JavaFX

JavaFX Architecture and APIs

Release 2.2.21

E22902-05

April 2013

This document describes the key components of the JavaFX architecture and briefly covers some of the JavaFX APIs for media streaming, web rendering, and user interface styling.

ORACLE®

JavaFX Architecture and APIs, Release 2.2.21

E22902-05

Primary Author: Monica Pawlan

Contents

JavaFX Architecture and APIs

JavaFX is built on a layered architecture of interrelating components that supports high-performance user interfaces (UIs) that feature audio, video, graphics, and animation. Fully integrated with Oracle's implementation of the Java Platform, Standard Edition (Java SE), the JavaFX architecture provides APIs, graphics pipelines, web and media engines, and cross-platform portability.

The first part of this document describes the layers and components of the JavaFX architecture and how they interrelate. The second part introduces the JavaFX APIs for styling a JavaFX scene graph. The scene graph defines the structure and appearance of the UI for a JavaFX application.

- Layers and Components
- Scene Graph Styling

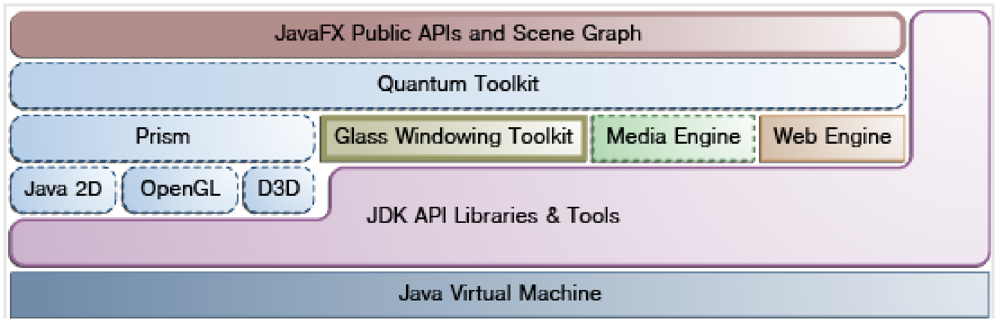
See the What is JavaFX? document for a high-level description of JavaFX features and how to get started creating a JavaFX application.

Layers and Components

Figure 1–1 shows how the JavaFX architecture components are structured in layers with the Java Virtual Machine in the bottom layer, and the JavaFX public APIs and scene graph in the top layer. The middle layers contain the graphics system in blue, the small and efficient windowing toolkit called Glass, and the media and web engines. The layered components work together to reduce the amount of code required to implement high-performance applications with modern UIs that feature audio, video, graphics, and animation.

Although the components below the top layer are not public, their descriptions in the following subsections can help you to better understand what runs a JavaFX application.

Figure 1–1 JavaFX Architecture Diagram



Scene Graph

The JavaFX scene graph is a hierarchical tree of nodes that represents all of the elements in an application UI (scene) and their relationships. Developers start JavaFX application development by using the JavaFX APIs to construct the scene graph. The scene graph simplifies UI development by separating the scene in the top layer from the operations performed on the scene, which are powered by the layers below. The scene graph and JavaFX public APIs are shown in the top layer of Figure 1–1.

With the exception of the root node of a scene graph, which has no parent, each node in a scene graph has one parent and zero or more children. Nodes in a scene graph can handle input and be rendered. Nodes that represent shapes such as rectangles and text have the geometry primitives. **Primitives** are the building blocks for other more complex shapes, and their presence in the scene graph streamlines the creation of complex shapes.

The `javafx.scene` package provides the `Node` class for creating a scene that starts with a root node, continues with the child nodes of the root node, their child nodes, and so on. The `javafx.scene.Node` class has properties and methods to define and manage the nodes in a scene graph. The following list summarizes the primary functionality available to node objects:

- Classes for 2-D and 3-D shapes, images, media, embedded web browser, text, and charts
- Unique string ID to locate the node
- A traditional coordinate system for shapes
- Transforms for translation, rotation, scaling, or shearing
- Bounding rectangle variables that define the transformed and untransformed shape
- Styling variables that apply cascading style sheet (CSS) styles to a node
- Visual effects such as blurs, shadows, color adjustment, and opacity
- Event handlers such as mouse, key, and input method
- Application-specific states for activities such as mouse hovering, button presses, and disabling and enabling nodes

In addition, the `javafx.scene` package provides classes for setting up camera rendering, ordered rendering, group rendering, mouse cursor representation, and node snapshots.

