

## **JavaFX**

Deploying JavaFX Applications

Release 2.2.40

**E20472-11**

September 2013

JavaFX Deploying JavaFX Applications Release 2.2.40

E20472-11

Copyright © 2008, 2013 Oracle and/or its affiliates. All rights reserved.

Primary Author: Dmitry Kostovarov

Contributor:

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate failsafe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

---

---

# Contents

## Part I About This Guide

### 1 What's New

### 2 Getting Started

2.1	Deployment Quick Start .....	2-1
2.2	Write Once, Deploy Anywhere .....	2-1
2.3	Application Package .....	2-2
2.3.1	Self-Contained Applications .....	2-2
2.4	Packaging Tools .....	2-2
2.4.1	NetBeans IDE .....	2-3
2.4.2	Ant Tasks .....	2-3
2.4.3	JavaFX Packager Command-Line Tool .....	2-4
2.5	User Experience .....	2-4
2.6	Getting the Most Out of the Execution Environment .....	2-5
2.7	Deploying Swing and SWT Applications with Embedded JavaFX Content .....	2-6

### 3 Application Execution Modes

3.1	Execution Modes .....	3-1
3.2	Understanding Feature Differences .....	3-2
3.2.1	Preloader Support .....	3-2
3.2.2	Desktop Integration via Shortcut .....	3-3
3.2.3	Built-In Proxy Support .....	3-3
3.2.4	Run in Sandbox Unless Signed and Trusted .....	3-3
3.2.5	Auto-Updates .....	3-3
3.2.6	Deployment Toolkit .....	3-4
3.2.7	Communicate to the Host Web Page .....	3-4
3.2.8	Managing Platform Dependencies .....	3-4
3.3	Coding Tips .....	3-5
3.3.1	Detecting Embedded Applications .....	3-5
3.3.2	Accessing Application Parameters .....	3-5
3.3.3	Consider the Use of Host Services .....	3-5
3.3.4	Loading Resources .....	3-6
3.3.5	Resize-Friendly Applications .....	3-6

## 4 Application Startup

4.1	Application Startup Process, Experience, and Customization .....	4-1
4.1.1	Startup Process .....	4-1
4.1.1.1	Visual Feedback During Phase One Initialization .....	4-2
4.1.1.2	Visual Feedback After Initialization .....	4-2
4.1.2	Default User Experience .....	4-3
4.1.3	Customization Options .....	4-4
4.2	Helping Users Start the Application .....	4-5
4.2.1	No JavaFX Runtime .....	4-5
4.2.1.1	Standalone Launch .....	4-5
4.2.1.2	Launch with the Deployment Toolkit .....	4-5
4.2.1.3	Launch a Remote Application without the Deployment Toolkit .....	4-6
4.2.2	Runtime Errors .....	4-6

## 5 Packaging Basics

5.1	JavaFX Packaging Overview .....	5-1
5.2	Base Application Package .....	5-2
5.3	Overview of Packaging Tasks .....	5-2
5.3.1	JavaFX Packaging Tools .....	5-4
5.4	Stylesheet Conversion .....	5-5
5.5	Create the Main Application JAR File .....	5-5
5.6	Sign the JAR Files .....	5-6
5.7	Run the Deploy Task or Command .....	5-7
5.7.1	Configure the Deployment Descriptor .....	5-7
5.7.2	Application Resources .....	5-8
5.7.3	Package Custom JavaScript Actions .....	5-9
5.7.4	Web Page Templates .....	5-9
5.8	Packaging Cookbook .....	5-11
5.8.1	Passing Parameters to the Application .....	5-11
5.8.2	Customizing JVM Setup .....	5-11
5.8.2.1	Specifying User JVM Arguments .....	5-12
5.8.2.2	Macro Expansion of Application Directory for <code>jvmarg</code> and <code>jvmuserarg</code> .....	5-13
5.8.3	Packaging Complex Applications .....	5-13
5.8.4	Publishing an Application that Fills the Browser Window .....	5-15
5.9	Performance Tuning for Web Deployment .....	5-16
5.9.1	Background Update Check for the Application .....	5-16
5.9.2	Embed the Deployment Descriptor into the Web Page .....	5-16
5.9.3	Embed Signing Certificate into Deployment Descriptor .....	5-17
5.9.4	Use New JavaFX Signing Method (Signed Applications) .....	5-18

## 6 Self-Contained Application Packaging

6.1	Introduction .....	6-1
6.2	Pros and Cons of Self-Contained Application Packages .....	6-1
6.3	Basics .....	6-2
6.3.1	Self-Contained Application Structure .....	6-3
6.3.2	Basic Build .....	6-3

6.3.3	Customization Using Drop-In Resources .....	6-4
6.3.3.1	Preparing Custom Resources .....	6-5
6.3.3.2	Substituting a Built-In Resource .....	6-5
6.3.4	Customization Options .....	6-6
6.3.5	Platform-Specific Customization for Basic Packages .....	6-7
6.3.5.1	Mac OS X .....	6-7
6.4	Installable Packages .....	6-7
6.4.1	EXE Package .....	6-9
6.4.2	MSI Package .....	6-10
6.4.3	DMG Package .....	6-11
6.4.4	Linux Packages .....	6-12
6.5	Working Through a Deployment Scenario .....	6-13

## 7 Deployment in the Browser

7.1	API Overview .....	7-1
7.1.1	Application Descriptor (dtjava.App) .....	7-2
7.1.2	Platform (dtjava.Platform) .....	7-3
7.2	Callbacks .....	7-5
7.2.1	onDeployError .....	7-6
7.2.2	onGetNoPluginMessage .....	7-7
7.2.3	onGetSplash .....	7-7
7.2.4	onInstallFinished .....	7-7
7.2.5	onInstallNeeded .....	7-8
7.2.6	onInstallStarted .....	7-8
7.2.7	onJavascriptReady .....	7-9
7.2.8	onRuntimeError .....	7-9
7.3	Examples .....	7-9
7.3.1	Embedded Application Starts After the DOM Tree Is Constructed .....	7-9
7.3.2	Launch a Web Start Application from a Web Page .....	7-10
7.3.3	Pass Parameters to a Web Application .....	7-11
7.3.4	Specify Platform Requirements and Pass JVM Options .....	7-12
7.3.5	Access JavaFX Code from JavaScript .....	7-12
7.3.6	Disable the HTML Splash Screen .....	7-13
7.3.7	Add a Custom HTML Splash Screen .....	7-14
7.3.8	Create a Handler for an Unsupported Platform .....	7-15
7.3.9	Check for Presence of JavaFX Runtime .....	7-16

## 8 JavaFX and JavaScript

8.1	Accessing a JavaFX Application from a Web Page .....	8-1
8.2	Accessing the Host Web Page from an Embedded JavaFX Application .....	8-3
8.3	Advanced topics .....	8-4
8.4	Threading .....	8-5
8.5	Security .....	8-6
8.6	Tab Pane Example .....	8-6

## 9 Preloaders

9.1	Implementing a Custom Preloader .....	9-1
9.2	Packaging an Application with a Preloader .....	9-3
9.2.1	Packaging a Preloader Application in NetBeans IDE .....	9-4
9.2.2	Packaging a Preloader Application in an Ant Task .....	9-5
9.3	Preloader Code Examples .....	9-7
9.3.1	Show the Preloader Only if Needed .....	9-7
9.3.2	Enhance Visual Transitions .....	9-8
9.3.3	Using JavaScript with a Preloader .....	9-8
9.3.4	Using a Preloader to Display the Application Initialization Progress .....	9-10
9.3.5	Cooperation of Preloader and Application: A Login Preloader .....	9-13
9.3.6	Cooperation of Preloader and Application: Sharing the Stage .....	9-15
9.3.7	Customizing Error Messaging .....	9-17
9.4	Performance Tips .....	9-18

## 10 JavaFX in Swing Applications

10.1	Overview .....	10-1
10.2	Packaging with JavaFX Ant Tasks .....	10-1
10.2.1	Enabling an HTML Splash Screen .....	10-2
10.3	Packaging without JavaFX Tools .....	10-3
10.3.1	Using the Deployment Toolkit .....	10-3

## 11 The JavaFX Packager Tool

javafxpackager .....	11-2
----------------------	------

## 12 JavaFX Ant Tasks

12.1	Requirements to Run JavaFX Ant Tasks .....	12-1
12.2	JavaFX Ant Elements .....	12-1
12.3	Using JavaFX Ant Tasks .....	12-2
12.4	Ant Script Examples .....	12-2
	JavaFX Ant Task Reference .....	13-1
	<fx:csstobin> .....	13-2
	<fx:deploy> .....	13-3
	<fx:jar> .....	13-7
	<fx:signjar> .....	13-9
	JavaFX Ant Helper Parameter Reference .....	13-11
	<fx:application> .....	13-12
	<fx:argument> .....	13-14
	<fx:callback> .....	13-15
	<fx:callbacks> .....	13-16
	<fx:fileset> .....	13-17
	<fx:htmlParam> .....	13-19
	<fx:icon> .....	13-21
	<fx:info> .....	13-22

<fx:jvmarg> .....	13-24
<fx:jvmuserarg> .....	13-25
<fx:param> .....	13-26
<fx:permissions> .....	13-27
<fx:platform> .....	13-28
<fx:preferences> .....	13-30
<fx:property> .....	13-32
<fx:resources> .....	13-33
<fx:splash> .....	13-35
<fx:template> .....	13-36

## 13 Troubleshooting

13.1	Running Applications .....	14-1
13.2	Development Process Issues .....	14-1
13.3	Runtime Issues .....	14-2
13.3.1	Standalone Execution .....	14-3
13.3.2	Self-Contained Applications .....	14-3
13.3.3	Web Start .....	14-4
13.3.4	Applications Embedded in the Browser .....	14-4
13.3.5	Disabling the Autoproxy Configuration in the Code .....	14-4





# Part I

---

## About This Guide

This guide provides basic and advanced information about building, packaging, and deploying your JavaFX application. JavaFX deployment requires no special code in your application and has many other differences from Java deployment. Even if you are an advanced Java developer, it is a good idea to review the Getting Started page.

**Tip:** For updates to this information, watch the following blogs and forums:

- The "JavaFX 2.0 and Later" Forum on OTN
- The Java Tutorials Blog
- The FX Experience Blog

This guide contains the following topics:

- [What's New](#)  
Describes new and changed features in the current release.
- [Getting Started](#)  
This page shows you how to get a basic JavaFX application running.
- [Application Execution Modes](#)  
An introduction of the terms and concepts used in JavaFX deployment.
- [Application Startup](#)  
How users experience JavaFX application startup and how to customize the default behavior.
- [Packaging Basics](#)  
An introduction to packaging JavaFX applications.
- [Self-Contained Application Packaging](#)  
How to create self-contained application packages, which can be distributed either as zip files or as operating-system-specific installers.
- [Deployment in the Browser](#)  
Topics related to applications embedded in the browser.
- [JavaFX and JavaScript](#)  
How to make JavaFX and JavaScript communicate with each other.
- [Preloaders](#)

Information about creating your own preloader.

- [JavaFX in Swing Applications](#)

How to deploy Swing applications in which JavaFX is embedded.

- [The JavaFX Packager Tool](#)

Information about Using the JavaFX Packager tool and a reference to the command line syntax.

- [JavaFX Ant Tasks](#)

[JavaFX Ant Task Reference](#)

[JavaFX Ant Helper Parameter Reference](#)

Reference pages for Ant Task elements and attributes that work with the JavaFX Ant task API.

- [Troubleshooting](#)

Basic troubleshooting tips if your first deployment efforts are not successful.

---

---

## What's New

This chapter enumerates new deployment features.

The following list describes the new features in JavaFX 2.2 that affect deployment.

- **Starting with JRE 7 Update 6, JavaFX Runtime is part of JRE installation**

This integration significantly simplifies the process of getting the system ready to run JavaFX applications. No separate installation is needed for JavaFX, and there is no need to write custom code to detect where JavaFX is installed. Moreover, JavaFX Runtime will be autoupdated in the same way as Java Runtime.

There is nothing needed to be changed in the application code but you may need to repackage your application to update your copy of deployment toolkit javascript APIs.

- **Deployment of JavaFX applications is now supported on Mac and Linux**

JavaFX applications are expected to work on Mac and recent Linux distributions, as long as they meet system requirements. No code changes or packaging changes are needed.

However, it is a good idea to repackage the application using the latest version of the packaging tools, to update the built-in launchers plus the copy of the current Deployment Toolkit, to be aware of new supported platforms.

- **JavaFX applications can be redistributed as self-contained application packages**

These platform-specific packages include all application resources and a private copy of Java and JavaFX Runtimes. Distributed as a native installable package, they provide the same installation and launch experience as native applications for that operating system. See [Chapter 6, "Self-Contained Application Packaging."](#)

- **Pass parameters to a Web Start Application from a Web Page**

See [Section 7.3.3, "Pass Parameters to a Web Application."](#)

- **Better support for packaging Swing applications with integrated JavaFX content**

Packaging tools are extended to simplify deploying hybrid applications. You now use the same deployment approach as you would for pure JavaFX applications.

The resulting package provides support for the same set of execution modes as a package for a JavaFX application; in other words, the application can be run standalone, using Web Start, embedded into a web page, or distributed as a self-contained application bundle. See [Chapter 10, "JavaFX in Swing Applications."](#)

- **Support for relative sizes for embedded applications**

---

Embedded applications can now be deployed with size relative to the browser window. See [Section 5.8.4, "Publishing an Application that Fills the Browser Window."](#)

---

---

## Getting Started

This page shows you the basics of taking your JavaFX application from code to deployment.

- [Section 2.1, "Deployment Quick Start"](#)
- [Section 2.2, "Write Once, Deploy Anywhere"](#)
- [Section 2.3, "Application Package"](#)
- [Section 2.4, "Packaging Tools"](#)
- [Section 2.5, "User Experience"](#)
- [Section 2.6, "Getting the Most Out of the Execution Environment"](#)
- [Section 2.7, "Deploying Swing and SWT Applications with Embedded JavaFX Content"](#)

### 2.1 Deployment Quick Start

Mastered JavaFX basics and have your application ready? Now want to learn what to do to publish it?

Here is all you have to do:

- Decide how you want to deploy the application.
- Use JavaFX tools to create the application package.
- If you plan to embed the application in a web page, update your HTML page to include the generated JavaScript snippet.
- Copy the package to the place you want to deploy it.
- You are done.

If you would like to try these steps in packaging one of the tutorial applications in the *JavaFX Getting Started* series using the NetBeans IDE, see the page on basic deployment and try out any of the tutorials.

### 2.2 Write Once, Deploy Anywhere

The same JavaFX application can be used in multiple execution environments including:

- Launching a desktop application
- Launching from the command line using the Java launcher

- Launching by clicking a link in the browser to download an application
- Viewing in a web page when opened

## 2.3 Application Package

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

- An application JAR file (or multiple JAR files for large applications)
  - 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 JavaFX content from the web page

For the Colorful Circles application from the JavaFX Getting Started tutorials, the basic package consists of these files:

- ColorfulCircles.jar
- ColorfulCircles.jnlp
- ColorfulCircles.html

### 2.3.1 Self-Contained Applications

Starting from JDK 7 update 6, JavaFX applications can be packaged as a platform-specific, self-contained application. These applications include all application resources, the Java and JavaFX runtimes, and a launcher, and they 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 for Mac, or RPM or DEB for Linux.

Depending on your requirements, this may be a good distribution vehicle for your application:

- They **resemble native applications for the target platform**, in that 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 Java Runtime used by the application is fully controlled by the application developer.
- Your application is easily deployed on fresh systems with **no requirement for Java Runtime to be installed**.
- Deployment occurs with **no need for admin permissions** when using ZIP or user-level installers.

For more information, see [Chapter 6, "Self-Contained Application Packaging."](#)

## 2.4 Packaging Tools

There are three different tools that you can use to package your application:

- [NetBeans IDE](#)
- [Ant Tasks](#)
- [JavaFX Packager Command-Line Tool](#)

The HTML page generated by default 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 these code snippets into an existing web page. For more information, see [Section 5.7.4, "Web Page Templates."](#)

For more information about JavaFX packaging, see [Chapter 5, "Packaging Basics."](#)

## 2.4.1 NetBeans IDE

If you use Netbeans IDE (see the JavaFX Getting Started tutorials for information about how to use JavaFX projects in Netbeans IDE), then it will do most of the work for you. Open Project Properties to specify preferred dimensions for your application scene. Enter 800 for width and 600 for height if you use the Colorful Circles example. Then build the project with Clean and Build. Your application package is generated to the dist folder. Open it in Windows Explorer and try double-clicking the HTML, JNLP, or JAR files.

For more information about packaging and deploying simple JavaFX applications using NetBeans IDE, see the basic deployment page in the *Getting Started with JavaFX* tutorials.

If you want to package a self-contained application, you need to customize the build.xml script in the NetBeans IDE. For more information, see [Section 6.3.2, "Basic Build."](#)

## 2.4.2 Ant Tasks

If you are using another IDE, then you can add JavaFX packaging as a post-build step, using Ant tasks that are included in the JavaFX SDK. [Example 2–1](#) shows an Ant package task for Colorful Circles.

When you add the attribute `nativeBundles="all"` attribute into the `<fx:deploy>` Ant task, all possible packages will be created: a standalone application package, one or more self-contained application packages for that platform, 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 will get three packages: a folder with the application, an .exe installer file, and an .msi installer file. For more information, see [Chapter 5, "Packaging Basics."](#) A simple Ant task with the `nativeBundles` attribute is shown in [Example 2–1](#).

### **Example 2–1 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="{javafx.sdk.path}/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>
```

### 2.4.3 JavaFX Packager Command-Line Tool

If you cannot use Ant and prefer command-line tools, use the JavaFX Packager tool that comes with the JavaFX SDK. The JavaFX Packager tool has several commands for packaging applications, described in [Chapter 11, "The JavaFX Packager Tool."](#) For a quick test build, you can use the `javafxpackager -makeall` command, such as the one in [Example 2-2](#). This command compiles source code and combines the `javafxpackager -createjar` and `javafxpackager -deploy` commands, with simplified options.

#### **Example 2-2** *javafxpackager -makeall Command to Build an Application*

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

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 any other execution mode.

---

---

**Note:** Stage width and height must always be specified for applications embedded in the browser.

---

---

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 more information, see [Section 5.3.1, "JavaFX Packaging Tools."](#)

**Tip:** For even more flexibility of options, use an Ant task instead of the JavaFX Packager tool.

## 2.5 User Experience

Users are easily annoyed if they are unable to start an application, do not understand what are the next steps, or perceive the application as slow or hung and for many other reasons.

Default JavaFX packaging takes care of many problem areas including:

- Ensuring the user has the required JRE and JavaFX installed



- Auto installing missing dependencies or offering to install them as needed
- Providing visual feedback to the user while the application is being loaded
- Showing descriptive error messages

For example, when users do not have JavaFX installed and double-click the JAR file for your application, they see a dialog box explaining that they need to download and install the JavaFX Runtime.

Moreover, developers have a wide range of options on how to tune the experience for their users, such as:

- Customize the messaging (for example explain to users why they need to install JavaFX Runtime in a language other than English)
- Show your own splash screen and use a custom loading progress indicator
- Switch to alternative content if the user's system is not capable of running JavaFX applications

For example, you could pass the following string as a value of the `javafx.default.preloader.stylesheet` parameter to add a company logo to the default preloader:

```
".default-preloader { -fx-preloader-graphic:url
    ('http://my.company/logo.gif'); }"
```

In [Example 2-3](#), the text in bold shows what must be changed in the Ant code used to deploy the `ColorfulCircles` example.

**Example 2-3 Ant Task to Add a Company Logo to the Default Preloader**

```
<fx:deploy width="800" height="600"
    outdir="dist-web" outfile="ColorfulCircles">
  <fx:info title="Colorful Circles"/>
  <fx:application name="Colorful Circles example"
    mainClass="colorfulcircles.ColorfulCircles">
    <fx:param name="javafx.default.preloader.stylesheet"
      value=".default-preloader
        { -fx-preloader-graphic: url('http://my.company/logo.gif'); }" />
  </fx:application>
  <fx:resources>
    <fx:fileset dir="dist-web" includes="ColorfulCircles.jar"/>
  </fx:resources>
</fx:deploy>
```

See the following chapters for more information and examples:

- [Chapter 4, "Application Startup"](#)
- [Chapter 7, "Deployment in the Browser"](#)
- [Chapter 9, "Preloaders"](#)

## 2.6 Getting the Most Out of the Execution Environment

Different execution environments have different specifics, and taking these specifics into account can help to make an application more natural and powerful when run in this execution environment.

One specific could be a unique feature that is not applicable to other environments, for example that applications embedded in a web page can use JavaScript to communicate

to the host web page. Another specific could be an important peculiarity such as a presized stage that is provided to a JavaFX application embedded into a web page.

[Example 2-4](#) shows an example of using JavaScript to go to a new page:

**Example 2-4 Using JavaScript to Go to a New Page**

```
final HostServices services = getHostServices();
JSObject js = services.getWebContext();
js.eval("window.location='http://javafx.com'");
```

See the following pages for more information and examples:

- [Chapter 3, "Application Execution Modes"](#)
- [Chapter 8, "JavaFX and JavaScript"](#)
- [Chapter 9, "Preloaders"](#)

## 2.7 Deploying Swing and SWT Applications with Embedded JavaFX Content

If you are developing Swing applications with embedded JavaFX content, then you must follow deployment scenarios for Swing applications and applets (see the Swing tutorial for tips on coding).

While most of the techniques discussed in this guide are not directly applicable to Swing application with JavaFX content some of them are:

- You can use the same packaging tools to package your Swing applications.
- You can use the Deployment Toolkit to embed your Swing applet into a web page or launch it from a web page.
- Your application can be bundled and packaged as an installable package, using the same technique as for self-contained JavaFX applications.

For more information, see [Chapter 10, "JavaFX in Swing Applications"](#).

SWT applications with embedded JavaFX content are deployed in the same manner as Swing applications, but the SWT library must be included as a resource. For an example of deploying an SWT-JavaFX application, see *JavaFX Interoperability with SWT*.

---



---

## Application Execution Modes

This chapter explains different application execution modes.

One of the main features of the JavaFX application model is that you can write one application and easily deploy it several different ways. The user can experience the same application running on the desktop, in a browser, or starting from a link in a web page.

However, different execution modes are not completely equivalent. There are some important differences to keep in mind while developing the application.

This page contains the following topics:

- [Section 3.1, "Execution Modes"](#)
- [Section 3.2, "Understanding Feature Differences"](#)
- [Section 3.3, "Coding Tips"](#)

### 3.1 Execution Modes

One of the main features of the JavaFX application model is that the applications you develop can be deployed in several different ways, as described in [Table 3-1](#).

**Table 3-1** JavaFX Execution Modes

Execution Mode	Description
Run as a standalone program	The application package is available on a local drive. Users launch it using a Java launcher, such as <code>java -jar MyApp.jar</code> , or by double-clicking the application JAR file.
Launched from a remote server using Web Start	Users click a link in a web page to start the application from a remote web server. Once downloaded, a Web Start application can also be started from a desktop shortcut.
Embedded into a web page	JavaFX content is embedded in the web page and hosted on a remote web server.
Launched as a self-contained application	Application is installed on the local drive and runs as a standalone program using a private copy of Java and JavaFX runtimes. The application can be launched in the same way as other native applications for that operating system, for example using a desktop shortcut or menu entry.

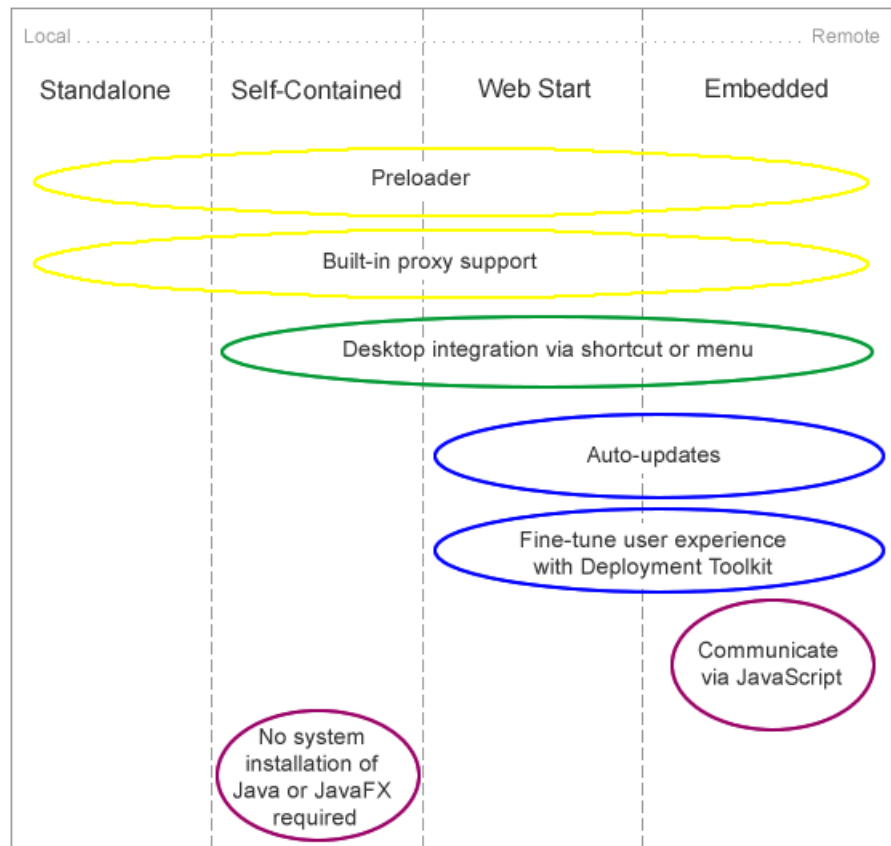
Each execution environment has its own specific complications and usability issues. For example, for remote applications the loading phase can be very long because the

application has to be loaded from the network. This is less of an issue for applications that run on a local drive.

### 3.2 Understanding Feature Differences

Figure 3-1 lists some of the features that behave differently in different environments. The following sections describe the figure in more detail.

Figure 3-1 Features of Deployment Types



#### 3.2.1 Preloader Support

The preloader is a small JavaFX application that receives notifications about application loading and initialization progress. The preloader is used with all execution modes, but depending on the execution mode, the preloader implementation receives a different set of events and optimal behavior may be different.

For example, in self-contained application or standalone execution mode or when launched from a shortcut, the preloader will not get any loading progress events, because there is nothing to load. See Chapter 9, "Preloaders" for information about preloader implementation and differences in behavior.

### 3.2.2 Desktop Integration via Shortcut

Most of the operating systems allow applications to simplify subsequent launch and integrate with the user's desktop by creating a desktop shortcut or adding a link to the programs menu or dock.

Built-in support for desktop shortcuts is available for self-contained and web-deployed applications. There is no built-in support for standalone applications.

### 3.2.3 Built-In Proxy Support

Properly packaged JavaFX application have proxy settings initialized according to Java Runtime configuration settings. By default, this means proxy settings will be taken from the current browser if the application is embedded into a web page, or system proxy settings will be used. Proxy settings are initialized by default in all execution modes.

### 3.2.4 Run in Sandbox Unless Signed and Trusted

WebStart and embedded applications are, by default, run in a restricted environment, known as a sandbox. In this sandbox, Java Runtime does the following:

- Protects users against malicious code that could affect local files.
- Protects enterprises against code that could attempt to access or destroy data on networks.

Applications fall into three categories: signed and trusted, signed and not trusted, and unsigned. When running on a client (unless launched as a standalone application), un-trusted applications operate with maximum restrictions within a security sandbox that allows only a set of safe operations.

Un-trusted applications cannot perform the following operations:

- They cannot access client resources such as the local filesystem, executable files, system clipboard, and printers.
- They cannot connect to or retrieve resources from any third-party server (in other words, any server other than the server it originated from).
- They cannot load native libraries.
- They cannot change the SecurityManager.
- They cannot create a ClassLoader.
- They cannot read certain system properties. See System Properties for a list of forbidden system properties.

### 3.2.5 Auto-Updates

JavaFX applications that run from a web page or were earlier installed from the web automatically check for availability of updates to the application at the original location where the application is loaded. This happens every time an application starts, and, by default, the update runs in the background. The application is automatically updated if updates are detected.

For standalone and self-contained applications, you are responsible for handling updates.

### 3.2.6 Deployment Toolkit

The Deployment Toolkit performs two important functions:

- It helps to simplify web deployment of JavaFX applications by managing updates.
- It improves the end user experience while waiting for applications to start.

These two functions are intertwined, because the application startup phase is a critical component of user satisfaction. For example, the Deployment Toolkit verifies that the user has JavaFX Runtime installed, and if not, it will offer to install it before trying to run the application, without much effort on the user's part.

The Deployment Toolkit provides a JavaScript API and is only available for applications embedded in a web page or launched from a web page.

For more information about the Deployment Toolkit, see [Chapter 7, "Deployment in the Browser."](#)

### 3.2.7 Communicate to the Host Web Page

Applications embedded into a web page can communicate with it using JavaScript. To initiate communication, the application must get the web context from the JavaFX `HostServices` API. For any other execution environment, an attempt to get a reference to web context returns null.

See [Chapter 8, "JavaFX and JavaScript"](#) for information about using JavaScript to communicate with the browser.

### 3.2.8 Managing Platform Dependencies

To run JavaFX content, recent versions of the Java and JavaFX runtimes are required. Unless the application is self-contained, the Java and JavaFX runtimes need to be installed on the user's system.

For most users, JavaFX 2.2 and later will be installed as part of the Java Runtime and will be auto-updated to the latest secure version. If the user does not have the required version of the Java or JavaFX runtime he or she will be guided to install it.

However, there are situations in which system installations of the Java and JavaFX runtime and the auto-update functionality are not sufficient. For example:

- The user does not have admin permissions to install the runtimes.
- The user requires an older version of Java for other applications.
- Users who need multiple system installations feel they are too complicated.
- You want to set the exact version of Java and JavaFX to be used by your application.
- Your distribution channel disallows dependencies on external frameworks

These issues are resolved if you choose to deploy your application as a self-contained application. The Java and JavaFX runtimes will be included in your application package, and users do not need to install them separately. The self-contained application package can be as simple as a .zip file distribution, or it can be wrapped into an installable package using technology that is native to the target operating system. See the topic [Chapter 5, "Packaging Basics"](#) for more details about self-contained application packages.

## 3.3 Coding Tips

The following small programming tips work well in all environments and simplify the development and deployment of applications.

### 3.3.1 Detecting Embedded Applications

When an application is run in embedded mode, it gets staged with predefined dimensions and cannot update them directly. [Example 3–1](#) shows a very simple code snippet to detect if the application is embedded in the browser. The code can be used in either the main application or the preloader start method.

**Example 3–1 Detect if the Application is Embedded in the Browser**

