JavaFX

Getting Started with JavaFX

Release 8

E50607-02

August 2014

Get started with JavaFX by getting an overview of the available features, learning the architecture, and creating simple applications that introduce you to layouts, CSS, FXML, visual effects, and animation.

JavaFX Getting Started with JavaFX, Release 8

E50607-02

Contributing Author: Jasper Potts, Nancy Hildebrandt, Joni Gordon, Cindy Castillo

Contents

Preface

This preface gives an overview about this tutorial and also describes the document accessibility features and conventions used in this tutorial - *Getting Started with JavaFX*

About This Tutorial

This tutorial is a compilation of three documents that were previously delivered with the JavaFX 2.x documentation set: JavaFX Overview, JavaFX Architecture, and Getting Started with JavaFX. The combined content has been enhanced with updated information about the new JavaFX features included with the Java SE 8 release. This document contains the following parts:

- What Is JavaFX?
- Getting Started with JavaFX Sample Applications

Each part contains chapters that introduce you to the JavaFX technology and gets you started in learning how to use it for your application development.

Audience

This document is intended for JavaFX developers.

Documentation Accessibility

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

Access to Oracle Support

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

Related Documents

For more information, see the rest of the JavaFX documentation set at http://docs.oracle.com/javase/javase-clienttechnologies.htm.

Conventions

The following text conventions are used in this document:

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

What's New

This chapter summarizes the new features and significant product changes made in the JavaFX component of the Java SE 8 release.

- The new Modena theme is now the default theme for JavaFX applications. See the Modena theme section of Key Features.
- Support for additional HTML5 features has been added. See Adding HTML Content to JavaFX Applications for more information.
- The new `SwingNode` class improves the Swing interoperability feature. See Embedding Swing Content in JavaFX Applications.
- New built-in UI controls, `DatePicker` and `TableView`, are now available. See Using JavaFX UI Controls document for more information.
- 3D Graphics library has been enhanced with several new API classes. See 3D Graphics features section of Key Features and Getting Started with JavaFX 3D Graphics for more information.
- The `javafx.print` package is now available and provides the public JavaFX printing APIs.
- Rich text support has been added.
- Support for Hi-DPI displays have been made available.
- CSS styleable classes became public APIs.
- Scheduled service class has been introduced.

Part I

What Is JavaFX?

Part I contains the following chapters:

1 JavaFX Overview

This chapter provides an overview of the types of applications you can build using JavaFX APIs, where to download the JavaFX libraries, and a high level information about the key JavaFX features being delivered.

JavaFX is a set of graphics and media packages that enables developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms.

See the Understanding the JavaFX Architecture chapter to learn about the JavaFX platform architecture and to get a brief description of the JavaFX APIs for media streaming, web rendering, and user interface styling.

JavaFX Applications

Since the JavaFX library is written as a Java API, JavaFX application code can reference APIs from any Java library. For example, JavaFX applications can use Java API libraries to access native system capabilities and connect to server-based middleware applications.

The look and feel of JavaFX applications can be customized. Cascading Style Sheets (CSS) separate appearance and style from implementation so that developers can concentrate on coding. Graphic designers can easily customize the appearance and style of the application through the CSS. If you have a web design background, or if you would like to separate the user interface (UI) and the back-end logic, then you can develop the presentation aspects of the UI in the FXML scripting language and use Java code for the application logic. If you prefer to design UIs without writing code, then use JavaFX Scene Builder. As you design the UI, Scene Builder creates FXML markup that can be ported to an Integrated Development Environment (IDE) so that developers can add the business logic.

Availability

The JavaFX APIs are available as a fully integrated feature of the Java SE Runtime Environment (JRE) and the Java Development Kit (JDK). Because the JDK is available for all major desktop platforms (Windows, Mac OS X, and Linux), JavaFX applications compiled to JDK 7 and later also run on all the major desktop platforms. Support for ARM platforms has also been made available with JavaFX 8. JDK for ARM includes the base, graphics and controls components of JavaFX.

The cross-platform compatibility enables a consistent runtime experience for JavaFX applications developers and users. Oracle ensures synchronized releases and updates on all platforms and offers an extensive support program for companies that run mission-critical applications.

On the JDK download page, you can get a zip file of JavaFX sample applications. The sample applications provide many code samples and snippets that show by example how to write JavaFX applications. See "How Do I Run a Sample Application?" for more information.

