting application ensures that your converted applet works as expected, and your application can take advantage of the Java Web Start features.

The major work needed is to convert your `applet` class to the `main` class of the application. The applet `init` and `start` methods are no longer present, instead, initialize the application at the beginning of the application's `main` method.

To quickly begin the migration, add the `main` method to your original `applet` class, and then start calling your applet initialization code where it normally gets called from the applet's `init` and `start` methods. When there is a `main` method in the `applet` class, you can begin launching it by using the Java Web Start technology, and then slowly remove the dependency on the `Applet` class and convert it completely to your application's `main` class.

Special Considerations

The following items are things to consider when migrating:

- A Java Web Start application does not run within the web browser. If your applet has any dependency on the browser (for example, Java-to-JavaScript or JavaScript-to-Java communications using the browser), the communication code will no longer work. The APIs that are affected include:
  - `JSObject` API (`netscape.javascript.JSObject.*`) for Java-to-JavaScript communication does not work for Java Web Start applications.
  - Common Document Object Model (DOM) APIs (`com.sun.java.browser.dom.*` and `org.w3c.dom.*`) available for Java Plug-in applets are not available to Java Web Start applications.
- Similar to Java Plug-in technology, for faster start-up performance, Java Web Start technology caches your application JARs and resources downloaded by your application.
- Java Web Start technology provides permanent cookie support on Windows using the cookie store in Internet Explorer (IE), and the cookie-handling behavior is determined by the cookie control in IE. On Linux and Solaris, Java Web Start technology provides permanent cookie support using its own cookie store implementation. See Cookies in the Java Tutorials.
- If you deploy an applet with the JNLP `applet-desc` element, your applet is created using the `AppletContainer` provided by Java Web Start technology. When your applet calls `Applet.getAppletContext()`, it returns the `AppletContainerContext` provided by Java Web Start technology. The following list describes minor differences in implementation between the Java Plug-in `AppletContext` and the Java Web Start `AppletContext`:
  - The following Applet Persistence API methods are not implemented by Java Web Start technology:

    ```
    AppletContext.getStream(String key)
    AppletContext.getStreamKeys()
    AppletContext.setStream(String key, InputStream s)
    ```

    For Java Web Start applications, you can use the JNLP Persistence Service API for storing data locally on the client's system. See the `PersistenceService` interface.

    - For `AppletContext.showDocument(URL url, String target)`, the target argument is ignored by Java Web Start technology.
    - For `AppletContext.showStatus(String status)`, when launched with Java Web Start technology, this sets the `JLabel` text that is below the applet, hosted by the Java Web Start `AppletContainer`.

- Similar to `AppletContext.showDocument`, Java Web Start applications are capable of showing an HTML page using the system's default web browser by using the `BasicService.showDocument` API.

  For a Java Plug-in applet:

  ```
  AppletContext.showDocument(URL url)
  AppletContext.showDocument(URL url, String target)
  ```

  For a Java Web Start application:

  ```
  BasicService.showDocument(URL url)
  ```

  See the `BasicService` interface.

- In an applet, if you obtain a resource using the following calls:

  ```
  Applet.getImage(URL url)
  Applet.getImage(URL url, String name)
  Applet.getAudioClip(URL url)
  Applet.getAudioClip(URL url, String name)
  AppletContext.getAudioClip(URL url)
  AppletContext.getImage(URL url)
  ```

  Then in Java Web Start technology, the best practice is to include the resources in your application JAR files, and access the resources using the `JNLPClassLoader`:

  ```
  ClassLoader cl = this.getClass().getClassLoader();
  URL url = cl.getResource(url);
  Image im = Toolkit.getDefaultToolkit().createImage(url);
  ```

  See Retrieving Resources from JAR Files.

- The pack200 JAR packing tool is supported by both the Java Web Start and the Java Plug-in technologies. If you are already deploying your applet JARs with pack200, no change is needed when migrating to Java Web Start technology.
- By using the `OBJECT` tag in Java Plug-in technology, you can detect whether Java is available on the client's machine with the plug-in `CLSID` and then auto-download Java if necessary. The same support is available with Java Web Start technology by using the Java Web Start `CLSID`. See Ensuring the Presence of the JRE Software in the Java Tutorials.
- If you want to deploy extensions for your Java Web Start application, then use the JNLP extension protocol mechanism. See the JSR 56: Java Network Launching Protocol and API, section 3.8 "Extension Descriptor."

  One advantage of the JNLP extensions mechanism over Java Plug-in technology is that the installed extensions are available to all Java Web Start applications running on the system, no matter what version of the JRE the application is running. For Java Plug-in technology, only applets running in the same JRE version can use the installed extensions.

- For signed JAR files, similar to Java Plug-in technology, you can sign your application JAR files and request your application to be run with all permissions using the JNLP file. In Java Plug-in technology, your applet JARs can be signed by different certificates. In Java Web Start technology, the same certificate must be used to sign all JAR files (`jar` and `nativelib` elements) that are part of a single JNLP file. This simplifies user management because only one certificate must be presented to the user during a launch per JNLP file. If you must use JARs signed with different certificates, then you can put them in a component extension JNLP file, and reference the extension file from the main JNLP file. See the JSR 56: Java Network Launching Protocol and API, section 5.4 "Signed Applications" and section 4.7 "Extension Resources."

7

JNLP File Syntax

This chapter includes the following topics:

Introduction to JNLP File Syntax

The format used in this release is that specified in the Java Network Launching Protocol & API Specification (JSR-56) version 6.0. The Java Network Launch Protocol (JNLP) file for rich internet applications (RIAs) is an XML file. All elements and their attributes must be entered in lowercase. The following table describes the most commonly used elements of a JNLP file. For a complete description of the format, see the specification.

Element	Description	Attributes	Required

`jnlp`	This is the main XML element for a JNLP file. Everything is contained within the `jnlp` element.	**`spec`**
		Specifies the minimum version of the JNLP specification that this JNLP file can work with. Valid values are 1.0, 1.5, 6.0, 6.0.10, 6.0.18, 7.0, 8.20, 9 or a wildcard such as 1.0+.
		
		`codebase`
		Specifies the base location for all relative URLs specified in `href` attributes in the JNLP file.
		
		`href`
		Specifies the URL of the JNLP file itself.
		
		`version`
		Specifies the version of the application being launched and the version of the JNLP file itself.
			Yes

`information`	Contains other elements that describe the application and its source.	**`os`**
		Specifies the operating systems for which this information element should be considered. Added in 1.5.0.
		
		`arch`
		Specifies the architecture for which this information element should be considered. Added in 1.5.0.
		
		`platform`
		Specifies the platform for which this information element should be considered. Added in 1.5.0.
		
		`locale`
		Specifies the locale for which this information element should be considered. Added in 1.5.0.
			No

`title`	Specifies the title of the application.	NA	No
`vendor`	Specifies the provider of the application.	NA	No
`homepage`	Home page of the application.	**`href`** Specifies the URL where more information about this application can be found. Required.	No

Element	Description	Attributes	Required
`description`	Short statement describing the application.	**`kind`** Indicates the type of description this is. Valid values are `one-line`, `short`, and `tooltip`.	No

`icon`	Describes an icon that can be used to identify the application to the user.	**`href`**
		Specifies the URL for the icon file. Can be in one of the following formats: GIF, JPG, PNG, ICO. Required.
		
		`kind`
		Indicates the suggested use of the icon. Valid values are `default`, `splash`, and `shortcut`.
		
		`width`
		Can be used to indicate the resolution of the image.
		
		`height`
		Can be used to indicate the resolution of the image.
		
		`depth`
		Can be used to indicate the resolution of the image.
			No

`offline-allowed`	Indicates that this application can operate when the client system is disconnected from the network.	NA	No

`shortcut`	Can be used to indicate an application's preferences for desktop integration. Added in 1.5.0.	**`online`**
		Can be used to describe the application's preference for creating a shortcut to run online or offline.
		
		`install`
		Can be used in a shortcut element to describe the application's preference for being considered installed. If the value is `true`, then the application prefers to be considered installed. The default value of the install attribute is `false`. On Windows, this determines if the application appears in the Add and Remove Programs panel. Added in 7.0.
			No

Element	Description	Attributes	Required
`desktop`	Can be used to indicate an application's preference for putting a shortcut on the users desktop. Added in 1.5.0.	NA	No
`menu`	Can be used to indicate an application's preference for putting a menu item in the user's start menus. Added in 1.5.0.	**`sub-menu`** Can be used to indicate an application's preference for where to place the menu item.	No

`association`	Can be used to hint to the JNLP client that it wants to be registered with the operating system as the primary handler of certain extensions and a certain MIME type. If this element is included, either the `offline-allowed` element must also be included, or the `href` attribute must be set for the jnlp element. Added in 1.5.0.	**`extensions`**
		Contains a list of file extensions (separated by spaces) that the application requests it be registered to handle.
		
		`mime-type`
		Contains the MIME type that the application requests it be registered to handle.
			No

`related-content`	Describes an additional piece of related content that can be integrated with the application. Added in 1.5.0.	**`href`** Specifies the URL to the related content. Required.	No

`update`	Indicates the preferences for how application updates should be handled by the JNLP client. Added in 6.0.	**`check`**
		Indicates the preference for when the JNLP client should check for updates. Valid values are `always`, `timeout`, and `background`.
		
		`policy`
		Indicates the preference for how the JNLP client should handle an application update when it is known an update is available before the application is launched. Valid values are `always`, `prompt-update`, and `prompt-run`.
			No

`security`	Requests enhanced permissions.	NA	No
`all-permissions`	Requests that the application be run with all permissions.	NA	No
`j2ee-application-client-permissions`	Requests that the application be run with a permission set that meets the security specifications of the J2EE application client environment.	NA	No

Element	Description	Attributes	Required

`resources`	Describes all the resources that are needed for an application.	**`os`**
		Specifies the operating system for which the `resources` element should be considered.
		
		`arch`
		Specifies the architecture for which the `resources` element should be considered.
		
		`locale`
		Specifies that the locales for which the `resources` element should be considered.
			Yes

`java` (or `j2se`)	Specifies the versions of Java to use to run the application. The `java` element was added in 6.0.	**`version`**
		Describes an ordered list of version ranges to use. Required.
		
		`href`
		URL denoting the supplier of this version of Java, and where it can be downloaded from.
		
		`java-vm-args`
		Indicates an additional set of standard and non-standard virtual machine arguments that the application would prefer the JNLP client to use when launching Java.
		
		`initial-heap-size`
		Indicates the initial size of the Java heap.
		
		`max-heap-size`
		Indicates the maximum size of the Java heap.
			No

Element	Description	Attributes	Required

`jar`	Specifies a JAR file that is part of the application's class path.	**`href`**
		URL of the JAR file. Required.
		
		`version`
		Requested version of the JAR file. Requires using the version-based download protocol.
		
		`main`
		Indicates if this JAR contains the class containing the main method of the application.
		
		`download`
		Can be used to indicate that this JAR can be downloaded eagerly, lazily, or for progress indication.
		
		`size`
		Indicates the downloadable size of the JAR file in bytes.
		
		`part`
		Can be used to group resources so they are downloaded at the same time.
			No, however, if a main JNLP file does not specify a JAR file, then the JNLP file must contain a component extension whose resources section has at least one `jar` element.

`nativelib`	Specifies a JAR file that contains native libraries in its root directory.	**`href`**
		URL of the JAR file. Required.
		
		`version`
		The requested version of the JAR file. Requires using the version-based download protocol.
		
		`download`
		Can be used to indicate that this JAR file can be downloaded lazily.
		
		`size`
		Indicates the downloadable size of the JAR file in bytes.
		
		`part`
		Can be used to group resources together so they will be downloaded at the same time.
			No

Element	Description	Attributes	Required

`extension`	Contains a pointer to an additional `component-desc` or `installer-desc` to be used with this application.	**`href`**
		URL to the additional extension JNLP file. Required.
		
		`version`
		Version of the additional extension JNLP file.
		
		`name`
		Name of the additional extension JNLP file.
			No

`ext-download`	Can be used in an `extension` element to denote the parts contained in a component extension.	**`ext-part`**
		Describes the name of a part that is expected to be found in the extension. Required.
		
		`download`
		Can be used to indicate that this extension can be downloaded eagerly or lazily.
		
		`part`
		Denotes the name of a part in this JNLP file in which to include the extension.
			No

`package`	Can be used to indicate to the JNLP client which packages are implemented in which JAR files.	**`name`**
		Package name contained in the JAR files of the given part. Required.
		
		`part`
		Part name that contains the JAR files that include the given package name. Required.
		
		`recursive`
		Can be used to indicated that all package names beginning with the given name can be found in the given part.
			No

`property`	Defines a system property that will be available through the `System.getProperty` and `System.getProperties` methods.	**`name`**
		Name of the system property. Required.
		
		`value`
		Value for the property. Required.
			No

Element	Description	Attributes	Required

`application-desc`	Denotes this is the JNLP file for an application.	**`main-class`**
		Name of the class containing the `public static void main(String[])` method of the application. Required.
		
		`progress-class`
		Name of the class that contains the implementation of `DownloadServiceListener` that can be used to indicate download progress. Added in 6.0.18.
		
		`type`
		Type of application to be deployed. Valid values are `Java` and `JavaFX`. The default is `Java`. Use this attribute in place of the XML extensions that were previously used to define a JavaFX application.
			A JNLP file must contain one of the following elements:
			• `application-desc`
			• `applet-desc`
			• `component-desc`
			• `installer-desc`

`argument`	Each argument contains (in order) an additional argument to be passed to the application's `main` method.	NA	No

`applet-desc`	Denotes this is the JNLP file for an applet.
	Note: Although available and supported in JDK 9, the Applet API is marked as deprecated in preparation for removal in a future release. Instead of applets, consider alternatives such as Java Web Start or self-contained applications.
		`main-class`
		Name of the main `Applet` class. Required.
		
