

# JavaFX: Getting Started with JavaFX

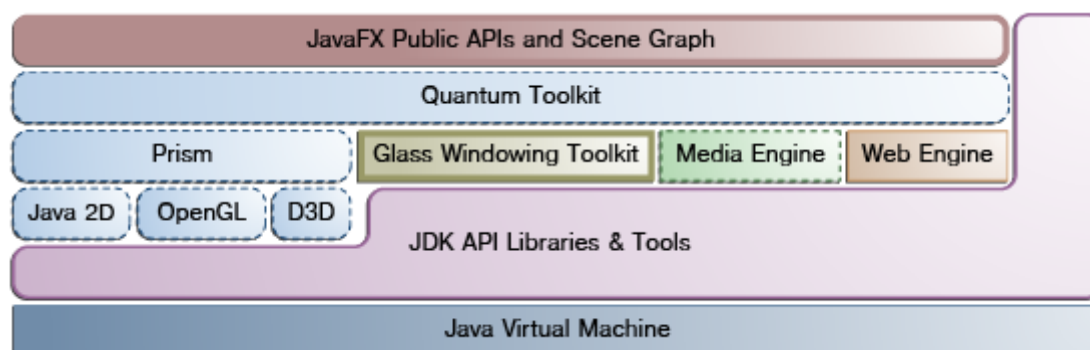
## 2 Understanding the JavaFX Architecture

The chapter gives a high level description of the JavaFX architecture and ecosystem.

[Figure 2-1](#) illustrates the architectural components of the JavaFX platform. The sections following the diagram describe each component and how the parts interconnect. Below the JavaFX public APIs lies the engine that runs your JavaFX code. It is composed of subcomponents that include a JavaFX high performance graphics engine, called Prism; a small and efficient windowing system, called Glass; a media engine, and a web engine. Although these components are not exposed publicly, their descriptions can help you to better understand what runs a JavaFX application.

- [Scene Graph](#)
- [Java Public APIs for JavaFX Features](#)
- [Graphics System](#)
- [Glass Windowing Toolkit](#)
- [Media and Images](#)
- [Web Component](#)
- [CSS](#)
- [UI Controls](#)
- [Layout](#)
- [2-D and 3-D Transformations](#)
- [Visual Effects](#)

**Figure 2-1 JavaFX Architecture Diagram**



[Description of "Figure 2-1 JavaFX Architecture Diagram"](#)

### Scene Graph

The JavaFX scene graph, shown as part of the top layer in [Figure 2-1](#), is the starting point for constructing a JavaFX application. It is a hierarchical tree of nodes that represents all of the visual elements of the application's user interface. It can handle input and can be rendered.

A single element in a scene graph is called a node. Each node has an ID, style class, and bounding volume. With the exception of the root node of a scene graph, each node in a scene graph has a single parent and zero or more children. It can also have the following:

- Effects, such as blurs and shadows
- Opacity

- Transforms
- Event handlers (such as mouse, key and input method)
- An application-specific state

Unlike in Swing and Abstract Window Toolkit (AWT), the JavaFX scene graph also includes the graphics primitives, such as rectangles and text, in addition to having controls, layout containers, images and media.

For most uses, the scene graph simplifies working with UIs, especially when rich UIs are used. Animating various graphics in the scene graph can be accomplished quickly using the `javafx.animation` APIs, and declarative methods, such as XML doc, also work well.

The `javafx.scene` API allows the creation and specification of several types of content, such as:

- **Nodes:** Shapes (2-D and 3-D), images, media, embedded web browser, text, UI controls, charts, groups, and containers
- **State:** Transforms (positioning and orientation of nodes), visual effects, and other visual state of the content
- **Effects:** Simple objects that change the appearance of scene graph nodes, such as blurs, shadows, and color adjustment

For more information, see the [Working with the JavaFX Scene Graph](#) document.

## Java Public APIs for JavaFX Features

The top layer of the JavaFX architecture shown in [Figure 2-1](#) provides a complete set of Java public APIs that support rich client application development. These APIs provide unparalleled freedom and flexibility to construct rich client applications. The JavaFX platform combines the best capabilities of the Java platform with comprehensive, immersive media functionality into an intuitive and comprehensive one-stop development environment. These Java APIs for JavaFX features:

- Allow the use of powerful Java features, such as generics, annotations, multithreading, and Lambda Expressions (introduced in Java SE 8).
- Make it easier for Web developers to use JavaFX from other JVM-based dynamic languages, such as Groovy and JavaScript.
- Allow Java developers to use other system languages, such as Groovy, for writing large or complex JavaFX applications.
- Allow the use of binding which includes support for the high performance lazy binding, binding expressions, bound sequence expressions, and partial bind reevaluation. Alternative languages (like Groovy) can use this binding library to introduce binding syntax similar to that of JavaFX Script.
- Extend the Java collections library to include observable lists and maps, which allow applications to wire user interfaces to data models, observe changes in those data models, and update the corresponding UI control accordingly.