```
public void start(Stage stage) {
    boolean isEmbedded = (stage.getWidth() > 0);
    ...
}
```

As an alternative, you can try to get a reference to the web context from the `Application.getHostServices()` method. It will be null unless the applications is embedded.

### 3.3.2 Accessing Application Parameters

JavaFX applications support both named and unnamed parameters that can be passed in a variety of ways:

- They can be specified on the command line for a standalone launch.
- They can be hardcoded in the application package (jar and deployment descriptor).
- They can be passed from the HTML page in which the application is embedded.

To access parameters from a preloader or main application, use the `getParameters()` method. For example, the code in [Example 3–2](#) gets a list of all named parameters and their values:

**Example 3–2 Get a List of Named Deployment Parameters and Values**

```
Map m = getParameters().getNamed();
int cnt = 0;
String labelText = "List of application parameters: \n";
for(String st: (Set<String>) m.keySet()) {
    labelText += " ["+st+"] : ["+m.get(st)+"]\n";
    cnt++;
}
```

### 3.3.3 Consider the Use of Host Services

The `Application.getHostServices()` method provides access to execution-mode-specific services, including:

- Access to information about the code base and the document base.

For example, for embedded applications this is the URL of the application and URL of the host web page, respectively.

- Access to the host web page using the JavaScript engine, only available to embedded applications.
- Ability to open a web page in the browser.

[Example 3-3](#) shows a few things you can do with `getHostServices()`.

**Example 3-3 Using `getHostServices()`**

```
final HostServices services = getHostServices();

Button jsButton = new Button("Test Javascript");
jsButton.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent t) {
        JSObject js = services.getWebContext();
        js.eval("window.alert('Hello from JavaFX')");
    }
});

Button openButton = new Button("Test openDocument()");
openButton.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent t) {
        services.showDocument("http://javafx.com/");
    }
});
```

### 3.3.4 Loading Resources

Using the `File` API and explicit relative references to external data files or resources may not work when the application is loaded from the web.

The best way to refer to resources relative to your application is to use the `getResource()` method on one of the application classes, as shown in [Example 3-4](#).

**Example 3-4 Use the `getResource()` Method to Load Resources**

```
scene.getStylesheets().
    add(this.getClass().getResource("my.css").toExternalForm());
```

As an alternative, consider using `getCodeBase()` or `getDocumentBase()` from the `HostServices` class to refer to resources relative to the application or the location where the application is used.

### 3.3.5 Resize-Friendly Applications

When an application is embedded into a web page, it cannot control stage dimensions. Dimensions you specify at packaging time are preferences only and can be overridden by the user, for example if the user has custom browser zoom settings. Moreover, the stage can be resized at runtime any time by the user.

To provide a good user experience, it is necessary to be prepared for arbitrary stage size. Otherwise, the application might be cropped, or there could be garbage painted in the unused area of the stage.

If your application uses layouts, then you do not need to do anything. Layouts take care of resizing for you. Otherwise, implement resizing logic and listen to stage dimension changes to update the application, as shown in the simplified code in [Example 3-5](#).



**Example 3-5 Using Listeners to Resize an Embedded Application**

```
public class ResizeFriendlyApp extends Application implements
    ChangeListener<Number> {
    private Stage stage;
    public void start(Stage stage) {
        //be resize friendly
        this.stage = stage;
        stage.widthProperty().addListener(this);
        stage.heightProperty().addListener(this);

        ...

        //resize content
        resize(stage.getWidth(), stage.getHeight());

        stage.show();
    }

    private void resize(double width, double height) {
        //relayout the application to match given size
    }

    public void changed(ObservableValue<? extends Number> ov,
        Number t, Number t1) {
        resize(stage.getWidth(), stage.getHeight());
    }
}
```



---

---

# Application Startup

This chapter provides information about application startup process, user experience, and customization.

The user experience (UE) is an extremely important factor for success of the application. The way the application is deployed creates a first impression on the user and has a crucial impact on user satisfaction, even before the application itself is ready.

Users are easily annoyed if they fail to launch the application, if they do not understand what the next steps are, if they perceive the application to be slow or it hangs, or for other reasons.

A properly packaged and deployed JavaFX application includes tweaks for many typical user experience problems, and with JavaFX, developers have a wide range of options to customize the user experience for their application and its audience.

In this section, the default experience for users of JavaFX applications is explained, and the options the developer has to customize the user experience are presented. This page contains the following sections:

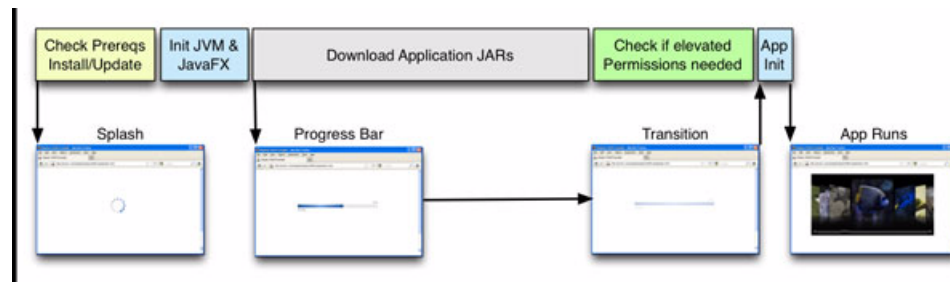
- [Section 4.1, "Application Startup Process, Experience, and Customization"](#)
- [Section 4.2, "Helping Users Start the Application"](#)

## 4.1 Application Startup Process, Experience, and Customization

Out of the box, JavaFX application startup was designed for a good user experience. The following sections describe the transition phases of application startup, how users experience those phases, and how the default visual feedback to the user can be customized.

### 4.1.1 Startup Process

Between the time an application is started and the time the user sees the main application, a sequence of events occurs on screen while operations are carried out in the background, as shown in [Figure 4-1](#) and described in the following paragraphs. This startup sequence partially depends on the execution mode and on the speed with which the background operations complete. [Figure 4-1](#) shows a series of boxes that depict the background operations over time, along with screenshots of what the user sees while these operations occur.

**Figure 4–1 Background Operations and Screen Events During Application Startup**

There are four phases in the application startup process:

- Phase 1: Initialization
 

Initialization of Java Runtime and an initial examination identifies components that must be loaded and executed before starting the application. The initialization phase is depicted in the first two boxes in the upper row in [Figure 4–1](#).
- Phase 2: Loading and preparation
 

The required resources are loaded from either the network or a disk cache, and validation procedures occur. All execution modes see the default or a custom preloader. This phase is depicted in the third box in the upper row in [Figure 4–1](#).
- Phase 3: Application-specific initialization
 

The application is started, but it may need to load additional resources or perform other lengthy preparations before it becomes fully functional. An example of this is checking whether elevated permissions are needed and displaying the appropriate request for permission to the user.
- Phase 4: Application execution
 

The application is displayed and is ready to use. This occurs after the background operations shown in [Figure 4–1](#) are finished.

#### 4.1.1.1 Visual Feedback During Phase One Initialization

Options to provide visual feedback during the first phase of initialization are limited. At that moment, it is not yet known what must be done to launch the application, and the Java platform has not initialized yet. Visual feedback must be provided using external means, for example using JavaScript or HTML if the application is embedded into the web page. By default, a splash screen is displayed during this phase, as described in [Section 4.1.2, "Default User Experience."](#)

#### 4.1.1.2 Visual Feedback After Initialization

To provide visual feedback after the initialization phase, JavaFX provides a preloader, which gets notifications about the loading progress and can also get application-specific notifications. By default, a default preloader with a progress bar is displayed during this phase, as described in [Section 4.1.2, "Default User Experience."](#) You can customize the default preloader (see [Section 4.1.3, "Customization Options"](#)), or you can create your own preloaders to customize the display and messaging (see [Chapter 9, "Preloaders"](#)).

## 4.1.2 Default User Experience

A properly packaged JavaFX application provides default visual feedback to the user for the first two phases for all execution modes. The actual behavior depends on the execution mode.

When launched in standalone mode most applications start quickly, and no visual feedback is required.

Table 4–1 summarizes the default behavior according to execution mode when the application is launched for the first time (in other words, is loaded from the network).

**Table 4–1 Default Behavior During First-Time Launch**









Startup Phase	Web Start Launch	Embedded into Web Page
Phase 1 Initialization	Splash screen: 	Splash screen: 
Phase 2 Loading Code	Progress window: 	Progress window: 
Phase 3 Transition to Application	Hide progress window	Fade-out progress window

Table 4–2 summarizes the default behavior for subsequent launches, in which the JAR files are loaded from the cache. In this case, the process has fewer visual transitions because nothing needs to be loaded from the network during the Loading Code phase, so launch time is substantially shorter.

**Table 4–2 Default Startup Behavior During Subsequent Launches**

	Web Start Launch or Launch from Shortcut	Embedded into Web Page
Phase 1 Initialization	Splash screen: 	Splash screen: 
Phase 2 Loading Code		

**Table 4–2 (Cont.) Default Startup Behavior During Subsequent Launches**

	Web Start Launch or Launch from Shortcut	Embedded into Web Page
Phase 3 Transition to Application	Hide splash screen	Hide splash screen

### 4.1.3 Customization Options

The splash screen for embedded applications is displayed in the web page and can be easily customized by using an `onGetSplash` callback, as shown in [Section 7.2.3, "onGetSplash."](#)

The default preloader can be customized by using a CSS stylesheet, similar to other JavaFX components. Pass the customized style data using the `javafx.default.preloader.stylesheet` parameter for your application. The value of the parameter can be any of following:

- Absolute or relative URI of the CSS stylesheet, either as a text file with a `.css` extension or in binary form with a `.bss` extension. For more information about binary conversion, see [Section 5.4, "Stylesheet Conversion."](#)
- Actual CSS code.

To customize the preloader, use the `.default-preloader` class. In addition to standard CSS keys, the preloader has two special keys:

- `-fx-preloader-status-text`  
Status text to be shown in the preloader
- `-fx-preloader-graphic`  
Absolute or relative URI of the image to be used by the preloader

[Example 4–1](#) shows an example of CSS file `my.css`:

**Example 4–1 Example CSS Class to Customize the Preloader**

```
.default-preloader {
    -fx-background-color: yellow;
    -fx-text-fill: red;
    -fx-preloader-graphic: url("http://host.domain/duke.gif");
    -fx-preloader-text: "Loading, loading, LOADING!";
}
```

Add the stylesheet to the `<fx:deploy>` Ant task as shown in [Example 4–2](#):

**Example 4–2 Adding a Preloader Stylesheet to the `<fx:deploy>` Ant Task**

```
<fx:deploy ...>
  <fx:application ...>
    <fx:htmlParam name="javafx.default.preloader.stylesheet"
      value="my.css"/>
  </fx:application>
</fx:deploy>
```

If your customizations are small, then it is more efficient to pass CSS code instead of a file name, because there is no need to download the file. [Example 4–3](#) shows to change the background color to yellow.

**Example 4–3 Changing the Default Preloader Background Color**

```

<fx:deploy ...>
  <fx:application ...>
    <!-- Using fx:param here, so it will be applicable to all web
         execution environemnts -->
    <fx:param name="javafx.default.preloader.stylesheet"
              value=".default-preloader { -fx-background-color: yellow; }"/>
    <fx:application>
  </fx:deploy>

```

If customizing the default preloader is not enough and you need a different visualization or behavior, see [Chapter 9, "Preloaders"](#) for information about how to implement your own preloader and see [Chapter 5, "Packaging Basics"](#) for information about how to add it to your package.

## 4.2 Helping Users Start the Application

There are various reasons why a particular user might have difficulty getting your application to run, such as the following:

- The user has an unsupported platform.
- The user's system does not have JavaFX Runtime or Java Runtime installed.
- Java is not configured correctly, for example the proxy information is not set.
- The user declined to grant permissions to a signed application.

Because most users never experience any of these problems, it is important to plan for users who experience problems, either providing guidance to resolve the issue or having the application fail gracefully and explaining to the user why it cannot be resolved.

The following sections describe approaches to some common issues.

### 4.2.1 No JavaFX Runtime

If the user does not have JavaFX Runtime installed, then the application cannot start. The JavaFX application package includes several hooks to improve user experience if this is the case.

You can customize the default behavior described in the following sections. See [Chapter 5, "Packaging Basics"](#) for information about how to embed various fallback applications into the application package. See [Chapter 7, "Deployment in the Browser"](#) for information about how to customize prompts to install JavaFX Runtime and error handling for web applications.

#### 4.2.1.1 Standalone Launch

The main application JAR file includes a launcher program, which is used to detect JavaFX runtime. If JavaFX Runtime or Java Runtime is not found, then a dialog box displays that explains where to get JavaFX Runtime and Java Runtime.

#### 4.2.1.2 Launch with the Deployment Toolkit

If the JavaFX application is embedded into a web page or launched from a web page using the Deployment Toolkit (see [Chapter 7, "Deployment in the Browser"](#)), then the Deployment Toolkit takes care of JavaFX Runtime and Java Runtime detection before trying to launch the application. If JavaFX Runtime or Java Runtime is missing, the Deployment Toolkit initiates installation of JavaFX Runtime, either by offering the user

a link to the installer, or by triggering download and installation automatically. By default, automatic download only occurs when the user launches an application using Web Start and has a recent version of the Java Runtime.

#### **4.2.1.3 Launch a Remote Application without the Deployment Toolkit**

The application package includes a fallback Swing application, which is used if an attempt to launch the application is made but the JavaFX Runtime cannot be found.

### **4.2.2 Runtime Errors**

An application can fail to launch due to various runtime errors or user mistakes, such as the absence of a network connection needed to load some of application JAR files.

One of the most common errors is when the user does not grant permissions to the application. In that case, the application fails, and the user has to restart the application and then grant permissions to get it to run. In a case like this, it is helpful to explain to the user why the application failed and what the user must do to restart it successfully. By default, an error message will display, either in a dialog box or inside the web page in the location where the application is embedded. In either case, the messaging can be customized.

If the Deployment Toolkit is used, then the `onDeployError` handler can be used to display an error message in the application area in the web page. You can also consider including some instructions to the splash screen to alert users about granting permissions. For more information, see [Chapter 7, "Deployment in the Browser"](#).

You can also include a custom preloader in your application to get notifications about errors, unless the error occurs while launching the preloader. For more information about preloaders and code examples, see [Chapter 9, "Preloaders."](#)



---

---

## Packaging Basics

This chapter provides an overview of JavaFX packaging and tools.

This page contains the following topics:

- [Section 5.1, "JavaFX Packaging Overview"](#)
- [Section 5.2, "Base Application Package"](#)
- [Section 5.3, "Overview of Packaging Tasks"](#)
- [Section 5.4, "Stylesheet Conversion"](#)
- [Section 5.5, "Create the Main Application JAR File"](#)
- [Section 5.6, "Sign the JAR Files"](#)
- [Section 5.7, "Run the Deploy Task or Command"](#)
- [Section 5.8, "Packaging Cookbook"](#)
- [Section 5.9, "Performance Tuning for Web Deployment"](#)

### 5.1 JavaFX Packaging Overview

A properly packaged JavaFX application runs in one or more of the following deployment modes:

- As a standalone application, using the system Java Runtime
- As a self-contained standalone application, using a private copy of Java Runtime
- As a Web Start application
- Embedded in a web page

By default, JavaFX packaging tools produce a package that includes everything needed to provide a good user experience for various user environments, including:

- Ensuring that the required Java and JavaFX Runtimes are installed
- Autodownloading missing dependencies or offering to install them as needed
- Providing visual feedback to the user while the application is being loaded
- Showing descriptive error messages

JavaFX application packages work out of the box in multiple execution environments, including:

- Launching from the command line using the Java launcher
- Double-clicking the JAR file or self-contained application launcher

- Embedding the application into a web page

Optionally, JavaFX packaging tools can produce self-contained application packages that simplify redistribution by avoiding dependencies on external software. For more information about self-contained application packages, see [Chapter 6, "Self-Contained Application Packaging."](#)

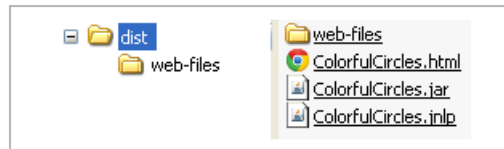
## 5.2 Base Application Package

The JavaFX application package that is generated by default includes:

- An executable application JAR file, which contains application code and resources and can be launched by double-clicking the file
- Additional application JAR and resource files
- A deployment descriptor for web deployment (kept in the JNLP file)
- An HTML file containing sample JavaScript code to embed and launch JavaFX content from a web page

[Figure 5–1](#) shows an example of the structure of a base application package. By default, NetBeans IDE will also include a copy of other support files in the web-files folder, but for production it is recommended that you use a public copy of the dtjava.js file, because it is always up to date.

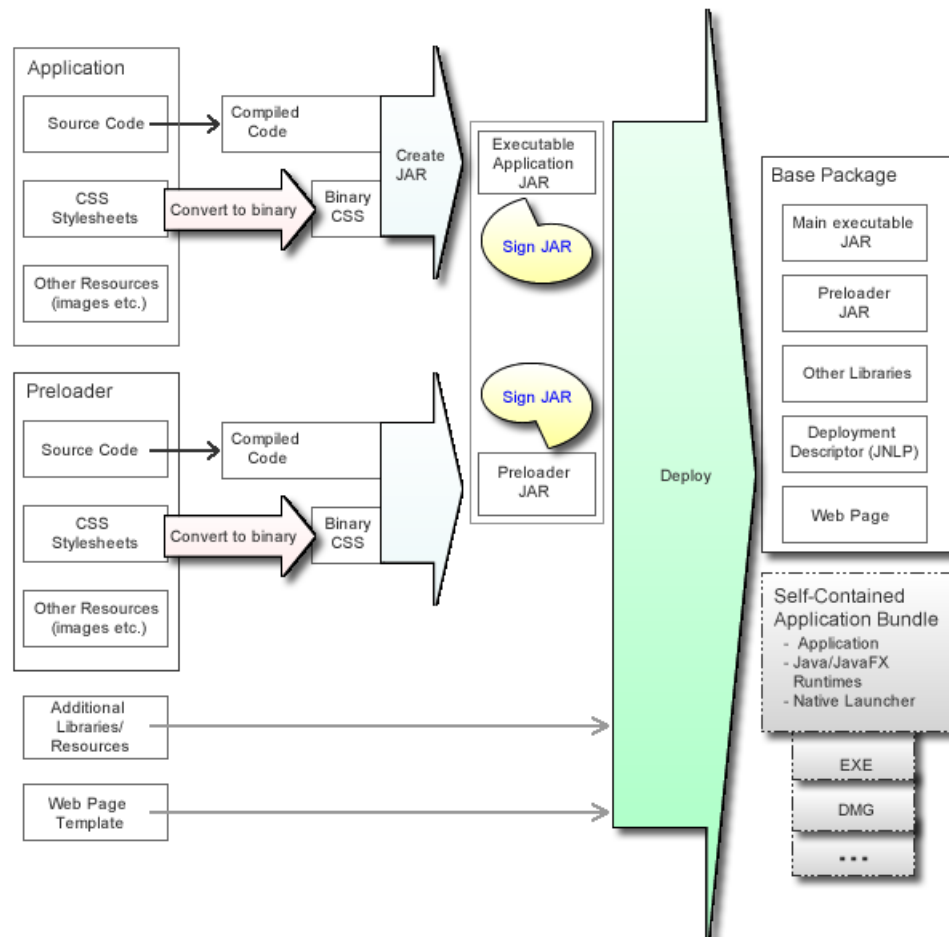
**Figure 5–1** Example of a Package for Web Deployment



## 5.3 Overview of Packaging Tasks

The build process for JavaFX applications extends the normal build process with several additional steps, as outlined in [Figure 5–2](#).

Figure 5–2 The Build Process for JavaFX Applications



New or modified steps are marked with colored arrows and described as follows:

- (Optional) Convert stylesheets to binary form  
Converts CSS files to binary form to reduce parsing overhead at application runtime.
- Create JAR  
Packages code and resources needed for the JavaFX application into a JAR file and embeds the utility classes to support autodetection of JavaFX Runtime, launch on double-click, and integration with the preloader JAR, if needed.  
See [Section 5.5, "Create the Main Application JAR File."](#)
- (Optional) Sign the JAR files  
Signing JAR files is needed only when the application requires elevated privileges, such as accessing files on the local file system or accessing nonsecure system properties. Signing is not a new concept, and you can sign the JAR files for your JavaFX application in the same way as you would for Swing/AWT applications.

JavaFX Runtime supports a new method to sign JAR files that reduces the JAR size overhead for signing, thereby improving the download time.

See [Section 5.6, "Sign the JAR Files."](#)

- Run the Deploy task

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. See [Section 5.7, "Run the Deploy Task or Command"](#) and [Chapter 6, "Self-Contained Application Packaging."](#)

### 5.3.1 JavaFX Packaging Tools

The recommended way to package JavaFX applications is to use a collection of Ant tasks (`ant-javafx.jar`), provided with the JavaFX SDK and also with JDK 7 Update 6 or later.

NetBeans IDE uses these Ant tasks to package JavaFX projects. Embedded packaging support in NetBeans IDE covers most of the typical use cases. However, if you need something special, you can always tune packaging by adding custom packaging hooks to the `build.xml` file (for example, as a `-post-jar` target).

Most of the other popular IDEs can easily use custom Ant build scripts. Other popular build frameworks, for example Maven or Gradle, support integration with Ant also.

The JavaFX SDK and JDK 7 Update 6 or later include a command-line packaging utility, `javafxpackager`, which can be used for simple packaging tasks. Note that `javafxpackager` is a convenience utility and does not provide as much flexibility or as many options as Ant tasks.

[Table 5–1](#) summarizes how to accomplish the build steps using the various packaging tools available. Note that `javafxpackager` also provides a `-makeall` macro command to create a complete application package for simple applications (for more information, see [Chapter 11, "The JavaFX Packager Tool."](#)).

**Table 5–1 JavaFX Packaging Tasks and Tools**

Task	JavaFX Packager Library Ant Task	JavaFX Packager Tool Command	NetBeans IDE
Convert any CSS files to binary format (See <a href="#">Section 5.4, "Stylesheet Conversion."</a> )	<code>&lt;fx:csstobin&gt;</code>	<code>javafxpackager -createbss</code>	Packaging category in Project Properties <ul style="list-style-type: none"> <li>■ <b>Select Binary Encode JavaFX CSS Files</b> check box.</li> </ul>
Create a JAR archive (See <a href="#">Section 5.5, "Create the Main Application JAR File."</a> )	<code>&lt;fx:jar&gt;</code>	<code>javafxpackager -createjar</code>	Occurs by default with a Build command, using the configuration in Project Properties.
Sign a JAR archive as one binary object (See <a href="#">Section 5.6, "Sign the JAR Files."</a> )	<code>&lt;fx:signjar&gt;</code>	<code>javafxpackager -signJar</code>	Deployment category in Project Properties <ul style="list-style-type: none"> <li>■ <b>Request unrestricted access</b> check box.</li> <li>■ To attach a certificate, click <b>Edit</b>.</li> </ul>

**Table 5–1 (Cont.) JavaFX Packaging Tasks and Tools**

Task	JavaFX Packager Library Ant Task	JavaFX Packager Tool Command	NetBeans IDE
Assemble application package for deployment (See <a href="#">Section 5.7, "Run the Deploy Task or Command"</a> ) and <a href="#">Chapter 6, "Self-Contained Application Packaging."</a>	<code>&lt;fx:deploy&gt;</code>	<code>javafxpackager -deploy</code>	Base application package is produced by default with a <code>Build</code> command. To produce self-contained applications, see <a href="#">Section 6.3.2, "Basic Build."</a>

## 5.4 Stylesheet Conversion

Converting stylesheets to binary format is optional but improves application performance. This is especially noticeable on larger CSS files.

To use a binary CSS file, refer to it instead of the CSS file, as shown in [Example 5–1](#):

### Example 5–1 Using a Binary Stylesheet