		`documentbase`
		Document base for the applet as a URL.
		
		`name`
		Name of the applet. Required.
		
		`width`
		Width of the applet in pixels. Required.
		
		`height`
		Height of the applet in pixels. Required.
		
		`progress-class`
		Name of the class that contains the implementation of `DownloadServiceListener` that can be used to indicate download progress. Added in 6.0.18.
			A JNLP file must contain one of the following elements:
			• `application-desc`
			• `applet-desc`
			• `component-desc`
			• `installer-desc`

Element	Description	Attributes	Required
`param`	Set of parameters that can be passed into the application. This element can be used as a sub element of the `applet-desc` or `application-desc` elements. If used in an `application-desc` element, then the values are used only if the `type` attribute is set to `JavaFX`. The values are ignored if the `type` attribute is set to `Java`.	**`name`** Name of the parameter. Required. **`value`** Value of the parameter. Required.	No
`component-desc`	Indicates this is the JNLP file for a component extension.	**`progress-class`** Name of the class that contains the implementation of `DownloadServiceListener` that can be used to indicate download progress. Added in 6.0.18.	A JNLP file must contain one of the following elements: • `application-desc` • `applet-desc` • `component-desc` • `installer-desc`
`installer-desc`	Indicates this is the JNLP file for an installed extension.	**`main-class`** Name of the class that contains the `public static void main(String[])` method of the installer. Required.	A JNLP file must contain one of the following elements: • `application-desc` • `applet-desc` • `component-desc` • `installer-desc`

The JNLP file is an XML document. See Examples of a JNLP File.

Examples of a JNLP File

For a basic application that does not require a lot of customization, the JNLP file can be simple. For more complex applications, additional elements can be added to the JNLP file.

The elements are described in JNLP Elements.

Basic JNLP File

This example shows a simple JNLP file. The root element is `jnlp`, which has two subelements: `resources` and `application-desc`.

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="9">
    <resources>
        <jar href="https://docs.oracle.com/javase/tutorialJWS/samples/deployment/
NotepadJWSProject/Notepad.jar" />
    </resources>
    <application-desc main-class="Notepad"/>
</jnlp>
```

JNLP File with Application Customizations

This example shows the basic outline of the JNLP file with settings for application attributes. The root element is `jnlp`, which has four subelements: `information`, `security`, `resources`, and `application-desc`. In addition, Java Web Start also supports launching applets by using the `applet-desc` element.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- JNLP File for SwingSet2 Demo Application -->
<jnlp
  spec="6.0+"
  codebase="http://my_company.com/jaws/apps"
  href="swingset2.jnlp">
  <information>
    <title>SwingSet2 Demo Application</title>
    <vendor>Sun Microsystems, Inc.</vendor>
    <homepage href="docs/help.html"/>
    <description>SwingSet2 Demo Application</description>
    <description kind="short">A demo of the capabilities
    of the Swing Graphical User Interface.</description>
    <icon href="images/swingset2.jpg"/>
    <icon kind="splash" href="images/splash.gif"/>
    <offline-allowed/>
    <association mime-type="application-x/swingset2-file"  extensions="swingset2"/>
    <shortcut online="false" install="false">
      <desktop/>
      <menu submenu="My Corporation Apps"/>
    </shortcut>
  </information>
  <information os="linux">
    <title> SwingSet2 Demo on Linux </title>
    <homepage href="docs/linuxhelp.html">
  </information>
  <security>
      <all-permissions/>
  </security>
  <resources>
    <j2se version="1.6+" java-vm-args="-esa -Xnoclassgc"/>
    <jar href="lib/SwingSet2.jar"/>
  </resources>
  <application-desc main-class="SwingSet2"/>
</jnlp>
```

JNLP Elements

Information about commonly used JNLP elements is provided in the following sections:

- jnlp Element
- information Element
- security Element
- update Element
- resources Element
- application-desc Element
- applet-desc Element

- component-desc Element
- installer-desc Element

jnlp Element

`spec` attribute: This attribute must be 1.0 or higher to work with this release. The default value is `1.0+`. Thus, it can typically be omitted. Note that the JDK 9 version of Java Web Start supports all versions of the spec through version 9. Previous versions of Java Web Start support only those versions of the specification (spec) that were available at the time of the JDK release. A JNLP file specifying `spec="9+"` will work with this version, but not previous versions of Java Web Start.

`codebase` attribute: All relative URLs specified in `href` attributes in the JNLP file are using this URL as a base.

`href` attribute: If present, this attribute points to the JNLP file that is used to download and run the application. If not present, then depending on the method used to start the application, Java Web Start might not know where the JNLP file came from, which can affect the security dialog boxes.

information Element

`os` attribute: This attribute contains a list of operating system names for this element. Read the discussion of the `resources` element later for a full discussion of the `os` attribute.

`title` element: The name of the application.

`vendor` element: The name of the vendor of the application.

`homepage` element: Contains a single attribute, `href`, which is a URL locating the home page for the application. It is used by the Java Application Cache Viewer to point the user to a web page where more information about the application can be found.

`description` element: A short statement about the application. Description elements are optional. The `kind` attribute defines how the description should be used. The following values are valid for `kind`:

- `one-line`: If a reference to the application is going to appear on one row in a list or a table, then this description is used.
- `short`: If a reference to the application is going to be displayed in a situation where there is room for a paragraph, then this description is used.
- `tooltip`: If a reference to the application is going to appear in a tooltip, then this description is used.

Only one `description` element of each kind can be specified. A `description` element without a `kind` attribute is used as a default value. Thus, if Java Web Start needs a description of kind `short`, and it is not specified in the JNLP file, then the text from the description without an attribute is used.

All descriptions contain plain text. No formatting, such as with HTML tags, is supported.

`icon` element: Contains an HTTP URL to an image file in either GIF, JPEG, ICO, or PNG format. The icons are used to represent the application:

- During the launch when Java Web Start presents the application to the user
- In the Java Application Cache Viewer
- In desktop shortcuts

A 64x64 icon is shown during the download. In the Java Application Cache Viewer and in desktop shortcuts, a 32x32 icon is used. Java Web Start automatically resizes an icon to the appropriate size.

Optional `width` and `height` attributes can be used to indicate the size of the images.

The optional `kind` attribute can have one of the following values:

- `default`: The specified image is used in the Java Application Cache Viewer. It is also used if an `icon` element with a particular `kind` attribute has not been specified. Not specifying the `kind` attribute is the same as specifying it with the `default` value.
- `splash`: The specified image is used as the splash image on the second and subsequent launches of the application. During the initial launch of the application, before resources are downloaded, the default splash image is used.
- `shortcut`: The specified image is used in desktop shortcuts.

A JNLP file can contain multiple `icon` elements that differ by their `kind` attribute. This enables you to specify different icon images for your application.

`offline-allowed` element: The optional `offline-allowed` element indicates if the application can be launched offline.

If `offline-allowed` is specified, then the application can be launched offline by the Java Application Cache Viewer, and shortcuts can be created that launch the application offline.

If an application is launched offline, then it does not check for updates, and the API call `BasicService.isOffline()` returns `true`.

The `offline-allowed` element also controls how Java Web Start checks for an update to an application. If the element is not specified—that is, the application is required to be online to run—Java Web Start checks for an updated version before launching the application. If an update is found, then the new application is downloaded and launched. Thus, it is guaranteed that the user always runs the latest version of the application. The application, however, must be run online.

If `offline-allowed` is specified, then Java Web Start also checks if an update is available. However, if the application is already downloaded, the check times out after a few seconds, in which case the cached application is launched instead. Given a reasonably fast server connection, the latest version of the application is usually run, but it is not guaranteed. The application, however, can be run offline.

`shortcut` element: The optional `shortcut` element can be used to indicate an application's preferences for desktop integration. The `shortcut` element and its subelements provide hints that the JNLP client may or may not use. The `shortcut` element can contain the optional `online` and `install` attributes, and the two optional subelements, `desktop` and `menu`.

`association` element: The optional `association` element is a hint to the JNLP client that it wants to be registered with the operating system as the primary handler of certain extensions and a certain MIME type. The `association` element must have the `extensions` and `mime-type` attributes. If the `association` element is included, then either the `offline-allowed` element must also be included, or the `href` attribute must be set for the `jnlp` element to ensure that the application can be located and run.

`related-content` element: The optional `related-content` element describes an additional piece of related content, such as a readme file, help pages, or links to registration pages, as a hint to a JNLP client. The application is asking that this content be included in its desktop integration. The `related-content` element has a mandatory `href` and `title` attribute. It can contain any of the following subelements:

- `description` element: A short description of the related content.
- `icon` element: The icon can be used by the JNLP client to identify the related content to the user.

security Element

Each sandbox application is run in a restricted execution environment, similar to the applet sandbox. The security element can be used to request unrestricted access.

If the `all-permissions` element is specified, then the application has full access to the client machine and local network. All JAR files must be signed. The user is prompted to accept the certificate and agree to run the application.

update Element

The `update` element indicates the preferences for how application updates are handled by Java Web Start.

The `update` element can contain the following two optional attributes:

`check` attribute: The `check` attribute indicates the preference for when the JNLP Client should check for updates. The following values are valid:

- `always`: Check for updates before launching the application.
- `timeout` (default): means to check for updates until the timeout before launching the application. If the update check is not completed before the timeout, then the application is launched, and the update check continues in the background.
- `background`: Launch the application while checking for updates in the background.

`policy` attribute: The `policy` attribute indicates the preference for how the JNLP client handles an application update when it is known an update is available before the application is launched. The following values are valid:

- `always` (default): Download updates without any prompt.
- `prompt-update`: Ask users if they want to download and run the updated version, or launch the cached version.
- `prompt-run`: Ask users if they want to download and run the updated version, or cancel and stop running the application.

For example:

```
<update check="always" policy="prompt-update">
```

resources Element

The `resources` element is used to specify all of the resources, such as Java class files, native libraries, and system properties, that are part of the application. To restrict a resource definition to a specific operating system, architecture, or locale, use the `os`, `arch`, and `locale` attributes.

The `os` attribute contains a list of operating system names for a resource. For example, you could use multiple `resources` definitions with different `os` attributes to supply a native library for multiple operating systems.

The `os` attribute contains a list of operating system names separated by spaces. At runtime, the `os` values are compared with the beginning of the `os.name` system property to find a match. For example, an `os` attribute value of "Windows" matches both "Windows 8" and "Windows 10" operating systems.

If you want to list an operating system whose name contains a space, then use a backslash to indicate that the space is part of the operating system name. The following example specifically matches "Windows 8" and "Windows 10":

```
<resources os="Windows\ 8 Windows\ 10">
  <jar href="hello.jar"/>
</resources>
```

The `resources` element has six different possible subelements: `jar`, `nativelib`, `j2se`, `property`, `package`, and `extension`. The `package` and `extension` elements are not discussed in this developer's guide.

A `jar` element specifies a JAR file that is part of the application's class path. For example:

```
  <jar href="myjar.jar"/>
```

The `jar` file is loaded into the JVM using a `ClassLoader` object. The `jar` file typically contains Java classes that contain the code for the particular application, but can also contain other resources, such as icons and configuration files, that are available through the `getResource` mechanism.

A `nativelib` element specifies a JAR file that contains native libraries, for example:

```
  <nativelib href="lib/windows/corelib.jar"/>
```

The JNLP client must ensure that each file entry in the root directory of the JAR file (i.e., `/`) can be loaded into the running process using the `System.loadLibrary` method. Each entry must contain a platform-dependent shared library with the correct naming convention, for example, `*.dll` on Windows or `lib*.so` on Solaris/Linux. The application is responsible for doing the actual call to `System.loadLibrary`.

Native libraries would typically be included in a `resources` element that is geared toward a particular operating system and architecture, for example:

```
<resources os="SunOS" arch="sparc">
  <nativelib href="lib/solaris/corelibs.jar"/>
</resources>
```

By default, `jar` and `nativelib` resources are downloaded eagerly, that is, they are downloaded and available locally to the JVM running the application before the application is launched. The `jar` and `nativelib` elements also allow a resource to be specified as `lazy`. This means the resource does not have to be downloaded onto the client system before the application is launched.

The `download` attribute is used to control whether a resource is downloaded eagerly, lazily, or contains a custom progress implementation, for example:

```
    <jarhref="sound.jar" download="lazy"/>
    <nativelibhref="native-sound.jar" download="eager"/>

    <jarhref="progress.jar" download="progress"/>
```

A JAR file denoted with `download="progress"` is downloaded eagerly, and can be used to indicate progress while downloading other resources.

The `j2se` element specifies what Java Platform, Standard Edition Runtime Environment (JRE) versions an application is supported on and standard parameters to the Java Virtual Machine. If several JREs are specified, then this indicates a prioritized list of the supported JREs, with the most preferred version first, for example:

```
       <j2se version="9" initial-heap-size="64m" max-heap-size="128m"/>
      <j2se version="1.8.0_101+" href="http://java.sun.com/products/autodl/j2se"
java-vm-args="-esa -Xnoclassgc"/>
```

The `version` attribute refers, by default, to a platform version (specification version) of the Java Platform Standard Edition. Currently defined platform versions are 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, and 9. (A platform version does not usually contain a micro version number; for example 1.4.2.)