The JavaFX APIs and programming model are a continuation of the JavaFX 1.x product line. Most of the JavaFX APIs have been ported directly to Java. Some APIs, such as Layout and Media, along with many other details, have been improved and simplified based on feedback received from users of the JavaFX 1.x release. JavaFX relies more on web standards, such as CSS for styling controls and ARIA for accessibility specifications. The use of additional web standards is also under review.

## Graphics System

The JavaFX Graphics System, shown in blue in [Figure 2-1](#), is an implementation detail beneath the JavaFX scene graph layer. It supports both 2-D and 3-D scene graphs. It provides software rendering when the graphics hardware on a system is insufficient to support hardware accelerated rendering.

Two graphics accelerated pipelines are implemented on the JavaFX platform:

- **Prism** processes render jobs. It can run on both hardware and software renderers, including 3-D. It is responsible for rasterization and rendering of JavaFX scenes. The following multiple render paths are possible based on the device being used:
  - DirectX 9 on Windows XP and Windows Vista
  - DirectX 11 on Windows 7
  - OpenGL on Mac, Linux, Embedded

- Software rendering when hardware acceleration is not possible

The fully hardware accelerated path is used when possible, but when it is not available, the software render path is used because the software render path is already distributed in all of the Java Runtime Environments (JREs). This is particularly important when handling 3-D scenes. However, performance is better when the hardware render paths are used.

- **Quantum Toolkit** ties Prism and Glass Windowing Toolkit together and makes them available to the JavaFX layer above them in the stack. It also manages the threading rules related to rendering versus events handling.

## Glass Windowing Toolkit

The Glass Windowing Toolkit, shown in beige in the middle portion of [Figure 2-1](#), is the lowest level in the JavaFX graphics stack. Its main responsibility is to provide native operating services, such as managing the windows, timers, and surfaces. It serves as the platform-dependent layer that connects the JavaFX platform to the native operating system.

The Glass toolkit is also responsible for managing the event queue. Unlike the Abstract Window Toolkit (AWT), which manages its own event queue, the Glass toolkit uses the native operating system's event queue functionality to schedule thread usage. Also unlike AWT, the Glass toolkit 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. This introduces a lot of issues, many of which are resolved in JavaFX by using the single JavaFX application thread approach.

## Threads

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

- **JavaFX application thread:** This 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 care must be taken when embedding JavaFX code into Swing applications.
- **Prism render thread:** This thread handles the rendering separately from the event dispatcher. It allows frame N to be rendered while frame N+1 is being processed. This ability to perform concurrent processing is a big advantage, especially on modern systems that have multiple processors. The Prism render thread may also have multiple rasterization threads that help off-load work that needs to be done in rendering.
- **Media thread:** This thread runs in the background and synchronizes the latest frames through the scene graph by using the JavaFX application thread.

## Pulse

A pulse is an event that indicates to the JavaFX scene graph that it is time to synchronize the state of the elements on the scene graph with Prism. A pulse is throttled at 60 frames per second (fps) maximum and is fired whenever animations are running on the scene graph. Even when animation is not running, a pulse is scheduled when something in the scene graph is changed. For example, if a position of a button is changed, a pulse is scheduled.

When a pulse is fired, the state of the elements on the scene graph is synchronized down to the rendering layer. A pulse enables application developers a way to handle events asynchronously. This important feature allows the system to batch and execute events on the pulse.

Layout and CSS are also tied to pulse events. Numerous changes in the scene graph could lead to multiple layout or CSS updates, which could 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.

The Glass Windowing Toolkit is responsible for executing the pulse events. It uses the high-resolution native timers to make the execution.

## Media and Images

JavaFX media functionality is available through the `javafx.scene.media` APIs. JavaFX supports both visual and audio media. Support is provided for MP3, AIFF, and WAV audio files and FLV video files. JavaFX media functionality is provided as three separate components: the `Media` object represents a media file, the `MediaPlayer` plays a media file, and a `MediaView` is a node that displays the media.

The Media Engine component, shown in green in [Figure 2-1](#), has been designed with performance and stability in mind and provides consistent behavior across platforms. For more information, read the [Incorporating Media Assets into JavaFX Applications](#) document.

## Web Component

The Web component is a JavaFX UI control, based on Webkit, that provides a Web viewer and full browsing functionality through its API. This Web Engine component, shown in orange in [Figure 2-1](#), is based on WebKit, which is an open source web browser engine that supports HTML5, CSS, JavaScript, DOM, and SVG. It enables developers to implement the following features in their Java applications:

- Render HTML content from local or remote URL
- Support history and provide Back and Forward navigation
- Reload the content
- Apply effects to the web component
- Edit the HTML content
- Execute JavaScript commands
- Handle events

This embedded browser component is composed of the following classes:

- **WebEngine** provides basic web page browsing capability.
- **WebView** encapsulates a WebEngine object, incorporates HTML content into an application's scene, and provides fields and methods to apply effects and transformations. It is an extension of a **Node** class.