```
scene.getStylesheets().add(this.getClass().getResource
    ("mystyles.bss").toExternalForm());
```

Usage:

- Ant task: [Convert CSS Files to Binary](#)
- JavaFX Packager tool: `-createbss` command in the `javafxpackager` reference
- NetBeans IDE: In the Packaging category of Project Properties, select **Binary Encode JavaFX CSS Files**.

## 5.5 Create the Main Application JAR File

In addition to application classes and resources, you can provide the following information when you create the main application JAR file:

- Platform requirements
- Required versions of Java Runtime and JavaFX Runtime
- Any required Java VM arguments
- The following details about the application:
  - Name of the main application class (Required)
  - Name of preloader class (if there is a preloader)
  - Name of fallback class to use if the platform does not support JavaFX
- Details about application resources, if applicable
- Set of class files and other resources included in the JAR file
- List of auxiliary JAR files needed by the application, including a custom preloader JAR file (if needed)

The use of JavaFX packaging tools to create the main application JAR file is very important for packaging double-clickable jars and self-contained applications. The main application JAR file will include a launcher program that takes care of the bootstrap launch. This also improves launch by:

- Checking for the JavaFX Runtime
- Guiding the user through any necessary installations
- Setting the system proxy for your application

---

---

**Note:** If you have a preloader, as shown in [Figure 5-2](#), create a separate JAR file with the preloader resources, using any of the packaging tools listed below. For more information, see [Chapter 9](#), "Preloaders."

---

---

Usage:

- Ant task: [<fx:jar> Usage Examples](#)
- JavaFX Packager tool: `-createjar` command (see the [javafxpackager](#) reference)
- NetBeans IDE: Handled automatically when you specify this information in the project's properties.

## 5.6 Sign the JAR Files

Before adding code to sign your application, ensure that signing is needed, because it carries a cost of overhead to perform validation and often causes additional dialog boxes to be shown to the end user on application startup. See [Section 3.2.4, "Run in Sandbox Unless Signed and Trusted"](#) to find out when an application needs elevated permissions.

If you want to use traditional methods to sign JAR files, consult the Java Tutorial's steps for code signing and the description of the standard Ant `signjar` task for information about the traditional signing method.

JavaFX also provides a new signing method that helps to reduce the size overhead of signing the JAR file. In this method, you sign the JAR file as one large object instead of signing every JAR entry individually. This saves up to 10 percent of the total JAR size.

To use the new signing method provided by JavaFX, you need the keystore and signing key. See the Java Tutorial on generating keys for instructions.

Usage:

- Ant task: [<fx:signjar> Usage Examples](#)
- JavaFX Packager tool: `-signJar` command in the [javafxpackager](#) reference
- NetBeans IDE: Netbeans IDE users enable signing when they request elevated permissions for the application by selecting the **Request unrestricted access** check box in the project properties. To sign with a specific certificate, click **Edit** next to the check box.

---

**Note:** All JAR files must be signed or unsigned in the context of a single deployment descriptor file. If you need to mix signed and unsigned JAR files, use an additional `<fx:deploy>` Ant task to generate an additional deployment descriptor for each JAR file. These additional deployment descriptors are called extension descriptors. Use `<fx:resources>` to refer to the extension descriptors when the main descriptor is generated. For an example of how to do this, see [Using `<fx:resources>` for Extension Descriptors](#).

---

## 5.7 Run the Deploy Task or Command

A basic redistribution package consists of the following items:

- The main executable JAR file
- (Optional) A set of auxiliary JAR files, including a JAR file with preloader code
- A deployment descriptor, defining how to deploy the application
- Either a basic HTML file with sample code to embed the application into your own web page or a custom web page that is the result of preprocessing an HTML template

JavaFX packaging tools can also package the application as a self-contained application bundle. This is an opt-in scenario, and it is disabled by default. For more details, see [Chapter 6, "Self-Contained Application Packaging."](#)

To assemble the redistributable package, you can use one of the following ways:

- Ant task: `<fx:deploy>` [Task Usage Examples](#)
- JavaFX Packager tool: `-deploy` command in the [javafxpackager](#) reference
- NetBeans IDE: A redistributable package is created every time you build the project. Packaging options are set in the project's properties.

### 5.7.1 Configure the Deployment Descriptor

The key part of this task is providing information to fill the deployment descriptor for web deployment. This information includes:

- Entry points: the main application class, preloader class, and other details  
Defined as attributes of the `<fx:application>` tag.
- Parameters to be passed to the application  
Defined using `<fx:param>` and `<fx:htmlParam>` tags under `<fx:application>`.
- The preferred application stage size  
It is crucial to reserve the display area for embedded content.  
Width and height are defined using `width` and `height` attributes in the `<fx:deploy>` tag for Ant tasks, the `javafxpackager -deploy` command in the [javafxpackager](#) tool, or in the Run category of NetBeans project properties.
- A description of application to be used in any dialog boxes that the user sees during application startup  
Defined using the `<fx:info>` tag.

- Platform requirements, including required versions of Java and JavaFX Runtimes and JVM settings  
Defined using the `<fx:platform>` tag. For an example, see [<fx:platform> Parameter to Specify JVM Options](#).
- Desktop integration preferences of the application, such as adding a shortcut to the desktop or a reference to the Start menu.  
Defined using the optional `<fx:preferences>` tag. See [<fx:preferences> Usage Examples](#).
- Permissions needed to run the application.  
By default web applications run in the sandbox. To request elevated permissions, use the `<fx:permissions>` tag. Note that in order for permissions to be granted, application JAR files must be signed, and the user must trust the security certificate used for signing. If the application requests elevated permissions but requirements for the user granting permissions are not met, then the application will fail to launch.

## 5.7.2 Application Resources

Supported application resource files include:

- JAR files
- Native JAR files
- JNLP files
- Icons
- License files
- Data files

Every resource has additional metadata associated with it, such as operating system and architecture for which this resource is applicable, plus a priority preference defining the point in the application lifecycle at which this resource is needed. Careful use of metadata may have a significant impact of the application startup experience. For a list of supported values, see [Table 12-8](#).

All files in the resource set will be copied to the build output folder. However, not all of them are used in all execution modes, as described in the following paragraphs.

Regardless of execution mode, all regular JAR files from the resource set will be added to the application classpath.

Native JAR files and JNLP files are only used for web deployment. Additional JNLP files are typically used to refer to external JNLP extensions or if the application itself is packaged as a set of components. See [Using <fx:resources> for Extension Descriptors](#) in the JavaFX Ant Task Reference.

Native JAR files are used to deploy native libraries used by application. Each native JAR file can contain a set of native dynamic libraries and is inherently platform-specific. For more details, see [Example 5-11](#) and [Section 5.8.3, "Packaging Complex Applications."](#)

License files are currently applicable to self-contained applications only and are used to add a click-through license to installable packages. See [Section 6.4, "Installable Packages."](#)



Data files do not have special semantics, and applications are free to use them for anything. For example if your application needs to bundle a movie file, then you can mark it as "data," and it will be included into the application package.

For further details, see [Table 12-8](#).

### 5.7.3 Package Custom JavaScript Actions

The Deployment Toolkit provides a set of hooks that can be used to customize the startup behavior when an application is deployed in the browser. Developers must install a callback function to the hook, so it will be utilized by the Deployment Toolkit.

[Chapter 7, "Deployment in the Browser"](#) describes in detail what hooks are available and how to use them in the code. However, in order to ensure that they are correctly installed, they also must be specified at packaging time.

To specify callbacks, list them in the `<fx:callbacks>` tag under `<fx:deploy>`. Add an `<fx:callback>` entry for every callback you want to install and specify the name of the hook in the name attribute. The content of the `<fx:callback>` tag is the JavaScript function to be used. You can use a full function definition or refer to a function defined elsewhere.

Usage:

- Ant task: [<fx:callback> Usage Examples](#)
- JavaFX Packager tool: See `-deploy` command in the [javafxpackager](#) reference.
- NetBeans IDE: Add callbacks in the Deployment category of Project Properties. Click the **Edit** button to the right of Custom JavaScript Actions.

### 5.7.4 Web Page Templates

By default, JavaFX packaging tools generate a simple web page with a placeholder for the embedded application. You can manually copy code from this generated page to your real web page, but this is error prone and time consuming if you need to do this often.

JavaFX packaging tools also support injecting required code into an existing web page through the use of an input template. This is especially useful when the application is tightly integrated with the web page, for example if the application uses JavaScript to communicate to the web page, or if callbacks are used and their code is kept in the web page itself.

An input template is an HTML file containing markers to be replaced with JavaScript or HTML snippets needed to deploy the JavaFX application on the web page.

[Example 5-2](#) shows an example of an input template.

#### **Example 5-2 HTML Input Template**

```
<html>
  <head>
    <title>Host page for JavaFX Application</title>
    #DT.SCRIPT.CODE#
    #DT.EMBED.CODE.ONLOAD#
  </head>
  <body>
    <h1>Embed JavaFX application into existing page</h1>
    <!-- application will be inserted here -->
    <div id="ZZZ"></div>
  </body>
```

```
</html>
```

`#DT.SCRIPT.CODE#` and `#DT.EMBED.CODE.ONLOAD#` are markers that will be substituted with JavaScript and HTML code snippets when the template is processed. Markers have the form of `#MARKERNAME#` or `#MARKERNAME(id)#`,

*id* is the identifier of an application (specified using the `id` attribute of the `<fx:deploy>` tag if you are using Ant), and *MARKERNAME* is the type of the marker. If *id* is not specified, then *MARKER* matches any application. For a list of supported markers, see `<fx:template>` in the Ant task reference.

Templates can be used to deploy multiple applications into the same page. Use the full form of the marker including application ID (an alphanumeric string without spaces) and pass the partially processed template file when packaging each applications to insert.

[Example 5-3](#) shows an example of a template that is used to deploy multiple applications.

### **Example 5-3 An Input Template Used to Deploy Multiple Applications**

```
<html>
  <head>
    <title>Page with two application</title>
    <script src="#DT.SCRIPT.URL#"></script>

    <!-- code to load first app with id 'firstApp'
         (specified as attribute to fx:application) -->
    <!-- #DT.EMBED.CODE.ONLOAD(firstApp)# -->

    <!-- code to load first app with id 'secondApp' -->
    <!-- #DT.EMBED.CODE.ONLOAD(secondApp)# -->
  </head>
  <body>
    <h1>Multiple applications in the same page</h1>

    JavaFX app: <br>
    <!-- First app. Ant task need to use "ZZZ_1 as placeholderId -->
    <div id="ZZZ_1"></div>

    Another app: <br>
    <!-- Second app. Ant task need to use "ZZZ_2 as placeholderId -->
    <div id="ZZZ_2"></div>
  </body>
</html>
```

[Example 5-3](#) demonstrate one useful feature of the JavaFX template processor: the markers can be placed in the HTML comments. If the comment does not contain anything other than marker code, then the comment tags will be removed from the content in the resulting HTML. This keeps the HTML in the template page well formed.

Usage:

- Ant task: Add a template tag. See `<fx:template>` [Usage Examples](#).
- JavaFX Packager tool: `-deploy` command in the [javafxpackager](#) reference
- NetBeans IDE: Specify the input HTML template file in the Run category of Project Properties.

## 5.8 Packaging Cookbook

This sections presents several examples for popular deployment tasks.

The examples use Ant APIs, but in most cases the same result can be achieved using the JavaFX Packager tool. See [Chapter 12, "JavaFX Ant Tasks"](#) and [Chapter 11, "The JavaFX Packager Tool."](#)

### 5.8.1 Passing Parameters to the Application

JavaFX applications support two types of application parameters: named and unnamed (see the API for `Application.Parameters`).

Static named parameters can be added to the application package using `<fx:param>` and unnamed parameters can be added using `<fx:argument>`. They are applicable to all execution modes including standalone applications.

It is also possible to pass parameters to a JavaFX application from a Web page that hosts it, using `<fx:htmlParam>`. Prior to JavaFX 2.2, this was only supported for embedded applications. Starting from JavaFX 2.2, `<fx:htmlParam>` is applicable to Web Start applications also.

Passing parameters from the HTML page is the most useful if parameters are dynamic. To use this technique, we recommend the following approach:

- Use a web page template (see [Section 5.7.4, "Web Page Templates"](#)) to provide JavaScript code to prepare dynamic parameters.
- Pass a JavaScript snippet as a value for `<fx:htmlParam>` and specify `escape="false"`. Then it will be evaluated at runtime

[Example 5-4](#) shows the use of various parameter types:

#### **Example 5-4 Using Various Parameter Types**

```
<fx:application name="Test" mainClass="tests.Params">
<!-- unnamed parameters -->
  <fx:argument>Arg1</fx:argument>
  <fx:argument>Arg2 with spaces </fx:argument>

  <!-- name parameters -->
  <param name="sampleParam" value="Built with ${java.version}"/>
  <param name="noValueParam"/>

  <!-- parameters passed from HTML page -->
  <htmlParam name="staticParamFromWebPage"
    value="(new Date()).getTime()"/>
  <htmlParam name="dynamicParamFromWebPage" escape="false"
    value="(new Date()).getTime()"/>
</fx:application>
```

### 5.8.2 Customizing JVM Setup

Does your application need a larger heap size? Do you want to tune garbage collector behavior? Trace class loading?

You can specify required JVM options and set system properties using the `<fx:jvmarg>` and `<fx:property>` tags in your Ant task. These tags are applicable for all execution modes except standalone applications. In other words, you always get the default JVM if you double-click the JAR file, but you can tailor the JVM to your requirements if you

are running a self-contained application, a Web Start application, or an application embedded into a web page.

If you use any nonsecure JVM options or system properties, the application will need to have elevated permissions. A set of "secure" JVM command-line arguments and system properties is defined in the Java Web Start Developers' Guide.

In [Example 5-5](#), the Ant task will package the application so that, when the application is run, the JVM will be launched with the following arguments: `"-Xmx400 -verbose:jni -Dpurpose=sample"`.

Neither `"-verbose:jni"` nor `purpose` are secure, so elevated permissions are required for Web Start and embedded execution modes.

#### **Example 5-5 Specifying Custom JVM Options and Properties in the Ant Task**

```
<fx:platform javafx="2.1+">
  <fx:jvmarg value="-Xmx400m"/>
  <fx:jvmarg value="-verbose:jni"/>
  <fx:property name="purpose" value="sample"/>
</fx:platform>
```

### 5.8.2.1 Specifying User JVM Arguments

If you require a capability to specify user overridable jvm options, use the `<fx:jvmuserarg>` attribute in `<fx:platform>`. This attribute explicitly defines an attribute that can be overridden by the user.

---

**Note:** The user overridable arguments are implemented for Ant tasks only.

---

#### **Example 5-6 Specifying User Overridable Options**

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

In [Example 5-6](#), `-Xmx768m` is passed as a default value for heap size. The user can override this value in a user configuration file on linux and mac or in the registry on windows. The configuration file and the registry uses the conventions of the Java Preferences API for location and format.

The node for the applications user preferences is based on the application id (or if that hasn't been specified, the fully qualified main class), which is passed as `-Dapp.preferences.id` to the Application so it can provide a preferences UI if required. The application can access the jvm user options with node `-Dapp.preferences.id` and key `"JVMOptions"`.

The following examples provide code for overriding the JVM heap size value of 768m to 400m on different platforms.

#### **Example 5-7 Overriding Default Value on Mac**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>JVMUserOptions</key>
  <dict>
    <key>-Xmx</key>
```

```

        <string>400m</string>
    </dict>
</dict>
</plist>

```

#### **Example 5–8 Overriding Default Value on Windows in Registry**

```

Computer\HKEY_CURRENT_
USER\Software\JavaSoft\Prefs\com\companyx\appy\J/V/M/Options
    name: -Xmx
    type: REG_SZ
    data: 400m

```

#### **Example 5–9 Overriding Default Value on Linux**

```

~/ .java/.userPrefs/com/companyx/appy/JVMOptions/prefs.xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE map SYSTEM "http://java.sun.com/dtd/preferences.dtd">
<map MAP_XML_VERSION="1.0">
    <entry key="-Xmx" value="400m"/>
</map>

```

### **5.8.2.2 Macro Expansion of Application Directory for `jvmarg` and `jvmuserarg`**

You can provide string substitution to the root of the install directory for parameters passed into the application.

#### **Example 5–10 Substituting Parameters Passed to the Application**

```

<fx:platform>
    <fx:jvmarg value="-Djava.policy.file=$APPDIR/app/whatever.policy"/>
    <fx:jvmuserarg name="-Xmx" value="768m" />
</fx:platform>

```

## **5.8.3 Packaging Complex Applications**

Real-life applications often have more than just a single JAR artifact. They may have third-party libraries, data files, native code, and so on. For the complex application you may need special packaging tweaks to support different execution modes

There are too many possible scenarios to cover all cases, but here are some guidelines:

- Mark platform-specific resources accordingly.
- For the double-clickable JAR file, consider repackaging everything into a single giant JAR file and loading native libraries and data files from inside the JAR.
 

Alternatively, if you prefer to have multiple files:

  - Make sure all dependent JAR files are listed in the `<fx:resources>` tag in the `<fx:jar>` task that will create the main JAR file.
  - List all data files and libraries in filesets with `type="data"` to copy them into the output folder.
  - Load native libraries and resources from locations relative to the main JAR file.
  - See the example [<fx:jar> Ant Task for a Simple Application](#) in the Ant Task Reference chapter.
- For self-contained applications:

- Avoid packaging anything but the main JAR file using `<fx:jar>`. Multiple embedded launchers can confuse the native launcher.
- List all dependent JAR files in the `<fx:resources>` section of `<fx:deploy>` and `<fx:jar>` for the main application JAR file.
- Either use an explicit location relative to the application JAR file to load native libraries, or copy native libraries into the root application folder. (Use `type="data"` to copy native library files.)
- See [Example 6–5](#).
- For Web Start and embedded applications:
  - List all dependent JAR files in the `<fx:resources>` section of `<fx:deploy>`.
  - Native libraries should be wrapped into the JAR files for redistribution. Use one JAR file per platform. Ensure the JAR files contain native libraries only and that the libraries are all in the top-level folder of the JAR file.
  - See [Example 5–11](#).

**Example 5–11 Packaging Native Libraries into JAR Files**

```
<jar destfile="${basedir}/build/native-libs-win-x86.jar"
    basedir="native/windows/x86" includes="*" />
<jar destfile="${basedir}/build/native-libs-win-x86_64.jar"
    basedir="native/windows/x86_64" includes="*" />
....

<!-- sign all jar files --->
<signjar keystore="test.keystore" alias="TestAlias" storepass="xyz123">
    <fx:fileset dir="dist" includes="**/*.jar" />
</signjar>

<!-- assemble package -->
<fx:deploy width="600" height="400"
    outdir="${basedir}/${bundle.outdir}"
    outfile="Demo">
    <fx:info title="Demo app" />
    <fx:application name="${bundle.name}"
        mainClass="{javafx.main.class}" />
    <fx:permissions elevated="true" />
    <fx:resources>
        <!-- jar files with classes and shared data -->
        <fx:fileset dir="dist" includes="*.jar" />
        <fx:fileset dir="dist" includes="lib/*.jar" />

        <!-- add native libs for deployment descriptor -->
        <fx:fileset dir="build" type="native"
            os="windows" arch="x86">
            includes="native-libs-win-x86.jar" />
        </fx:fileset>
        <fx:fileset dir="build" type="native"
            os="windows" arch="x64"
            includes="native-libs-win-x86_64.jar" />
        </fx:fileset>
        ...
    </fx:resources>
</fx:deploy>
```

## 5.8.4 Publishing an Application that Fills the Browser Window

Prior to JavaFX 2.2, it was not easy to size embedded applications relative to the size of browser window. The `width` and `height` attributes of the `<fx:deploy>` task can only take numeric values, because they are used not just in the generated HTML/Javascript code but also in the deployment descriptor.

In JavaFX 2.2, two new attributes were added to the `<fx:deploy>` task: `embeddedWidth` and `embeddedHeight`. These attributes enable you to specify the size relative to the browser window (for example, as "50%").

These optional `embeddedWidth` and `embeddedHeight` attributes are only used for embedded applications, and only in the generated HTML/Javascript code. Also note that `width` and `height` values in pixels are still required.

To fill the browser window completely, you must set `embeddedWidth` and `embeddedHeight` to "100%". This alone does not produce a perfect result, because scrollbars will be added to the browser window, for the following reasons:

- The default HTML template has some other content.
- The default style of HTML tags may reserve space for things like margins.

The resulting web page will appear to be larger than the view area, and therefore the browser will add scrollbars.

The full solution consists of the following steps:

- Specify `embeddedWidth="100%"` and `embeddedHeight="100%"` in the `<fx:deploy>` task. (See [Example 5–12](#).)
- Add a custom web page template. (See [Example 5–13](#) and [Section 5.7.4, "Web Page Templates."](#))
- Reset the style of used HTML elements to ensure the application is the only element in the view area.

### Example 5–12 Packaging

```
<fx:deploy width="100" height="100"
           embeddedWidth="100%" embeddedHeight="100%"
           outdir="${basedir}/${dist.dir}" outfile="${application.title}">
  <fx:application name="${application.title}"
                 mainClass="${javafx.main.class}"/>
  <fx:template file="${basedir}/web/index_template.html"
              tofile="${dist.dir}/TestApp.html"/>
  <fx:resources>
    <fx:fileset dir="${basedir}/${dist.dir}" includes="*.jar"/>
  </fx:resources>
  <fx:info title="${application.title}"
          vendor="${application.vendor}"/>
</fx:deploy>
```

### Example 5–13 Web Page Template (web/index\_template.html)

```
<html>
  <head>
    <!-- This will be replaced with javascript code to embed the application -->
    <!--      #DT.SCRIPT.CODE#-->
    <!--      #DT.EMBED.CODE.ONLOAD#-->

    <!-- Reset html styles to ensure these elements do not waste space -->
    <style>
```

```
        html, body {
            margin: 0;
        }
    </style>
</head>
<body>
<!-- Application will be added to the div below -->
    <div id='javafx-app-placeholder'></div>
</body>
</html>
```

## 5.9 Performance Tuning for Web Deployment

There are several options that can be used to improve application launch time for the first and subsequent launches of web applications.

**Tip:** While you actively develop an application it is a good idea to disable optimizations to avoid unneeded complexity. Use them at the final packaging stage.

### 5.9.1 Background Update Check for the Application

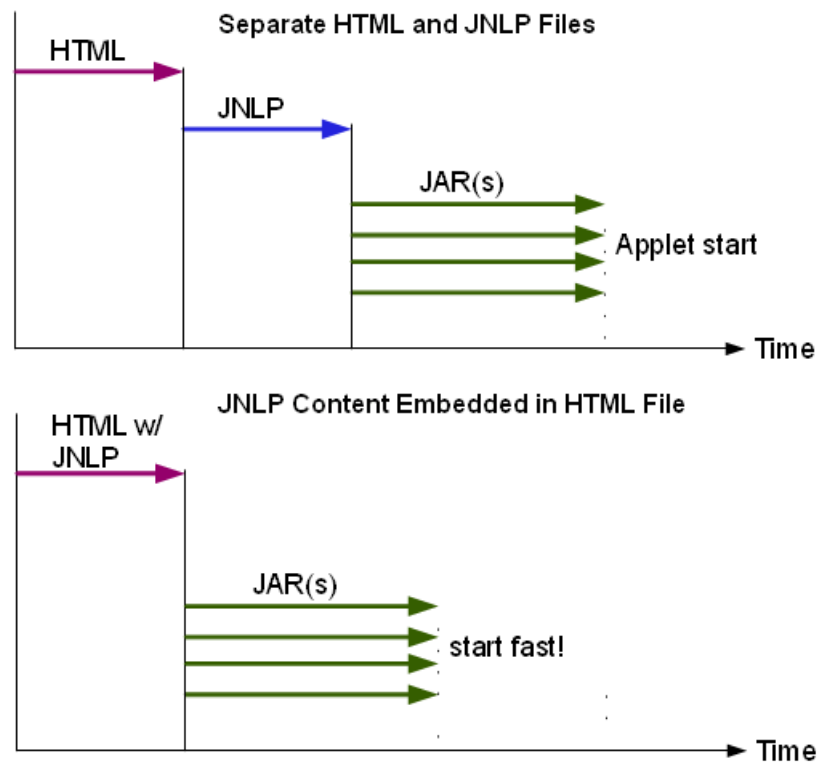
Every time a web application starts (Web Start or embedded), a background update check is conducted for whether updates are required. By default, JavaFX applications perform "lazy" update checks in the background while the application runs. This helps to avoid wait time to check for updates when the application starts. If updates are found, then they will be used only after the application restarts. To switch between update modes, use the following mechanisms in the JavaFX packaging tools:

- Ant task: `updatemode` attribute of the `<fx:deploy>` task.
- JavaFX Packager tool: `-updatemode` option of the `javafxpackager -deploy` command in the `javafxpackager` tool.
- NetBeans IDE: In the Deployment category of Project Properties, select **Check for Application Updates in Background**.

### 5.9.2 Embed the Deployment Descriptor into the Web Page

You can embed the content of the deployment descriptor into the HTML page, which helps to reduce the number of network connections needed to launch the application, as shown in [Figure 5-3](#).



**Figure 5-3 Embedding JNLP Content Reduces Network Connections**

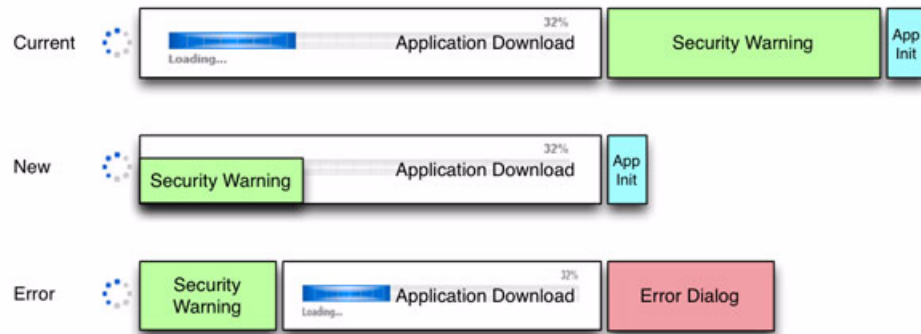
The original JNLP file will be loaded in the background to perform application update checks.

To embed the content of the deployment descriptor:

- Ant task: `embedjnlp` attribute of the `<fx:deploy>` task.
- JavaFX Packager tool: `-embedjnlp` option of the `javafxpackager -deploy` command in the `javafxpackager` tool.
- NetBeans IDE: The content is embedded by default.

### 5.9.3 Embed Signing Certificate into Deployment Descriptor

If the application is signed, then this option embeds into the JNLP file a copy of the details of the certificate used to sign the JAR files. If the user needs to approve permissions, this certificate, followed by a security prompt, is shown while the application JAR files are being loaded, as shown in [Figure 5-4](#).

**Figure 5–4 Advantage of Embedding the Certificate in the Deployment Descriptor**

To use this feature:

- Ant task: `cachecertificates` attribute of the `<fx:permissions>` task.
- JavaFX Packager tool: `-embedCertificates` option of the `javafxpackager -deploy` command in the [javafxpackager](#) tool.