Exact product versions (implementation versions) can also be specified by including the `href` attribute, such as 1.7.0_111, or 1.8.0_92, for example:

```
<j2se version="1.7.0_111" href="http://java.sun.com/products/autodl/j2se"/
```

or

```
<j2se version="1.8.0_92" href="http://java.sun.com/products/autodl/j2se"/>
```

If a platform version is specified (that is, no `href` attribute is provided), Java Web Start does not consider an installed non-General Availability (GA milestone) JRE as a match. For example, a request in the following the form does not consider an installed 9-ea JRE as a match for the request:

```
<j2se version="9+"/>
```

The `java-vm-args` attribute of the `j2se` element specifies a preferred set of virtual machine arguments to use when launching `java`.

```
<j2se version="9+" java-vm-args="-ea -Xincgc"/>
```

The following `java-vm-args` are supported by this version:

```
 -d32,                                                          /* use a 32-bit data
model if available (unix platforms only) */
 -client,                                                       /* to select the
client VM */
 -server,                                                       /* to select the
server VM */
 -verbose,                                                      /* enable verbose
output */
 -version,                                                      /* print product
version and exit */
```

```
 -showversion,                                                /* print product 
version and continue */
 -help,                                                       /* print this help 
message */
 -X,                                                          /* print help on non-
standard options */
 -ea,                                                         /* enable assertions 
*/
 -enableassertions,                                           /* enable assertions 
*/
 -da,                                                         /* disable 
assertions */
 -disableassertions,                                          /* disable 
assertions */
 -esa,                                                        /* enable system 
assertions */
 -enablesystemassertions,                                     /* enable system 
assertions */
 -dsa,                                                        /* disable system 
assertione */
 -disablesystemassertions,                                    /* disable system 
assertione */
 -Xmixed,                                                     /* mixed mode 
execution (default) */
 -Xint,                                                       /* interpreted mode 
execution only */
 -Xnoclassgc,                                                 /* disable class 
garbage collection */
 -Xincgc,                                                     /* enable 
incremental garbage collection */
 -Xbatch,                                                     /* disable 
background compilation */
 -Xprof,                                                      /* output cpu 
profiling data */
 -Xdebug,                                                     /* enable remote 
debugging */
 -Xfuture,                                                    /* enable strictest 
checks, anticipating future default */
 -Xrs,                                                        /* reduce use of OS 
signals by Java/VM (see documentation) */
 -XX:+ForceTimeHighResolution,                                /* use high 
resolution timer */
 -XX:-ForceTimeHighResolution,                                /* use low 
resolution (default) */
-XX:+PrintGCDetails,          /* Gives some details about the GCs */
-XX:+PrintGCTimeStamps,       /* Prints GCs times happen to the start of the 
application */
-XX:+PrintHeapAtGC,           /* Prints detailed GC info including heap occupancy */
-XX:+PrintTenuringDistribution,  /* Gives the aging distribution of the allocated 
objects */
-XX:+TraceClassUnloading,     /* Display classes as they are unloaded */
-XX:+CMSClassUnloadingEnabled,/* It needs to be combined with -XX:
+CMSPermGenSweepingEnabled */
-XX:+CMSIncrementalPacing,    /* Automatic adjustment of the incremental mode duty 
cycle */
-XX:+UseConcMarkSweepGC,      /* Turns on concurrent garbage collection */
-XX:-ParallelRefProcEnabled,
-XX:+DisableExplicitGC,       /* Disable calls to System.gc() */
-XX:+UseG1GC,
-XX:+HeapDumpOnOutOfMemoryError,
-XstartOnFirstThread,
-XX:+UseG1GC,
-XX:+UseStringDeduplication,
-XX:+PrintStringDeduplicationStatistics,
-XX:+UseParallelOldGC,
-XX:-UseParallelOldGC",
-XX:+UseParallelOldGCCompacting",
-XX:-UseParallelOldGCCompacting",
-XX:+UseParallelGC,
-XX:-UseParallelGC,
-XX:+UseGCTimeLimit,
-XX:-UseGCTimeLimit,
-XX:+UseGCOverheadLimit,
-XX:-UseGCOverheadLimit,
-XX:+ScavengeBeforeFullGC,
-XX:-ScavengeBeforeFullGC,
-XX:+UseParallelScavenge,
-XX:-UseParallelScavenge,
-XX:-TransmitErrorReport,
```

Also supported are any arguments that start with one of the following strings:

```
-ea,                         /* enable assertions for classes */
-enableassertions,           /* enable assertions for classes */
-da,                         /* disable assertions for classes */
-disableassertions,          /* disable assertions for classes */
-verbose,                    /* enable verbose output */
-Xms,                        /* set initial Java heap size */
-Xmx,                        /* set maximum Java heap size */
-Xss,                        /* set java thread stack size */
-XX:NewRatio,                /* set Ratio of new/old gen sizes */
-XX:NewSize,                 /* set initial size of new generation */
-XX:MaxNewSize,              /* set max size of new generation */
-XX:PermSize,                /* set initial size of permanent gen */
-XX:MaxPermSize,             /* set max size of permanent gen */
-XX:MaxHeapFreeRatio,        /* heap free percentage (default 70) */
-XX:MinHeapFreeRatio,        /* heap free percentage (default 40) */
-XX:UseSerialGC,             /* use serial garbage collection */
-XX:ThreadStackSize,         /* thread stack size (in KB) */
-XX:MaxInlineSize,           /* set max num of bytecodes to inline */
-XX:ReservedCodeCacheSize,   /* Reserved code cache size (bytes) */
-XX:MaxDirectMemorySize,
-XX:PrintCMSStatistics,              /* If > 0, Print statistics about the
concurrent collections */
-XX:SurvivorRatio,                   /* Sets the ratio of the survivor spaces */
-XX:MaxTenuringThreshold,            /* Determines how much the objects may age */
-XX:CMSMarkStackSize,
-XX:CMSMarkStackSizeMax,
-XX:CMSIncrementalDutyCycleMin,      /* The percentage which is the lower bound on
the duty cycle */
-XX:ParallelCMSThreads,
-XX:ParallelGCThreads,               /* Sets the number of parallel GC threads */
-XX:CMSInitiatingOccupancyFraction,  /* Sets the threshold percentage of the used
heap */
-XX:+UseCompressedOops,              /* Enables compressed references in 64-bit
JVMs */
-XX:GCPauseIntervalMillis,
-XX:MaxGCPauseMillis,                /* A hint to the virtual machine to pause
times */
-XX:+CMSIncrementalMode,             /* Enables the incremental mode */
-XX:StringDeduplicationAgeThreshold,
-XX:GCTimeLimit",
-XX:GCHeapFreeLimit",
-XX:MarkStackSize,
-XX:MarkStackSizeMax,
-XX:ConcGCThreads,
```

The `property` element defines a system property that is available through the `System.getProperty` and `System.setProperties` methods. It has two required attributes: `name` and `value`, for example:

```
<property name="key" value="overwritten"/>
```

Properties that are considered secure are set by Java Web Start after the VM is started but before the application is invoked. These properties are passed as `-Dkey=value` arguments to the `java` command when invoked. The following properties are considered secure:

- Properties set in a signed JNLP file
- Properties set in an unsigned JNLP file that are prefixed with one of the following strings: `jnlp.`, `javaws.`, or `javapi.`
- Predefined secure properties:

```
sun.java2d.noddraw,
javaws.cfg.jauthenticator,
swing.useSystemFontSettings,
swing.metalTheme,
http.agent,
http.keepAlive,
sun.awt.noerasebackground,
sun.java2d.opengl,
sun.java2d.d3d,
java.awt.syncLWRequests,
java.awt.Window.locationByPlatform,
sun.awt.erasebackgroundonresize,
sun.awt.keepWorkingSetOnMinimize,
swing.noxp,
swing.boldMetal,
awt.useSystemAAFontSettings,
sun.java2d.dpiaware,
sun.awt.disableMixing,
sun.lang.ClassLoader.allowArraySyntax,
java.awt.smartInvalidate"
apple.laf.useScreenMenuBar,
java.net.preferIPv4Stack,
java.util.Arrays.useLegacyMergeSort",
sun.locale.formatasdefault,
sun.awt.enableExtraMouseButtons,
com.sun.management.jmxremote.local.only,
sun.nio.ch.bugLevel,
sun.nio.ch.disableSystemWideOverlappingFileLockCheck,
jdk.map.althashing.threshold
```

application-desc Element

The `application-desc` element indicates that the JNLP file is launching an application (as opposed to an applet). The application element has an optional attribute, `main-class`, for specifying the name of the application's main class, which is the class where execution must begin:

- For Java applications, the main class is the class that contains the `public static void main(String argv[])` method.
- For JavaFX applications, the main class is the class that extends `javafx.application.Application`.

The `main-class` attribute can be omitted if the first JAR file specified in the JNLP file contains a manifest file that contains the `Main-Class` attribute.

The optional `type` attribute can be used to indicate the type of application, either Java, which is the default, or JavaFX. Java Web Start must know if it is a JavaFX application to know how to invoke it.

Arguments can be specified for the application by including one or more nested `argument` elements, for example:

```
<application-desc main-class="Main">
    <argument>arg1</argument>
    <argument>arg2</argument>
</application-desc>
```

Parameters can be added to applications with the `type` attribute set to `JavaFX` just as in applets by including one or more `param` elements, for example:

```
<application-desc type="JavaFX" main-class="fxApp">
    <param name="key1" value="value1"/>
    <param name="key2" value="value2"/>
</application-desc>
```

The (optional) `progress-class` attribute can be used to indicate that the class of this name implements the `javax.jnlp.DownloadServiceListener` interface. This class can be loaded first and used to indicate the progress of other resources being downloaded and verified.

applet-desc Element

Java Web Start has support for launching Java applets. This support provides easy migration of existing code to Java Web Start.

Note:

Although available and supported in JDK 9, the Applet API is marked as deprecated in preparation for removal in a future release.

An applet is launched using the `applet-desc` element instead of the `application-desc` element, for example:

```
<applet-desc
    documentBase="http://..."
    name="TimePilot"
    main-class="TimePilot.TimePilotApp"
    width="527"
    height="428">
  <param name="key1" value="value1"/>
  <param name="key2" value="value2"/>
</applet-desc>
```

The JAR files that make up the applet are described using the `resources` element, the same as for applications. The `documentBase` must be provided explicitly because a JNLP file is not embedded in an HTML page. The rest of the attributes correspond to the respective HTML applet tag elements.

The `main-class` attribute is used instead of the `code` attribute. The `main-class` attribute is assigned the name of the `Applet` class (without the `.class` extension). This attribute can be omitted if the `Applet` class can be found from the `Main-Class` manifest entry in the main JAR file.

The (optional) `progress-class` attribute can be used to indicate that the class of this name implements the `javax.jnlp.DownloadServiceListener` interface. This class can be loaded first and used to indicate the progress of other resources being downloaded and verified.

Note:

Applets must be packaged in JAR files to work with Java Web Start.

component-desc Element

The `component-desc` element denotes that this JNLP file is not an application or an applet but an extension that can be used as a resource in an application, applet, or another extension.

A component extension is typically used to factor out a set of resources that are shared among multiple applications or that have separate security needs.

The (optional) `progress-class` attribute can be used to indicate that the class of this name implements the `javax.jnlp.DownloadServiceListener` interface. This class can be loaded first and used to indicate the progress of other resources being downloaded and verified.

installer-desc Element

The `installer-desc` element denotes that this JNLP file is an installer extension that defines an application that will be run only once, the first time this extension JNLP file is used in an application, applet, or another extension.

An installer extension is typically used to install platform-specific native code that requires a more complicated setup than simply loading a native library into the VM.

8
JNLP API Examples

The JNLP API provides additional information to the application that would otherwise not be available using the standard Java Platform Standard Edition API. For untrusted applications, the JNLP API provides methods for operations such as reading and writing files or accessing the clipboard or printers, which would otherwise be prevented by the security manager.

The public classes and interfaces in the JNLP API are included in the `javaws.jar` file, which is in the `lib` directory. This JAR file must be included in the class path when compiling source files that use the JNLP API. For example on Windows:

```
javac -classpath .;javaws.jar *.java
```

The following code examples show how the JNLP services can be used:

- Using the BasicService Service
- Using the ClipboardService Service
- Using the DownloadService Service
- Using the DownloadService2 Service
- Implementing the DownloadServiceListener Service
- Using the FileOpenService Service
- Using the FileSaveService Service
- Using the IntegrationService Service
- Using the PrintService Service
- Using the PersistenceService Service
- Using FileContents
- Using a JNLPRandomAccessFile
- Using the SingleInstanceService Service
- Using an ExtendedService Service

Using the BasicService Service

The `javax.jnlp.BasicService` service provides a set of methods for querying and interacting with the environment similar to what the `AppletContext` provides for a Java applet.

The `showURL` method uses the JNLP API to direct the default browser on the platform to show the given URL. The method returns true if the request succeeds; otherwise, it returns false.

```
import javax.jnlp.*;
   ...
   // Method to show a URL
   boolean showURL(URL url) {
       try {
           // Lookup the javax.jnlp.BasicService object
           BasicService bs =
(BasicService)ServiceManager.lookup("javax.jnlp.BasicService");
           // Invoke the showDocument method
           return bs.showDocument(url);
       } catch(UnavailableServiceException ue) {
           // Service is not supported
           return false;
       }
    }
```

Using the ClipboardService Service

The `javax.jnlp.ClipboardService` service provides methods for accessing the shared system-wide clipboard, even for applications that are running in the restricted execution environment.