See "Scene Graph Styling" for an overview of CSS and the APIs for styling scene graph nodes. See also the Working with the JavaFX Scene Graph document.

JavaFX Public APIs

The top layer of the JavaFX architecture shown in Figure 1–1 provides a complete set of public APIs that support rich client application development. The graphics, media, and web components in the layers below provide optimizations and enhancements that reduce coding time and ensure fast, high-quality rendering and streaming. The layered architecture with the APIs and scene graph on top is particularly effective when the UI features a lot of audio, video, graphics, and animation, which is typical of JavaFX applications.

The JavaFX public APIs also:

- Rely on web standards. UI controls are styled with CSS and adhere to accessibility standards through the Web Accessibility Initiative - Rich Internet Applications (WAI-ARIA) specification.
- Make it easier for web developers to use JavaFX from other dynamic languages such as Groovy and Scala to write large or complex JavaFX applications.
- Allow the use of binding which includes support for high-performance lazy binding, binding expressions, bound sequence expressions, and partial bind reevaluation. Alternative languages such as Groovy can use the binding library to introduce binding syntax.
- Extend the Java collections library to include observable lists and maps, which allow applications to wire UI controls to data models, observe changes in those data models, and update the UI controls accordingly.

Graphics System

The JavaFX graphics system, shown in blue in Figure 1–1, supports 2-D and 3-D scene graphs and provides software rendering when the graphics hardware on a system is insufficient to support hardware-accelerated rendering. The JavaFX releases implement two accelerated graphics pipelines: Prism and Quantum Toolkit.

Prism

Prism processes render jobs and handles the rasterization and rendering of JavaFX scenes. Prism can run on hardware and software renderers, including 3-D renderers. The DirectX API libraries for multimedia and video tasks on Windows platforms include Direct3D (D3D). D3D renders 3-D graphics in applications, such as games, where performance is important.

Prism uses one of the following render paths based on the device being used:

- DirectX 9 on Windows XP and Windows Vista (hardware-accelerated path)
- DirectX 11 on Windows 7 (hardware-accelerated path)
- OpenGL on Mac, Linux, and embedded (hardware-accelerated path)
- Java2D when hardware acceleration is not possible

One of the fully hardware-accelerated paths is used when possible. Hardware acceleration enables high-performance graphics rendering because the graphics instructions are processed by the graphics processing unit (GPU) on the card rather than by a software stack. This is particularly important when rendering 3-D graphics scenes. If the system does not feature one of the recommended GPUs supported by JavaFX, then Prism defaults to the Java2D software stack. The Java2D render path is distributed in the Java Runtime Environment (JRE) for all Java SE platforms.

Quantum Toolkit

Quantum Toolkit ties Prism and the Glass Windowing Toolkit together and makes them available to the JavaFX layer above them in the stack. Quantum Toolkit also manages the threading rules related to rendering and event handling. See "Threads" in the next section.

Glass Windowing Toolkit

The Glass Windowing Toolkit (Glass), shown in the middle portion of Figure 1–1, is the lowest level of the JavaFX graphics stack. Glass is the platform-dependent layer that connects JavaFX to the native operating system with the Java Native Interface

(JNI) API. It provides native operating services such as managing the windows, timer, surfaces, and event queue.

Unlike the Abstract Window Toolkit (AWT), which manages its own event queue, Glass uses the native operating system event queue to schedule thread usage. Also unlike AWT, Glass runs on the same thread as the JavaFX application. In AWT, the native half of AWT runs on one thread, and the Java level runs on another thread. The two threads cause problems in AWT that do not exist in JavaFX because of the single JavaFX application thread approach.

Threads

The JavaFX system runs two or more of the following threads at any given time.

The **JavaFX application thread** is the primary thread used by JavaFX application developers. Any live scene, which is a scene that is part of a window, must be accessed from this thread. A scene graph can be created and manipulated in a background thread, but when its root node is attached to any live object in the scene, that scene graph must be accessed from the JavaFX application thread. This enables developers to create complex scene graphs on a background thread while keeping animations on live scenes smooth and fast. The JavaFX application thread is a different thread from the Swing and AWT event dispatch thread (EDT), so be very aware of this difference when you embed JavaFX code in Swing applications.