#### 5.9.4 Use New JavaFX Signing Method (Signed Applications)

Reducing the JAR file size helps to reduce download time and improve startup time. See [Section 5.6, "Sign the JAR Files."](#)

---

# Self-Contained Application Packaging

A self-contained application is a wrapper for your JavaFX application, making it independent of what the user might have installed.

- [Section 6.1, "Introduction"](#)
- [Section 6.2, "Pros and Cons of Self-Contained Application Packages"](#)
- [Section 6.3, "Basics"](#)
- [Section 6.4, "Installable Packages"](#)
- [Section 6.5, "Working Through a Deployment Scenario"](#)

## 6.1 Introduction

JavaFX packaging tools provide built-in support for several formats of self-contained application packages. The basic package is simply a single folder on your hard drive that includes all application resources as well as Java Runtime. It 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 very similar to producing a basic JavaFX application package as discussed in [Chapter 5, "Packaging Basics,"](#) with the following differences:

- Self-contained application packages can only be built using JDK 7 Update 6 or later. (The standalone JavaFX SDK does not support self-contained applications.)
- Self-contained application packages must be explicitly requested by passing additional arguments to the `<fx:deploy>` Ant task or `javafxpackager` tool.
- Operating system and tool requirements must be met to be able to build a package in a specific format.

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.

## 6.2 Pros and Cons of Self-Contained Application Packages

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

Self-contained application packages have several benefits:

- They **resemble native applications for the target platform**, in that 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 Java Runtime used by the application is fully controlled by the application developer.
- Your application is easily deployed on fresh systems with **no requirement for Java Runtime to be installed**.
- Deployment occurs with **no need for admin permissions** when using ZIP or user-level installers.

On the other hand, there are a few caveats:

- "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 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 will be noticeably larger than the size of a standalone application, because a private copy of Java Runtime is included.
- Package per target platform  
Self-contained application packages are platform specific and can only be produced for the same system that you build on. If you want to deliver self-contained application packages on Windows, Linux and Mac you will have to 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 Java and JavaFX Runtimes to the latest secure version several times every year. There is no built-in support for this in self-contained applications.

## 6.3 Basics

Each self-contained application package includes the following:

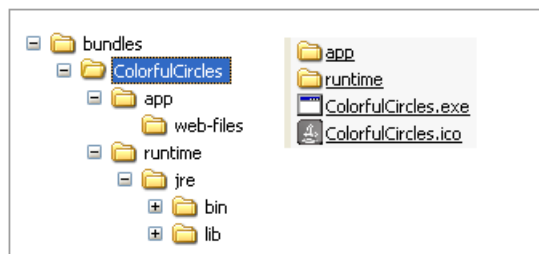
- The application code, packaged into a set of JAR files, plus any other application resources (data files, native libraries)
- A private copy of the Java and JavaFX Runtimes, to be used by this application only
- A native launcher for the application
- Metadata, such as icons

Multiple package formats are possible. Built-in support is provided for several types of packages, but you can 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.

### 6.3.1 Self-Contained Application Structure

The basic form of a self-contained application is a single folder on your hard drive, such as the example in [Figure 6–1](#). When any of the installable packages are installed, the result is a folder with the same content.

**Figure 6–1** Example of a Self-Contained Application Package



The internal structure of a self-contained application folder is platform-specific and may change in future. However, the following points are guaranteed:

- The application package, as defined in [Chapter 5, "Packaging Basics,"](#) is included as a folder, preserving the application directory structure.
- A copy of Java Runtime is included as another folder, and the Java Runtime 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:** Only a subset of Java Runtime is included by default. Some optional and rarely used files are excluded to reduce the package size, such as all executables. If you need something that is not included by default, then you need to 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 [Section 6.3.3, "Customization Using Drop-In Resources."](#)

---

### 6.3.2 Basic Build

The easiest way to produce a self-contained application is to modify the deployment task. To request creation of all applicable self-contained application packages simply add `nativeBundles="all"` to the `<fx:deploy>` task, as shown in [Example 6–1](#).

**Example 6–1** Simple Deployment Task to Create All Self-contained Application Packages

```
<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 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.

If you have a JavaFX project in Netbeans 7.2, you can add the above snippet as a post-build step by overriding the `"-post-jfx-deploy"` target, as shown in [Example 6-2](#). Add the following code to the `build.xml` file in the main project directory.

**Example 6-2 Custom build.xml Script to Create Self-contained Application Packages in NetBeans IDE**

```
<target name="-post-jfx-deploy">
    <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>
</target>
```

You can also produce native packages using the JavaFX Packager tool. self-contained application packages are built by default if you use the `-makeall` command, or you can request them explicitly using the `-native` option in the `-deploy` command. See the [javafxpackager](#) command reference.

[Example 6-3](#) 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 6-3 JavaFX Packager Command to Generate Self-Contained Application Packages**

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

### 6.3.3 Customization Using Drop-In Resources

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

For this you need to:

- Know what resources are used
- Drop custom resources into a location where the packaging tool will look for them

The following sections explain how to do this.

### 6.3.3.1 Preparing 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 `javafxpackager -deploy` command.

Verbose mode does the following:

- It prints the following items:
  - A list of config resources used for the package you are generating
  - The role of each resource
  - The expected custom resource name
- It saves a copy of all config files to the temp folder, so you can use and customize it.

[Example 6–4](#) shows sample output, with the important parts highlighted:

#### **Example 6–4 Sample Output in Verbose Mode**

```
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.fxbundle/macosx. Use them
to customize package.
```

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

Note: It is recommended that you disable verbose mode once you are done with customization or add a custom cleanup action to remove sample config files.

### 6.3.3.2 Substituting a Built-In Resource

Packaging tools look for customized resources on the classpath before reverting to built-in resource. The JavaFX Packager has `.` (the current working directory) added to the classpath by default. Hence, to replace the application icon, simply copy your custom icon to `./package/macosx/DemoApp.icns` in the directory where `javafxpackager` is run (typically, the root project directory).

The classpath for JavaFX 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 6–5](#) shows how to add `.` to the classpath. For a more detailed code snippet, see [Example 10–1](#).

#### **Example 6–5 Enabling Resource Customization for JavaFX Ant Tasks**

```
<taskdef resource="com/sun/javafx/tools/ant/antlib.xml"
uri="javafx:com.sun.javafx.tools.ant"
classpath=".:path/to/sdk/lib/ant-javafx.jar"/>
```

If you created a JavaFX project using Netbeans 7.2 or later, then the JavaFX Ant tasks are predefined, and `.` is already added to the classpath by default.

Once you provide a customized resource, verbose build output will report that it is used. For example, if you added a custom icon to an application, then the verbose output would report the addition, shown in [Example 6-6](#).

**Example 6-6 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
```

### 6.3.4 Customization Options

Many of the existing JavaFX Ant elements are used to customize self-contained application packages. Different sets or parameters are needed for different packages, and the same element might have different roles. [Table 6-1](#) introduces most of the customization options.

**Table 6-1 Customization Options with Ant Elements and Attributes**

Tag	Attribute	Details
<fx:application>	id	Identifier of application. Format is platform/package specific. If not specified, then a value will be generated.
	version	Application version. Default: 1.0.
	name	Short name of the application. Most bundlers use it to create the name of the output package. If not specified, then the name of the main class is used.
<fx:preferences>	shortcut	If set to true, then a desktop shortcut is requested.
	menu	If set to true then an entry in the applications menu is requested.
	install	If set to false, then a user-level installer is requested. Default behavior depends on the package format. See <a href="#">Table 6-2</a> .
<fx:fileset>	type	Defines the role of files in the process of assembling the self-contained application package. Resources of types <code>jnlp</code> and <code>native</code> are not used for building self-contained application packages. Resources of type <code>license</code> are used as a source of content for a click-through license or a license embedded into the package
<fx:info>	title	Application title.
	vendor	Application vendor.
	category	Application category. Category names are package-format specific.
	license	License type (for example, GPL). As of JavaFX 2.2, this attribute is used only for Linux bundles.
	copyright	Short copyright statement.
	description	Application description.



**Table 6–1 (Cont.) Customization Options with Ant Elements and Attributes**

Tag	Attribute	Details
<fx:jvmarg>	—	JVM arguments to be passed to JVM and used to run the application (for example, large heap size).
<fx:property>	—	Properties to be set in the JVM running the application.

### 6.3.5 Platform-Specific Customization for Basic Packages

Creation and customization of 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, and on Mac it is .icns. No icon is embedded into launcher on Linux.
- To ensure that the icon is set in runtime, you also need to add it to the application stage. For example, add the following code to to the `start()` method of your application:

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

- Consider signing files in output folder if you plan to redistribute them.  
For example, on Windows, the launcher executable can be signed using `signtool.exe`.

#### 6.3.5.1 Mac OS X

The resulting package on Mac OS X is an "application bundle" (or .app) in the Mac OS "dialect/jargon".

Several config parameters end up in the Info.plist file in the application bundle and need to conform to the rules:

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

Mac OS X 10.8 introduces Gatekeeper, which prevents execution of untrusted code by default (regardless of whether this code was 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, you need to obtain a Developer ID Certificate from Apple and sign the .app folder produced by JavaFX packaging tools, as follows:

```
% codesign -s "Developer ID Application" ExampleApp.app
```

For more details, see the Developer ID and Gatekeeper topic at the Apple Developer site.

## 6.4 Installable Packages

A self-contained application can be wrapped into a platform-specific installable package to simplify redistribution. JavaFX 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 that 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. On the other hand it assumes admin permissions and will likely result in additional steps 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 enables you to show as little dialogs as possible and run the program even if 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.

Think carefully if you really need this, because extra dialogs degrade the user experience.

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

The user should be able to launch your application easily. Therefore, we assume that 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 to install the package. Developers also have limited control of the installation location—they can only choose system or private user location).

If this is not sufficient for your needs you can try advanced customizations by tuning the config file templates (see [Section 6.3.3, "Customization Using Drop-In Resources"](#)) or packaging a basic self-contained application and then wrapping it into an installable package on your own.

As of JDK 7u6, the following installable package formats are supported:

**Table 6–2** *Installable Package Formats*

Package format	Installation Location (Default mode in bold)	Click-Through License	Prerequisites
EXE	<b>Per user:</b> %LOCALAPPDATA%	Yes (option)	■ Windows
	<b>System:</b> %ProgramFiles%		■ Inno Setup 5 or later
MSI	<b>Per user:</b> %LOCALAPPDATA%	No special support	■ Windows
	<b>System:</b> %ProgramFiles%		■ WiX 3.0 or later
DMG	<b>Per user:</b> user's desktop folder	Yes (option)	■ Mac OS X
	<b>System:</b> /Applications		

**Table 6–2 (Cont.) Installable Package Formats**

Package format	Installation Location (Default mode in bold)	Click-Through License	Prerequisites
RPM	Per user: unsupported System: /opt	No special support	<ul style="list-style-type: none"> <li>■ Linux</li> <li>■ RPMBuild</li> </ul>
DEB	Per user: unsupported System: /opt	No special support	<ul style="list-style-type: none"> <li>■ Linux</li> <li>■ Debian packaging tools</li> </ul>

## 6.4.1 EXE Package

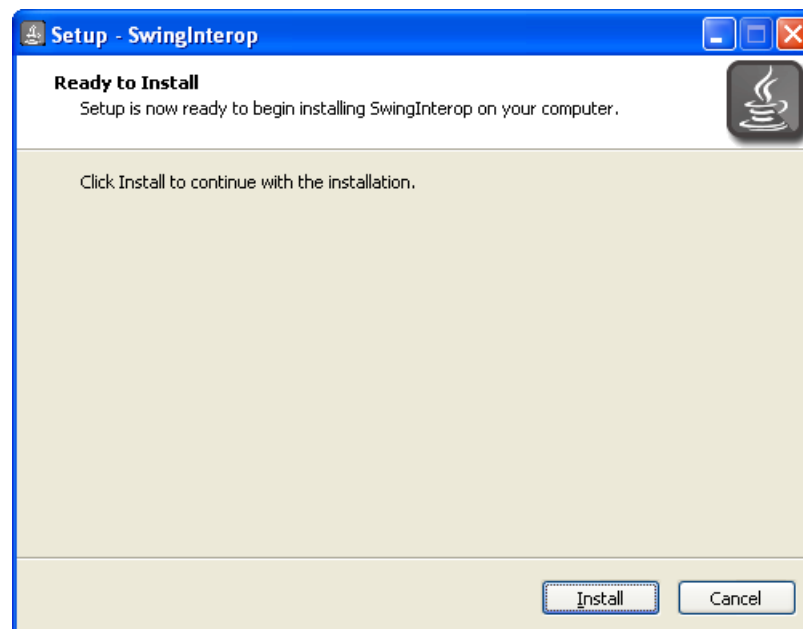
As of JavaFX 2.2, in order to generate an EXE package, you must have Inno Setup 5 or later installed and available on the `PATH`. To validate that it is available, try running `iscc.exe` from the command line where you launch the build or from your build script.

By default, the generated package:

- Does not require admin privileges to install
- Is optimized to have a minimum number of dialogs
- Must be referenced from programs menu or have desktop shortcut (having both are fine)
- Is configured so the application launches at the end of installation

Figure 6–2 shows a typical dialog box for a self-contained JavaFX application being installed on Windows.

**Figure 6–2 Windows Installation Dialog for a Self-Contained JavaFX Application**



Customization tips:

- If you chose system-wide installation, then the user will need to have admin permissions, and the application will not be launched at the end of installation.

- A click-through license is supported (an .rtf file is required).
- The image shown in the installation dialogs is different from the application icon. You can customize it using the "drop-in technique" described in [Section 6.3.3, "Customization Using Drop-In Resources."](#)

The current version of Inno Setup assumes the image is a bitmap file with maximum size of 55x58 pixels.
- To ensure the icon is set in the runtime, make sure to explicitly add it to the stage. See [Section 6.3.5, "Platform-Specific Customization for Basic Packages."](#)
- Consider signing the resulting .exe package. If you distribute an unsigned executable, then many versions of Windows will scare the user with an "Unknown Publisher" warning dialog.

If you sign your package, then this warning will be removed. You will need to get a certificate and then can use the signtool.exe utility to sign the code.
- You can fine tune the self-contained application folder before it is wrapped into an .exe file, for example to sign the launcher executable.

To do this, provide a Windows script file, using the technique from [Section 6.3.3, "Customization Using Drop-In Resources."](#)

---

---

**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.

---

---

## 6.4.2 MSI Package

MSI packages are generated using the Windows Installer XML (WiX) toolset (also known as WiX). As of JavaFX 2.2, WiX 3.0 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:

- Is optimized for deployment using enterprise deployment tools
- Installs to a system-wide location
- Does not have any click-through UI. Only a progress dialog is shown.
- Must to be referenced from the programs menu or have a desktop shortcut (having both is fine)
- Will remove all files in the installation folder, even if they were created outside of the installation process. (WiX 3.5 or later is required.)
- Will try to use the application identifier as `UpgradeCode`.

If the application identifier is not a valid GUID, then a random GUID for `UpgradeCode` is generated.
- `ProductCode` is randomly generated. To use a fixed Product code, customize the WiX template file using the technique from [Section 6.3.3, "Customization Using Drop-In Resources."](#)

If you plan to distribute your MSI package on the network, then consider signing 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). For details, see [Section 6.4.1, "EXE Package."](#)

To add a custom UI to the MSI package, you can customize WiX template file used by JavaFX Packager using technique from [Section 6.3.3, "Customization Using Drop-In Resources."](#) Consult WiX documentation for more details.

### 6.4.3 DMG Package

By default, a DMG package provides a simple drag-and-drop installation experience. [Figure 6-3](#) shows an example of the default behavior during installation.

**Figure 6-3** Example of Default Installer for Mac OS X



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 tweak sizes and positions of elements on the install view. For details about how to do this, see [Section 6.3.3, "Customization Using Drop-In Resources."](#)

To fine tune the self-contained application folder before it is wrapped, you can provide your own bash script to be executed after the application folder is populated. You can use it, for example, to enrich it with localization files, and so on. [Figure 6-4](#) shows an example of a "tuned" application installer.

**Figure 6–4** Example of Customized Appearance of Installable Package for Mac OS X

To create a Gatekeeper-friendly package (for Mac OS X 10.8 or later, see [Section 6.3.5.1, "Mac OS X"](#)), the application in the DMG package must be signed. It is not necessary to sign the DMG file itself. To sign the application, you can use a technique described in [Section 6.3.3, "Customization Using Drop-In Resources"](#) to provide a config script to be executed after the application bundle is populated. For sample DemoApp, the config script is located in the package/macosx/DemoApp-post-image.sh and has the content shown in [Example 6–7](#).

**Example 6–7** Example of Config Script to Sign the Application

```
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-through license provided in the text format. If use of rich text format is desired, then prepare the license.plist file externally, then add it to the package using the technique from [Section 6.3.3, "Customization Using Drop-In Resources."](#)

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

## 6.4.4 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:

- Will install the application to /opt
- Will add a shortcut to the application menu
- Does not have any UI for installation (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>`.

Refer to 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 from [Section 6.3.3, "Customization Using Drop-In Resources."](#)

Consult the DEB/RPM packaging guides to get more background on available options.

## 6.5 Working Through a Deployment Scenario

Consider following scenario. You have a JavaFX application that:

- 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

How do you package it as a self-contained application that does not need admin permissions to install?

It is assumed that your application works fine as a standalone, 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 6–8](#). The approach is:

- 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()` will work fine.

[Example 6–8](#) shows an example `<fx:deploy>` task.

### **Example 6–8 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:resources>
</fx:deploy>
```

```
<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>
```



---

---

## Deployment in the Browser

This page explains how to deploy JavaFX applications in the browser using the Deployment Toolkit. The chapter includes an API overview, information about the callback methods, and typical examples of use.

The recommended way to embed a JavaFX application into a web page or launch it from inside a web browser is to use the Deployment Toolkit library.

The Deployment Toolkit provides a JavaScript API to simplify web deployment of JavaFX applications and improve the end user experience with getting the application to start.

In addition to providing functionality to add HTML tags needed to insert JavaFX content into the web page, the Deployment Toolkit also does the following:

- Detects whether the user environment is supported
- Offers to install the JavaFX Runtime if needed
- Provides visual feedback to the user while the application is loaded
- Reports to the user in case of unexpected error
- Provides other helper APIs to the developer, which can be used to simplify deployment and integration with the web page

The recommended way to use the Deployment Toolkit is to import the Deployment Toolkit JavaScript file from a shared location at <http://java.com/js/dtjava.js>.

This way your application always uses the latest recommended way to integrate into a web page. If you cannot use the shared location, you can include the necessary files with your application by using the `includeDT` option in the `<fx:deploy>` Ant task.

Note that for the majority of simple use cases, most if not all of the code needed for deployment is generated by JavaFX packaging tools (see [Section 5.3.1, "JavaFX Packaging Tools"](#)), and you can use callbacks to tune default behavior further. However, if you need more flexibility, you can use the Deployment Toolkit APIs directly.

This page contains the following topics:

- [Section 7.1, "API Overview"](#)
- [Section 7.2, "Callbacks"](#)
- [Section 7.3, "Examples"](#)

### 7.1 API Overview

The Deployment Toolkit API provides several important methods:

- `dtjava.embed(app, platform, callbacks)`

Embeds the application into the browser based on a given application descriptor. If Java Runtime or JavaFX Runtime installation is required, by default it will guide the user to click to install it or click to redirect to the page to download the installer.
- `dtjava.launch(app, platform, callbacks)`

Launches applications that are deployed outside the browser, based on a given application descriptor. If Java or JavaFX Runtime installation is required, then by default it performs the following actions:

  - It attempts to start the Java or JavaFX Runtime installer.
  - If automated installation of the Runtime is not possible, it presents a popup request for the user to install manually.
- `dtjava.install(platform, callbacks)`

Initiates installation of required components according to platform requirements.
- `dtjava.validate(platform)`

Validates that the user environment satisfies platform requirements (for example, if a required version of JavaFX is available). Returns `PlatformMismatchEvent` describing problems, if any.
- `dtjava.hideSplash(id)`

Hides the HTML splash panel for an application with a given id. If the splash panel does not exist, this method has no effect. For JavaFX applications, this method is called automatically after the application is ready.

See the following sections for more information about arguments taken by these methods.

### 7.1.1 Application Descriptor (`dtjava.App`)

The application launch descriptor is an instance of the `dtjava.App` object. It contains all of the details describing what needs to be launched. The `dtjava.App` constructor takes the following two parameters:

```
dtjava.App(url, {attribute_map});
```

The first parameter contains the required `url` attribute, which contains a URL or a relative link to the JNLP file. The second parameter contains a subset of the attributes described in the following list.

- `id`

Identifier of the application. Expected to be unique on the web page. If null, then it is autogenerated. Later it can be used to get the handle of the application in JavaScript code.
- `jnlp_content`

(optional) Content of the JNLP file in BASE64 encoding.
- `params`

The set of named parameters to pass to an application, if any. This attribute takes the following form:

```
params: { variable: 'value' }
```

- `width` and `height`  
Dimensions used to reserve space for an embedded application. The values can be absolute (in pixels) or relative (for example, 50 or 50%).
- `placeholder`  
Reference to a DOM node, or an identifier of a DOM node, in which to embed an application.

---



---

**Note:** The `width`, `height`, and `placeholder` attributes are required for applications that run in the browser (in other words, applications that use the `dtjava.embed()` method, but not for other types of deployment (applications that use the `dtjava.launch()` method).

---



---

Example 7–1 shows an example of application launch descriptor code for an application that will run in the browser.

**Example 7–1 Creating a `dtjava.App` Object**

```
var app = new dtjava.App(
    'ssh.jnlp',
    {
        id: 'SSH',
        width: 100,
        height: 25,
        placeholder: 'SSH_application_container',
        params: {
            config: 'ssh_application.conf',
        }
    }
);
```

## 7.1.2 Platform (`dtjava.Platform`)

Platform requirements for an application to launch are defined by the `dtjava.Platform` object. The following attributes are supported:

- `javafx`  
The minimum JavaFX version needed. Default is `null`.
- `jvm`  
Minimum version of Java Runtime needed. Default is 1.6+.
- `jvmargs`  
List of requested JVM arguments. The default is `null`.

Version strings are treated according to the following rules:

- The null version string is treated as if there is no requirement to have it installed. Validation passes whether this component is installed or not.
- The version pattern strings are of the form `#[.#[.#[_#]]][+|*]`, which includes strings such as "1.6", "2.1\*", and "1.6.0\_18+".
- An asterisk (\*) means any version within this family, where family is defined by a prefix.

- A plus sign (+) means any version greater or equal to the specified version.
- If the version pattern does not include all four version components but does not end with an asterisk or plus sign, it will be treated as if it ended with an asterisk.

To clarify differences, consider the following examples:

- "1.6.0\*" matches 1.6.0\_25 but not 1.7.0\_01, whereas "1.6.0+" or "1.\*" matches both.
- "2.1" is equivalent to "2.1\*", and will match any version number beginning with "2.1".
- Both asterisk and plus sign patterns will match any installed version of the component. However, if the component is not installed, then validation fails.

[Example 7-2](#) shows a `dtjava.Platform` object:

**Example 7-2 Example of a `dtjava.Platform` Object**

```
var platform = new dtjava.Platform({
    javafx: "2.1+",
    jvmargs: "-Dapp.property= ? -Xmx1024m"
});
```

If platform requirements are not met, then `dtjava.PlatformMismatchEvent` is returned, either as a return value from the `validate()` method or passed to the `onDeployError` callback. This object provides a set of helper methods to identify root causes:

- `javafxStatus()`

Returns `ok` if the error was not due to a missing JavaFX Runtime. Otherwise, this method returns one of the following error codes characterizing the problem:

  - `none`  
No JavaFX runtime is detected on the system.
  - `old`  
A version of JavaFX Runtime is detected, but it does not match the platform requirements.
  - `disabled`  
A matching version of JavaFX Runtime is detected, but it is disabled.
  - `unsupported`  
JavaFX is not supported on this platform.
- `jreStatus()`

Returns `ok` if the error was not due to a missing JRE. Otherwise, this method returns one of the following error codes characterizing the problem:

  - `none`  
No JRE was detected on the system.
  - `old`  
A version of the JRE is detected, but it does not match the platform requirements.
  - `oldplugin`

A matching JRE was found, but it is configured to use a deprecated Java plug-in.

- `canAutoInstall()`  
Returns true if the installation of missing components can be triggered automatically or if there are no missing components.
- `isRelaunchNeeded()`  
Returns true if a browser must be restarted before the application can be loaded. This often is true in conjunction with the need to perform an installation.
- `isUnsupportedBrowser()`  
Returns true if the current web browser is not supported (for example, if it is out of date).
- `isUnsupportedPlatform()`  
Returns true if this platform (OS/hardware) does not satisfy the supported platform requirements. For example, a JavaFX 2.0 application attempts to launch on Solaris, which is not supported.
- `javafxInstallerURL (locale)`  
Returns the URL of a page to download and install the required version of JavaFX Runtime manually. If a matching JavaFX Runtime is already installed or not supported, then the return value is null.
- `jreInstallerURL (locale)`  
Returns the URL of a page to visit to install the required version of Java. If a matching Java Runtime is already installed or not supported, then the return value is null.

## 7.2 Callbacks

The Deployment Toolkit provides a set of hooks that can be used to customize startup behavior. To use the hook, the developer must provide a callback function. The following hooks are supported:

- `onDeployError`: `function(app, mismatchEvent)`  
Called when platform requirements are not met.
- `onInstallFinished`: `function(placeholder, component, status, relaunchNeeded)`  
Called after the installation of a required component is completed, unless installation was started manually.
- `onInstallNeeded`: `function(app, platform, cb, isAutoinstall, needRelaunch, launchFunc)`  
Called if embedding or launching an application needs additional components to be installed.
- `onInstallStarted` - `function(placeholder, component, isAuto, restartNeeded)`  
Called before installation of the required component is triggered.

The following hooks are specific to embedded applications:

- `onGetNoPluginMessage` `function(app)`

Called to get content to be shown in the application area if the Java plug-in is not installed and none of the callbacks have helped.

- `onGetSplash`: `function(app)`

For embedded applications, called to get the content of the splash panel.

- `onJavascriptReady`: `function(id)`

Called after the application is ready to accept JavaScript calls.

- `onRuntimeError`: `function(id)`

Called if the application failed to launch.

## 7.2.1 onDeployError

This handler can be utilized to customize error handling behavior, such as showing messages in the user's language. See the example in [Section 7.3.8, "Create a Handler for an Unsupported Platform"](#) for an example of the `onDeployError` handler.

A callback function is called if the application cannot be deployed because the current platform does not match the given platform requirements. It is also called if a request to install missing components cannot be completed due to platform incompatibility.

Function signature:

```
onDeployError : function(app, mismatchEvent)
```

The problem can be a fatal error or a transient issue, such as a required relaunch. Further details can be extracted from the `mismatchEvent` that is provided. Here are some typical combinations:

- The browser is not supported by Java.  
`mismatchEvent.isUnsupportedBrowser()` returns `true`
- The browser must be restarted before the application can be launched.  
`mismatchEvent.isRelaunchNeeded()` returns `true`

JRE-specific codes:

- JRE is not supported on this platform.  
`mismatchEvent.jreStatus() == "unsupported"`
- JRE is not detected and must be installed.  
`mismatchEvent.jreStatus() == "none"`
- The installed version of JRE does not match application requirements.  
`mismatchEvent.jreStatus() == "old"`
- A matching JRE is detected, but a deprecated Java plug-in is used.  
`mismatchEvent.jreStatus() == "oldplugin"`

JavaFX-specific codes:

- JavaFX is not supported on this platform.  
`mismatchEvent.javafxStatus() == "unsupported"`
- JavaFX Runtime is missing and must be installed manually.  
`mismatchEvent.javafxStatus() == "none"`
- The installed version of JavaFX Runtime does not match application requirements.

```
mismatchEvent.javafxStatus() == "old"
```

- JavaFX Runtime is installed but is disabled.

```
mismatchEvent.javafxStatus() == "disabled"
```

The default error handler handles both application launch errors and embedded content.