Java Web Start will warn the user of the potential security risk of letting an untrusted application access potentially confidential information stored in the clipboard, or overwriting contents stored in the clipboard.

```
import javax.jnlp;
    ...

    private ClipboardService cs;

    try {
        cs = (ClipboardService)ServiceManager.lookup
                 ("javax.jnlp.ClipboardService");
    } catch (UnavailableServiceException e) {
        cs = null;
    }

    if (cs != null) {
        // set the system clipboard contents to a string selection
        StringSelection ss = new StringSelection("Java Web Start!");
        cs.setContents(ss);
        // get the contents of the system clipboard and print them
        Transferable tr = cs.getContents();
        if (tr.isDataFlavorSupported(DataFlavor.stringFlavor)) {
           try {
                String s = (String)tr.getTransferData(DataFlavor.stringFlavor);
                System.out.println("Clipboard contents: " + s);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

Using the DownloadService Service

The `javax.jnlp.DownloadService` service allows an application to control how its resources are cached.

The service allows an application to determine which of its resources are cached, to force resources to be cached, and to remove resources from the cache.

```
import javax.jnlp.*;
    ...

    DownloadService ds;

    try {
        ds = (DownloadService)ServiceManager.lookup("javax.jnlp.DownloadService");
    } catch (UnavailableServiceException e) {
        ds = null;
    }

    if (ds != null) {

        try {
            // determine if a particular resource is cached
            URL url =
                    new URL("http://www.example.com/draw.jar");
            boolean cached = ds.isResourceCached(url, "1.0");
            // remove the resource from the cache
            if (cached) {
                ds.removeResource(url, "1.0");
            }
            // reload the resource into the cache
            DownloadServiceListener dsl = ds.getDefaultProgressWindow();
            ds.loadResource(url, "1.0", dsl);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Using the DownloadService2 Service

The `javax.jnlp.DownloadService2` service, introduced in the Java SE 6 update 18 release, provides the following methods:

- `getCachedResources` – Lists cached resources that match the given version, URL, and resource type.
- `getUpdateAvailableResources` – Checks and lists resources for which updates are available. If an application uses the version download protocol, then specify a version in the `DownloadService2.ResourceSpec`. If not, then specify a null value for the version.

An instance of the `DownloadService2.ResourceSpec` class specifies details about the resource to be checked.

```
import javax.jnlp.*;
...
DownloadService2 service = (DownloadService2)
                        ServiceManager.lookup("javax.jnlp.DownloadService2");

// create a new instance of ResourceSpec. In this example:
// - resource is downloaded from a directory on http://foo.bar.com:8080
// - version is 2. [0-9]+
// - resource type is JAR
ResourceSpec spec = new ResourceSpec("http://foo.bar.com:8080/.*", 2.*, service.JAR)

// returns all cached resources that match the given ResourceSpec
ResourceSpec results[] = service.getCachedResources(spec);

// returns all resources for which an update is available on the
// server http://foo.bar.com:8080.
results = service.getUpdateAvailableResources(spec);
```

Implementing the DownloadServiceListener Service

The `javax.jnlp.DownloadServiceListener` service provides methods to specify a customized loading progress indicator that indicates the progress of an application's download.

Using the FileOpenService Service

The `javax.jnlp.FileOpenService` service provides methods for importing files from the local disk, even for applications that are running in the restricted execution environment.

This interface is designed to provide the same type of disk access to potentially untrusted web-deployed applications that a web developer has when using HTML. HTML forms support the inclusion of files by displaying an Open dialog box.

```
import javax.jnlp.*;
    ...

    FileOpenService fos;

    try {
        fos = (FileOpenService)ServiceManager.lookup("javax.jnlp.FileOpenService");
    } catch (UnavailableServiceException e) {
        fos = null;
    }

    if (fos != null) {
        try {
            // ask user to select a file through this service
            FileContents fc = fos.openFileDialog(null, null);
            // ask user to select multiple files through this service
            FileContents[] fcs = fos.openMultiFileDialog(null, null);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Using the FileSaveService Service

The `javax.jnlp.FileSaveService` service provides methods for exporting files to the local disk, even for applications that are running in the restricted execution environment.

This interface is designed to provide the same level of disk access to potentially untrusted web-deployed applications that a web browser provides for contents that it is displaying. Most browsers provide a Save As dialog box as part of their user interface.

```
import javax.jnlp.*;
    ...

    FileSaveService fss;
    FileOpenService fos;

    try {
        fos = (FileOpenService)ServiceManager.lookup("javax.jnlp.FileOpenService");
        fss = (FileSaveService)ServiceManager.lookup
                                   ("javax.jnlp.FileSaveService");
    } catch (UnavailableServiceException e) {
        fss = null;
        fos = null;
    }

    if (fss != null && fos != null) {
        try {
            // get a file with FileOpenService
            FileContents fc = fos.openFileDialog(null, null);
            // one way to save a file
            FileContents newfc = fss.saveFileDialog(null, null,
            fc.getInputStream(), "newFileName.txt");
            // another way to save a file
            FileContents newfc2 = fss.saveAsFileDialog(null, null, fc);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Also see Using `FileContents`.

Using the IntegrationService Service

The `javax.jnlp.IntegrationService` service provides methods for programmatic management of shortcuts. By using this service, an application can perform the following operations:

- Create a desktop shortcut
- Create a menu shortcut
- Query and delete shortcuts
- Create, query, and delete associations of an application with a MIME type or file extensions.

```
import javax.jnlp.*;
...

IntegrationService is = null;
try {
    is = (IntegrationService) ServiceManager.lookup("javax.jnlp.IntegrationService");
} catch(UnavailableServiceException use){
    ...
}

// creates a desktop and system menu shortcut; returns true if the shortcuts
// were created successfully
boolean result = is.requestShortcut(true, true, null);

//removes all shortcuts for application
result = is.removeShortcuts();

// checks to see if there are shortcuts for the application
result = is.hasMenuShortcut() && is.hasDesktopShortcut());

// associates the application with the specified mime-type and file extensions
String mime = "x-application/aaa";
String [] exts = {"aaa", "abc"};
result = is.requestAssociation(mime, exts);

// checks if the application is associated with the specified mime-type and file
extensions
result = is.hasAssociation(mime, exts);

// removes association between the application and the specified mime-type and file
extensions
is.removeAssociation(mime, exts);
```

Using the PrintService Service

The `javax.jnlp.PrintService` service provides methods for access to printing, even for applications that are running in the restricted execution environment.

Using this service, an application can submit a print job. Java Web Start then shows this request to the user and, if accepted, queues the request to the printer.

Starting in Java Web Start 5.0, you can directly use the Java Printing APIs, and Java Web Start pops up a security dialog box that asks the user to grant `PrintPermission` if the application is running in a sandbox. There is no need to use the JNLP Printing APIs anymore. You can have full access to the Java Printing APIs in any JNLP application.

```
import javax.jnlp.*;
    ...

    PrintService ps;

    try {
        ps = (PrintService)ServiceManager.lookup("javax.jnlp.PrintService");
    } catch (UnavailableServiceException e) {
        ps = null;
    }

    if (ps != null) {
        try {

            // get the default PageFormat
            PageFormat pf = ps.getDefaultPage();

            // ask the user to customize the PageFormat
            PageFormat newPf = ps.showPageFormatDialog(pf);
            // print the document with the PageFormat above
            ps.print(new DocToPrint());

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

    // Code to construct the Printable Document
    class DocToPrint implements Printable {
        public int print(Graphics g, PageFormat pageformat, int PageIndex){
            // code to generate what you want to print
        }
    }
```

Using the PersistenceService Service

The `javax.jnlp.PersistenceService` service provides methods for storing data locally on the client system, even for applications that are running in the restricted execution environment.

The service is designed to be similar to that which the cookie mechanism provides to HTML-based applications. Cookies allow a small amount of data to be stored locally on the client system. That data can be securely managed by the browser and can only be retrieved by HTML pages that originate from the same URL as the page that stored the data.

```
import javax.jnlp.*;
    ...

    PersistenceService ps;
    BasicService bs;

    try {
        ps =
(PersistenceService)ServiceManager.lookup("javax.jnlp.PersistenceService");
        bs = (BasicService)ServiceManager.lookup("javax.jnlp.BasicService");
    } catch (UnavailableServiceException e) {
        ps = null;
        bs = null;
    }

    if (ps != null && bs != null) {

        try {
            // find all the muffins for our URL
            URL codebase = bs.getCodeBase();
            String [] muffins = ps.getNames(url);

            // get the attributes (tags) for each of these muffins.
            // update the server's copy of the data if any muffins
            // are dirty
            int [] tags = new int[muffins.length];
            URL [] muffinURLs = new URL[muffins.length];
            for (int i = 0; i < muffins.length; i++) {
                muffinURLs[i] = new URL(codebase.toString() + muffins[i]);
                tags[i] = ps.getTag(muffinURLs[i]);
```

```
                    // update the server if anything is tagged DIRTY
                    if (tags[i] == PersistenceService.DIRTY) {
                        doUpdateServer(muffinURLs[i]);
                    }
                }

                // read in the contents of a muffin and then delete it
                FileContents fc = ps.get(muffinURLs[0]);
                long maxsize = fc.getMaxLength();
                byte [] buf = new byte[fc.getLength()];
                InputStream is = fc.getInputStream();
                long pos = 0;
                while((pos = is.read(buf, pos, buf.length - pos)) > 0) {
                    // just loop
                }
                is.close();

                ps.delete(muffinURLs[0]);

                // re-create the muffin and repopulate its data
                ps.create(muffinURLs[0], maxsize);
                fc = ps.get(muffinURLs[0]);
                // don't append
                OutputStream os = fc.getOutputStream(false);
                os.write(buf);
                os.close();

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

        void doUpdateServer(URL url) {
            // update the server's copy of the persistent data
            // represented by the given URL
            ...
            ps.setTag(url, PersistenceService.CACHED);
        }
```

Using FileContents

javax.jnlp.FileContents objects encapsulate the name and contents of a file. An object of this class is used by the FileOpenService, FileSaveService, and PersistenceService. Here is an example of how an instance of FileContents can be used to read from and write to a file:

```
import javax.jnlp.*;
    ...

    FileOpenService fos;

    //Initialize fos (see Using the FileOpenService Service example)
    ...

    if (fos != null) {

        try {
                // get a FileContents object to work with from the
                // FileOpenService
                FileContents fc = fos.openFileDialog(null, null);

                // get the InputStream from the file and read a few bytes
                byte [] buf = new byte[fc.getLength()];
                InputStream is = fc.getInputStream();
                int pos = 0;
                while ((pos = is.read(buf, pos, buf.length - pos)) > 0) {
                    // just loop
                }
                is.close();

                // get the OutputStream and write the file back out
                if (fc.canWrite()) {
                   // don't append
                   OutputStream os = fc.getOutputStream(false);
                   os.write(buf);
                }

            } catch (Exception e) {
                e.printStackTrace();
            }
        }
```

Using a JNLPRandomAccessFile

Instances of `javax.jnlp.JNLPRandomAccessFile` support both reading and writing to a random access file. A random access file behaves like a large array of bytes stored in the file system. Here is an example of how an instance of a `JNLPRandomAccessFile` can be used to write to a random access file:

```
import javax.jnlp.*;
    ...

    FileOpenService fos;

    //Initialize fos (see Using the FileOpenService Service example)
    ...

    if (fos != null) {
        try {
           // ask the user to choose a file to open
           FileContents fc = fos.openFileDialog(null, null);

           // attempt to increase the maximum file length
           long grantedLength = fc.getLength();
           if (grantedLength + 1024 > fc.getMaxLength()) {
               // attempt to increase the maximum file size defined by
               // the client
               grantedLength = fc.setMaxLength(grantedLength + 1024);
           }

           // if we were able to increase the maximum allowable file size,
           // get a JNLPRandomAccessFile representation of the file, and
           // write to it
           if (fc.getMaxSize() > fc.getLength() && fc.canWrite()) {
               JNLPRandomAccessFile raf = fc.getRandomAccessFile("rw");
```

```
                raf.seek(raf.length() - 1);
                raf.writeUTF("Java Web Start!");
                raf.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Using the SingleInstanceService Service

The `javax.jnlp.SingleInstanceService` provides a set of methods for applications to register themselves as singletons, and to register listeners for handling arguments passed in from different instances of applications.

```
import javax.jnlp.*;
    ...

    SingleInstanceService sis;

    ...

    try {
        sis =
(SingleInstanceService)ServiceManager.lookup("javax.jnlp.SingleInstanceService");
    } catch (UnavailableServiceException e) { sis=null; }

    ...


    // Register the single instance listener at the start of your application

    SISListener sisL = new SISListener();
    sis.addSingleInstanceListener(sisL);

    ...


    // Remember to remove the listener before your application exits

    sis.removeSingleInstanceListener(sisL);
    System.exit(0);


    // Implement the SingleInstanceListener for your application

    class SISListener implements SingleInstanceListener {
        public void newActivation(String[] params) {

            // your code to handle the new arguments here

            ...
        }
    }
```

Using an ExtendedService Service

The `javax.jnlp.ExtendedService` provides additional support to the current JNLP API. It allows applications to open specific files in the client's file system.

```
import javax.jnlp.*;
    ...

    ExtendedService es;

    ...

    try {
        es =
(ExtendedService)ServiceManager.lookup("javax.jnlp.ExtendedService");
    } catch (UnavailableServiceException e) { es=null; }

    ...


    // Open a specific file in the local machine

    File a = new File("c:\somefile.txt");

    ...