The **Media thread** runs in the background and synchronizes the latest frames through the scene graph by using the JavaFX application thread.

The Prism render thread handles the rendering separately from the event dispatcher. It allows frame *n* to be rendered while frame *n* +1 is being processed. Concurrent processing provides a big performance advantage, especially on modern systems with multiple processors. The Prism render thread can also have multiple rasterization threads that handle rendering work.

Pulse

A **pulse** is an event that indicates to the JavaFX scene graph that it is time to synchronize the state of the elements in the scene graph with Prism. Glass executes the pulse events using the high-resolution native timers.

A pulse is throttled at 60 frames per second (fps) maximum and executes whenever animations are running on the scene graph or when something in the scene graph changes. For example, if the user changes the position of a button, then Glass schedules a pulse.

When a pulse executes, the state of the elements in the scene graph is synchronized down to the rendering layer. This synchronization gives application developers a way to handle events asynchronously, because the system can batch and execute events on the pulse.

Layout and CSS are tied to pulse events. Numerous changes in the scene graph can lead to multiple layout or CSS updates, which can seriously degrade performance. The system automatically performs a CSS and layout pass once per pulse to avoid performance degradation. Application developers can also manually trigger layout passes as needed to take measurements prior to a pulse.

Media Engine and APIs

The media engine shown in Figure 1–1 is redesigned for performance and stability and provides consistent media behavior across platforms.

The `javafx.scene.media` package provides audio and visual media functionality that supports MP3, AIFF, and WAV audio files and flash video (FLV) files. A JavaFX `Media` object represents the media file. The media is displayed by a `MediaView` object and played by a `MediaPlayer` object.

For more information, see the Incorporating Media Assets into JavaFX Applications document.

Web Component

The Web component (labeled Web engine in Figure 1–1) is a new JavaFX UI control that is based on Webkit. The Web component provides a Web viewer and full browsing functional ity through its API.

WebKit is an open source web browser engine that supports HTML5, CSS, JavaScript, Document Object Model (DOM), and scalable vector graphics (SVG) let developers implement the following features in their applications:

- HTML content rendering from local or remote URLs
- Navigation history and backward and forward navigation
- Content reloading
- Web component effects
- HTML content editing
- JavaScript command execution
- Event handling

The `javafx.scene.web` package provides APIs for loading and displaying web content. A `WebEngine` object provides basic web page browsing capability. A `WebView` object encapsulates a `WebEngine` object, incorporates HTML content into an application scene and provides fields and methods to apply effects and transformations. The `WebView` class extends the `Node` class.

In addition, Java calls can be controlled through JavaScript and vice versa to allow developers to make the best of both environments. For more information about the JavaFX embedded browser, see the Adding HTML Content to JavaFX Applications document.

Java Virtual Machine

JavaFX applications execute in the Java runtime environment (JRE) provided by the Java Virtual Machine (JVM) shown in Figure 1–1. In addition to running compiled Java and JavaFX applications, the JVM can run programs in many other languages (for example, JRuby and Groovy).

The Java VM handles basic application tasks such as object and stack management, loading and storing variables, arithmetic, branching, method invocation and return, exception throwing, type conversions, and concurrency.

For more information, see The Java Virtual Machine Specification document.

Scene Graph Styling

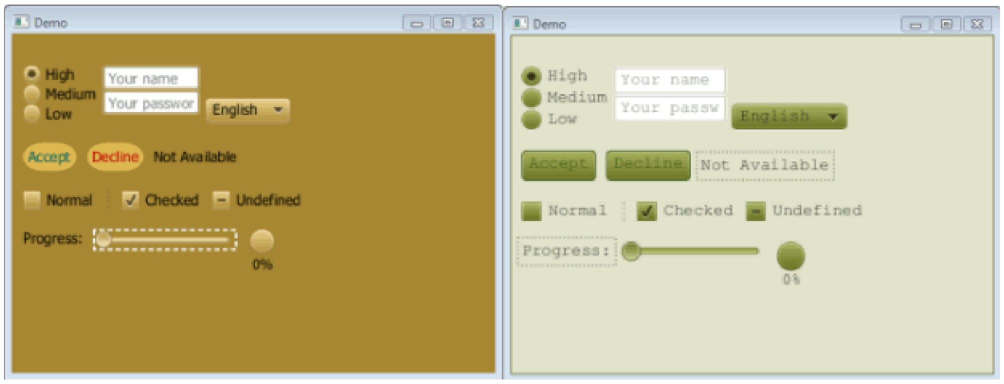
This section presents an overview of ways to style scene graph nodes to achieve the right look for your application UI. For more information, see the Working with the JavaFX Scene Graph document.