## 7.2.2 onGetNoPluginMessage

This handler is called to get content to be shown in the application area if the Java plug-in is not installed and none of the callbacks helped to resolve this.

Function signature:

```
onGetNoPluginMessage : function(app)
```

## 7.2.3 onGetSplash

This handler gets the content of the splash panel when the application is embedded in a browser.

Function signature:

```
onGetSplash: function(app)
```

Gets the application launch descriptor as input. If null is returned, then the splash panel is disabled. A non-null return value is expected to be an HTML snippet to be added into the splash overlay. Note that the splash overlay does not enforce any specific size. You must ensure that the custom splash image is sized to fit the area in which the application will run.

For examples of customizing the splash panel, see [Section 7.3.7, "Add a Custom HTML Splash Screen"](#) and [Section 7.3.6, "Disable the HTML Splash Screen."](#)

---



---

**Note:** Autohiding the splash panel is only supported by JavaFX applications. If you are deploying a Swing applet, then the application must call `dtjava.hideSplash()` explicitly to hide the splash panel. See [Example 10-6](#) on the Swing deployment page.

---



---

## 7.2.4 onInstallFinished

This handler is called after the installation of a required component is completed. This method will not be called if the installation is performed in manual mode.

Function signature:

```
onInstallFinished: function(placeholder, component, status,
relaunchNeeded)
```

Parameters:

- `placeholder`: A DOM element that was passed to `onInstallStarted` to insert visual feedback.
- `component`: String "jre" or "javafx"
- `status`: The status code is a string categorizing the status of the installation. ("success", "error:generic", "error:download" or "error:canceled")

- `relaunchNeeded`: Boolean value to specify whether a browser restart is required to complete the installation

### 7.2.5 `onInstallNeeded`

This handler is called if the embedding or launching application needs additional components to be installed. This callback is responsible for handling situations such as reporting to the user the need to install something, initiating installation using `install()`, and hiding the splash panel for embedded applications (if needed). After installation is complete, the callback implementation may retry the attempt to launch the application using the provided launch function.

This method is not called if the platform requirement could not be met (for example, if the platform is not supported or if installation is not possible).

The default handler provides a click to install solution for applications embedded in a browser and attempts to perform installation without additional questions for applications that have been started using `launch()`.

If the handler is null, then it is treated as a no-op handler.

Function signature:

```
onInstallNeeded: function(app, platform, cb, isAutoinstall, needRelaunch, launchFunc)
```

Parameters:

- `app`: Application launch descriptor. For embedded applications, `app.placeholder` refers to the root of the embedded application area in the DOM tree (to be used for visual feedback).
- `platform`: Application platform requirements.
- `cb`: The set of callbacks to be used during the installation process.
- `isAutoinstall`: True if the installation can be launched automatically.
- `needRestart`: True if restarting the browser is required after installation is complete.
- `launchFunction`: Function to be executed to retry launching the application after the installation is finished.

### 7.2.6 `onInstallStarted`

This handler is called before the installation of the required component is triggered. For a manual installation scenario, it is called before the installation page is opened.

This hook can be used to provide visual feedback to the user during the installation. The placeholder points to the area that can be used for visualization. For embedded applications it will be the area into which the application will be embedded. If null, then the callee must find a place for display on its own.

In case of automatic launch of the installation, `onInstallFinished` will be called after installation is complete (successfully or not).

If the handler is null, then it is treated as no-op handler.

Function signature:

```
onInstallStarted: function(placeholder, component, isAuto, restartNeeded)
```

Parameters:



- `placeholder`: The DOM element to insert visual feedback into. If null, then the callee must add visual feedback to the document on its own (for example, embedding the application into a web page).
- `component`: String "Java", "JavaFX", or "Java bundle".
- `isAutoInstall`: True if the installer will be launched automatically.
- `restartNeeded`: Boolean value to specify whether a browser restart is required.

### 7.2.7 onJavascriptReady

This handler is called after the embedded application is ready to accept JavaScript calls.

Function signature:

```
onJavascriptReady: function(id)
```

### 7.2.8 onRuntimeError

This handler is called if the embedded application fails to launch.

Function signature:

```
onRuntimeError: function(id)
```

## 7.3 Examples

This section shows examples of deployment that use the Deployment Toolkit.

---



---

**Note:** Some of the Deployment Toolkit methods may not be fully operational if used before the web page body is loaded, because the Deployment Toolkit plug-ins cannot be instantiated. If you intend to use the Deployment Toolkit before the web page DOM tree is constructed, then `dtjava.js` must be loaded inside the `<body>` element of the page and called before any other Deployment Toolkit APIs.

---



---

The recommended way to initiate deployment is to use `onload` handlers to add the application after the HTML content is loaded, for the following reasons:

- The DOM tree must exist when the application content is inserted. The DOM tree is created after the HTML content page loads.
- The Deployment Toolkit's detection code does not prevent loading of the HTML page.
- It avoids loading concurrently with other resources that are needed for the page.

### 7.3.1 Embedded Application Starts After the DOM Tree Is Constructed

[Example 7-3](#) shows the default deployment scenario for applications embedded in a web page. The application is added to the web page in the `deployIt()` function. This function is called after the page is loaded and its DOM tree is constructed. The position of the application in the DOM tree is determined by the value of the `placeholder` parameter. In this case, the application will be inserted into a `<div>` tag with `id="place"`.

Adding the application to a web page after the main page is loaded helps to get a complete web page (including the embedded application) to load faster, because the application starts concurrently with loading other resources required by the web page.

Note that the value of the placeholder element can be a string or a JavaScript variable pointing to the container DOM node. If that node is detached from the document DOM tree, then the application is not initialized until the parent node is attached to the DOM tree.

**Example 7-3 Insert the Application into the HTML Body with a Placeholder**

```
<head>
  <script type="text/javascript" src="http://java.com/js/dtjava.js"></script>
  <script>
    function deployIt() {
      dtjava.embed(
        {
          id: "myApp",
          url: "Fish.jnlp",
          width: 300,
          height: 200,
          placeholder: "place"
        },
        { javafx: "2.1+" },
        {}
      );
    }
    dtjava.addOnloadCallback(deployIt);
  </script>
</head>
<body>
  <div id="place"></div>
</body>
```

## 7.3.2 Launch a Web Start Application from a Web Page

The canonical form for the link to launch a Web Start application is shown in [Example 7-4](#). Either relative or absolute links are acceptable.

**Example 7-4 Canonical Link to Launch a Web Start Application**

```
<a href="my.jnlp"
  onclick="dtjava.launch(new dtjava.App('my.jnlp'));
  return false;">Launch me!</a>
```

An alternative simplified syntax is shown in [Example 7-5](#).

**Example 7-5 Link to Launch a Web Start Application, Simplified**

```
<a href="my.jnlp"
  onclick="dtjava.launch({url: 'my.jnlp'});
  return false;">Launch me!</a>
```

A third form for simple applications is shown in the following example.

**Example 7-6 Link to Launch a Web Start Application, Third Example**

```
<a href="my.jnlp"
  onclick="dtjava.launch('my.jnlp');
  return false;">Launch me!</a>
```

If JavaScript is disabled in the browser, the Deployment Toolkit does not work. In this case, the browser tries to use the href attribute. If JavaFX Runtime is installed, then the application will still launch. The `return false;` statement ensures that the browser will not leave the current page if the launch is successful.

### 7.3.3 Pass Parameters to a Web Application

You can pass dynamic parameters to JavaFX applications from a web page. For this you need to add a set of named parameters as the `params` attribute of the application descriptor.

In [Example 7-7](#), the embedded application gets two dynamic parameters from the web page. Note that the "zip" parameter is assigned with the value of the JavaScript variable.

#### **Example 7-7 Pass Parameters to an Embedded Application**

```
<head>
  <script type="text/javascript" src="http://java.com/js/dtjava.js"></script>
  <script>
    function deployIt() {
      var zipcode = 95054;

      dtjava.embed(
        {
          id: "myApp",
          url: "Map.jnlp",
          width: 300,
          height: 200,
          placeholder: "place",
          params: {
            mode: "streetview",
            zip: zipcode
          }
        },
        { javafx: "2.1+" },
        {}
      );
    }
    dtjava.addOnloadCallback(deployIt);
  </script>
</head>
<body>
  <div id="place"></div>
</body>
```

The same approach works for Web Start applications if the user has JavaFX Runtime 2.2 or later. Here is an example:

#### **Example 7-8 Pass Parameters to a Web Start Application**

```
<script>
  var zipcode = 95054;

  function launchApp() {
    dtjava.launch(
      {
        url: 'my.jnlp',
        params: {
          mode: "streetview",
          zip: zipcode
        }
      }
    );
  }
  dtjava.addOnloadCallback(launchApp);
</script>
```

```

        }
    },
    {   javafx : '2.2+'   },
    {}
    );
    return false;
}
</script>

<a href="my.jnlp" onclick="launchApp(); return false;">
    Launch me!
</a>

```

To access parameters in the application code, use the `getParameters()` method of the `Application` class. For example:

```
String zipcode = app.getParameters().getNamed("zip");
```

### 7.3.4 Specify Platform Requirements and Pass JVM Options

Use the second argument group of the `dtjava.embed()` function to specify platform requirements for the application.

In [Example 7-9](#), JRE 1.6 or later is specified in the `jvm` parameter; JavaFX 2.1 or later is specified in the `javafx` parameter; and the `jvmargs` parameter indicates that the application should be executed in the Java Virtual Machine (JVM) with a 1 gigabyte heap size and given options.

#### **Example 7-9** Specify Platform Requirements and Passing JVM Properties

```

<head>
  <script type="text/javascript" src="http://java.com/js/dtjava.js"></script>
  <script>
    function deployIt() {
      dtjava.embed(
        {
          id: "my",
          url: "app.jnlp",
          width: 300,
          height: 200,
          placeholder: "place"
        },
        {
          jvm: "1.6.0+",
          javafx: "2.1+",
          jvmargs: "-Dapp.property=somevalue -Xmx1024m"
        }
      );
    }
    dtjava.addOnloadCallback(deployIt);
  </script>
</head>
<body>
  <div id="place"></div>
</body>

```

### 7.3.5 Access JavaFX Code from JavaScript

To access a Java or JavaFX application from JavaScript, you must get a reference to the JavaScript object representing the application. The best way to do this is to specify an

`id` parameter in the first argument group of the `dtjava.embed()` function, as shown in [Example 7-3](#)

For example, if an `id` parameter is set to `my`, then a public method of the application can be accessed with the script shown in [Example 7-10](#).

**Example 7-10 Access an Application Method with an `id` Parameter**

```
<script>
    var a = document.getElementById("my");
    a.java_method();
</script>
```

Attempts to call application methods may not work until the application finishes the initialization phase. (For a description of startup phases, see [Application Startup Process, Experience, and Customization](#).) To access an application while it is loading, use an `onJavascriptReady` callback, as shown in [Example 7-11](#).

**Example 7-11 Access an Application Method While the Application Is Loading**

```
<head>
    <script type="text/javascript" src="http://java.com/js/dtjava.js"></script>
    <script>
        function callApp(id) {
            //it is safe to call now
            var a = document.getElementById(id);
            a.java_method();
        }
        function deployIt() {
            dtjava.embed(
                {
                    id: "my",
                    url: "fxapp.jnlp",
                    width: 300,
                    height: 200,
                    placeholder: "place"
                },
                {},
                {
                    onJavascriptReady: callApp
                }
            );
        }
        dtjava.addOnloadCallback(deployIt);
    </script>
</head>
<body>
    <div id="place"></div>
</body>
```

### 7.3.6 Disable the HTML Splash Screen

To disable the HTML splash screen for a JavaFX application, add an `onGetSplash` handler that returns `null`; as shown in [Example 7-12](#). For information about application startup phases, see [Section 4.1, "Application Startup Process, Experience, and Customization."](#)

**Example 7–12 Disable the HTML Splash Screen**

```

<head>
  <script type="text/javascript" src="http://java.com/js/dtjava.js"></script>
  <script>
    function deployIt() {
      dtjava.embed(
        {
          id: "my",
          url: "app.jnlp",
          width: 300,
          height: 200,
          placeholder: "place"
        },
        {
          jvm: "1.6.0+",
          javafx: "2.1+",
        },
        {
          onGetSplash: function(app) {return null;}
        }
      );
    }
    dtjava.addOnloadCallback(deployIt);
  </script>
</head>
<body>
  <div id="place"></div>
</body>

```

**7.3.7 Add a Custom HTML Splash Screen**

[Example 7–13](#) shows how to replace the default HTML splash screen with a green rectangle, defined in a JavaScript function.

**Example 7–13 Replace the Splash Screen with a Custom One**

```

<head>
  <script type="text/javascript" src="http://java.com/js/dtjava.js"></script>
  <script>
    function getSplash(app) {
      //custom splash - green rectangle
      var p = document.createElement('div');
      p.style.width = app.width;
      p.style.height = app.height;
      p.style.background="green";
      return p;
    }

    function deployIt() {
      dtjava.embed(
        {
          id: "my",
          url: "app.jnlp",
          width: 300,
          height: 200,
          placeholder: "place"
        },
        {
          jvm: "1.6.0+",
          javafx: "2.1+",
        },

```

```

        {
            onGetSplash: getSplash
        }
    );
}
dtjava.addOnloadCallback(deployIt);
</script>
</head>
<body>
    <div id="place"></div>
</body>

```

### 7.3.8 Create a Handler for an Unsupported Platform

[Example 7–14](#) shows the use of JavaScript to handle an unsupported browser.

#### **Example 7–14** Handle an Unsupported Browser with JavaScript

```

<head>
<script type="text/javascript" src="http://java.com/js/dtjava.js"></script>
<script>
    function reportError(app, r) {
        //overwrite behavior for unsupported browser
        var a = app.placeholder;
        if (a != null && r.isUnsupportedBrowser()) {
            var p = document.createElement('div');
            p.id = "splash";
            p.style.width = app.width;
            p.style.height = app.height;
            p.style.background="red";
            p.appendChild(
                document.createTextNode("This browser is not supported.");
            );

            //clear embedded application placeholder
            while(a.hasChildNodes()) a.removeChild(a.firstChild);

            //show custom message
            a.appendChild(p);
        } else {
            //use default handlers otherwise
            var def = new dtjava.Callbacks();
            return def.onDeployError(app, r);
        }
    }

    function deployIt() {
        dtjava.embed(
            {
                id: "my",
                url: "app.jnlp",
                width: 300,
                height: 200
            },
            {
                jvm: "1.6.0+",
                javafx: "2.1+",
            },
            {
                onDeployError: reportError
            }
        );
    }

```

```

        );
    }
    dtjava.addOnloadCallback(deployIt);
</script>
</head>
<body>
    <div id="place"></div>
</body>

```

### 7.3.9 Check for Presence of JavaFX Runtime

[Example 7–15](#) shows the use of JavaScript to replace the default text offering to install JavaFX. It is replaced with an offer to start the application only if the correct version of JavaFX is found on the user's system.

#### **Example 7–15** *Launch Only If JavaFX Is Installed*

```

<html>
  <head>
    <script type="text/javascript"
      src="http://java.com/js/dtjava.js"></script>
    <script>
      function maybeOfferLaunch() {
        var platform = new dtjava.Platform({'javafx' : '2.1+'});

        //check if validate find any problems
        if (dtjava.validate(platform) == null) {
          var t = document.getElementById("text");
          t.innerHTML = "<a href='my.jnlp' " +
            "onclick='dtjava.launch({url: 'my.jnlp'})'; return false;'" +
            "Launch me!</a>";
        }
      }
      dtjava.addOnloadCallback(maybeOfferLaunch);
    </script>
  </head>
  <body>
    <div id="text">To view this content you need
    <a href="http://javafx.com">to install
      JavaFX Runtime 2.1</a>.</div>
  </body>
</html>

```



---

---

# JavaFX and JavaScript

This chapter shows how JavaFX applications can be accessed from JavaScript and vice versa.

A JavaFX application can communicate with the web page in which it is embedded by using a JavaScript engine. The host web page can also communicate to embedded JavaFX applications using JavaScript.

---

---

**Note:** To a large extent, this functionality is based on the Java-to-JavaScript communication bridge that is implemented in the Java plug-in. Therefore most of the available documentation and examples for Java applets are also applicable to JavaFX applications. For more information about the Java implementation, see the Java LiveConnect documentation.

---

---

This page contains the following sections.

- [Section 8.1, "Accessing a JavaFX Application from a Web Page"](#)
- [Section 8.2, "Accessing the Host Web Page from an Embedded JavaFX Application"](#)
- [Section 8.3, "Advanced topics"](#)
- [Section 8.4, "Threading"](#)
- [Section 8.5, "Security"](#)
- [Section 8.6, "Tab Pane Example"](#)

## 8.1 Accessing a JavaFX Application from a Web Page

To access a JavaFX application from JavaScript, the first step is to get a reference to a JavaScript object representing the JavaFX application. The easiest way to get the reference is to use a standard JavaScript `getElementById()` function, using the identifier that was specified in the `id` attribute of the Ant `<fx:deploy>`, as shown in [Example 8-1](#).

**Example 8-1 Use JavaScript to Access an Application Object ID**

```
var fxapp = document.getElementById("myMapApp")
```

The result corresponds to the main class of the JavaFX application.

By getting the reference to a JavaScript object, you can use JavaScript code to access any public methods and fields of a Java object by referencing them as fields of the

corresponding JavaScript object. After you have the `fxapp` reference, you can do something similar to the following:

```
var r = fxapp.doSomething()
```

The implementation of the `doSomething()` method in Java code returns a Java object. The variable `r` becomes a reference to the Java object. You can then use code such as `r.doSomethingElse()` or `fxapp.doSomethingWithR(r)`.

You can access static fields or invoke static methods for classes loaded by a given application, by using a synthetic `Packages` keyword attached to the application object. You can use the same approach to create new instances of Java objects. For example, [Example 8-2](#) contains Java code, and [Example 8-3](#) contains JavaScript that interacts with that code. Look at both examples to see how they work together.

#### **Example 8-2 Java Code Example**

```
package testapp;

public class MapApp extends Application {
    public static int ZOOM_STREET = 10;

    public static class City {
        public City(String name) {...}
        ...
    }

    public int currentZipCode;

    public void navigateTo(City location, int zoomLevel) {...}
    ....
}
```

The JavaScript snippet in [Example 8-3](#) passes several values to the Java code in [Example 8-2](#). Before these values are used in the Java code, they are automatically converted to the closest Java type.

#### **Example 8-3 JavaScript Code for Example 8-2**

```
function navigateTo(cityName) {
    //Assumes that the Ant task uses "myMapApp" as id for this application
    var mapApp = document.getElementById("myMapApp");
    if (mapApp != null) {
        //City is nested class. Therefore classname uses $ char
        var city = new mapApp.Packages.testapp.MapApp$City(cityName);
        mapApp.navigateTo(city, mapApp.Packages.testapp.MapApp.ZOOM_STREET);
        return mapApp.currentZipCode;
    }
    return "unknown";
}
window.alert("Area zip: " + navigateTo("San Francisco"));
```

The JavaScript string, numeric, and Boolean objects can be converted into most of the Java primitive types—Boolean, byte, char, short, int, long, float, and double—and `java.lang.String`.

For JavaScript objects representing Java objects (in other words, objects that have previously been returned from Java), conversion results in extracting a reference to that Java object.

Conversion into one and multidimensional arrays is supported according to rules similar to rules for conversion of individual objects. If conversion cannot be performed successfully, then the JavaScript engine raises an exception.

All Java objects returned to the web browser are associated with a particular JavaFX application instance. References held by the JavaScript engine to a Java objects act as persistent references, preventing that Java object from being garbage-collected in the hosting JVM. However, if a particular application is destroyed, for example by leaving the web page hosting the application or by detaching the application from the HTML DOM tree, then references are immediately invalidated and further attempts to use those object in JavaScript will raise exceptions.

For more information about data type conversion and object lifetimes, see

[http://jdk6.java.net/plugin2/liveconnect/#JS\\_JAVA\\_CONVERSIONS](http://jdk6.java.net/plugin2/liveconnect/#JS_JAVA_CONVERSIONS)

---

**Note:** If a Java object has overloaded methods, in other words if it has multiple methods with the same name, but different sets of argument types, then the heuristic will be adopted of using the method with the closest types. For information, see the Java LiveConnect documentation.

The general recommendation is to avoid overloaded methods if you plan to use them from JavaScript code.

---

## 8.2 Accessing the Host Web Page from an Embedded JavaFX Application

JavaFX applications can call the following JavaScript components:

- Functions
- The get, set, and remove fields of JavaScript objects
- The get and set elements of JavaScript arrays

JavaFX applications can also evaluate snippets of JavaScript code. Through the JavaScript DOM APIs, JavaFX applications can modify the web page dynamically by adding, removing and moving HTML elements.

To bootstrap JavaFX-to-JavaScript communication, the JavaFX application must get a reference to the JavaScript window object containing the application. This reference can be used for subsequent operations such as evaluation, function calls, and fetches of variables.

Both the main and preloader application can get this reference by accessing the `HostServices` class in the JavaFX API and requesting `getWebContext()`, as shown in [Example 8-4](#).

### **Example 8-4 Access the HostServices Class from JavaFX Code**

```
public class MyApp extends Application {
    private void communicateToHostPage() {
        JSObject jsWin = getHostServices().getWebContext();
        //null for non-embedded applications
        if (jsWin != null) {
            //use js
            ...
        }
    }
    ...
}
```

```
}
```

All instances of JavaScript objects, including references to the DOM window, appear within Java code as instances of `netscape.javascript.JSObject`.

[Example 8-5](#) shows how to use JavaScript to implement a function to resize an embedded application with `id='myMapApp'` at runtime.

**Example 8-5 Use JavaScript to Resize an Application in the Browser**

```
public void resizeMyself(int w, int h) {
    JSObject jsWin = getHostServices().getWebContext();
    if (jsWin != null) {
        jsWin.eval("var m = document.getElementById('myMapApp');" +
            "m.width=" + w + "; m.height=" + h + ";");
    } else {
        // running as non embedded => use Stage's setWidth()/setHeight()
    }
}
```

## 8.3 Advanced topics

JavaFX applications embedded in a web page can call JavaScript methods in a web page after the `init()` method is called for the preloader or main application class.

JavaScript can access JavaFX applications at any time, but if the application is not ready yet, then this request may be blocked until the application is ready. Specifically, this will happen if the `init()` method of the main application class has not finished yet and the main application did not perform calls to the web page itself. A JavaScript call from the preloader does not fully unblock JavaScript-to-Java communication.

Most browsers use single-threaded JavaScript engines. This means that when blocking occurs, the host web page and the browser appear to be frozen.

To access a JavaFX application from the host web page early and avoid blocking, either notify the web page when the application is ready by calling a Java function from the application, or use an `onJavaScriptReady` callback in the Ant task.

[Example 8-6](#) shows an HTML template for an Ant task that uses an `onJavaScriptReady` callback to call the `doSomething()` method in the main application without blocking the browser.

**Example 8-6 HTML Input Template for an Ant Task**

```
<html>
  <head>
    <!-- template: code to load DT JavaScript will be inserted here -->
    #DT.SCRIPT.CODE#
    <!-- template: code to insert application on page load will be
    inserted here -->
    #DT.EMBED.CODE.ONLOAD#

    <script>
      function earlyCallFunction(id) {
        //it is safe to call application now
        var a = document.getElementById(id);
        if (a != null) a.doSomething();
      }
    </script>
  </head>
</body>
```

```

        <!-- application is inserted here -->
        <div id="ZZZ"></div>
    </body>
</html>

```

[Example 8-7](#) shows the relevant part of the Ant task used to generate an HTML page from the template in [Example 8-6](#). For this example, it is assumed that the template has the path `src/web/test_template.html`.

**Example 8-7 Ant `<fx:deploy>` Task to Generate an HTML Page from a Template**

```

<fx:deploy placeholderId="ZZZ" ...>
    ....
    <fx:template file="src/web/test_template.html"
        tofile="dist/test.html"/>
    <fx:callbacks>
        <fx:callback name="onJavascriptReady">earlyCallFunction</fx:callback>
    </fx:callbacks>
</fx:deploy>

```

## 8.4 Threading

Java code called from JavaScript is executed on a special thread that is not the JavaFX application thread. Use the `Platform.runLater()` method in the JavaFX code to ensure that something is executed on the JavaFX application thread.

In general, return as quickly as possible from functions that are called from JavaScript. In most modern browsers, JavaScript engines are single-threaded. If the call sticks, then the web page can appear frozen, and the browser will be unresponsive. In particular, it is recommended that you avoid writing code to wait for work to be done on a JavaFX application thread. If JavaScript code depends on the result of this work, then it is recommended that you use a callback from Java to notify the JavaScript code of the result of the execution of the work.

[Example 8-8](#) shows an example of code to avoid in JavaScript.

**Example 8-8 Naive implementation Blocking JavaScript Thread**

```

function process(r) {
    window.alert("Result: "+r);
}

var result = myApp.doSomethingLong();
process(result);

```

[Example 8-9](#) shows a better pattern to follow in JavaScript code.

**Example 8-9 A Better Implementation of [Example 8-8](#)**

```

function process(r) {
    window.alert("Result: "+r);
}

myApp.doSomethingLong(function(r) {process(r)});

```

[Example 8-10](#) shows a better example in Java code.

**Example 8–10 Java Code Using a Callback**

```

public void doSomethingLong(JSObject callback) {
    Object result;
    //do whatever is needed to get result

    //Invoke callback
    // callback is a function object, and every function object
    // has a "call" method
    Object f[] = new Object[2];
    f[0] = null; //first element is object instance but this is global function
                //not applying it to any specific object
    f[1] = new String(result); //real argument
    callback.call("call", f);
}

```

Java code can call JavaScript from any thread, including the JavaFX application thread. However, if the JavaScript engine in the browser is busy, then a call to JavaScript may stick for some time. If there is a call on the JavaFX application thread, then it may make your application appear frozen, because it will not be able to update the screen and handle user events. It is recommended that you offload execution of LiveConnect calls from the JavaFX application thread.

## 8.5 Security

JavaScript code on the web page can always make JavaScript-to-Java calls against an application on the page, and it can access all public methods and fields of Java classes loaded by the application. However, when a JavaScript-to-Java call is made, it is treated as called from the sandbox environment. Moreover, if the HTML document and the application originate from different sites, then JavaScript on the web page cannot cause any network connections to be made on its behalf.

Aside from this restriction, calling Java from JavaScript does not have any other consequences if the application is running in the sandbox. However, if the application is signed and trusted and therefore can request elevated permissions, then a call to a Java method from JavaScript is executed in the sandbox without elevated permissions. If elevated permissions are needed, then `AccessController.doPrivileged` in the Java API can be used to request them in the trusted code.