    // Java Web Start will pop up a dialog asking the user to grant permission
    // to read/write the file c:\somefile.txt

    FileContents fc_a = es.openFile(a);


    // You can now use the FileContents object to read/write the file

    ...


    // Open a specific set of files in the local machine

    File[2] fArray = new File[2];

    fArray[0] = a;
    fArray[1] = new File("c:\anotherFile.txt");


    // Java Web Start will pop up a dialog asking the user to grant permission
    // to read/write files in fArray

    FileContents[] fc_Array = es.OpenFiles(fArray);


    // You can now read/write the set of files in fc_Array using the
    // FileContents objects

    }
```

For detailed information about using the `javaws` command, see `javaws` in the *Java Platform, Standard Edition Tools Reference*.

Part IV
Configuring and Monitoring Deployment

The topics in this part provide information about the Java Control Panel, which is used to configure deployment options, and deployment rule sets, which are used to manage the Java desktop environment.

9

Java Control Panel

The Java Control Panel is used to control how Java and JavaFX applications that are embedded in a browser or are launched from a browser run on your computer. The settings in the Java Control Panel are not used by standalone and self-contained applications.
The Java Control Panel includes the following tabs:

- General
- Update
- Desktop Settings
- Web Settings
- Security
- Advanced

Every tab contains a search field. Use this field to find settings related to the search term entered.

Overview of Java Control Panel

The Java Control Panel maintains settings that manage how Java and JavaFX applications embedded in or launched from a browser are run.

Note:

Although available and supported in JDK 9, the Applet API and the Java Plug-in are marked as deprecated in preparation for removal in a future release. Alternatives for applets and embedded JavaFX applications include Java Web Start and self-contained applications.

In JDK 9, the Java Control Panel was rewritten as a JavaFX application and the location of some functions has changed.

To start the Java Control Panel from the command line, enter <*JRE installation home*>`\bin\javacpl.exe` on Windows, or <*JRE installation home*>`/bin/jcontrol` on macOS or Linux. The Java Control Panel provides the following capabilities:

- View and delete temporary files used by the Java Plug-in, which runs applets and JavaFX applications that are embedded in a browser, and by Java Web Start, which enables you to run Java and JavaFX applications over the network.
- Update your version of the Java platform so that you always have the latest Java Runtime Environment (JRE).
- Manage the JREs on your system and set runtime parameters for them.
- Manage certificates.

- View the active deployment rule set on your system, if any.
- Manage the exception site list for your system.
- Configure proxy settings.
- Enable enhanced security restrictions for Java and JavaFX applications embedded in or launched from a browser.
- Configure settings for debugging, applet handling, and other functions.
- Search the Java Control Panel for settings to configure.

General Tab in the Java Control Panel

The General tab shows the version of the Java runtime (JRE) that you are running and the security status of the JRE.

The JRE that is running is identified by the Java version number and the build number. Security status is determined by the following attributes:

- Security Baseline - Minimum recommended update for Java.
- Expiration Date - Date related to the scheduled release of the next Critical Patch Update. After the expiration date, additional security fixes might be available.

If your JRE is below the security baseline or past the expiration date, you are encouraged to upgrade to the latest version.

Update Tab in the Java Control Panel

The Update tab shows when the check for updates is done and enables you to change the settings for the update process.

Automatic updates are supported only on Microsoft Windows and macOS. The update feature works with the Java Update Scheduler (`jusched.exe`) to provide you with the latest Java updates. You must have Administrative privileges to update the JRE.

From this tab, you can automatically or manually update the system JRE that is installed. If you have more than one JRE installed, the Desktop Settings tab shows you which JRE is considered the system JRE. See Desktop Settings Tab in the Java Control Panel.

The Update tab provides the options shown in the following table, not all options are available on both platforms:

Option	Description
Notify me before an update is	Indicates when you want to be notified that an update is available. The options are: • Downloaded - Notifies you before the update is downloaded. • Installed - Notifies you after the update is downloaded, but before the update is installed.
Automatically check for updates (Recommended)	Indicates if the check for updates is done automatically. This option is enabled by default. The time when the check is scheduled is shown. See Scheduling the Check for Updates to set the schedule.
Check Now	Checks for updates when clicked. The time of the last check is shown above the button.

Option	Description
Download the latest version of Java from `java.com`	Provides a link to where you can download the latest JRE.

Scheduling the Check for Updates

Set the time and frequency for automatic updates of your JRE from the Update tab of the Java Control Panel. A manual check can be done at any time from the same tab.

You must have Administrative privileges to update the JRE. The following instructions are for Microsoft Windows. Not all options are available for macOS. To check for an update:

1. To immediately check for an update, click **Check Now**. The time of the last check is shown above the button.
2. To schedule an automatic check for updates:
   a. Select **Automatically Check for Updates**. Your JREs are updated automatically on a schedule that you set.
   b. From the **Notify Me Before an Update is** drop-down list, choose to be notified either before the update is downloaded, or after the update is downloaded but before the update is installed.
   c. Click the date and time shown for **Check for Updates** to set up the schedule for updates.

      The Automatic Update Advanced Settings window is shown.
   d. Select how often you want the check to run and the day and time to run it.

      Choose **Daily**, **Weekly**, or **Monthly**. For daily updates, select the time of the day for the update. For weekly updates, select the day of the week and the time of the day. For monthly updates, select the day of the week and the time of the day. Monthly updates check weekly and notify you within 30 days that an update is available. However, if an update is considered critical, you are notified within a week of its release.
   e. Close the Automatic Update Advanced Settings window to see your selection in the Update tab.
   f. Click **Apply** to save your changes, or **OK** to save your changes and close the Java Control Panel.

Java Update Scheduler

On Microsoft Windows platforms, the Java Update Scheduler, `jusched.exe`, is used to launch automatic updates when the option to update automatically is selected in the Update tab. `jusched.exe` runs as a background process that launches the Update Manager at predefined intervals set by the in the Update tab of the Java Control Panel. The Update Manager coordinates the update process.

`jusched.exe` is launched when the user reboots the computer after installing the JDK or JRE. It is normally transparent to the user, but can be viewed in the Processes tab of the Windows Task Manager. If you do not want the scheduler to run, use the **End Process** button of the Processes tab to kill the process.

Desktop Settings Tab in the Java Control Panel

The Desktop Settings tab shows information about the JREs that are installed on your system and enables you to choose the JREs that you want to use to run applications that are embedded in a web page or launched from a browser.

The following table describes the information that is shown for each JRE found on your computer:

Setting	Description
Web Enabled	Flag that indicates which of the JRE versions are considered when running an application using Java Plug-in or Java Web Start. Settings in the Java Control Panel do not apply to standalone or self-contained applications. If the check box for a JRE is not selected, then Java Plug-in and Java Web Start will not use the JRE to launch Java applications. However, the current JRE might be used even if it is not marked as enabled. **Note:** If Java content in the browser is disabled in the Security tab of the Java Control Panel, enabling the JRE in the Desktop Settings tab has no effect.
Platform	Java platform number for the JRE, for example, `1.8` or `9`
Product	Full version number of the JRE, including the update number, for example, `1.8.0_101`
Architecture	Architecture of the JRE
Type	Type of JRE found, which is one of the following values: • System - JRE that was used to start the Java Control Panel • User - All of the registered JREs and the JREs that the user added
Path	Full path to the JRE
Runtime Parameters	Optional custom options used to override the Java Plug-in default startup parameters, see Java Runtime Parameters

The table always has at least one entry, which is the most recently installed JRE. This is the JRE that is associated with the Java Control Panel.

On Microsoft Windows all of the JREs that are installed on a computer are shown. The Java Control Panel finds the JREs by looking in the registry. On Solaris, Linux, and macOS, the JRE that Java Web Start or Java Plug-in is using to deploy applications is the JRE that is considered registered. Use the **Add** and **Remove** buttons to change which JREs are listed in the table, see Editing Desktop Settings. On macOS, only the currently installed JRE is displayed, JDKs are not included.

Editing Desktop Settings

JREs can be added and removed from the table in the Desktop Settings tab and runtime parameters can be set for each JRE.

The following functions are available for managing JREs on a computer:

- To change the runtime parameters for a user JRE, select the JRE, click the cell in the **Runtime Parameters** column, and edit the value.

- To add a JRE to the table, click **Add**. Browse to the location of the JRE and select the home folder.
- To remove a JRE from the table, select the JRE and click **Remove**.

  The System JRE cannot be removed.

Java Runtime Parameters

To override Java Plug-in default startup parameters, specify custom options in the **Runtime Parameters** column for a JRE shown in the Desktop Settings tab of the Java Control panel.

> **Note:**
>
> Although available and supported in JDK 9, the Java Plug-in has been marked as deprecated in preparation for removal in a future release. Alternatives for applets and embedded JavaFX applications, which require the plug-in, include Java Web Start and self-contained applications.

With the exception of setting `classpath` and `cp`, the syntax is the same as that used with parameters for the `java` command line invocation.

The following sections provide examples of Java runtime parameters:

- Setting classpath or cp
- Enabling and Disabling Assertion Support
- Tracing and Logging Support
- Debugging Applets in Java Plug-in
- Default Connection Timeout

See the `java` command in *Java Platform, Standard Edition Tools Reference* for a full list of command line options.

Setting classpath or cp

The following format should be used for setting `classpath` or `cp` in Java Plug-in. It differs slightly from the `java` command line format, which uses a space instead of the equal (=) sign.

```
-classpath=path
-cp=path
```

The following example shows a class path for Windows:

```
-cp=C:\apps\java\MyClasses;C:\java\OtherClasses
```

The following example shows a class path for Linux:

```
-cp=apps/java/MyClasses:/java/OtherClasses
```

Enabling and Disabling Assertion Support

System properties are used to enable and disable assertion support.

The following system property is used to enable assertion support:

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

The following system property is used to disable assertion in the Java Plug-in:

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

Assertion is disabled in Java Plug-in code by default. The effect of assertion is determined during Java Plug-in startup. If you change the assertion settings in the Java Plug-in Control Panel, you must restart the browser 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 using the following parameter:

```
-[ 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<random-number>.trace` or `javaws<random-number>.trace`). Use the following parameters to turn on tracing:

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

If you do not want to use the default trace file name, use the following parameter to specify a different name:

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

Similar to tracing, logging is a facility to redirect any output in the Java Console to a log file (`plugin<random-number>.log` or `javaws<random-number>.log`) using the Java Logging API. Use the following parameter to turn on logging:

```
-Ddeployment.logging=true
```

If you do not want to use the default log file name, use the following parameter to specify a different name:

```
-Ddeployment.log.filename=<logfilename>
```

Furthermore, if you do not want to overwrite the trace and log files each session, you can use the following parameter:

```
-Ddeployment.outputfiles.overwrite=false
```

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

Debugging Applets in Java Plug-in

The following parameters are used when debugging applets in the Java Plug-in:

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

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

Note:

Although available and supported in JDK 9, the Applet API and the Java Plug-in are marked as deprecated in preparation for removal in a future release. Alternatives for applets and embedded JavaFX applications include Java Web Start and self-contained applications.

Default Connection Timeout

The default network timeout value for all HTTP connections is two minutes. You can override this setting by using the following parameter:

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

Another networking property that you can set is `sun.net.client.defaultReadTimeout`, as shown in the following example:

```
-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.

The following networking parameters can also be used to set the connect and read timeout values for the protocol handlers used by `java.net.URLConnection`. The default value 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.

Web Settings Tab in the Java Control Panel

The Web Settings tab shows information about permissions for Java applications and how the applications connect to the network. The tab also enables you to manage temporary files and the Java cache.
The Web Settings tab contains the following tabs:

- Exception Site List Tab
- Deployment Rule Set Tab
- Temporary Files Settings Tab

- Network Settings Tab
- Java Cache Viewer Tab

Exception Site List Tab

The Exception Site List tab in the Web Settings tab enables you to manage Rich Internet Applications (RIAs) that users want to run even if the RIAs are normally blocked by security checks.

RIAs from the locations listed are allowed to run with applicable security prompts. Use the following controls to manage the list:

- Click **Add** to add a location.
- Select an entry and click **Remove** to remove a location.
- Double-click an entry to edit it.
- Use the **Filter** field to search the list for sites that contain the search term.

The following rules apply to the format of the location URL:

- A protocol is required.

  Supported protocols are HTTPS (`https://`), HTTP (`http://`), and FILE (`file:///`). HTTPS is recommended. FILE and HTTP protocols are considered a security risk.
- A domain is required.

  Wildcards are not supported. If only a domain is provided, any RIA from that domain is allowed to run. A domain can have multiple entries, for example, `https://www.example.com` and `http://www.example.com`.
- A port number is required only if the default port is not used.
- A path is optional.

  Wildcards are not supported. If the path ends with a slash, for example, `file:///C:\local\apps\`, RIAs in that directory and any subdirectory are allowed to run. If the path does not end with a slash, for example, `file:///C:\local\apps\applet.html`, only that specific RIA is allowed to run.
- The format must be the same as the format used for the RIA URL or `href` attribute.

  For example, `https://www.example.com/sample/app/sample1/../sample2` and `https://www.example.com/sample//app/sample2` are not considered matches to `https://www.example.com/sample/app/sample2`.

Deployment Rule Set Tab

The Deployment Rule Set tab in the Web Settings tab shows the active deployment rule set, which manages the running and blocking of Rich Internet Applications (RIAs).

If an active deployment rule set is installed on the system, the following information is shown:

- Notice that the rule set is valid or a warning that it is not valid
- Text box that shows the rules in the **Rules** tab and information about the certificate used to sign the rule set in the **Certificate Details** tab
- Timestamp of the rule set signature

- Location of the rule set
- Expiration date of the rule set signature

When a rule set is available, the rules determine if a RIA is run without security prompts, run with security prompts, or blocked. Deployment rules and rule sets are described in Deployment Rule Set.

Temporary Files Settings Tab

The Temporary Files Settings tab in the Web Settings tab enables you to manage files that are cached for applications that are embedded in a web page or launched from a web page.

From this tab, you can perform the following actions:

- Select if you want to keep temporary files on your computer.
- Set the location where temporary files are kept.
- Set the compression level for JAR files that are cached. The higher the compression level, the more compressed the file.
- Set the amount of disk space for storing temporary files.
- Delete temporary files by clicking **Delete Files** , which shows the Delete Files and Applications dialog. From this dialog, you can select the types of files that you want to delete:
  - Trace and Log Files
  - Cached Applications and Applets
  - Installed Applications and Applets
- Restore default settings for the Temporary Files Settings dialog by clicking **Restore Defaults** .

Network Settings Tab

The Network Settings tab in the Web Settings tab enables you to configure your connection to the network.

The available options are shown in the following table:

Setting	Description
Use browser settings	Select this option to use the browser default proxy settings. This is the default setting.
Use proxy server	Select this option to provide the address and port number of the proxy server that you want to use. The option to bypass the proxy server for local addresses is available. To provide separate addresses for different protocols, click Advanced. You can also specify address that bypass the proxy server.

Setting	Description
Use automatic proxy configuration script	Select this option to specify 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.
Direct Connection	Select this option if you do not want to use a proxy.

Java Cache Viewer Tab

The Java Cache Viewer tab in the Web Settings tab shows the applications, resources, and deleted applications stored in the Java cache.

From this tab, you can perform the following actions for users or for the system by using the icons or by right-clicking an application:

- For applications:
  - Run applications.
  - Visit the Web page of applications.
  - View the JNLP file of applications.
  - Install shortcuts to applications.
  - Remove applications from the list. Applications are moved to the list of deleted applications.
- For resources:
  - View JNLP file resources.
  - Remove resources.
- For deleted applications:
  - Install deleted applications.
  - Remove applications from the cache.
- View JNLP file resources.
- Install deleted applications.

Security Tab in the Java Control Panel

The Security tab shows general security settings and information about certificates used to sign RIAs.
The Security tab contains the following tabs:

- General Security Settings Tab
- Manage Certificates Tab

General Security Settings Tab

The General tab of the Security tab shows the security settings that are in place. This tab also enables you to restore security prompts.

The following table shows the options that are available.

Option	Description
Enable Java Content in the Browser	Enables Java applications to be run in a browser or launched from a browser. To prevent these types of applications from running, do not select this option. This option is selected by default.
Enable enhanced security restrictions	Adds the additional restriction of requiring that the system must be able to check the revocation status of the certificate used to sign the application, or the application is blocked. If not selected, applications that are signed with a valid certificate that is located in the Signer CA keystore, and include the Permissions attribute in the manifest for the main JAR file are allowed to run with security prompts. This option is not selected by default.
Restore Security Prompts	Restores the security prompts that were previously hidden. When asked to confirm the selection, click **Restore All**. The next time an application is started, the security prompt is shown. To insure the continued security of your system, it is recommended that you periodically restore the prompts that were hidden. Seeing the prompts again provides an opportunity to review the applications and ensure that you still want them to run.

Manage Certificates Tab

User-level and system-level certificates used to verify RIAs that you run can be managed from the Manage Certificates tab of the Security tab.

From this tab, you can import, export, remove, and view the details for certificates. Information is provided for the following types of certificates:

- Trusted Certificates - Certificates for signed RIAs that are trusted.
- Secure Site - Certificates for secure sites.
- Signer CA - Certificates of Certificate Authorities (CAs) who issue the certificates to the signers of trusted certificates.
- Secure Site CA - Certificates of CAs who issue the certificates for secure sites.
- Client Authentication - Certificates used by a client to authenticate itself to a server.

User-Level Certificates

You can export, import, remove, and view the details of user-level certificates using the buttons provided in the Certificates dialog. To export, remove, or view the details, first select a certificate from the list.

The following table shows the default location of the `keystore` files.

Table 9-1   Default Keystore Location

Operating System	Location
Solaris, Linux, macOS	`${user.home}/.java/deployment/security`
Microsoft Windows	`${deployment.user.home}\security`

For example, the default location on Microsoft Windows 7 for user `jsmith` is