Key Features

The following features are included in JavaFX 8 and later releases. Items that were introduced in JavaFX 8 release are indicated accordingly:

- **Java APIs**. JavaFX is a Java library that consists of classes and interfaces that are written in Java code. The APIs are designed to be a friendly alternative to Java Virtual Machine (Java VM) languages, such as JRuby and Scala.
- **FXML and Scene Builder**. FXML is an XML-based declarative markup language for constructing a JavaFX application user interface. A designer can code in FXML or use JavaFX Scene Builder to interactively design the graphical user interface (GUI). Scene Builder generates FXML markup that can be ported to an IDE where a developer can add the business logic.
- **WebView**. A web component that uses WebKitHTML technology to make it possible to embed web pages within a JavaFX application. JavaScript running in WebView can call Java APIs, and Java APIs can call JavaScript running in WebView. Support for additional HTML5 features, including Web Sockets, Web Workers, and Web Fonts, and printing capabilities have been added in JavaFX 8. See Adding HTML Content to JavaFX Applications.
- **Swing interoperability**. Existing Swing applications can be updated with JavaFX features, such as rich graphics media playback and embedded Web content. The `SwingNode` class, which enables you to embed Swing content into JavaFX applications, has been added in JavaFX 8. See the SwingNode API javadoc and Embedding Swing Content in JavaFX Applications for more information.
- **Built-in UI controls and CSS**. JavaFX provides all the major UI controls that are required to develop a full-featured application. Components can be skinned with standard Web technologies such as CSS. The DatePicker and TreeTableView UI controls are now available with the JavaFX 8 release. See Using JavaFX UI Controls for more information. Also, the CSS Styleable* classes have become public API, allowing objects to be styled by CSS.
- **Modena theme.** The Modena theme replaces the Caspian theme as the default for JavaFX 8 applications. The Caspian theme is still available for your use by adding the `setUserAgentStylesheet(STYLESHEET_CASPIAN)` line in your Application start() method. For more information, see the Modena blog at fxexperience.com

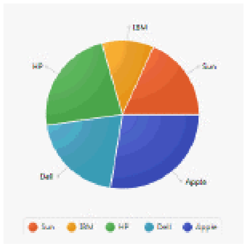
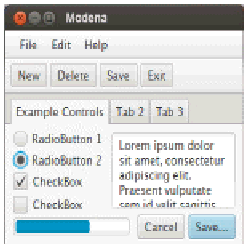
- **3D Graphics Features**. The new API classes for `Shape3D` (`Box, Cylinder, MeshView, and Sphere` subclasses), `SubScene, Material, PickResult, LightBase (AmbientLight` and `PointLight` subclasses), and `SceneAntialiasing` have been added to the 3D Graphics library in JavaFX 8. The `Camera` API class has also been updated in this release. For more information, see the Getting Started with JavaFX 3D Graphics document and the corresponding API javadoc for `javafx.scene.shape.Shape3D`, `javafx.scene.SubScene`, `javafx.scene.paint.Material`, `javafx.scene.input.PickResult`, and `javafx.scene.SceneAntialiasing`.
- **Canvas API**. The Canvas API enables drawing directly within an area of the JavaFX scene that consists of one graphical element (node).
- **Printing API.** The `javafx.print` package has been added in Java SE 8 release and provides the public classes for the JavaFX Printing API.
- **Rich Text Support**. JavaFX 8 brings enhanced text support to JavaFX, including bi-directional text and complex text scripts, such as Thai and Hindu in controls, and multi-line, multi-style text in text nodes.
- **Multitouch Support**. JavaFX provides support for multitouch operations, based on the capabilities of the underlying platform.
- **Hi-DPI support.** JavaFX 8 now supports Hi-DPI displays.
- **Hardware-accelerated graphics pipeline**. JavaFX graphics are based on the graphics rendering pipeline (Prism). JavaFX offers smooth graphics that render quickly through Prism when it is used with a supported graphics card or graphics processing unit (GPU). If a system does not feature one of the recommended GPUs supported by JavaFX, then Prism defaults to the software rendering stack.
- **High-performance media engine**. The media pipeline supports the playback of web multimedia content. It provides a stable, low-latency media framework that is based on the GStreamer multimedia framework.
- **Self-contained application deployment model.** Self-contained application packages have all of the application resources and a private copy of the Java and JavaFX runtimes. They are distributed as native installable packages and provide the same installation and launch experience as native applications for that operating system.

What Can I Build with JavaFX?

With JavaFX, you can build many types of applications. Typically, they are network-aware applications that are deployed across multiple platforms and display information in a high-performance modern user interface that features audio, video, graphics, and animation.

Table 1–1 shows images of a few of the sample JavaFX applications that are included with the JavaFX 8.*n* release.