CSS

Cascading style sheets (CSS) separate appearance and style from implementation so that developers can concentrate on coding. Graphic designers can customize the appearance and style of the application through the CSS. The CSS can be applied asynchronously to any node in the JavaFX scene graph and can be assigned to the scene at runtime to change the application appearance dynamically.

Figure 1–2 demonstrates two different CSS styles applied to the same set of UI controls.

Figure 1–2 CSS Style Sheet Sample



JavaFX CSS is based on the W3C CSS version 2.1 specifications with some additions from current work on version 3. The JavaFX CSS support and extensions are designed to allow JavaFX CSS style sheets to be parsed cleanly by any compliant CSS parser, even one that does not support JavaFX extensions. This enables the mixing of CSS styles for JavaFX and for other purposes (such as for HTML pages) into a single style sheet. All JavaFX property names begin with the `-fx-` vendor extension, including those that might seem to be compatible with standard HTML CSS. The prefix is needed because some JavaFX values have slightly different semantics.

The following example is a CSS style definition that customizes the appearance of the font on buttons. The definition uses the -fx- vendor extension.

```
.custom-button {
    -fx-font: 16px "Serif";
    -fx-padding: 10;
    -fx-background-color: #CCFF99;
}
```

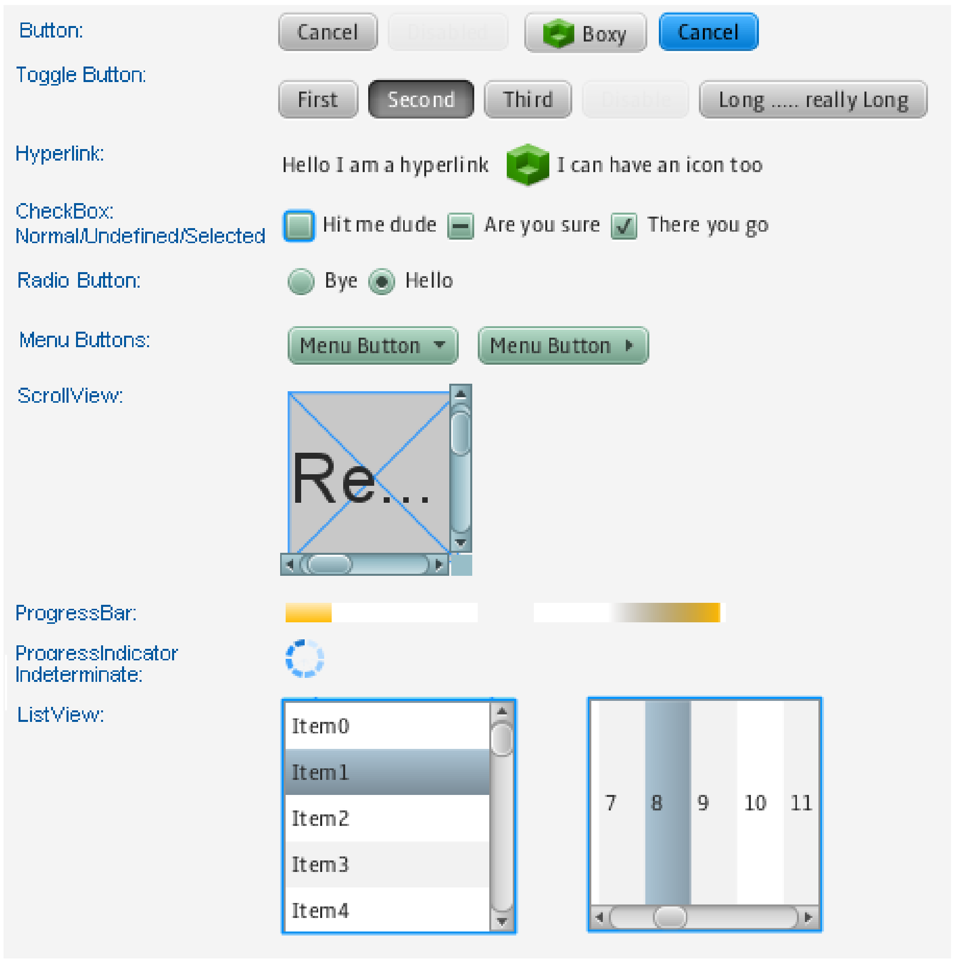
For more information about JavaFX CSS, see the Skinning JavaFX Applications with CSS article.