```
C:\Users\jsmith\AppData\LocalLow\Sun\Java\Deployment\security
```

To specify a user-level keystore in a location other than the default location, set properties in the user-level `deployment.properties` file. The following table describes the property to set for each type of certificate.

Table 9-2   Properties for User-Level Keystore Locations

Certificate Type	Property Name
Trusted Certificates	`deployment.user.security.trusted.certs`
Secure site	`deployment.user.security.trusted.jssecerts`
Signer CA	`deployment.user.security.trusted.cacerts`
Secure site CA	`deployment.user.security.trusted.jssecacerts`
Client Authentication	`deployment.user.security.trusted.clientcerts`

System-Level Certificates

You can export and view the details of system-level certificates using the buttons provided in the Certificates dialog. System-level certificates cannot be imported or removed by an end user.

Trusted, Secure Site, and Client Authentication certificate `keystore` files do not exist by default. The following table shows the default location for the Signer CA keystore file.

Table 9-3   Default Location for the Signer CA Keystore

Operating System	Location
Linux, or macOS	`$JAVA_HOME/lib/security/cacerts`
Microsoft Windows	`$JAVA_HOME\lib\security\cacerts`

The following table shows the default location for the Secure Site CA keystore.

Table 9-4   Default Location for the Secure Site CA Keystore

Operating System	Location
Solaris, Linux, or macOS	`$JAVA_HOME/lib/security/jssecacerts`
Microsoft Windows	`$JAVA_HOME\lib\security\jssecacerts`

To specify a system-level keystore in a location other than the default location, set properties in the system-level `deployment.properties` file. The System-Level `deployment.properties` file does not exist by default. The following table describes the property to set for each type of certificate.

Table 9-5 Properties for System-Level Keystore Locations

Certificate Type	Property Name
Trusted Certificates	`deployment.system.security.trusted.certs`
Secure site	`deployment.system.security.trusted.jssecerts`
Signer CA	`deployment.system.security.trusted.cacerts`
Secure site CA	`deployment.system.security.trusted.jssecacerts`
Client Authentication	`deployment.system.security.trusted.clientcerts`

Advanced Tab in the Java Control Panel

The Advanced tab enables you to set the options that are available for the JRE. The following options are available:

- Debugging
- Java Console
- Shortcut Creation
- JNLP File/MIME Association
- Application Installation
- Execution Environment Security Settings
- Mixed code (sandboxed vs. trusted) security verification
- Perform signed code certificate revocation checks on
- Check for signed code certificate revocation using
- Advanced Security Settings
- Miscellaneous

Debugging

Enable tracing, logging, and the showing of applet lifecycle exceptions by selecting the appropriate check boxes. If the boxes are not checked, the options are disabled.

Java Console

The Java Console is a debugging aid for Java applets and Java Web Start applications. System.out and System.err messages and tracing and logging output are shown in the console.

The following choices for viewing the console are available:

- Show the console
- Hide the console (default)

- Do not start the console

Shortcut Creation

This option provides the following choices for Java Web Start for creating shortcuts on the desktop, select only one:

- Always allow
- Ask user if untrusted (default)
- Always ask user
- Never allow

JNLP File/MIME Association

This option enables you to associate files with the JNLP MIME type. The following choices are available, select only one:

- Always allow
- Prompt user (default)
- Never allow

Application Installation

The following choices are available, select only one:

- Install if hinted (default)
- Install if shortcut created
- Install if hinted and shortcut
- Never install

A Java application or applet that is launched with Java Web Start can either be installed or cached on the client computer. If the Java application is cached, then Java Web Start stores the entire application in its cache; the application is removed from the client computer when Java Web Start empties its cache. If the Java application is installed, then the application has an entry in the Add or Remove Programs applet in Windows Control Panel.

A Java application or applet can specify if it prefers to be cached or installed; if the Java application specifies that it prefers to be installed, then it is *hinted*. By default, Java applications that are hinted are installed on the client computer. You can also specify that a Java application is installed if it creates a shortcut on the client computer's desktop.

Execution Environment Security Settings

The following choices are available, more than one can be selected:

- Allow user to grant permissions to signed content
- Show sandbox warning banner
- Allow user to accept JNLP security requests

- Don't prompt for client certificate selection when no certificates or only one exists
- Warn if site certificate does not match hostname
- Show site certificate from server even if it is valid (not checked by default)

Mixed code (sandboxed vs. trusted) security verification

The following choices are available, select only one:

- Enable - show warning if needed (selected by default)
- Enable - hide warning and run with protections
- Enable - hide warning and don't run untrusted code
- Disable verification (not recommended)

Perform signed code certificate revocation checks on

Before a signed applet or Java Web Start application is run, the certificates used to sign the JAR file can be checked to ensure that none have been revoked. You can have all certificates checked, or only the certificate from the publisher of the app. If a certificate has been revoked, any RIA that is signed with the certificate is not allowed to run. This check can be disabled, but that is not recommended. The following choices are available, select only one:

- Publisher's certificate only
- All certificates in the chain of trust (selected by default)
- Do not check (not recommended)

Check for signed code certificate revocation using

The following options indicate what to use to determine if a certificate has been revoked, select only one:

- Certificate Revocations Lists (CRLs)
- Online Certificate Status Protocol (OCSP)
- Both CRLs and OCSP (selected by default)

If Do Not Check is selected for Perform signed code certificate revocation checks on, this setting is ignored.

Perform TLS certificate revocation checks on

Before a signed applet or Java Web Start application is run from a secure server, the certificates used to authenticate the secure server can be checked to ensure that none have been revoked. You can have all certificates checked, or only the certificate from the server. If a certificate has been revoked, any RIA that is signed with the certificate is not allowed to run. This check can be disabled, but that is not recommended. The following choices are available, select only one:

- Server certificate only
- All certificates in the chain of trust (selected by default)
- Do not check (not recommended)

Check for signed code certificate revocation using

The following options indicate what to use to determine if the certificate for a secure server has been revoked, select only one:

- Certificate Revocations Lists (CRLs)
- Online Certificate Status Protocol (OCSP)
- Both CRLs and OCSP (selected by default)

If Do Not Check is selected for Perform TLS certificate revocation checks on, this setting is ignored.

Advanced Security Settings

The following choices are available, more than one can be selected:

- Enable the operating system's restricted environment (native sandbox) (Windows only, not checked by default)
- Use certificates and keys in browser keystore
- Enable blacklist revocation check
- Enable caching password for authentication
- Use SSL 2.0 compatible ClientHello format (not checked by default)
- Use TLS 1.0
- Use TLS 1.1
- Use TLS 1.2

Native Sandbox

The native sandbox option is available only on Windows. When the native sandbox is enabled, sandbox applets and Java Web Start applications run in a restricted environment that is provided by the operating system. All-permission applications are not affected and continue to run as before.

The following conditions apply:

- The native sandbox is disabled for applications included in the Exception Site List or when a Deployment Rule Set is used.
- Sandbox applets deployed with the HTML applet tag, which includes all-permissions JAR files from the Class-Path manifest attribute, run in the native sandbox. In this case, a special warning dialog is shown to inform the user that the applet might not work properly when it tries to access the all-permission JAR files.
- The custom preloader is disabled in the following cases when the native sandbox is enabled:
  - The custom preloader is disabled when a sandbox applet or Java Web Start application is initializing. The default preloader is used instead. After the application is initialized, Java VM restarts with the native sandbox enabled and the custom preloader is used.
  - For all-permission applications, the custom preloader is disabled if it is located in a JNLP file that has sandbox permission, until the user agrees to run the

application from the Security Dialog, which grants unrestricted access (privileged) to the application.

Miscellaneous

The following choices are available depending on your platform, none are checked by default:

- Store user settings in the roaming profile (Windows only)

  By default, user settings are stored in `*user_home*\AppData\LocalLow\Sun\Java\Deployment`. Select this option to store user settings in `*user_home*\AppData\Roaming\Sun\Java\Deployment`. When selected, the `deployment.properties` file is copied to the `Roaming` directory. When deselected, the file is removed from the `Roaming` directory. In addition, when the option is selected, the following items are also stored in the `Roaming` directory:

  – Local application properties
  – Security baselines
  – Blacklisted certificates
  – Blacklisted JAR files
  – User certificate stores
  – Exception site list

- Place Java icon in system tray
- Suppress sponsor offers when installing or updating Java

  Select this option if you do not want to be provided with offers from sponsors during the installation or update process.

10 Deployment Rule Set

The Deployment Rule Set feature is for enterprises that manage their Java desktop environment directly, and provides a way for enterprises to continue using legacy business applications in an environment of ever-tightening Java applet and Java Web Start application security policies. This feature also provides the ability to control the version of the JRE that is used for specific applications.

Note:

The Deployment Rule Set feature is optional and shall only be used internally in an organization with a controlled environment. If a JAR file that contains a rule set is distributed or made available publicly, then the certificate used to sign the rule set will be blacklisted and blocked in Java.

This topic contains the following sections:

Overview of Deployment Rule Sets

The Deployment Rule Set feature enables an enterprise to establish a whitelist of known applications that can run without security prompts.

Java applets, Java Web Start applications, and JavaFX applications launched from or embedded in a browser are known collectively as Rich Internet Applications (RIAs). To protect the user and minimize the possibility that a RIA was compromised, security checks are performed when a RIA is started, and the user is prompted for permission to run the RIA. Applications on the whitelist defined by a Deployment Rule Set can be run without most security prompts, however, the following prompts are not suppressed:

- HTTPS security warnings
- Authentication dialogs that require the user to provide credentials to connect
- Security warnings from unsigned Java Web Start applications that want to perform such actions as creating a shortcut or an association

Rules for deployment are defined in an XML file and packaged in a signed JAR file. The rules are tested sequentially until one matches the RIA. Depending on the action assigned to the rule, the RIA is then run without security prompts, blocked from running, or run with default processing that shows any security prompts that are applicable. If no rules are matched, then default processing is used. The rules also provide the ability to specify the version of the JRE that is used to run the RIA and suppress the notification of out-of-date JREs.

An active rule set that is installed on the system can be viewed from the Web Settings tab of the Java Control Panel.

Create the Rule Set

The rule set is an XML file that must be named `ruleset.xml`. Use any text editor to create this file.

Define the Rules

Define the rules that you need to run or block RIAs for your organization. See Java Deployment Rule Set DTD for the syntax of the rule set. If unknown elements or attributes are included in the rule set, warnings are written to the Java Console.

The following elements are valid:

- <ruleset>
- <rule>
- <id>
- <certificate>
- <checksum>
- <jnlp-checksum>
- <action>
- <message>
- <customer>

If the rule set is invalid, then an error that describes the problem is shown and all RIAs are blocked. Either the `ruleset.xml` file must be corrected, or the `DeploymentRuleSet.jar` file must be removed from the deployment directory (see Installing the Rule Set for the location of this directory) before RIAs can be run. If a rule set is reported as invalid, then check for the following problems based on the error you received:

- The file is unreadable.
- The structure of the file is invalid.
- The rules are not properly defined.
- A rule with an action of `run` has no selection criteria provided and therefore matches all RIAs.
- The JAR file is not properly signed with a valid certificate.

If the `DeploymentRuleSet.jar` file is removed, RIAs are handled by the default deployment process.

See Examples for some sample rule sets.

<ruleset>

The `<ruleset>` element is the top-level element of the policy file.

The valid child elements are <rule> and <customer>.

The following table describes the valid attribute.

Table 10-1 Attribute for <ruleset>

Attribute	Description	Required
`version`	Minimum version of the Deployment Rule Set specification that is required to process this file. Use a plus sign (+) to indicate that later versions can also be used, for example `1.0+`. If your JRE does not support the version specified, all RIAs are blocked.	Yes

<rule>

The `<rule>` element defines the action taken for the RIA or set of RIAs that is matched by the criteria specified for the rule.

This element contains one <id> element, one <action> element, and optional <customer> elements. Rules are processed sequentially until a rule is matched. When a match is found, no further rules are processed. If no rule is matched, then default processing is used, and any relevant security prompts or warnings are shown.

Note:

When a RIA has artifacts that are signed with a different certificate or that are in a different location, ensure that the rule set contains rules for all artifacts of the RIA. For mixed code cases, which are calls between JAR files with different permission levels or calls from JavaScript code to privileged Java code, see Set Up Rules for Mixed Code.

The valid parent element is <ruleset>. The valid child elements are <id> and <action>.

This element has no attributes.

<id>

The `<id>` element identifies the RIA or set of RIAs to which the rule applies. To be considered a match, the RIA must match all attributes and child elements present. If no attributes or child elements are present, then the rule matches all RIAs.

Note:

If the action for a rule is `run`, then at least one attribute or child element must be present.

The valid parent element is <rule>. The valid child elements are <certificate>, <checksum>, and <jnlp-checksum>. Use the `<certificate>` element for signed JAR files. Use the `<checksum>` element in Deployment Rule Set 1.2 and higher for unsigned JAR files. Use the `<jnlp-checksum>` element in Deployment Rule Set 1.3 and higher with a location-based run rule to allow insecure properties in an unsigned JNLP file to be used.

The following table describes the valid attributes.

Table 10-2 Attributes for <id>

Attribute	Description	Required
`location`	URL of the source of the RIA. For RIAs that use JNLP, this value is matched to the `href` attribute in the main JNLP file, or the `jnlp_href` parameter for the applet tag. If there is no `href` attribute or `jnlp_href` parameter provided, use the `<jnlp-checksum>` element with the location. For a JNLP extension, this value is matched to the location of the extension element in the resource element of the main artifact. For RIAs that do not use JNLP, this value is matched to the URL for the HTML file. The path is case sensitive and UTF-8 encoding is assumed. Use of the HTTPS protocol is strongly recommended to avoid potential man-in-the middle attacks. A location is matched based on the following guidelines: • If provided, then the protocols must match exactly. • The host name can start with an asterisk followed by a dot and a host name substring (*.*host-name-substring*), which then matches any host that ends with the host name substring provided after the dot. For example, `*.example.com` matches `host.example.com` and `host.test.example.com`. The host name cannot be just an asterisk or an asterisk with no host name substring, such as *.com. • If provided, then the port numbers must match exactly. • If provided, then the beginning of the paths must match exactly. If the location attribute does not contain a path, then all paths from the host are considered a match. For example, `host.example.com/samples` matches `host.example.com/samples` and `host.example.com/samples/test`, but does not match `host.example.com/test`. However, `host.example.com` matches `host.example.com/samples`, `host.example.com/samples/test`, and `host.example.com/test`. If the `location` attribute is not present, or the value is null, then the location matches all RIAs.	No

Table 10-2 (Cont.) Attributes for <id>

Attribute	Description	Required
`title`	String used in the title element of the JNLP file, or as used by the Java Plug-in. If the `title` attribute is present and the value is not null, then the value must match the title of the RIA exactly. If the `title` attribute is not present, or the value is null, then the title matches all RIAs. If the action for a rule is `run` or `default` and the `title` attribute is present, another `id` attribute or child element must be specified with the `title` attribute, otherwise the rule is invalid.	No

<certificate>

The `<certificate>` element identifies the certificate used to sign the RIA. The `hash` attribute is required.

The valid parent element is <id>. This element has no child elements.

The following table describes the valid attributes:

Table 10-3 Attributes for <certificate>

Attribute	Description	Required
`algorithm`	String that defines the hashing algorithm used to generate the value for the hash attribute. Currently, only security hash algorithm SHA-256 is supported. If the value is null, then SHA-256 is used.	No
`hash`	String of hexadecimal digits that represent the hash value of the code signing certificate. The value is based on the hashing algorithm specified for the `algorithm` attribute. See Get the Certificate Hash for information on getting the value to use.	Yes

<checksum>

The `<checksum>` element identifies the checksum for an unsigned JAR file. The `hash` attribute is required. This element is available in Deployment Rule Set 1.2 and higher.

The valid parent element is <id>. This element has no child elements.

The following table describes the valid attributes:

Table 10-4 Attributes for <checksum>

Attribute	Description	Required
`algorithm`	String that defines the hashing algorithm used to generate the value for the hash attribute. Currently, only security hash algorithm SHA-256 is supported. If the value is null, then SHA-256 is used. If a non-null value other than SHA-256 is used, a warning is issued and SHA-256 is used.	No

Table 10-4 (Cont.) Attributes for <checksum>

Attribute	Description	Required
`hash`	String of hexadecimal digits that represent the hash value of the checksum for the uncompressed form of the JAR file (compression level 0). The value is based on the hashing algorithm specified for the `algorithm` attribute.	Yes

<jnlp-checksum>

The `<jnlp-checksum>` element identifies the checksum for a JNLP file. The `hash` attribute is required. This element is available in Deployment Rule Set 1.3 and higher.

The valid parent element is <id>. This element has no child elements.

If the `location` attribute of the `<id>` element is null, the `<jnlp-checksum>` element is ignored. You can specify more than one `<jnlp-checksum>` element. A rule matches if the checksum for the JNLP file equals the hash attribute of one of the `<jnlp-checksum>` elements in the rule. The following table describes the valid attribute:

Table 10-5 Attributes for <jnlp-checksum>

Attribute	Description	Required
`hash`	String of hexadecimal digits that represent the hash value of the checksum. The value is based on the hashing algorithm specified for the `algorithm` attribute.	Yes

<action>

The `<action>` element defines the action taken for any RIA that matches the rule.

The valid parent element is <rule>. The valid child element is <message>.

The following table describes the valid attributes:

Table 10-6 Attributes for <action>

Attribute	Description	Required
`permission`	Action taken. The valid values are `block`, `default`, and `run`. `block` - RIA is not run. A message is shown to the user. To provide a custom message, include the <message> element. `default` - RIA is run with default processing and any applicable security prompts are shown. To include a custom message when default processing blocks the RIA, use version 1.2 or higher of the Deployment Rule Set and include the <message> element. `run` - The following types of RIAs are allowed to run without prompts: • Signed with a valid certificate from a trusted certificate authority • Signed with an expired certificate • Self-signed • Unsigned • Missing required JAR file manifest attributes To include a custom message with the run action, use version 1.2 or higher of the Deployment Rule Set and include the <message> element.	Yes

Table 10-6 (Cont.) Attributes for <action>

Attribute	Description	Required
`version`	String that identifies the version of the JRE to use to run the RIA. This attribute applies only when the value for the `permission` attribute is `run`. Use the `version` attribute when an older JRE is needed for compatibility with specific RIAs. If the `version` attribute is not present, the RIA is run with the latest JRE available. The following values are valid: • Platform version, such as 1.7, 1.7+, 1.8, 9. A platform version requests the use of any version of the specified platform, or the specified platform or later when a plus sign (+) follows the version. • Implementation version, such as 1.7.0_40, 1.8.0_20. An implementation version requests the use of a specific version. • `SECURE`. The `SECURE` keyword requests the use of any version at or above the security baseline. • `SECURE-version`, where `version` is a valid platform version, such as `SECURE-1.8`. The `SECURE-version` keyword requests the use of any secure version of the specified platform, or the specified platform or later when a plus sign (+) follows the platform. The version of the JRE that is used is determined by the following order of precedence: • The current version of the JRE is used if it is available and matches both the `version` attribute and the version requested by the RIA. • The latest available version of the JRE is used if it matches both the `version` attribute and the version requested by the RIA. • The current version of the JRE is used if it is available and matches the `version` attribute. • The latest available version of the JRE is used if it matches the `version` attribute. If no version is available that meets the criteria, then the RIA is blocked, and a message is shown to the user. To provide a custom message, include the `message` element.	No
`force`	Boolean that indicates if the JRE specified for the `version` attribute must be used to run the RIA. If this attribute is set to `true`, the JRE specified in the rule overrides any JRE requested by the RIA. If the JRE specified in the rule is not available, the RIA is blocked. The default is `false`. For example, if the RIA requests a JRE in the 1.7 family (1.7*) and you want only secure versions of JRE 8 run in your enterprise, you can create a rule that specifies `SECURE-1.8` for the `version` attribute and set the `force` attribute to `true`. This rule forces the RIA to run only with a secure version from the 1.8 family. This attribute is available in 1.1 and later versions of the Deployment Rule Set.	No

<message>

The `<message>` element defines a custom message shown to the user. This message can be used to explain why the RIA is blocked, or provide additional information to the user.

Only plain text is allowed, HTML tags and other special formatting are not supported. If this element is not present, then a default message is shown when a RIA is blocked. In Deployment Rule Set 1.2 and higher, if this element is present for a rule with the action set to run, then an additional dialog is shown to the user. To support multiple locales, include a `message` element for each locale.

If the `locale` attribute is not specified, then the message is used for any locale for which a `message` element is not provided. If a `message` element for the user's locale is not provided and a `message` element without a locale is not present, then a default message is shown.

To ensure that the dialog box that shows the message fits the screen, keep the message under 1024 characters and test for all locales provided.

The valid parent element is <action>. This element has no child elements.

The following table describes the valid attribute:

Table 10-7 Attribute for the <message> element

Attribute	Description	Required
`locale`	Locale to which the message applies.	No

<customer>

Information provided in the `<customer>` element is written to the deployment trace file and the Java console when tracing and the console are enabled. Use this element to provide debug information or associate custom data with a rule or rule set.

You can enter any information in the `<customer>` element as either plain text or valid XML. Include a `<customer>` element within a rule to provide information specific to that rule. Include the element outside of the rules to provide information about the rule set. Prior to Deployment Rule Set 1.2, this element is ignored. In Deployment Rule Set 1.2 and higher, information from this element is copied to the trace file and Java console when they are enabled.

The valid parent elements are <ruleset> and <rule>. This element has no child elements.

This element has no attributes.

Set Up Rules for Calls From JavaScript Code (LiveConnect)

If you need to make calls to your RIA from JavaScript code, then apply the following guidelines to prevent the calls from being blocked:

- If the rule set contains a rule with the action of `run` that matches your RIA, then the rule set must also contain a rule with the action of `run` that matches the location of the JavaScript code.

- If the rule set contains a rule with the action of `default` that matches your RIA, or no rule matches your RIA so default processing is used, then one of the following must be true:
  - The rule set contains a rule with the action of `run` that matches the location of the JavaScript code.
  - The rule set contains a rule with the action of `default` that matches the location of the JavaScript code.
  - No rule matches the location of the JavaScript code, so default processing is used.

If the JavaScript code is calling privileged code and you want to suppress mixed code warnings, see Set Up Rules for Mixed Code.

Set Up Rules for Mixed Code

When you create your rule set, ensure that you have rules for all of the artifacts that are associated with the RIAs. Additional rules might be needed to suppress mixed code security warnings that are generated when calls are made between code with different permission levels, or from JavaScript code to privileged Java code.

To suppress the mixed code security warnings, include rules in your rule set based on the following requirements of your RIA:

- To make calls between Java code with different permission levels, add a rule with an action of `run` that matches the code being called.

  For example, the following rule suppresses the mixed code prompt for calls to privileged code located at `https://host.example.com/apps` from sandbox code:

  ```
  <rule>
   <id location="https://host.example.com/apps"/>
   <action permission="run"/>
   </rule>
  ```

- To call privileged Java code from JavaScript code, add a rule with an action of `run` that matches the location of the JavaScript code.

  For example, the following rule suppresses the mixed code prompt for calls to privileged Java code from JavaScript code that is located in any page on `https://host.example.com`.