Table 1–1 Sample JavaFX Applications

Sample Application	Description
	**JavaFX Ensemble8** Ensemble8 is a gallery of sample applications that demonstrate a large variety of JavaFX features, including animation, charts, and controls. You can view and interact with each running sample on ALL platforms, and read its descriptions. On the desktop platforms, you can copy each sample's source code, adjust the properties of the sample components used in several samples, and follow links to the relevant API documentation when you're connected to the Internet. Ensemble8 also runs with JavaFX for ARM.
 Modena File Edit Help New Delete Save Exit Example Controls Tab 2 Tab 3 RadioButton 1 RadioButton 2 CheckBox CheckBox Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent vulputate sem id velit sagittis Cancel Save...	**Modena** Modena is a sample application that demonstrates the look and feel of UI components using the Modena theme. It gives you the option to contrast Modena and Caspian themes, and explore various aspects of these themes.
	3D Viewer 3DViewer is a sample application that allows you to navigate and examine a 3D scene with a mouse or a trackpad. 3DViewer has importers for a subset of the features in OBJ and Maya files. The ability to import animation is also provided for Maya files. (Note that in the case of Maya files, construction history should be deleted on all the objects when saving as a Maya file.) 3DViewer also has the ability to export the contents of the scene as Java or FXML files.

How Do I Run a Sample Application?

The steps in this section explain how to download and run the sample applications that are available as a separate download with the Java Platform (JDK 8).

Note: Before you can run a sample JavaFX application, you need to have the JavaFX runtime libraries on your machine. Before you proceed with these steps, either install the latest version of the JDK 8 or the latest version of the JRE.

To download and run the sample applications:

1. Go to the Java SE Downloads page at `http://www.oracle.com/technetwork/java/javase/downloads/`.
2. Scroll down to locate the JDK 8 and JavaFX Demos and Samples section.
3. Click the Demos and Samples **Download** button to go to the downloads page.
4. On the Java SE Development Kit 8 Downloads page, scroll down to the JavaFX Demos and Samples Downloads section.
5. Download the zip file for the correct operating system and extract the files.

The `javafx-samples-8.x` directory is created and contains the files for the available samples. The NetBeans projects for the samples are in the `javafx-samples-8.x\src` directory.

6. Double-click the executable file for a sample.

   For example, to run the Ensemble8 pre-built sample application, double-click the `Ensemble8.jar` file.

How Do I Run a Sample in an IDE?

You can use several Java development IDEs to develop JavaFX applications. The following steps explain how to view and run the source code in the NetBeans IDE.

To view and run the sample source code in NetBeans IDE:

1. Download the samples, as described above, and extract the files.
2. From a NetBeans 7.4 or later IDE, load the project for the sample you want to view.
   a. From the **File** menu, select **Open Project**.
   b. In the **Open Project** dialog box, navigate to the directory that lists the samples. The navigation path looks something like this:

      ```
      ..\javafx_samples-8.x-<platform>\javafx-samples-8.x\src
      ```

   c. Select the sample you want to view.
   d. Click the **Open Project** button.
3. In the Projects window, right click the project you just opened and select **Run**. Notice the Output window is updated and the sample project is run and deployed.

How Do I Create a JavaFX Application?

Because JavaFX applications are written in the Java language, you can use your favorite editor or any integrated development environment (IDE) that supports the Java language (such as NetBeans, Eclipse, or IntelliJ IDEA) to create JavaFX applications.

To create JavaFX applications:

1. Go to the Java SE Downloads page at `http://www.oracle.com/technetwork/java/javase/downloads/` to download the Oracle® JDK 8 with JavaFX 8.*n* support. Links to the certified system configurations and release notes are also available on that page..
2. Use Getting Started with JavaFX Sample Applications to create simple applications that demonstrates how to work with layouts, style sheets, and visual effects.
3. Use JavaFX Scene Builder to design the UI for your JavaFX application without coding. You can drag and drop UI components to a work area, modify their properties, apply style sheets, and integrate the resulting code with their application logic.
   a. Download the JavaFX Scene Builder from the JavaFX Downloads page at `http://www.oracle.com/technetwork/java/javase/downloads/`.
   b. Follow the Getting Started with JavaFX Scene Builder tutorial to learn more.

Resources

Use the following resources to learn more about the JavaFX technology.

- Download the latest JDK 8 release and the JavaFX samples from the Java SE Downloads page at:
  `http://www.oracle.com/technetwork/java/javase/downloads/`.
- Read Understanding the JavaFX Architecture.
- Browse JavaFX tutorials and articles for developers.