Developers should be careful not to expose APIs in their applications that would accidentally confer additional privileges on untrusted JavaScript code. Developers who must grant elevated privileges to JavaScript code are encouraged to serve their applications over verifiable HTTPS connections, and perform checks to ensure that the document base of the web page hosting the application is the same as the expected origin of the application's code.

## 8.6 Tab Pane Example

This section contains a sample that demonstrates how to use communication between JavaFX and JavaScript to integrate JavaFX web applications with the browser.

[Example 8–11](#) shows a JavaFX application that creates a tab pane on a web page, with 20 tabs.

**Example 8–11 Create Tabs on the Embedding Web Page**

```

public class TabbedApp extends Application {
    Group root = new Group();
    TabPane tabPane = new TabPane();
}

```

```

public void init() {
    // Prepare tab pane with set of tabs
    BorderPane borderPane = new BorderPane();
    tabPane.setPrefSize(400, 400);
    tabPane.setSide(Side.TOP);
    tabPane.setTabClosingPolicy(TabPane.TabClosingPolicy.UNAVAILABLE);

    for(int i=1; i<=20; i++) {
        final Tab t = new Tab("T" + i);
        t.setId(""+i);
        Text text = new Text("Tab "+i);
        text.setFont(new Font(100));
        BorderPane p = new BorderPane();
        p.setCenter(text);
        t.setContent(p);
        tabPane.getTabs().add(t);
    }
    borderPane.setCenter(tabPane);
    root.getChildren().add(borderPane);
}

@Override
public void start(Stage primaryStage) throws Exception {
    primaryStage.setScene(new Scene(root));
    primaryStage.show();
}
}

```

This application can be further improved to save the history of visited tabs into the browser history. This enables users to click the Back and Forward buttons in the browser to move between tabs.

The implementation is based on the `onhashchange` event introduced in HTML 5 and described at

<http://www.whatwg.org/specs/web-apps/current-work/#event-hashchange>

The JavaScript technique used by AJAX applications to achieve a similar effect is to save a reference to the current selection in the hash part of the document URL. When the user clicks the Back button, the URL is updated, and a selection state can be extracted that must be restored.

To implement this solution, two new methods are added to the sample: `onNavigate()` and `navigateTo()`. The `onNavigate()` method is called whenever a new tab is selected. It delivers information about the new selection to the web page by calling the JavaScript method `navigateTo()` and passing the tab ID to it. The JavaScript code saves the tab ID in the URL hash.

The `navigateTo()` method is responsible for reverse synchronization. After the web page URL is changed, this method is called with the ID of the tab to be selected.

**Example 8–12** shows the updated code of the application. The code that is different from **Example 8–11** appears in bold.

#### **Example 8–12 Improved Application that Saves Tab History**

```

public class TabbedApp extends Application {
    Group root = new Group();
    TabPane tabPane = new TabPane();

```

```

public void init() {
    // Prepare tab pane with set of tabs
    BorderPane borderPane = new BorderPane();
    tabPane.setPrefSize(400, 400);
    tabPane.setSide(Side.TOP);
    tabPane.setTabClosingPolicy(TabPane.TabClosingPolicy.UNAVAILABLE);

    for(int i=1; i<=20; i++) {
        final Tab t = new Tab("T" + i);
        t.setId(""+i);
        Text text = new Text("Tab "+i);
        text.setFont(new Font(100));
        BorderPane p = new BorderPane();
        p.setCenter(text);
        t.setContent(p);

        // When tab is selected, notify web page to save this in the
        // browser history
        t.selectedProperty().addListener(new ChangeListener<Boolean>() {
            public void changed(ObservableValue<? extends Boolean> ov,
                Boolean tOld, Boolean tNew) {
                if (Boolean.TRUE.equals((tNew)) {
                    onNavigate(t.getId());
                }
            }
        });
        tabPane.getTabs().add(t);
    }
    borderPane.setCenter(tabPane);
    root.getChildren().add(borderPane);
}

@Override
public void start(Stage primaryStage) throws Exception {
    primaryStage.setScene(new Scene(root));
    primaryStage.show();
}

public void navigateTo(String tab) {
    for (Tab t: tabPane.getTabs()) {
        if (tab.equals("#"+t.getId())) {
            tabPane.getSelectionModel().select(t);
            return;
        }
    }
}

private void onNavigate(String tab) {
    JSONObject jsWin = getHostServices().getWebContext();
    // Null for nonembedded applications
    if (jsWin != null) {
        //use js
        jsWin.eval("navigateTo('" + tab + "')");
    }
}
}

```

Part of the implementation logic is in the HTML page. [Example 8–13](#) shows a page that is used as an input template in an Ant script. When the Ant script is run, it inserts



code to embed the JavaFX application next to the custom JavaScript code. For more information about input templates, see [<fx:template>](#).

The implementation of JavaScript functions is straightforward. The `onhashchange` attribute of the `<body>` tag is used to subscribe to notifications of updates of the hash part of the URL. After the event is obtained, the JavaFX application is embedded in the web page, and the `navigateTo()` method is called.

If the application calls with an update on the selected tab, it is saved to the hash part of the URL.

### **Example 8–13 HTML Template Used as Input to the Ant Script**

```
<html>
  <head>
    <!-- template: code to load DT javascript will be inserted here -->
    #DT.SCRIPT.CODE#
    <!-- template: code to insert application on page load will be
         inserted here -->
    #DT.EMBED.CODE.ONLOAD#

    <script>
      function hashchanged(event) {
        var a = document.getElementById('tabbedApp');
        if (a != null) {
          try {
            a.navigateTo(location.hash);
          } catch (err) {
            alert("JS Exception: " + err);
          }
        }
      }

      function navigateTo(newtab) {
        if (window.location.hash != newtab) {
          window.location.hash = newtab;
        }
      }
    </script>
  </head>
  <body onhashchange="hashchanged(event)">
    <h2>Test page</h2>
    <!-- Application will be inserted here -->
    <div id='javafx-app-placeholder'></div>
  </body>
</html>
```

For completeness, [Example 8–14](#) shows the Ant script used to deploy this sample. The application is created with the ID `tabbedApp`. The JavaScript code uses this ID to find the application on the page, and the HTML template uses it to embed the application into the custom HTML page that is produced by the Ant task.

### **Example 8–14 Ant Script to Package the Application**

```
<fx:application id="tabbedApp"
  name="Example of browser integration"
  mainClass="docsamples.TabbedApp"/>

<fx:jar destfile="dist/docsamples/tabbedapp.jar">
  <fx:application refid="tabbedApp"/>
```

```
<fileset refid="appclasses"/>
</fx:jar>

<fx:deploy width="400" height="400"
  outdir="dist-web"
  outfile="BrowserIntegrationApp">
  <fx:info title="Doc sample"/>
  <fx:application refid="tabbedApp"/>
  <fx:template
    file="src/template/TabbedApp_template.html"
    tofile="dist-web/TabbedApp.html"/>
  <fx:resources>
    <fx:fileset requiredFor="startup" dir="dist/docsamples">
      <include name="tabbedapp.jar"/>
    </fx:fileset>
  </fx:resources>
</fx:deploy>
```

This chapter explains preloaders in JavaFX.

During the second phase of startup, a preloader application runs, either the default application in the JavaFX Runtime or a custom application that you supply. See [Section 4.1, "Application Startup Process, Experience, and Customization"](#) for information about how a preloader fits into the startup flow.

A custom preloader application is optional and can be used to tune the application loading and startup experience. For example, users tend to get irritated if they have to wait for an application to start or if they do not get status messages. Use of a preloader can help to reduce perceived application startup time by showing some content to the user earlier, such as a progress indicator or login prompt.

A preloader application can also be used to present custom messaging to the user. For example, you can explain what is currently happening and what the user will be asked to do next, such as grant permissions to the application, or you could create a preloader to present custom error messaging.

Not every application needs a preloader. For example, if the size of your application is small and does not have special requirements such as permissions, then it probably starts quickly. Even for larger applications, the default preloader included with the JavaFX Runtime can be a good choice, because it is loaded from the client machine rather than the network.

This page contains the following topics:

- [Section 9.1, "Implementing a Custom Preloader"](#)
- [Section 9.2, "Packaging an Application with a Preloader"](#)
- [Section 9.3, "Preloader Code Examples"](#)
- [Section 9.4, "Performance Tips"](#)

See [Section 4.1, "Application Startup Process, Experience, and Customization"](#) for information about how to customize the default preloader.

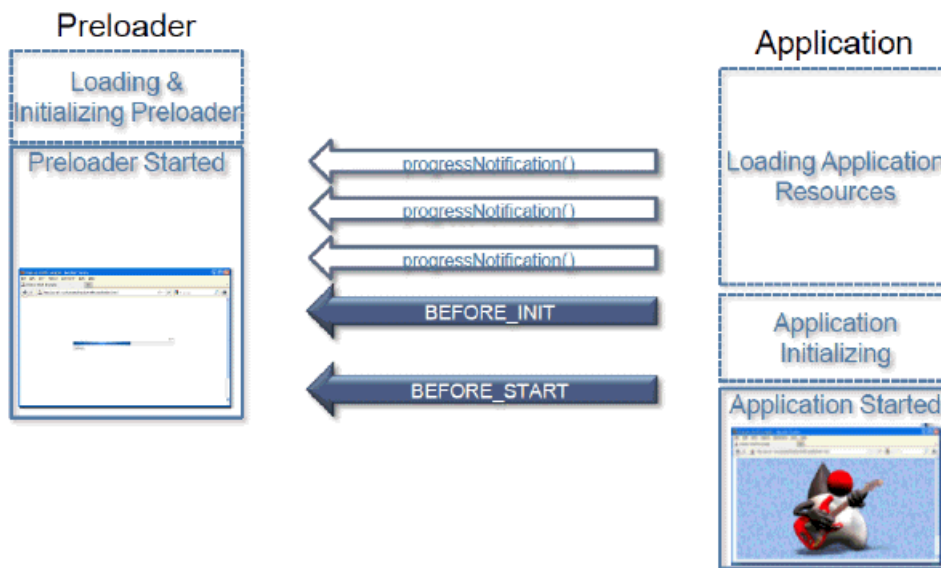
## 9.1 Implementing a Custom Preloader

A custom preloader is a specialized JavaFX application extending the `javafx.application.Preloader` class. Because the `Preloader` class is an extension of `javafx.application.Application`, a custom preloader has the same lifecycle and can use all of the features of the JavaFX Runtime.

The preloader startup sequence is shown in relation to the application startup in [Figure 9-1](#). The preloader application is started before the main application and gets

notification of the progress of the loading application resources, application initialization, and startup, as well as of errors.

**Figure 9–1 Preloader Startup Related to Application Startup**



**Example 9–1** shows a simple preloader that uses the `ProgressBar` control to visualize the loading progress.

**Example 9–1 Simple Preloader Using the `ProgressBar` Control**

```
public class FirstPreloader extends Preloader {
    ProgressBar bar;
    Stage stage;

    private Scene createPreloaderScene() {
        bar = new ProgressBar();
        BorderPane p = new BorderPane();
        p.setCenter(bar);
        return new Scene(p, 300, 150);
    }

    public void start(Stage stage) throws Exception {
        this.stage = stage;
        stage.setScene(createPreloaderScene());
        stage.show();
    }

    @Override
    public void handleProgressNotification(ProgressNotification pn) {
        bar.setProgress(pn.getProgress());
    }

    @Override
    public void handleStateChangeNotification(StateChangeNotification evt) {
        if (evt.getType() == StateChangeNotification.Type.BEFORE_START) {
            stage.hide();
        }
    }
}
```

As a regular JavaFX application, the `FirstPreloader` class uses the `start()` method to create a scene to display the loading progress. Updates on progress are delivered to the preloader using the `handleProgressNotification()` method, and the `FirstPreloader` implementation uses them to update the UI.

The preloader and main application have different Stage objects, and the preloader needs to take care of showing and hiding its own stage when needed. In [Example 9–1](#), the preloader stage is hidden after notification is received that the `start()` method of the main application is about to be called.

The implementation of the `FirstPreloader` class illustrates the main concept and will work in many scenarios, but it does not provide the best user experience for all use cases. See [Section 9.3, "Preloader Code Examples"](#) for examples of how to improve it further.

## 9.2 Packaging an Application with a Preloader

There are some special requirements for packaging applications with preloaders.

First, in most cases, the code for the preloader must be packaged into one or more JAR files that are separate from the rest of application. This enables faster loading when the application is deployed on the web. Using a single JAR file for both application and preloader code can be a good choice for some specialized cases, for example if the application is run in standalone mode only. In NetBeans IDE, the JAR files are packaged separately by creating two projects: one for the main application and a special JavaFX preloader project for the preloader. See [Section 9.2.1, "Packaging a Preloader Application in NetBeans IDE."](#)

Second, application deployment descriptors should include information about which class belongs to the preloader and where the preloader code is. The way to specify it depends on what tools you use for packaging. For more information about tools, see [Section 5.3.1, "JavaFX Packaging Tools."](#)

All of the packaging tools produce a deployment descriptor that includes the preloader, as in [Example 9–2](#). In this example, the main application is called `AnimatedCircles` and the preloader application is called `FirstPreloader`.

### **Example 9–2** Sample Deployment Descriptor for an Application with a Preloader

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0" xmlns:jfx="http://javafx.com" href="AnimatedCircles.jnlp">
  <information>
    <title>AnimatedCircles</title>
    <vendor>Oracle</vendor>
    <description>Animated Circles</description>
    <offline-allowed/>
  </information>
  <resources os="Windows">
    <jfx:javafx-runtime version="2.1+"
      href="http://javadl.sun.com/webapps/download/GetFile/
        javafx-latest/windows-i586/javafx2.jnlp" />
  </resources>
  <resources>
    <j2se version="1.6+" href="http://java.sun.com/products/autodl/j2se"/>
    <jar href="lib/FirstPreloader.jar" size="2801" download="progress" />
    <jar href="AnimatedCircles.jar" size="13729" download="always" />
  </resources>
  <applet-desc width="800" height="600"
    main-class="com.javafx.main.NoJavaFXFallback" name="AnimatedCircles" />
</jnlp>
```

```
<jfx:javafx-desc width="800" height="600"
  main-class="animatedcircles.AnimatedCircles" name="AnimatedCircles"
  preloader-class="firstpreloader.FirstPreloader" />
<update check="background" />
</jnlp>
```

The manifest must also contain the classpath to the preloader, shown in [Example 9-3](#).

### **Example 9-3 Sample Manifest for an Application with a Preloader**

```
Manifest-Version: 1.0
JavaFX-Version: 2.1
implementation-vendor: nhildebr
implementation-title: AnimatedCircles
implementation-version: 1.0
JavaFX-Preloader-Class: firstpreloader.FirstPreloader
JavaFX-Application-Class: animatedcircles.AnimatedCircles
JavaFX-Class-Path: lib/FirstPreloader.jar
JavaFX-Fallback-Class: com.javafx.main.NoJavaFXFallback
Created-By: JavaFX Packager
Main-Class: com/javafx/main/Main
```

## **9.2.1 Packaging a Preloader Application in NetBeans IDE**

If you are using NetBeans IDE, in the main application you can specify either another NetBeans project that contains the main preloader class or a JAR file in which the preloader was packaged.

The following procedures show two ways to package a preloader in NetBeans IDE, depending on your project configuration. You can either create a new NetBeans project and choose a preloader option, or you can add a preloader to an existing NetBeans project. Both procedures use the preloader class from [Example 9-1](#).

### **To create a new application with a preloader in NetBeans IDE:**

1. On the File menu, choose **New Project**.
2. Select the **JavaFX** category and **JavaFX Application** as the project type. Click **Next**.
3. Enter **FirstApp** as a project name and choose **Create Custom Preloader**. Click **Finish**.

Netbeans IDE creates two new projects for you: a **FirstApp-Preloader** project with basic implementation of a custom preloader, and **FirstApp** project with a sample JavaFX Application using your custom preloader.

4. Open the `SimplePreloader` class in Source Packages in the **FirstApp-Preloader** project.
5. Replace the implementation of the `SimplePreloader` class with the implementation of the `FirstPreloader` class, or any other sample from this page.

Be sure to fix imports if needed by going to the Source menu and choosing **Fix Imports**.

6. Select the **FirstApp** project and run **Clean and Build** to build both the sample application and the preloader.

The artifacts are placed in the `dist` folder in the **FirstApp** project.

7. Test the artifacts by running them in Netbeans.

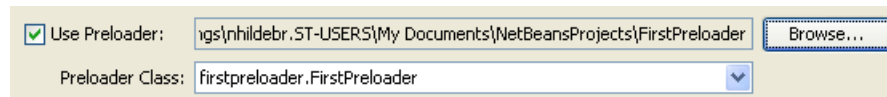
**Tip:** You can launch your application as standalone or in a browser by choosing a Run category in Project Properties, or you can directly open the build artifacts.

Note that for standalone launch, the preloader may be not visible if it displays loading progress only, because there is nothing to load. Even when testing web launch from a local hard drive, the preloader might show up for a very short time.

**To add a preloader to an existing NetBeans project:**

1. Create a separate NetBeans project of type JavaFX Preloader for the preloader class. In the example, the project name is FirstPreloader, which contains the firstpreloader package and the code for the FirstPreloader class.
2. In the Project Properties for the main application, click the **Run** category.
3. Select the check box **Use Preloader**, then Click **Browse**, then choose the NetBeans project for the preloader. The Preloader Class field is populated by default, as shown in [Figure 9-2](#).

**Figure 9-2** Preloader Option in the Run Category of NetBeans Project Properties



**Note:** As an alternative to selecting a NetBeans project for the preloader, when you click Browse you have the option of selecting a preloader JAR file.

4. Click **OK** to close the Project Properties dialog box.
5. Right-click the main application and choose **Clean and Build**.

The main application files are created for deployment in the dist directory, and the preloader JAR file is placed in a lib subdirectory. All of the necessary JNLP and manifest entries are handled by the IDE.

## 9.2.2 Packaging a Preloader Application in an Ant Task

Ant users must specify information about the preloader class and JAR files in both the `<fx:jar>` and `<fx:deploy>` tasks. Setting the proper parameters in the `<fx:jar>` task ensures that the preloader is registered for standalone applications. Setting the proper parameters in the `<fx:deploy>` task creates the configuration for web deployment.

Some settings are required in other parts of the Ant script. The preloader main class is specified as part of the `<fx:application>` element, as shown in [Example 9-4](#).

**Example 9-4** Specify the Preloader Class in `<fx:application>`

```
<fx:application id="app-desc"
  mainClass="sample.AppSample"
  preloaderClass="preloaders.SamplePreloader" />
```

Preloader resources are marked with the `requiredFor="preloader"` attribute in the description of application resources, nested under `<fx:application>`, as shown in [Example 9-5](#).

**Example 9-5 Use the `requiredFor` Attribute of `<fx:fileset>`**

```

<fx:application ... >
  <fx:resources>
    <fx:fileset id="preloader-files"
      requiredFor="preloader"
      dir="dist"
      includes="preloader.jar"/>
    <fx:fileset dir="dist" includes="myapp.jar"/>
  </fx:resources>
</fx:application>

```

With the help of the `refid` attribute creating a reference to an `id` attribute, elements can be reused to reduce code duplication. The preloader settings for the `<fx:jar>` and `<fx:deploy>` tasks are shown in [Example 9-6](#).

**Example 9-6 Preloader Settings in `<fx:jar>` and `<fx:deploy>` Tasks**

```

<fx:jar destfile="dist/application.jar">
  <fx:application refid="app-desc"/>
  <fx:resources>
    <fx:fileset refid="preloader-files"/>
  </fx:resources>
  <fileset dir="build/classes/" include="**"/>
</fx:jar>

<fx:deploy width="600" height="400"
  outdir="app-dist" outfile="SampleApp">
  <fx:info title="Sample application"/>
  <fx:application refid="app-desc"/>
  <fx:resources>
    <fx:fileset requiredFor="startup" dir="dist" include="application.jar"/>
    <fx:fileset refid="preloader-files"/>
  </fx:resources>
</fx:deploy>

```

See [Example 12-2](#) to see another preloader configuration in a full Ant task. In that example, both the preloader and the main application JAR files are signed in the `<fx:signjar>` task. If the preloader JAR file is unsigned and the main application JAR file is signed, then a multipart deployment descriptor is needed. Packaging is similar to any other JavaFX application using a mix of signed and unsigned code. For more information, see [Section 5.7.2, "Application Resources."](#)

Note the following preloader-specific details:

- The name of the preloader class is always specified in the main application descriptor, as in [Example 9-4](#).
- In most cases, it is a good idea to keep the preloader JAR files in the main application descriptor so they will start loading sooner.

The reasoning for the last point is as follows. There are two `<fx:deploy>` tasks to package this application, which generate two different JNLP files: one for the main application and another extension. The application will start from the link to the main JNLP, so whatever is referenced from the main JNLP file can start loading sooner and is ready faster.



## 9.3 Preloader Code Examples

The following code examples demonstrate various uses of preloaders:

- [Section 9.3.1, "Show the Preloader Only if Needed"](#)
- [Section 9.3.2, "Enhance Visual Transitions"](#)
- [Section 9.3.3, "Using JavaScript with a Preloader"](#)
- [Section 9.3.4, "Using a Preloader to Display the Application Initialization Progress"](#)
- [Section 9.3.5, "Cooperation of Preloader and Application: A Login Preloader"](#)
- [Section 9.3.6, "Cooperation of Preloader and Application: Sharing the Stage"](#)
- [Section 9.3.7, "Customizing Error Messaging"](#)

### 9.3.1 Show the Preloader Only if Needed

If the application runs standalone or is loaded from the web cache, then the preloader does not get any progress notifications because there is nothing to load, and the application will likely start quickly.

Using the FirstPreloader example as implemented in [Example 9-1](#), users only see the preloader stage briefly with 0 percent progress. Unless the application is embedded in a browser, a window also pops up that is briefly visible. In this case, a better user experience is to show nothing until the first progress notification.

When the application is embedded in a web page, something needs to be shown to avoid having a gray box (hole in the web page effect) where the application will appear. One possible approach is to display the HTML splash screen until the preloader has something to display or, if the preloader does not get any events, until the application is ready. Another option is to show a simplified version of the preloader and add a progress indicator after the first progress notification is received.

[Example 9-7](#) shows how to improve the relevant parts of the FirstPreloader implementation:

- Do not show the progress indicator until the first progress notification.
- If the preloader stage is not embedded, do not show it until the first progress notification.

#### **Example 9-7 Example of Tweaking When the Preloader Appears**

```
boolean isEmbedded = false;
public void start(Stage stage) throws Exception {
    //embedded stage has preset size
    isEmbedded = (stage.getWidth() > 0);

    this.stage = stage;
    stage.setScene(createPreloaderScene());
}

@Override
public void handleProgressNotification(ProgressNotification pn) {
    if (pn.getProgress() != 1 && !stage.isShowing()) {
        stage.show();
    }
    bar.setProgress(pn.getProgress());
}
```

See [Section 9.3.3, "Using JavaScript with a Preloader"](#) for an example of how to postpone hiding the splash screen.

## 9.3.2 Enhance Visual Transitions

The last state change notification received by the preloader before the application starts is `StateChangeNotification.Type.BEFORE_START`. After it is processed, the application's `start()` method is called. However, it can take time before the application is ready to display its stage after the `start()` method is called. If the preloader stage is already hidden, then there could be a period of time when the application shows nothing on the screen. When the application is embedded in a web page, this can result in a hole in the web page effect.

For this and other reasons, hiding the preloader instantly might not be the best visual transition from preloader to application. One approach to improve the visual transition between preloader and application is shown in [Example 9–8](#). If this `FirstPreloader` example is used for an application embedded in a web page, it will fade out over a period of 1 second instead of hiding instantly.

### **Example 9–8 Make the Preloader Fade Out**

```
@Override
public void handleStateChangeNotification(StateChangeNotification evt) {
    if (evt.getType() == StateChangeNotification.Type.BEFORE_START) {
        if (isEmbedded && stage.isShowing()) {
            //fade out, hide stage at the end of animation
            FadeTransition ft = new FadeTransition(
                Duration.millis(1000), stage.getScene().getRoot());
            ft.setFromValue(1.0);
            ft.setToValue(0.0);
            final Stage s = stage;
            EventHandler<ActionEvent> eh = new EventHandler<ActionEvent>() {
                public void handle(ActionEvent t) {
                    s.hide();
                }
            };
            ft.setOnFinished(eh);
            ft.play();
        } else {
            stage.hide();
        }
    }
}
```

If the preloader and application cooperate, then the transition is even smoother. See [Section 9.3.6, "Cooperation of Preloader and Application: Sharing the Stage"](#) for an example of a preloader that fades into the application.

If the application takes time to initialize, then it can be helpful to use a custom notification to initiate the transition from preloader to application when the application is ready. See [Section 9.3.4, "Using a Preloader to Display the Application Initialization Progress"](#) for further information.

## 9.3.3 Using JavaScript with a Preloader

Because a JavaFX application preloader has access to application features such as parameters and host services, the preloader can use JavaScript to communicate to the web page in which an application is embedded.

In [Example 9–9](#), JavaScript access is used to create a preloader that displays the loading progress in the HTML splash screen and hides the splash screen only when the application is ready. The code uses the following two JavaScript methods, which must be provided by the web page:

- `hide()` to hide the HTML splash screen
- `progress(p)` to update the progress

It is assumed that there is a custom HTML splash screen that is not hidden by default.

#### **Example 9–9 Use JavaScript from the Preloader**

```
import javafx.application.Preloader;
import javafx.stage.Stage;
import netscape.javascript.JSObject;

public class JSPreloader extends Preloader {
    public void start(Stage stage) throws Exception {}

    public void handleStateChangeNotification(StateChangeNotification evt) {
        if (evt.getType() == StateChangeNotification.Type.BEFORE_START) {
            JSObject js = getHostServices().getWebContext();
            if (js != null) {
                try {
                    js.eval("hide();");
                } catch (Throwable e) {
                    System.err.println("Ouch "+e);
                    e.printStackTrace();
                }
            }
        }
    }

    public void handleProgressNotification(ProgressNotification pn) {
        JSObject js = getHostServices().getWebContext();
        if (js != null) {
            try {
                js.eval("progress("+ ((int) (100*pn.getProgress()))+");");
            } catch (Throwable e) {
                e.printStackTrace();
            }
        }
    }
}
```

[Example 9–10](#) shows a sample web page template that uses the preloader in [Example 9–9](#). When this template page is processed during packaging, `#DT.SCRIPT.URL#` and `#DT.EMBED.CODE.ONLOAD#` will be replaced with code to embed the JavaFX application into the web page. For more information about templates, see [<fx:template>](#) in the JavaFX Ant reference.

#### **Example 9–10 Web Page Template Containing JavaScript for Preloader**

```
<html>
  <head>
    <style>
      div.label {
        position:absolute;
        bottom:100px;
        left:200px;
```

```
        font-family: 'tahoma';
        font-size:150px;
        color:silver;
    }
</style>

<SCRIPT src="#DT.SCRIPT.URL#"></SCRIPT>
<script>
    //Postpone the moment the splash screen is hidden
    // so it can show loading progress
    // save reference to hide function and replace it with no op for now
    var realHide = dtjava.hideSplash;
    dtjava.hideSplash = function(id) {}

    //hide splash
    function hide() {
        realHide('sampleApp');
    }

    //update progress data
    function progress(p) {
        var e = document.getElementById("splash");
        e.removeChild(e.firstChild);
        e.appendChild(document.createTextNode(""+p));
    }