  ```
  <rule>
   <id location="https://host.example.com/"/>
   <action permission="run"/>
   </rule>
  ```

  If the rule set has no rule with an action of `run` or `default` that matches the location of the JavaScript code, then calls from JavaScript code are blocked. If you want any applicable security prompts to be shown for calls from JavaScript code, you must define a rule with an action of `default` that matches the location of the JavaScript code.

Be aware that the rules shown in this section for suppressing the mixed code prompt also suppress the other security prompts for any RIA that matches the rule. Make sure that your rules are defined in the order needed to provide the control that you want.

Get the Certificate Hash

If you want to define a rule that uses the certificate hash to match RIAs, you need to obtain the correct string of hexadecimal digits. Follow these steps:

1. Use the following command to print out the certificate information for your JAR file, replacing `myjar.jar` with the name of your JAR file:

   ```
   keytool -printcert -jarfile myjar.jar | more
   ```

2. At the beginning of the output, find `Signer #1`
3. In the `Certificate fingerprints` section under `Signer #1`, find the line that begins with `SHA256`.

   The text that follows the `SHA256` identifier contains 32 pairs of hexadecimal numbers separated by colons. Use this string for the `hash` attribute of the `certificate` element. The string can be used with or without the colons.

Packaging the Rule Set

The rule set defined in the `ruleset.xml` file must be packaged in a signed JAR file named `DeploymentRuleSet.jar`. The JAR file must be signed with a valid certificate from a trusted certificate authority.

For information about creating and signing a JAR file, see the lesson Packaging Programs in JAR Files in the Java Tutorials.

Installing the Rule Set

The rule set must be installed on every system on which you need to run Java applications.

Install the `DeploymentRuleSet.jar` file on your users' systems in the following directories:

- On Windows platforms, install the file in the `<`*`Windows-directory`*`>\Sun\Java\Deployment` directory, for example, `c:\Windows\Sun\Java\Deployment`.
- On Solaris and Linux platforms, install the file in the `/etc/.java/deployment` directory.
- On macOS platforms, install the file in the `/Library/Application Support/Oracle/Java/Deployment/` directory.

Viewing the Active Rule Set

Only one deployment rule set can be active at a time. The active rule set can be viewed from the Java Control Panel.

To view the active rule set:

1. Start the Java Control Panel.
2. Go to the Web Settings tab.
3. Go to the Deployment Rule Set tab.

If a rule set is active, a link to the rule set is shown. If no active rule set exists, a message is shown.

4. Click the **View the active Deployment Rule Set** link.

Security Considerations

The Deployment Rule Set feature enables RIAs to run without notifying users of potential security risks. Review the following security considerations to be aware of the risks of using a rule set, and follow any recommendations provided:

- The `location` attribute of the `id` element is compared to the following information:
  - Location of the HTML file, for applets that do not use JNLP
  - Value of the `href` attribute in the JNLP file, for Java Web Start applications and applets that do use JNLP
  - Value of the `jnlp_href` parameter for the applet tag, for applets that use JNLP and do not provide the `href` attribute in the JNLP file

  If matched, then all of the content in the HTML file or JNLP file is considered trusted. However, if the web site that hosts the file is vulnerable to cross-site scripting attacks, malicious content could be injected into the HTML file or JNLP file.
- For applets that use JNLP, the location of the HTML file is not checked, so the applet could potentially be started from anywhere.
- If the `location` attribute is not used to match a rule to a RIA, then the HTML file or JNLP file that is used to start the RIA could be compromised. Use of the `location` attribute is recommended.
- Only include a path in the `location` attribute for a rule with an action of run if you trust the entire server. Using a path in a run rule when other locations on the server might not be trusted could present a security risk and is not recommended.
- When a path is included in the `location` attribute, avoid using complex paths or multi-byte characters, if possible. The path is case sensitive and UTF-8 encoding is assumed. Security exceptions occur when any unsupported characters, decoding errors, or overlong encoding is encountered. If the web server, file system, or browser normalizes the path differently, a rule based on the `location` attribute could return unexpected results.
- A blocking rule for a specific URI is not intended to be a robust security enforcement mechanism. For example, a rule established with a domain name can be bypassed if a user uses the IP address instead. The recommended practice is to have a final rule in your rule set with no identifiers and an action of block. Define the rules that you need to run RIAs without security prompts or with default processing, and let all other RIAs be matched by the final rule, which blocks them from running.
- Use of the HTTPS protocol is recommended for all locations.
- The order of the rules in the deployment rule set is critical. Rules are processed sequentially from the beginning of the file. When a match is found, no further rules are processed. Review your final rule set and look at both positive and negative cases to ensure that the rules cover the RIAs that you plan to manage without allowing matches to unknown RIAs.

- Rules are required for all artifacts of the RIA, such as multiple JAR files and JNLP extensions. Be careful when defining a rule for an artifact so that you do not inadvertently allow other RIAs that match the rule to run.
- Deployment rules allow RIAs to run with old versions of the JRE when needed for compatibility, however, older versions might have known security issues. Use the latest JRE whenever possible, and set the `version` attribute to `SECURE` or `SECURE-version`. If an older version of the JRE must be used, make any rule that requests the old version as restrictive as possible to limit the RIAs that match the rule and run with the old version. Use of all of the identifiers-location, title, and certificate hash-is recommended in this case.
- If a rule with an action of `run` exists for the RIA, the RIA is run even if the certificate used to sign the RIA is expired.

Examples

Sample deployment rule sets are provided to show how deployment rules are used to block or allow applications to run.

Example 10-1   Run RIAs from a Single Location

This example allows all RIAs from `https://host.example.com/` to run without security prompts. RIAs from other locations do not match the rule so default processing is used and security prompts are shown as applicable.

```
<ruleset version="1.0+">
  <rule>
    <id location="https://host.example.com" />
    <action permission="run" />
  </rule>
</ruleset>
```

Example 10-2   Block Any RIAs Not Matched

To ensure that all RIAs are handled by the rule set, you can provide a final rule that matches any RIA that was not matched by a previous rule. The action for this rule must be either `block` or `default`. This example allows all RIAs from `https://host.example.com:8080` to run without security prompts and blocks all other RIAs. The default message is shown when a RIA is blocked because no custom message is provided.

```
<ruleset version="1.0+">
  <rule>
    <id location="https://host.example.com:8080" />
    <action permission="run" />
  </rule>

  <rule>
    <id />
    <action permission="block" />
  </rule>
</ruleset>
```

Example 10-3   Rule Order Matters

Rules are processed in the order in which they appear in the rule set. Complex patterns can be defined for matching rules by placing the rules in the correct order. This example allows RIAs from `https://host.example.com` to run without security prompts using a secure version of the Java 1.7 platform, but uses default processing for RIAs from `https://host.example.com/games`, which shows applicable security prompts. RIAs from other locations do not match either rule, so default processing is used.

```
<ruleset version="1.0+">
  <rule>
    <id location="https://host.example.com/games" />
    <action permission="default" />
  </rule>

  <rule>
    <id location="https://host.example.com" />
    <action permission="run" version="SECURE-1.7" />
  </rule>
</ruleset>
```

Example 10-4 Manage a Specific RIA

This example modifies the rule set in Example 10-3 and requires only the RIA named *Solitaire* from `https://host.example.com/games` to run with default processing. Other RIAs from `https://host.example.com` are allowed to run without security prompts using a secure version of the Java 1.7 platform. All other RIAs are blocked, and a custom message is shown.

```
<ruleset version="1.0+">
  <rule>
    <id title="Solitaire" location="https://host.example.com/games" />
    <action permission="default" />
  </rule>

  <rule>
    <id location="https://host.example.com" />
    <action permission="run" version="SECURE-1.7" />
  </rule>

  <rule>
    <id />
    <action permission="block">
      <message>Blocked by corporate. Contact J. Smith, smith@host.example.com, if
you need to run this app.</message>
    </action>
  </rule>
</ruleset>
```

Example 10-5 Manage RIAs Based on Signing Certificate

To allow multiple RIAs from multiple locations to run, and all RIAs are signed with the same certificate, you can use the `certificate` element to identify the RIAs with one rule instead of creating rules for each location and title. This example allows all RIAs that are signed with the certificate used by Oracle to run without security prompts using a secure version of the Java platform. RIAs from any host ending with `example.com` are allowed to run with default processing. All other RIAs are blocked, and a custom message is shown.

```
<ruleset version="1.0+">
  <rule> <!-- allow anything signed with company's public cert -->
    <id>
      <certificate
hash="794F53C746E2AA77D84B843BE942CAB4309F258FD946D62A6C4CCEAB8E1DB2C6" />
```

```
    </id>
    <action permission="run" version="SECURE" />
  </rule>

  <rule>
    <id location="*.example.com" />
    <action permission="default" />
  </rule>

  <rule>
    <id />
    <action permission="block">
      <message>Blocked by corporate. Contact J. Smith, smith@host.example.com, if
you need to run this app.</message>
    </action>
  </rule>
</ruleset>
```

Example 10-6   Force the Use of a Specific JRE

To force the use of a specific JRE, use the `force` attribute of the `action` element. This attribute is introduced in the 1.1 version of the Deployment Rule Set. This example allows RIAs from `https://host.example.com/apps` to run without security prompts using version 1.8_20 of the JRE. Any version requested by the RIA is ignored. If version 1.8_20 is not available, the RIA is blocked. All other RIAs are blocked, and a custom message is shown.

```
<ruleset version="1.1+">
  <rule>
    <id location="https://host.example.com/apps" />
    <action permission="run" version="1.8_20" force="true" />
  </rule>

  <rule>
    <id />
    <action permission="block">
      <message>Blocked by corporate. Contact J. Smith, smith@host.example.com, if
you need to run this app.</message>
    </action>
  </rule>
</ruleset>
```

Example 10-7   Include Customer Data in the Rule Set

The `customer` element enables you to provide comments, debug information, or other data about the rules and rule set. Starting in Deployment Rule Set 1.2, this information is written to the deployment trace file and the Java console when they are enabled. This example shows customer information for the rule set and for two of the rules. The XML elements used in the `customer` element is for illustration purposes, any valid XML can be used.

```
<ruleset version="1.2+">
  <rule>
    <id location="https://host.example.com/verified-apps" />
    <action permission="run" />
    <customer>Allowing applications from https://host.example.com, which has been
validated as a secure site</customer>
  </rule>

  <customer>
     <warning font=bold>Run rule not matched.</warning>
     <text>Application will either be blocked or will show security dialogs.</text>
  </customer>

  <rule>
    <id location="*.example.com" />
    <action permission="default" />
  </rule>

  <rule>
    <id />
    <action permission="block">
      <message>Blocked by corporate. Contact J. Smith, smith@host.example.com, if
you need to run this app.</message>
    </action>
    <customer>
       <warning font=bold>Blocked</warning>
       <text>No rule matched, application blocked by final rule.</text>
    </customer>
  </rule>
</ruleset>
```

Java Deployment Rule Set DTD

The following example shows the DTD for the version 1.3 of the Deployment Rule Set. Version 1.3 is supported by 8u72 and higher. Version 1.2 is supported by JRE 8u60 and higher. Version 1.1 is supported by JRE 8u20 and higher. Version 1.0 is supported by JRE 7u40 and higher. Items introduced after version 1.0 are noted.

```
<!ELEMENT ruleset (rule*, customer*)>
<!ATTLIST ruleset version CDATA #REQUIRED>

<!ELEMENT rule (id, action, customer*)>

<!-- checksum introduced in 1.2, jnlp-checksum introduced in 1.3 -->
<!ELEMENT id (certificate?, checksum?, jnlp-checksum*)>
<!ATTLIST id title CDATA #IMPLIED>
<!ATTLIST id location CDATA #IMPLIED>

<!-- jnlp-checksum introduced in 1.3 -->
<!ELEMENT jnlp-checksum EMPTY>
<!ATTLIST jnlp-checksum hash CDATA #REQUIRED>

<!ELEMENT certificate EMPTY>
<!ATTLIST certificate algorithm CDATA #IMPLIED>
<!ATTLIST certificate hash CDATA #REQUIRED>

<!-- checksum introduced in 1.2 -->
<!ELEMENT checksum EMPTY>
<!ATTLIST checksum algorithm CDATA #IMPLIED>
<!ATTLIST checksum hash CDATA #REQUIRED>

<!ELEMENT action (message?)>
<!ATTLIST action permission (run | block | default) #REQUIRED>
<!ATTLIST action version CDATA #IMPLIED>
<!ATTLIST action force (true|false) "false">  <!-- force introduced in 1.1 -->

<!ELEMENT message (#PCDATA)>
<!ATTLIST message locale CDATA #IMPLIED>

<!ELEMENT customer ANY>
```