2

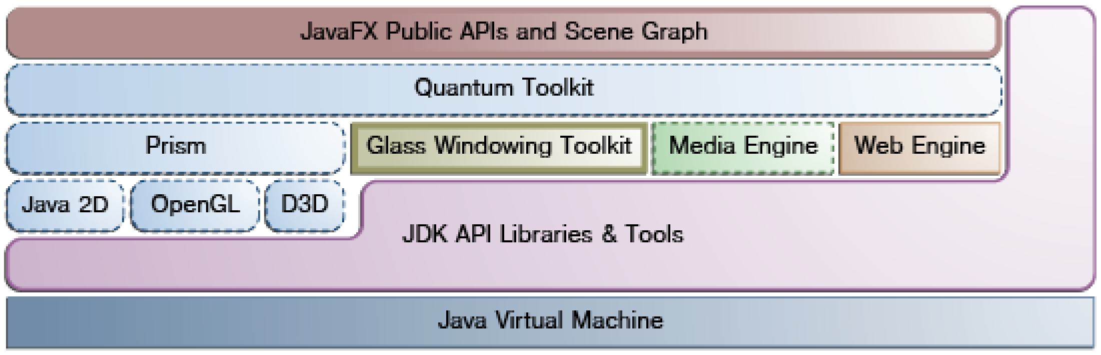
Understanding the JavaFX Architecture

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



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 Lamda 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.

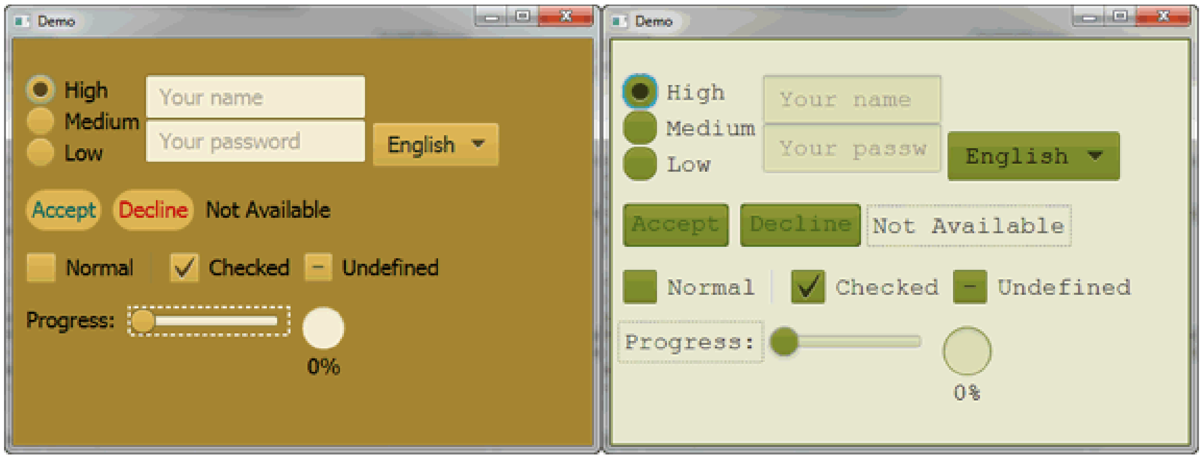
In addition, Java calls can be controlled through JavaScript and vice versa to allow developers to make the best of both environments. For more detailed overview of the JavaFX embedded browser, see the Adding HTML Content to JavaFX Applications document.

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



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.