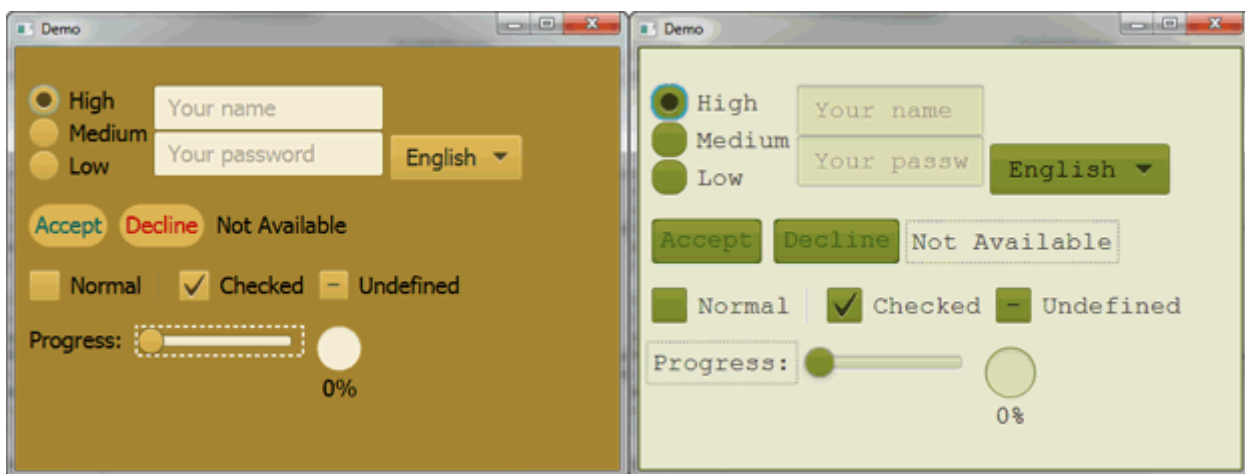
In addition, Java calls can be controlled through JavaScript and vice versa to allow developers to make the best of both environments. For more detailed overview of the JavaFX embedded browser, see the [Adding HTML Content to JavaFX Applications](#) document.

## CSS

JavaFX Cascading Style Sheets (CSS) provides the ability to apply customized styling to the user interface of a JavaFX application without changing any of that application's source code. CSS can be applied to any node in the JavaFX scene graph and are applied to the nodes asynchronously. JavaFX CSS styles can also be easily assigned to the scene at runtime, allowing an application's appearance to dynamically change.

[Figure 2-2](#) demonstrates the application of two different CSS styles to the same set of UI controls.

**Figure 2-2 CSS Style Sheet Sample**



[Description of "Figure 2-2 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 have been 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 are prefixed with a vendor extension of ”-fx-”, including those that might seem to be compatible with standard HTML CSS, because some JavaFX values have slightly different semantics.

For more detailed information about JavaFX CSS, see the [Skinning JavaFX Applications with CSS](#) document.

## UI Controls

The JavaFX UI controls available through the JavaFX API are built by using nodes in the scene graph. They can take full advantage of the visually rich features of the JavaFX platform and are portable across different platforms. JavaFX CSS allows for theming and skinning of the UI controls.

[Figure 2-3](#) shows some of the UI controls that are currently supported. These controls reside in the `javafx.scene.control` package.

**Figure 2-3 JavaFX UI Controls Sample**



[Description of "Figure 2-3 JavaFX UI Controls Sample"](#)

For more detailed information about all the available JavaFX UI controls, see the [Using JavaFX UI Controls](#) and the [API documentation](#) for the `javafx.scene.control` package.

## Layout

Layout containers or panes can be used to allow for flexible and dynamic arrangements of the UI controls within a scene graph of a JavaFX application. The JavaFX Layout API includes the following container classes that automate common layout models:

- 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 enables the developer to create a flexible grid of rows and columns in which to lay out content nodes.
- The `FlowPane` class arranges its content nodes in either a horizontal or vertical "flow," wrapping 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 enables developers to create anchor nodes to the top, bottom, left side, or center of the layout.

To achieve a desired layout structure, different containers can be nested within a JavaFX application.

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 `javafx.scene.layout` package.

## 2-D and 3-D Transformations

Each node in the JavaFX scene graph can be transformed in the x-y coordinate using the following `javafx.scene.transform` classes:

- `translate` – Move a node from one place to another along the x, y, z planes relative to its initial position.
- `scale` – Resize a node to appear either larger or smaller in the x, y, z planes, depending on the scaling factor.
- `shear` – Rotate 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.
- `rotate` – Rotate a node about a specified pivot point of the scene.
- `affine` – Perform a linear mapping from 2-D/3-D coordinates to other 2-D/3-D coordinates while preserving the 'straight' and 'parallel' properties of the lines. This class should be used with `Translate`, `Scale`, `Rotate`, or `ShearTransform` classes instead of being used 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](#).

## Visual Effects

The development of rich client interfaces in the JavaFX scene graph involves the use of Visual Effects or Effects to enhance the look of JavaFX applications in real time. The JavaFX Effects are primarily image pixel-based and, hence, they take the set of nodes that are in the scene graph, render it as an image, and apply the specified effects to it.

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

- `DropShadow` – Renders a shadow of a given content behind the content to which the effect is applied.
- `Reflection` – Renders a reflected version of the content below the actual content.
- `Lighting` – Simulates a light source shining on a given content and can give a flat object a more realistic, three-dimensional 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.