    //create custom splash to be used
    function getSplash(app) {
        var l = document.createElement('div');
        l.className="label";
        l.id="splash";
        l.appendChild(document.createTextNode("..."));
        return l;
    }
</script>
<!-- #DT.EMBED.CODE.ONLOAD# -->

</head>
<body>
    <h2>Test page for <b>JS preloader sample</b></h2>
    <!-- Application will be inserted here -->
    <div id='javafx-app-placeholder'></div>
</body>
</html>
```

### 9.3.4 Using a Preloader to Display the Application Initialization Progress

A JavaFX application can pass information about events to a preloader by using custom notifications. For example, the preloader can be used to display the application initialization progress.

Technically, any class implementing the `Preloader.PreloaderNotification` interface can serve as a custom notification, and the application can send it to the preloader by using the `Application.notifyPreloader()` method. On the preloader side, the application notification is delivered to the `handleApplicationNotification()` method.

[Example 9–11](#) is a variation of the `FirstPreloader` example. It does not hide the preloader after notification of application startup is received. It waits for

application-specific notifications, displays the progress notifications, and hides the splash screen after the application sends a state change notification.

**Example 9-11 Preloader to Display Progress of Application Initialization and Loading**

```
public class LongAppInitPreloader extends Preloader {
    ProgressBar bar;
    Stage stage;
    boolean noLoadingProgress = true;

    private Scene createPreloaderScene() {
        bar = new ProgressBar(0);
        BorderPane p = new BorderPane();
        p.setCenter(bar);
        return new Scene(p, 300, 150);
    }

    public void start(Stage stage) throws Exception {
        this.stage = stage;
        stage.setScene(createPreloaderScene());
        stage.show();
    }

    @Override
    public void handleProgressNotification(ProgressNotification pn) {
        //application loading progress is rescaled to be first 50%
        //Even if there is nothing to load 0% and 100% events can be
        // delivered
        if (pn.getProgress() != 1.0 || !noLoadingProgress) {
            bar.setProgress(pn.getProgress()/2);
            if (pn.getProgress() > 0) {
                noLoadingProgress = false;
            }
        }
    }

    @Override
    public void handleStateChangeNotification(StateChangeNotification evt) {
        //ignore, hide after application signals it is ready
    }

    @Override
    public void handleApplicationNotification(PreloaderNotification pn) {
        if (pn instanceof ProgressNotification) {
            //expect application to send us progress notifications
            //with progress ranging from 0 to 1.0
            double v = ((ProgressNotification) pn).getProgress();
            if (!noLoadingProgress) {
                //if we were receiving loading progress notifications
                //then progress is already at 50%.
                //Rescale application progress to start from 50%
                v = 0.5 + v/2;
            }
            bar.setProgress(v);
        } else if (pn instanceof StateChangeNotification) {
            //hide after get any state update from application
            stage.hide();
        }
    }
}
```

In [Example 9–11](#), note that the same progress bar is used to display the progress of both the application initialization and loading. For simplicity, 50 percent is reserved for each phase. However, if the loading phase is skipped, for example when the application is launched as standalone, then the entire progress bar is devoted to displaying the progress of the application initialization.

[Example 9–12](#) shows the code on the application side. The `longStart()` method is used to simulate a lengthy initialization process that happens on a background thread. After initialization is completed, the `ready` property is updated, which makes the application stage visible. During initialization, intermediate progress notifications are generated. At the end of initialization, the `StateChangeNotification` is sent, which causes the preloader to hide itself.

**Example 9–12 Application Code to Enable the Progress Display**

```
public class LongInitApp extends Application {
    Stage stage;
    BooleanProperty ready = new SimpleBooleanProperty(false);

    private void longStart() {
        //simulate long init in background
        Task task = new Task<Void>() {
            @Override
            protected Void call() throws Exception {
                int max = 10;
                for (int i = 1; i <= max; i++) {
                    Thread.sleep(200);
                    // Send progress to preloader
                    notifyPreloader(new ProgressNotification(((double) i)/max));
                }
                // After init is ready, the app is ready to be shown
                // Do this before hiding the preloader stage to prevent the
                // app from exiting prematurely
                ready.setValue(Boolean.TRUE);

                notifyPreloader(new StateChangeNotification(
                    StateChangeNotification.Type.BEFORE_START));

                return null;
            }
        };
        new Thread(task).start();
    }

    @Override
    public void start(final Stage stage) throws Exception {
        // Initiate simulated long startup sequence
        longStart();

        stage.setScene(new Scene(new Label("Application started"),
            400, 400));

        // After the app is ready, show the stage
        ready.addListener(new ChangeListener<Boolean>(){
            public void changed(
                ObservableValue<? extends Boolean> ov, Boolean t, Boolean t1) {
                if (Boolean.TRUE.equals(t1)) {
                    Platform.runLater(new Runnable() {
                        public void run() {
```

```

        stage.show();
    }
    });
}
});
}
}

```

In this example, standard events are reused, but in general the application can send arbitrary data to the preloader. For example, for application loading, image collection notifications can include sample preview images and so on.

### 9.3.5 Cooperation of Preloader and Application: A Login Preloader

As part of `StateChangeNotification`, the preloader receives a reference to the application, which enables the preloader to cooperate closely with the application.

The example in this section shows how to use this cooperation in a login preloader, which requests user credentials while the application is loading, then passes them to the application.

In order to cooperate, this preloader and application share the `CredentialsConsumer` interface, which the preloader uses to pass credentials to the application. In addition to implementing a shared interface, the only other special thing in this sample is that the application does not show itself until it has both user credentials and a reference to a `Stage` object.

[Example 9–13](#) shows the application code for the login preloader.

#### Example 9–13 Enable the Login Preloader

```

public class AppToLogInto extends Application implements CredentialsConsumer {
    String user = null;
    Label l = new Label("");
    Stage stage = null;

    private void maybeShow() {
        // Show the application if it has credentials and
        // the application stage is ready
        if (user != null && stage != null) {
            Platform.runLater(new Runnable() {
                public void run() {
                    stage.show();
                }
            });
        }
    }

    @Override
    public void start(Stage stage) throws Exception {
        this.stage = stage;
        stage.setScene(new Scene(1, 400, 400));
        maybeShow();
    }

    public void setCredential(String user, String password) {
        this.user = user;
        l.setText("Hello "+user+"!");
        maybeShow();
    }
}

```

```
    }  
}
```

The preloader stage is displayed unconditionally, because the user must provide credentials. However, the preloader is not hidden when the application is ready to start unless there are credentials to pass to the application.

To be able to pass credentials, you can cast a reference to the application from `StateChangeNotification` to `CredentialsConsumer`, assuming the application implements it.

In [Example 9-14](#), the login pane UI from the previous example above is simplistic, but it shows how to adapt it to execution mode. If there is no progress to display, then there is no point to adding a progress bar to the UI. Also, if the application has finished loading but is still waiting for user input, then the UI can be simplified by hiding unneeded progress.

#### **Example 9-14 Login Preloader Code**

```
public class LoginPreloader extends Preloader {  
    public static interface CredentialsConsumer {  
        public void setCredential(String user, String password);  
    }  
  
    Stage stage = null;  
    ProgressBar bar = null;  
    CredentialsConsumer consumer = null;  
    String username = null;  
    String password = null;  
  
    private Scene createLoginScene() {  
        VBox vbox = new VBox();  
  
        final TextField userNameBox = new TextField();  
        userNameBox.setPromptText("name");  
        vbox.getChildren().add(userNameBox);  
  
        final PasswordField passwordBox = new PasswordField();  
        passwordBox.setPromptText("password");  
        vbox.getChildren().add(passwordBox);  
  
        final Button button = new Button("Log in");  
        button.setOnAction(new EventHandler<ActionEvent>(){  
            public void handle(ActionEvent t) {  
                // Save credentials  
                username = userNameBox.getText();  
                password = passwordBox.getText();  
  
                // Do not allow any further edits  
                userNameBox.setEditable(false);  
                passwordBox.setEditable(false);  
                button.setDisable(true);  
  
                // Hide if app is ready  
                maybeHide();  
            }  
        });  
        vbox.getChildren().add(button);  
  
        bar = new ProgressBar(0);
```



```

        vbox.getChildren().add(bar);
        bar.setVisible(false);

        Scene sc = new Scene(vbox, 200, 200);
        return sc;
    }

    @Override
    public void start(Stage stage) throws Exception {
        this.stage = stage;
        stage.setScene(createLoginScene());
        stage.show();
    }

    @Override
    public void handleProgressNotification(ProgressNotification pn) {
        bar.setProgress(pn.getProgress());
        if (pn.getProgress() > 0 && pn.getProgress() < 1.0) {
            bar.setVisible(true);
        }
    }

    private void maybeHide() {
        if (stage.isShowing() && username != null && consumer != null) {
            consumer.setCredential(username, password);
            Platform.runLater(new Runnable() {
                public void run() {
                    stage.hide();
                }
            });
        }
    }

    @Override
    public void handleStateChangeNotification(StateChangeNotification evt) {
        if (evt.getType() == StateChangeNotification.Type.BEFORE_START) {
            //application is loaded => hide progress bar
            bar.setVisible(false);

            consumer = (CredentialsConsumer) evt.getApplication();
            //hide preloader if credentials are entered
            maybeHide();
        }
    }
}

```

Note that close cooperation between the preloader and application is subject to mixed code restrictions unless both the preloader and application are in the same trust domain, in other words both are signed or unsigned.

### 9.3.6 Cooperation of Preloader and Application: Sharing the Stage

This section demonstrates how to use cooperation between the preloader and the application to improve the transition from preloader to application.

[Example 9–15](#) shows how the preloader and application share the same stage, and the preloader fades into the application when the application is ready. As in [Example 9–14](#), the preloader and application need to share the `SharedScene` interface.

**Example 9–15 SharedScene Interface**

```
/* Contact interface between application and preloader */
public interface SharedScene {
    /* Parent node of the application */
    Parent getParentNode();
}
```

The Application class implements it to provide the preloader with access to the application scene. The preloader later uses it for setup transition.

Now, the interface must be implemented. The code in [Example 9–16](#) shows that the application is active during the transition.

**Example 9–16 Implement the SharedScene Interface**

```
public class SharedStageApp extends Application
    implements FadeInPreloader.SharedScene {
    private Parent parentNode;
    private Rectangle rect;

    public Parent getParentNode() {
        return parentNode;
    }

    public void init() {
        //prepare application scene
        rect = new Rectangle(0, 0, 40, 40);
        rect.setArcHeight(10);
        rect.setArcWidth(10);
        rect.setFill(Color.ORANGE);
        parentNode = new Group(rect);
    }

    public void start(Stage primaryStage) {
        //setup some simple animation to
        // show that application is live when preloader is fading out
        Path path = new Path();
        path.getElements().add(new MoveTo(20, 20));
        path.getElements().add(new CubicCurveTo(380, 0, 380, 120, 200, 120));

        PathTransition pathTransition = new PathTransition();
        pathTransition.setDuration(Duration.millis(4000));
        pathTransition.setPath(path);
        pathTransition.setNode(rect);
        pathTransition.setCycleCount(Timeline.INDEFINITE);
        pathTransition.setAutoReverse(true);

        pathTransition.play();
    }
}
```

On the preloader side, instead of hiding the preloader stage, the code initiates a fade-in transition by inserting the application scene behind the preloader scene and fading out the preloader scene over time. After the fade-out is finished, the preloader is removed from the scene so the application can own the stage and scene.

**Example 9–17 Preloader Use of Fade-Out for a Smooth Transition**

```
public class FadeInPreloader extends Preloader{
    Group topGroup;
```

```

Parent preloaderParent;

private Scene createPreloaderScene() {
    //our preloader is simple static green rectangle
    Rectangle r = new Rectangle(300, 150);
    r.setFill(Color.GREEN);
    preloaderParent = new Group(r);
    topGroup = new Group(preloaderParent);
    return new Scene(topGroup, 300, 150);
}

public void start(Stage stage) throws Exception {
    stage.setScene(createPreloaderScene());
    stage.show();
}

@Override
public void handleStateChangeNotification(StateChangeNotification evt) {
    if (evt.getType() == StateChangeNotification.Type.BEFORE_START) {
        //its time to start fading into application ...
        SharedScene appScene = (SharedScene) evt.getApplication();
        fadeInTo(appScene.getParentNode());
    }
}

private void fadeInTo(Parent p) {
    //add application scene to the preloader group
    // (visualized "behind" preloader at this point)
    //Note: list is back to front
    topGroup.getChildren().add(0, p);

    //setup fade transition for preloader part of scene
    // fade out over 5s
    FadeTransition ft = new FadeTransition(
        Duration.millis(5000),
        preloaderParent);
    ft.setFromValue(1.0);
    ft.setToValue(0.5);
    ft.setOnFinished(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent t) {
            //After fade is done, remove preloader content
            topGroup.getChildren().remove(preloaderParent);
        }
    });
    ft.play();
}
}

```

### 9.3.7 Customizing Error Messaging

A preloader can also be used to customize messaging to the end user. For example, if the application cannot be started because the user declines to grant permissions, then an unsigned preloader can be used to provide better feedback, as shown in [Example 9–18](#).

#### **Example 9–18 Preloader with Error Messaging**

```

@Override
public boolean handleErrorNotification(ErrorNotification en) {
    // Display error
}

```

```
Label l = new Label(  
    "This application needs elevated permissions to launch. " +  
    "Please reload the page and accept the security dialog.");  
stage.getScene().setRoot(l);  
  
// Return true to prevent default error handler to take care of this error  
return true;  
}
```

Note that the preloader cannot provide error messaging when the error affects the preloader itself. For example, if a user cannot run an application embedded in a web page because the Java proxy settings are incorrect, then the preloader code cannot be loaded and therefore cannot display an error message.

## 9.4 Performance Tips

Because preloaders are displayed while the main application is loading, it is critical that they load quickly and run smoothly.

Use the following guidelines to ensure your preloaders perform well.

- Put the preloader classes and resources in a separate JAR file from the main application and follow packaging instructions. See [Section 9.2, "Packaging an Application with a Preloader."](#)
- Keep it small.  
Optimizing visual assets for size can significantly reduce the size of JAR files.
- Plan for achieving smooth transitions.  
Take care of the transition both from the splash screen to the preloader and from the preloader to the application. Consider reducing the number of transitions, for example by doing the following:
  - Show the preloader in HTML. See [Section 9.3.3, "Using JavaScript with a Preloader."](#)
  - Share the stage between the preloader and application. For an example, see [Section 9.3.6, "Cooperation of Preloader and Application: Sharing the Stage."](#)
  - Do not display a preloader. See [Section 9.3.1, "Show the Preloader Only if Needed."](#)
- Ensure there is something to display before initiating a transition.  
Both the application and the preloader itself may need some initialization time before they can display something on the screen. If this is the case, consider explicitly hiding the splash screen when the preloader is ready (see [Section 9.3.3, "Using JavaScript with a Preloader"](#)) and hiding the preloader on custom notification from the application (see [Section 9.3.4, "Using a Preloader to Display the Application Initialization Progress"](#)).
- If application initialization takes time, use custom events to update the preloader on initialization progress. For example, see [Section 9.3.4, "Using a Preloader to Display the Application Initialization Progress."](#)
- Avoid signing the preloader if possible  
If the preloader is signed and the user needs to grant permissions, then the preloader is not visible until the user grant permissions.

The following guidelines are applicable to both the main application and the preloader. See also [Section 3.3, "Coding Tips"](#) for general tips.

- Avoid lengthy operations on the JavaFX application thread.

Blocking the JavaFX thread pauses any UI updates and event processing. To avoid freezing the application UI, use the JavaFX Worker API and offload lengthy operations to other threads.

- Try to keep the `start()` method implementation lightweight.

Doing more work in the `init()` method unclogs the JavaFX application thread.

- Enable embedding when packaging your application for web deployment.

Embedding a deployment descriptor (JNLP) and security certificates (if needed) reduces the time needed to collect all the information about the application and help to start the preloader and application faster.



---

---

## JavaFX in Swing Applications

You can create Swing applications with embedded JavaFX content. This page describes how to deploy such applications.

This page contains the following topics:

- [Section 10.1, "Overview"](#)
- [Section 10.2, "Packaging with JavaFX Ant Tasks"](#)
- [Section 10.3, "Packaging without JavaFX Tools"](#)

See also [Section 2.7, "Deploying Swing and SWT Applications with Embedded JavaFX Content"](#).

### 10.1 Overview

Developers working on existing Swing applications may still take advantage of new JavaFX features by integrating JavaFX content into their Swing applications. (See the tutorial *JavaFX and Swing Applications*.)

Deployment of a Swing application with embedded JavaFX content on the web is similar to deployment of a regular Swing application as a Web Start (JNLP) application or applet. See the Java Tutorials lessons on Java applets and Web Start applications.

However, to be able to use JavaFX, the deployment descriptor of your application (JNLP file) needs to express a dependency on JavaFX Runtime.

JavaFX Ant tasks are recommended for packaging hybrid applications, as described in [Section 10.2, "Packaging with JavaFX Ant Tasks."](#) Alternatively, you can tweak your existing packaging process manually, as described in [Section 10.3, "Packaging without JavaFX Tools."](#)

### 10.2 Packaging with JavaFX Ant Tasks

As of JavaFX 2.2, you can use same set of Ant tasks (see [Chapter 5, "Packaging Basics"](#)) to package Swing applications with integrated JavaFX content. You only need to mark that the application's primary UI toolkit is Swing, using the `toolkit="swing"` attribute of `<fx:application>`.

The resulting package will be very similar to the package for pure JavaFX applications. (See [Section 5.2, "Base Application Package."](#)) The only difference is that there will be two deployment descriptors - one for deploying as a Swing applet and another for launching your application using Web Start.

[Example 10-1](#) shows a sample Ant task for a Swing-JavaFX application that uses the `toolkit="swing"` attribute.

**Example 10–1 Using JavaFX Ant Tasks to Package a Swing Application with Integrated JavaFX Content**

```

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

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

<fx:deploy width="800" height="600" outdir="dist-web"
    outfile="SwingInterop">
    <fx:info title="Swing Interop"/>
    <!-- Mark application as a Swing app -->
    <fx:application id="myapp"
        mainClass="swinginterop.SwingInterop"
        toolkit="swing"/>
    <fx:resources>
        <fx:fileset dir="dist-web" includes="SwingInterop.jar"/>
    </fx:resources>
</fx:deploy>

```

**10.2.1 Enabling an HTML Splash Screen**

Swing applications do not have a default preloader, so there is no code to hide the HTML splash screen when the application is ready. Therefore, by default, the generated HTML page will not use an HTML splash screen.

To enable a splash screen, do both of the following:

- Explicitly define a code splash callback `onGetSplash`. For example; add the markup in [Example 10–2](#) to [Example 10–1](#):

**Example 10–2 Defining a Code Splash Callback `onGetSplash`**

```

<fx:callbacks>
    <!-- force use of default splash handler -->
    <fx:callback name="onGetSplash">
        (new dtjava.Callbacks()).onGetSplash
    </fx:callback>
</fx:callbacks>

```

See [Section 7.2, "Callbacks"](#) for more information about callbacks and `<fx:callbacks>` for further details on the use of `<fx:callbacks>`.

- Add code to explicitly hide the HTML splash screen when your application is ready, by using `LiveConnect` to call the JavaScript function `dtjava.hideSplash()` from Java code, as shown in [Example 10–3](#).

**Example 10–3 Hide the HTML Splash Screen When the Application is Ready**

```

Applet myApplet = ...;
//appId is id used in the dtjava.embed() call String appId = "sampleApp";

JSObject window = JSObject.getWindow(myApplet);
try {

```



```

        window.eval("dtjava.hideSplash('+appId+')");
    } catch(Throwable t) {
        ...
    }
}

```

## 10.3 Packaging without JavaFX Tools

If your project already has existing support for packaging as a Web Start application or applet, then it might be easier to tweak the template of the JNLP file directly.

To express the dependency on JavaFX Runtime, you need to edit the JNLP to do the following:

- Define the JavaFX namespace with `jfx:`.
- Add the `<jfx:javafx-runtime>` tag and specify the minimum required version of JavaFX.
- Specify that the minimum required version of Java Runtime is Java Runtime 7 update 6 or later

[Example 10–4](#) shows an example of these modifications to the deployment descriptor.

### Example 10–4 Modify the Deployment Descriptor

```

<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0"
    xmlns:jfx="http://javafx.com"
    href="SwingAppWithJavaFXContent.jnlp"
    ...>
    ...
    <resources>
        <j2se version="1.7.0_06+"
            href="http://java.sun.com/products/autodl/j2se"/>
        <jfx:javafx-runtime version="2.1+"
            href="http://javadl.sun.com/webapps/download/GetFile/
                javafx-latest/windows-i586/javafx2.jnlp"/>
    </resources>
    ...
</jnlp>

```

No changes are required on the HTML side. However, you may also use the JavaFX Deployment Toolkit to integrate your content with the web page, as shown in the following examples.

### 10.3.1 Using the Deployment Toolkit

It is recommended that you use the Deployment Toolkit to deploy your application as an applet or launch it from a web page. The Deployment Toolkit greatly simplifies deployment routines, such as ensuring that JavaFX Runtime is available. The Deployment Toolkit also has a number of bonus features, such as:

- Passing JVM options or arguments to your application dynamically. See [Section 7.3.4, "Specify Platform Requirements and Pass JVM Options"](#) and [Section 7.3.3, "Pass Parameters to a Web Application."](#)
- Usage procedures are almost the same as for JavaFX applications. The only requirement is to specify the Swing toolkit as one of the platform requirements.

For example [Example 10–5](#) shows an example of web page code for launching a Swing Web Start application. This code is the same as that used to launch JavaFX applications.

**Example 10–5 Scripts and Markup in HTML Page for a Web Start Application**

```
<html>
  <head>
    <SCRIPT src="http://java.com/js/dtjava.js"></SCRIPT>
    <script>
      function launchApplication(jnlpfile) {
        dtjava.launch(
          { url : jnlpfile },
          {
            javafx : '2.2+',
            toolkit: 'swing'
          },
          {}
        );
        return false;
      }
    </script>
  </head>
  <body>
    <h2>Test page</h2>
    <a href='SampleApp.jnlp'
      onclick="return launchApplication('SampleApp.jnlp');">Click</a>
    to launch test app.
  </body>
</html>
```

[Example 10–5](#) is simplistic, but you can use other features of the `dtjava.launch()` method in the Deployment Toolkit for Swing applications, as needed. For example, you can embed the JNLP file into the web page for faster startup.

For applets, the approach is similar. Applets can be embedded into a web page using the `dtjava.embed()` function. You must also specify the 'swing' toolkit to indicate a preference of application type for the Deployment Toolkit.

Another caveat concerns built-in HTML splash support in the Deployment Toolkit. Swing applications do not have a default preloader, so there is no code to hide the HTML splash screen when the application is ready. To address this difference, do either of the following:

- Add code to explicitly hide the HTML splash screen when your application is ready, by using LiveConnect to call the JavaScript function `dtjava.hideSplash()` from Java code, as shown in [Example 10–6](#).

**Example 10–6 Hide the HTML Splash Screen When the Application is Ready**

```
Applet myApplet = ...;
//appId is id used in the dtjava.embed() call String appId = "sampleApp";

JSObject window = JSObject.getWindow(myApplet);
try {
    window.eval("dtjava.hideSplash('"+appId+"');");
} catch(Throwable t) {
    ...
}
```

- Provide a custom splash callback that does nothing, as shown in [Example 10–7](#) and in [Section 7.3.6, "Disable the HTML Splash Screen"](#).

**Example 10–7 Custom Callback to Disable the HTML Splash Screen**

```
<html>
  <head>
    <SCRIPT src="http://java.com/js/dtjava.js"></SCRIPT>
    <script>
      function embedApp() {
        dtjava.embed(
          {
            id : 'sampleApp',
            url : 'SampleApp_browser.jnlp',
            placeholder : 'app-placeholder',
            width : 960,
            height : 720
          },
          {
            javafx : '2.2+',
            toolkit: 'swing'
          },
          { onGetSplash: function() {} } //disable splash
        );
      }
      <!-- Embed Swing application into web page after page is loaded -->
      dtjava.addOnloadCallback(embedApp);
    </script>
  </head>
  <body>
    <h2>Test page</h2>
    <!-- Applet will be inserted here -->
    <div id='javafx-app-placeholder'></div>
  </body>
</html>
```



---

---

## The JavaFX Packager Tool

The JavaFX Packager tool can be used to compile, package, sign, and deploy JavaFX applications from the command line. It can be used as an alternative to an Ant task or building the applications in an IDE.

You can access reference information about the JavaFX Packager tool in this document or by entering `javafxpackager` at the command line if you have set an environment variable for the path to the tool in your JavaFX SDK installation.

**Tip:** For Windows installations, add the path to the `javafxpackager` command to the `PATH` environment variable. The `javafxpackager.jar` file is located in the `bin` directory of your JavaFX SDK installation.

The `javafxpackager` command has several component task commands, described in [Table 11-1](#). The command-line options depend on which task command you are using, as described in the [javafxpackager](#) reference documentation.

**Table 11-1** Task Commands in the JavaFX Packager Tool

Task Command	Description
<code>javafxpackager -createbss</code>	Converts a CSS file into binary form
<code>javafxpackager -createjar</code>	Produces a JAR archive according to other parameters specified as options.
<code>javafxpackager -deploy</code>	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.
<code>javafxpackager -makeall</code>	Compiles source code and combines the <code>-createjar</code> and <code>-deploy</code> commands, with simplified options.
<code>javafxpackager -signjar</code>	Digitally signs JAR files and attaches a certificate.

## javafxpackager

A tool with commands that perform tasks related to packaging and signing JavaFX applications.

### Synopsis

```
javafxpackager -taskcommand [-options]
```

where *-taskcommand* is one of the following:

**-createjar**

Produces a JAR archive according to other parameters.

**-deploy**

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

**-createbss**

Converts CSS files into binary form.

**-signJar**

Signs JAR file(s) with a provided certificate.

**-makeall**

Performs compilation, `createjar`, and `deploy` steps as one call, with most arguments predefined. By default, it attempts to generate all applicable self-contained application packages. The source files must be located in a folder called `src`, and the resulting files (JAR, JNLP, HTML, and self-contained application packages) are put in a folder called `dist`. This command can only be configured in a minimal way and is as automated as possible.

Note that all options are case-insensitive.

### Options for the createjar Command

**-appclass <application class>**

Qualified name of the application class to be executed.

**-preloader <preloader class>**

Qualified name of the preloader class to be executed.

**-paramfile <file>**

A properties file with default named application parameters.

**-argument arg**

An unnamed argument to be inserted into the JNLP file as an `<fx:argument>` element.

**-classpath <files>**

List of dependent JAR file names.

**-manifestAttrs <manifest attributes>**

List of additional manifest attributes. Syntax:

```
"name1=value1,name2=value2,name3=value3"
```

**-noembedlauncher**

If present, the packager will not add the JavaFX launcher classes to the JAR file.

**-nocss2bin**

The packager will not convert CSS files to binary form before copying to JAR.

**-runtimeversion <version>**

Version of the required JavaFX Runtime.

**-outdir <dir>**

Name of the directory that will receive generated output files.

**-outfile <filename>**

Name (without the extension) of the file that will be generated.

**-srcdir <dir>**

Base directory of the files to package.

**-srcfiles <files>**

List of files in `srcdir`. If omitted, all files in `srcdir` (which is a mandatory argument in this case) will be used. Files in the list must be separated by spaces.

## Options for the deploy Command

**-title <title>**

Title of the application.

**-vendor <vendor>**

Vendor of the application.

**-description <description>**

Description of the application.

**-appclass <application class>**

Qualified name of the application class to be executed.

**-preloader <preloader class>**

Qualified name of the preloader class to be executed.

**-paramfile <file>**

Properties file with default named application parameters.

**-htmlparamfile <file>**

Properties file with parameters for the resulting application when it is run in the browser.

**-width <width>**

Width of the application.

**-height <height>**

Height of the application.

**-native <type>**

Generate self-contained application bundles (if possible). If *type* is specified, then only a bundle of this type is created. List of supported types includes: installer, image, exe, msi, dmg, rpm, deb.

**-name <name>**

Name of the application.

**-embedjnlp**

If present, the JNLP file will be embedded in the HTML document.

**-embedCertificates**

If present, the certificates will be embedded in the JNLP file.

**-allpermissions**

If present, the application will require all security permissions in the JNLP file.

**-updatemode <updatemode>**

Sets the update mode for the JNLP file.

**-isExtension**

If present, the `srcfiles` are treated as extensions.

**-callbacks**

Specifies user callback methods in generated HTML. The format is the following:

```
"name1:value1,name2:value2,..."
```

**-templateInFilename**

Name of the HTML template file. Placeholders are in the following form:

```
#XXXX.YYYY (APPID) #
```

Where APPID 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 `http://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.

**-templateOutFilename**

Name of the HTML file that will be generated from the template.

**-templateId**

Application ID of the application for template processing.

**-argument arg**

An unnamed argument to be inserted into an `<fx:argument>` element in the JNLP file.



**-outdir <dir>**

Name of the directory that will receive generated output files.

**-outfile <filename>**

Name (without the extension) of the file that will be generated.

**-srcdir <dir>**

Base directory of the files to package.

**-srcfiles <files>**

List of files in `srcdir`. If omitted, all files in `srcdir` (which is a mandatory argument in this case) will be used. Files in the list must be separated by spaces.

**Options for the createbss Command****-outdir <dir>**

Name of the directory that will receive generated output files.

**-srcdir <dir>**

Base directory of the files to package.

**-srcfiles <files>**

List of files in `srcdir`. If omitted, all files in `srcdir` (which is a mandatory argument in this case) will be used. Files in the list must be separated by spaces.

**Options for the signJar Command****-keyStore <file>**

Keystore file name.

**-alias**

Alias for the key.

**-storePass**

Password to check integrity of the keystore or unlock the keystore

**-keyPass**

Password for recovering the key.

**-storeType**

Keystore type. The default value is "jks".

**-outdir <dir>**

Name of the directory that will receive generated output files.

**-srcdir <dir>**

Base directory of the files to be signed.

**-srcfiles <files>**

List of files in `srcdir`. If omitted, all files in `srcdir` (which is a mandatory argument in this case) will be used. Files in the list must be separated by spaces.

**Options for the makeAll Command****-appclass <application class>**

Qualified name of the application class to be executed.

**-preloader <preloader class>**

Qualified name of the preloader class to be executed.

**-classpath <files>**

List of dependent JAR file names.

**-name <name>**

Name of the application.

**-width <width>**

Width of the application.

**-height <height>**

Height of the application.

## Notes

- A `-v` option can be used with any task command to enable verbose output.
- When the `-srcdir` option is allowed in a command, it can be used more than once. If the `-srcfiles` option is specified, the files named in the argument will be looked for in the location specified in the preceding `srcdir` option. In case there is no `-srcdir` preceding `-srcfiles`, the directory where the `javafxpackager` command is executed will be used.

## Examples

### Example 1 -createjar Command Usage

```
javafxpackager -createjar -appclass package.ClassName  
-srcdir classes -outdir out -outfile outjar -v
```

Packages the contents of the classes directory to `outjar.jar`, sets the application class to `package.ClassName`.

### Example 2 -deploy Command Usage

```
javafxpackager -deploy -outdir outdir -outfile outfile -width 34 -height 43  
-name AppName -appclass package.ClassName -v -srcdir compiled
```

Generates `outfile.jnlp` and the corresponding `outfile.html` files in `outdir` for application `AppName`, which is started by `package.ClassName` and has dimensions of 34 x 43.

### Example 3 -makeall command Usage

```
javafxpackager -makeall -appclass brickbreaker.Main -name BrickBreaker  
-width 600 -height 600
```

Does all the packaging work including compilation: compile, createjar, deploy.

### Example 4 -signJar Command Usage

```
javafxpackager -signJar --outdir dist -keyStore sampleKeystore.jks  
-storePass **** -alias javafx -keypass **** -srcdir dist
```

Signs all of the JAR files in the `dist` directory, attaches a certificate with the specified alias, `keyStore` and `storePass`, and puts the signed JAR files back into the `dist` directory.

---

---

## JavaFX Ant Tasks

This chapter shows how to use Ant to package JavaFX application.

JavaFX Ant tasks and the JavaFX Packager tool are currently the only supported ways to package JavaFX applications. This includes supported versions of the NetBeans IDE, which build JavaFX applications with JavaFX Ant tasks.

This page contains the following topics:

- [Section 12.1, "Requirements to Run JavaFX Ant Tasks"](#)
- [Section 12.2, "JavaFX Ant Elements"](#)
- [Section 12.3, "Using JavaFX Ant Tasks"](#)
- [Section 12.4, "Ant Script Examples"](#)

See also the following two Ant Task Reference sections:

- [JavaFX Ant Task Reference](#)
- [JavaFX Ant Helper Parameter Reference](#)

### 12.1 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`

### 12.2 JavaFX Ant Elements

There are two categories of Ant elements for JavaFX. Each of the following elements is described in [JavaFX Ant Task Reference](#).

#### JavaFX Ant Tasks

These elements accomplish the following tasks:

- Creating double-clickable JAR files
- 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

See [JavaFX Ant Task Reference](#). For general information about packaging for JavaFX applications, see [Chapter 5, "Packaging Basics"](#) and [Chapter 6, "Self-Contained Application Packaging."](#)

### Ant Helper Parameters

These elements are used by the JavaFX tasks. They are listed and described in [JavaFX Ant Helper Parameter Reference](#).

## 12.3 Using JavaFX Ant Tasks

To use the JavaFX Ant tasks in the your Ant script, you must load their definitions. An example is shown in the build.xml file in [Example 12-1](#):

### Example 12-1 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=".:path/to/sdk/lib/ant-javafx.jar"/>
  </target>
</project>
```

Notes about [Example 12-1](#):

- Ensure that you declare the `fx:` namespace, shown in bold in [Example 12-1](#), 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 [Section 6.3.3, "Customization Using Drop-In Resources."](#)

Once 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

## 12.4 Ant Script Examples

[Example 12-2](#) shows an Ant script that uses the `<fx:jar>` task to build the JAR file and the `<fx:deploy>` task to build the JNLP and HTML files for web deployment. Other elements, such as `<fx:application>` and `<fx:resources>` are types that are described in the `<fx:application>` and `<fx:resources>` in the Ant task reference.

### Example 12-2 Typical JavaFX Ant Script

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

<fx:application id="sampleApp"
  name="Some sample app"
```

```
        mainClass="test.MyApplication"
        <!-- This application has a preloader class -->
        preloaderClass="testpreloader.Preloader"
        fallbackClass="test.UseMeIfNoFX"/>

<fx:resources id="appRes">
    <fx:fileset dir="dist" requiredFor="preloader"
        includes="mypreloader.jar"/>
    <fx:fileset dir="dist" includes="myapp.jar"/>
</fx:resources>

<fx:jar destfile="dist/myapp.jar">
    <!-- Define what to launch -->
    <fx:application refid="sampleApp"/>

    <!-- Define what classpath to use -->
    <fx:resources refid="appRes"/>

    <manifest>
        <attribute name="Implementation-Vendor"
            value="{application.vendor}"/>
        <attribute name="Implementation-Title"
            value="{application.title}"/>
        <attribute name="Implementation-Version" value="1.0"/>
    </manifest>

    <!-- Define what files to include -->
    <fileset dir="{build.classes.dir}"/>
</fx:jar>

<fx:signjar keyStore="{basedir}/sample.jks" destdir="dist"
    alias="javafx" storePass="****" keyPass="****">
    <fileset dir='dist/*.jar'/>
</fx:signjar>

<fx:deploy width="{applet.width}" height="{applet.height}"
    outdir="{basedir}/{dist.dir}" embedJNLP="true"
    outfile="{application.title}">

    <fx:application refId="sampleApp"/>

    <fx:resources refid="appRes"/>

    <fx:info title="Sample app: {application.title}"
        vendor="{application.vendor}"/>

    <!-- Request elevated permissions -->
    <fx:permissions elevated="true"/>
</fx:deploy>
```

## JavaFX Ant Task Reference

The following items comprise the main JavaFX Ant tasks:

- [<fx:csstobin>](#)  
Used to convert CSS files to binary format for faster processing. See also [Section 5.4, "Stylesheet Conversion."](#)
- [<fx:deploy>](#)  
Used to assemble 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. See also [Section 5.7, "Run the Deploy Task or Command."](#)
- [<fx:jar>](#)  
Used to create one or more application JAR files. See also [Section 5.5, "Create the Main Application JAR File."](#)
- [<fx:signjar>](#)  
Used when the application needs a digital signature. See also [Section 5.6, "Sign the JAR Files."](#)

Items are in alphabetical order.

## <fx:csstobin>

### Description

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

### Parent Elements

None.

### Parameters

**Table 12–1** *fx:csstobin*

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

**Table 12–2** *fx:deploy*

Attribute	Description	Type	Required?
<code>embeddedHeight</code>	If present, this value will be used for Javascript/HTML code instead of width/height. Affects only embedded deployment mode.  Use it if you want to specify a relative dimension for an embedded application.  See <a href="#">Section 5.8.4, "Publishing an Application that Fills the Browser Window."</a>	String	No
<code>embeddedWidth</code>	Same description as for <code>embeddedHeight</code> .	String	No
<code>embedjnlp</code>	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.
<code>extension</code>	Treat the files named in <code>srcfiles</code> 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.
<code>height</code>	Height of the application scene, for embedding applications into a web page.	String	Yes
<code>includeDT</code>	If set to <code>true</code> , files related to the Deployment Toolkit will be copied to a <code>web-files</code> subdirectory of the directory specified in <code>outdir</code> . This setting is useful for offline development but is not advised for production.	Boolean	No Default is false.



**Table 12–2 (Cont.) *fx:deploy***

Attribute	Description	Type	Required?
nativeBundles	<p>Values:</p> <ul style="list-style-type: none"> <li>▪ all</li> <li>▪ deb</li> <li>▪ dmg</li> <li>▪ exe</li> <li>▪ image</li> <li>▪ msi</li> <li>▪ none</li> <li>▪ rpm</li> </ul> <p>Value <code>all</code> produces all applicable self-contained application packages. Value <code>none</code> produces no self-contained application packages. Or use another value to produce a specific package installer.</p>	String	No Default is none.
offlineAllowed	If the value is <code>true</code> , the cached application can operate even if the client system is disconnected from the network.	Boolean	Default is <code>true</code> .
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 <code>document.getElementById()</code> is used to resolve it.	String	Yes Either the reference or the ID of the placeholder is required.

**Table 12-2 (Cont.) fx:deploy**

Attribute	Description	Type	Required?
updatemode	Indicates the preferences for when checks for application updates are performed for embedded and Web Start applications.  A value of <code>always</code> means to always check for updates before launching the application.  A value of <code>background</code> means to launch the application while checking for updates in the background.  See <a href="#">Section 5.9.1, "Background Update Check for the Application."</a>	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: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 redistributable 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>
```

## <fx:jar>

### Description

Packages a JavaFX 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, which detects the presence of JavaFX Runtime, sets up the environment, and executes the application.
- Embeds the fallback AWT applet, to be used if JavaFX is not available.
- Creates a manifest in the JAR file.

The resulting JAR file supports launching by double-clicking.

### Parent Elements

None.

### Parameters

**Table 12–3** *fx:jar*

Attribute	Description	Type	Required?
<code>destfile</code>	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:jar> Usage Examples

See [Example 12–2](#) 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 auxiliary 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

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.

This is a new signing method in JavaFX. For traditional signing, the standard Ant `signjar` task should be used.

### Parent Elements

None.

### Parameters

**Table 12–4** *fx:signjar*

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:application>`
- `<fx:argument>`
- `<fx:callback>`
- `<fx:callbacks>`
- `<fx:fileset>`
- `<fx:htmlParam>`
- `<fx:icon>`
- `<fx:info>`
- `<fx:jvmarg>`
- `<fx:param>`
- `<fx:permissions>`
- `<fx:platform>`
- `<fx:preferences>`
- `<fx:property>`
- `<fx:resources>`
- `<fx:splash>`
- `<fx:template>`

Items are in alphabetical order.



## <fx:application>

### Description

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

### Parent Elements

- [<fx:deploy>](#)

### Parameters

**Table 12–5** *fx:application*

Attribute	Description	Type	Required?
name	--	String	--
fallbackClass	AWT-based applet to be used if application fails to launch due to missing FX runtime and installation of JavaFX is not possible.	String	No
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 <code>refid</code> ).	String	No
mainClass	Qualified name of the main application class, which should extend <code>javafx.application.Application</code>	String	Yes
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 <code>javafx.application.Preloader</code>	String	No Default is the preloader that is shipped with the JavaFX Runtime.
refid*	--	Reference	No

**Table 12–5 (Cont.) *fx:application***

Attribute	Description	Type	Required?
toolkit	Indicates your preference for the application to use a specific UI toolkit. Possible values: <ul style="list-style-type: none"><li>▪ fx</li><li>▪ swing</li></ul>	String	No Default value is fx.

\* 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

See [Example 12–2](#).

## <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 & false</fx:argument>
</fx:application>
```

## <fx:callback>

### Description

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

### Parent Elements

- [<fx:callbacks>](#)

### Parameters

**Table 12–6** *fx:callback*

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

*Table 12–7 fx:callbacks*

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 on 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. See [Section 5.7.2, "Application Resources"](#) for details.

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

**Table 12–8** *fx:fileset*

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.
excludes	--	String	--
includes	--	String	--
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.

**Table 12–8 (Cont.) *fx:fileset***

Attribute	Description	Type	Required?
requiredFor (used only when <fx:fileset> is nested under <fx:resources>)	Defines when resources are needed (affects loading priority). Supported values are: <ul style="list-style-type: none"> <li>▪ preloader - resources are needed to launch the preloader (first thing to be executed)</li> <li>▪ startup - resources are needed to launch the application.</li> <li>▪ runtime - resources are not critical to launch the application but may be needed later.</li> </ul>	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: <ul style="list-style-type: none"> <li>▪ auto for autodetect</li> <li>▪ data</li> <li>▪ jar</li> <li>▪ jnlp</li> <li>▪ license</li> <li>▪ native for JAR files containing native libraries</li> <li>▪ icon</li> </ul>	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

**Table 12–9** *fx:htmlParam*

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
    to embedded [nd 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.

Note that in JavaFX 2.2, <fx:icon> is not used for self-contained applications. For details on how to customize icon for self-contained application, see [Section 6.3.3, "Customization Using Drop-In Resources."](#)

### Parent Elements

- <fx:info>

### Parameters

**Table 12–10** *fx:icon*

Attribute	Description	Type	Required?
depth	Image depth	String	No
href	Location of image	String	Yes
height	Image height	String	No
kind	Icon type. Supported values are: <ul style="list-style-type: none"><li>▪ default</li><li>▪ disabled</li><li>▪ rollover</li><li>▪ selected</li><li>▪ shortcut</li></ul>	String	No Default value is default.
width	Image width	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

**Table 12–11** *fx:info*

Attribute	Description	Type	Required?
category	<p>Application category. Creates a link to an application in a specified category. Semantics of the value depends on the format of the package.</p> <p>For example:</p> <ul style="list-style-type: none"> <li>▪ For a self-contained application on Linux, it is used to define the application menu category where the application is listed.</li> <li>▪ On Mac: Creates key in info.plist  <pre>&lt;key&gt;LSApplicationCategoryType&lt;/key&gt; &lt;string&gt;unknown&lt;/string&gt;</pre> </li> <li>▪ 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</li> </ul>	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:info>

---

- [<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

**Table 12–12** *fx:jvmarg*

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

### Parameters Accepted as Nested Elements

None.

### <fx:jvmarg> Usage Examples

See [<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

**Table 12–13** *fx:jvmuserarg*

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

### Parameters Accepted as Nested Elements

None.

### <fx:jvmuserarg> Usage Examples

See [<fx:platform> Parameter to Specify JVM Options](#).

## <fx:param>

### Description

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

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

### Parent Elements

- [<fx:application>](#)

### Parameters

**Table 12–14** *fx:param*

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>](#)

### Parameters

**Table 12–15** *fx:permissions*

Attribute	Description	Type	Required?
cacheCertificates	If set to true, then the certificate used to sign the JAR files are cached in the deployment descriptor. Caching enables the user to accept elevated permissions earlier in the startup process, which improves startup time.  This setting has no effect if the application is run in the sandbox.	Boolean	No Default is false.
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 Embed Signing Certificate into Deployment Descriptor

See [Section 5.9.3, "Embed Signing Certificate into Deployment Descriptor."](#)

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



## <fx:platform>

### Description

Defines application platform requirements.

### Parent Elements

- [<fx:deploy>](#)
- [<fx:jar>](#)

### Parameters

**Table 12–16** *fx:platform*

Attribute	Description	Type	Required?
refid*	--	Reference	No
javafx	Minimum version of JavaFX required by the application.	String	No Default value matches the release of the JavaFX SDK; for example, if you use the JavaFX 2.2 SDK, the default value is '2.2'.
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 JavaFX Runtime version 2.1 or later and JRE version 7.0 or later.

```
<fx:platform javafx="2.1+" j2se="7.0"/>
```

#### Example 2 <fx:platform> Parameter to Specify JVM Options

In this example, the application needs JavaFX Runtime version 2.1 or later and needs to run in a JVM launched with "-Xmx400 -verbose:jni -Dpurpose='sample value'".

```
<fx:platform javafx="2.1+">
  <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.
- Java Runtime 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

**Table 12–17** *fx:preferences*

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

\* 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 [<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

*Table 12–18* *fx:property*

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

**Table 12–19** *fx:resources*

Attribute	Description	Type	Required?
<code>id</code>	ID that can be referred from another element with a <code>refid</code> attribute.	String	No
<code>refid*</code>	--	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

See also examples in [Chapter 5, "Packaging Basics"](#) and [Chapter 6, "Self-Contained Application Packaging."](#)

#### 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>
```

## <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

**Table 12–20** *fx:splash*

Attribute	Description	Type	Required?
href	Location of image	String	Yes
mode	Deployment mode. Supported values are: <ul style="list-style-type: none"><li>■ any (but currently only functional in Web Start mode)</li><li>■ webstart</li></ul>	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 `http://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

**Table 12–21** *fx:template*

Attribute	Description	Type	Required?
<code>file</code>	Input template file.	File	Yes

**Table 12–21 (Cont.) fx:template**

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>
```

---

---

## Troubleshooting

This page contains some troubleshooting practices to follow if you encounter any problems deploying your JavaFX applications. It contains the following sections.

- [Section 13.1, "Running Applications"](#)
- [Section 13.2, "Development Process Issues"](#)
- [Section 13.3, "Runtime Issues"](#)

### 13.1 Running Applications

Use the following checklist if you have trouble running applications after you package them.

- Verify that you have a supported environment. See the System Requirements document for your JavaFX release.
- Check the release notes for known issues.

If the tips below do not help to resolve the issue, then:

- Ask experts in the JavaFX Forum.
- File a bug to JIRA.

Describe in detail what you are trying to do, and what exactly does not work as expected. If possible, share a test case to reproduce the problem. Be sure to include information about your environment and your Java and JavaFX versions.

### 13.2 Development Process Issues

- Verify that you use the latest version of JavaFX tools.
- `javafxpackager` command fails:
  - Ensure that the JDK bin folder, where `javafxpackager` utility resides, is on the `PATH`.
  - Ensure `JAVA_HOME` is set.
  - On Mac, the problem could be due to simultaneous use of Apple's JDK 6 and Oracle's JDK 7. To work around them, set `JAVA_HOME` and add it to the path.
    - \* `export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.7.0_06.jdk/Contents/Home`
    - \* `export PATH=$JAVA_HOME/bin:$PATH`
- Packaging self-contained applications:

- If the package format you are trying to build depends on third-party tools, then make sure they are available on the `PATH`.
- Set `verbose="true"` in the `<fx:deploy>` task or pass `"-v"` to `javafxpackager` to get verbose build output.
- Set the environment variable `JAVAFX_ANT_DEBUG` to `true` to get additional details and keep intermediate build artifacts.
- If drop-in custom resources are not used, then verify that the JavaFX Ant task definitions are loaded with the correct classpath, typically with `"."` at the beginning. See [Using JavaFX Ant Tasks](#)
- Packaging self-contained applications on remote systems:
  - If packaging tools are executed on remote system (e.g. Jenkins, Hudson, and other build systems) to produce the artifact, then it is important to have same user logged on a desktop system. If using the same user login is not possible, you can create a custom `.dmg` bundle as follows:
    1. Create a `.dmg` image from manual build
    2. Convert the `.dmg` file to a read-write form
    3. Remove the content of you application folder, but keep the top level app directory
    4. Add the `dmg` to the build
    5. At the build time, mount it, copy `.app` contents to the image, then convert `.dmg` to a compressed read only form
- NetBeans issues:
  - Ensure you are using NetBeans 7.2 or later
  - If a Clean and Build fails to build sample application or double-clicking some or all of the files results in an error, check that the JavaFX platform is enabled properly in NetBeans. See "Setting Up NetBeans IDE With JavaFX".
  - To get more insight into the build process, enable verbose output by clicking the **Tools** icon in the build output window.
  - To build self-contained applications, ensure that the required third-party tools are added to the `PATH` before you start NetBeans. See [Section 6.4, "Installable Packages."](#)

## 13.3 Runtime Issues

Basic checklist:

- Verify that you have the latest version Java installed (Java 7 update 6 or later).
- Before trying to troubleshoot your application, ensure the JavaFX samples run properly.
  1. Download the JavaFX samples zip file.
  2. Double-click the JAR file, the JNLP file, and the HTML file for at least one sample to ensure it runs correctly.
- Validate that the `java` process is using your Java Runtime installed location.

- If you have a 64-bit system, you may be using either a 32- or 64-bit version of Java. The best way to avoid problems caused by unexpected use of an older version of Java Runtime is to keep both 32- and 64-bit versions of Java up to date.

### 13.3.1 Standalone Execution

- Run the same application from the command line as  

```
java -jar MyApp.jar
```

This way you can see actual exceptions and trace messages (if any).
- Pass `"-Djavafx.verbose=true"` to enable verbose output from the embedded launcher.
- If your application starts slow, then it could be due to network configuration. Try disabling the autoproxy configuration by passing `"-Djavafx.autoproxy.disable=true"` to see if it helps.
- Do not forget that it is just a Java application, and you can use your favorite techniques to troubleshoot it.

### 13.3.2 Self-Contained Applications

- Run the native launcher from the console window to see trace messages and so on.
  - On Mac, the launcher is `MyApp.app/Contents/MacOS/JavaAppLauncher`
  - On Windows, you must pass the `"/Debug"` option to the launcher to open a window to see the trace messages.
- You can pass debug options to your application with `<fx:jvmarg>` or `<fx:property>` at package time. Examples:
  - To list all classes loaded, add the following to your `<fx:deploy>` task:
 

```
<fx:jvmarg value="-verbose:class"/>
```
  - To debug:
 

```
<fx:jvmarg value="-agentlib:jdwp=transport=dt_socket,address=4000,server=y,suspend=y"/>
```

This instructs the agent to suspend after the JVM is initialized and wait for a debugger to connect on port 4000. Consult this guide for a full description of invocation options.
  - To profile in Netbeans 7.2 on Mac OS X:
 

```
<fx:jvmarg value="-agentpath:/Applications/NetBeans/NetBeans7.2.app/Contents/Resources/NetBeans/profiler/lib/installed/jdk16/mac/libprofilerinterface.jnilib=/Applications/NetBeans/NetBeans7.2.app/Contents/Resources/NetBeans/profiler/lib,5140"/>
```

This will stop the application after it is loaded until the profiler is attached. Consult the documentation on your profiler for exact values to pass.
- Review the tips for troubleshooting standalone applications.
- Mac OS X: if other users cannot run your application, then ensure it is signed See [Section 6.3.5.1, "Mac OS X."](#)
  - To validate signing, use
 

```
codesign -v -d --verbose=4 MyApp.app
```

### 13.3.3 Web Start

- If you rebuilt the application but do not see changes in the runtime, exit the application and clear the Java cache, as follows:
  1. Run `javaws -viewer` from the command line, or open Java Control Panel manually (for example, by choosing Java in Windows Control Panel).
  2. In Java Control Panel, in the Temporary Internet Files section, click **Settings**, then click **Delete Files**.
  3. Select **Cached Applications and Applets** and click OK.
- If the application fails with an error, then check the process list to see what java process is actually used to run it. Ensure it comes from correct place.
- Consult the Java SE 7 Desktop Troubleshooting Guide for tips on how to troubleshoot generic Java Web Start problems.

### 13.3.4 Applications Embedded in the Browser

- Validate your version of Java at <http://java.com>.
- Check the JavaScript error console for errors.
- Try a different browser. See if the problem is common to the system or is browser-specific.

Remember that not all browsers are supported. For example, Chrome on Mac OS X is not supported.
- Find out the architecture of the browser you are using. Most of the browsers are 32-bit even if you are using a 64-bit platform.

Install/upgrade 32-bit Java and JavaFX runtimes to resolve the problem.
- Review the browser's list of installed plugins.
  - There should be one and only Java Plugin on the list. For JavaFX 2.2 and Java 7 update 6 the correct plugin is version 10.6.\*.
  - On Windows, there should be one and only one Deployment Toolkit plugin, and it should be the same or a higher version than that of the Java Plugin.
- Safari 6 on Mac if the `file://` protocol is used:
  - According to default Safari 6 policy, "no `file://` is allowed to open any local resources that might run code". This means that JavaFX applications will not load and appear to get stuck on the spinning wheel.
  - Run a local web server or disable local file restrictions. To do this, enable the Develop menu, then check the **Disable Local File Restrictions** menu item.
- See the Java SE 7 Desktop Troubleshooting Guide for tips on how to troubleshoot generic Java plug-in problems.

### 13.3.5 Disabling the Autoproxy Configuration in the Code

By default, JavaFX application proxy settings are taken from the current browser if the application is embedded into a web page, or system proxy settings are used. Sometimes SOCKS proxy settings, specified in the `System.setProxy` are ignored.

If you need to disable the automatic proxy configuration in the application, specify a `JavaFX-Feature-Proxy` manifest entry in the `fx:jar` with the `None` as a value as in the following example.

```
<manifest>  
  <attribute name="JavaFX-Feature-Proxy" value="None"/>  
</manifest>
```

Once you enter the `JavaFX-Feature-Proxy` manifest, the network stack will not be initialized prior to application code gets executed and you can set socks properties in the code.