UI Controls

The `javafx.scene.control` package provides classes for building UI controls. Inside the application, UI controls are scene graph nodes that developers can visually enhance with any of the JavaFX APIs for styling nodes. JavaFX CSS enables theming and skinning of the UI controls.

Figure 1–3 shows some of the supported UI controls. Other Java UI controls (such as `TitlePane` or `Accordion)` were introduced with JavaFX 2.

Figure 1–3 JavaFX UI Controls Sample



For more information about all the available JavaFX UI controls, see the Using JavaFX UI Controls document and the API documentation for the `javafx.scene.control` package.

Layout

Layout containers (panes) enable flexible and dynamic arrangements of the UI controls within a scene graph of a JavaFX application. The JavaFX layout API provides the following container classes that automate common layout models. Note that multiple containers can be nested within a JavaFX application to accommodate UI content that requires different layout styles.

- The `BorderPane` class lays out its content nodes in the top, bottom, right, left, or center region.
- The `HBox` class arranges its content nodes horizontally in a single row.
- The `VBox` class arranges its content nodes vertically in a single column.
- The `StackPane` class places its content nodes in a back-to-front single stack.
- The `GridPane` class arranges its content nodes in a flexible grid of rows and columns.

- The `FlowPane` class arranges its content nodes in either a horizontal or vertical flow that wraps at the specified width (for horizontal) or height (for vertical) boundaries.
- The `TilePane` class places its content nodes in uniformly sized layout cells or tiles.
- The `AnchorPane` class anchors its content nodes to the top, bottom, left, or center of the layout.

To learn more about how to work with layouts, see the Working with Layouts in JavaFX article. For more information about the JavaFX layout API, see the API documentation for the `javax.scene.layout`.

Transformations

Each node in the JavaFX scene graph can be transformed with the following classes in the `javafx.scene.transform` package. Most of the class constructors accommodate 2-D and 3-D transformations.

- The `Translate` class moves a node from one place to another along the coordinate planes relative to its initial position.
- The `Scale` class resizes a node to appear either larger or smaller in its coordinate planes depending on the scaling factor.
- The `Shear` class rotates one axis so that the x-axis and y-axis are no longer perpendicular. The coordinates of the node are shifted by the specified multipliers. This transform applies to 2-D transformations only.
- The `Rotate` classes rotate a node around a specified pivot point of the scene.
- The `Affine` class performs a linear mapping from 2-D or 3-D coordinates to other 2-D or 3-D coordinates and preserves the straight and parallel properties of the lines. Use this class with the `Translate`, `Scale`, `Rotate`, or `Shear` classes. Do not use it directly.

To learn more about working with transformations, see the Applying Transformations in JavaFX document. For more information about the `javafx.scene.transform` API classes, see the API documentation for the `javafx.scene.transform` package.

Visual Effects

The `javafx.scene.effect` package provides classes that create rich visual effects to enhance the look of JavaFX applications, both when they load and in real time as the user interacts with the application.

An effect is a graphical algorithm that produces an image usually by modifying a source image contained within a scene graph node. Some effects change the color properties of the source image pixels, other effects combine multiple images, and other effects warp or move the pixels of the source image.

Some of the visual effects available in JavaFX include the use of the following classes:

- The `DropShadow` class renders a shadow behind the content to which the effect is applied.
- The `Reflection` class renders a reflected version of the content below the actual content.
- The `Lighting` class simulates a light source that shines on a given content and can give a flat object a more realistic, 3-D appearance.

For examples on how to use some of the available visual effects, see the Creating Visual Effects document. For more information about all the available visual effects classes, see the API documentation for the `javafx.scene.effect` package.