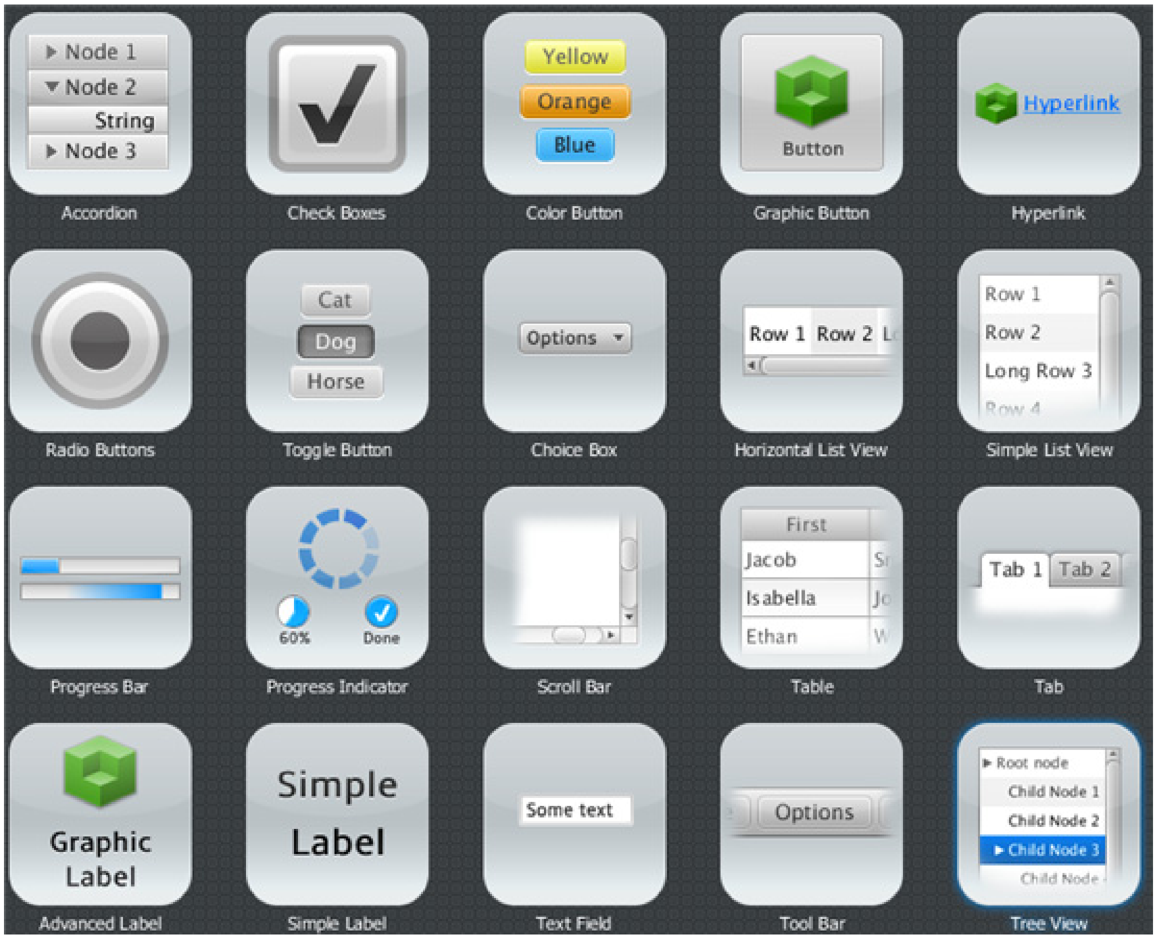
For more detailed information about JavaFX CSS, see the Skinning JavaFX Applications with CSS document.

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



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.tranform` 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 `Shear` transform 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:

- `Drop Shadow` – 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.

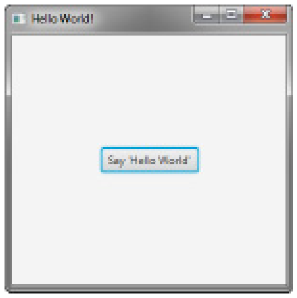
For examples on how to use some of the available visual effects, see the Creating Visual Effects document. For more information about all the available visual effects classes, see the API documentation for the `javafx.scene.effect` package.

Part II

Getting Started with JavaFX Sample Applications

This collection of sample applications is designed to get you started with common JavaFX tasks, including working with layouts, controls, style sheets, FXML, and visual effects.

Hello World, JavaFX Style

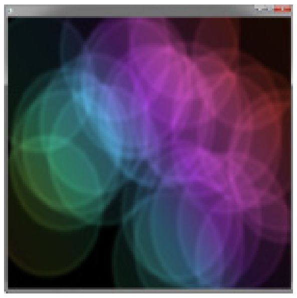


Form Design in JavaFX

Fancy Design with CSS

User Interface Design with FXML

Animated Shapes and Visual Effects



3

Hello World, JavaFX Style

The best way to teach you what it is like to create and build a JavaFX application is with a “Hello World” application. An added benefit of this tutorial is that it enables you to test that your JavaFX technology is properly installed.

The tool used in this tutorial is NetBeans IDE 7.4. Before you begin, ensure that the version of NetBeans IDE that you are using supports JavaFX 8. See the Certified System Configurations section of the Java SE 8 downloads page for details.

Construct the Application

1. From the **File** menu, choose **New Project**.
2. In the **JavaFX** application category, choose **JavaFX Application**. Click **Next**.
3. Name the project **HelloWorld** and click **Finish**.

   NetBeans opens the `HelloWorld.java` file and populates it with the code for a basic Hello World application, as shown in Example 3–1.

Example 3–1 Hello World

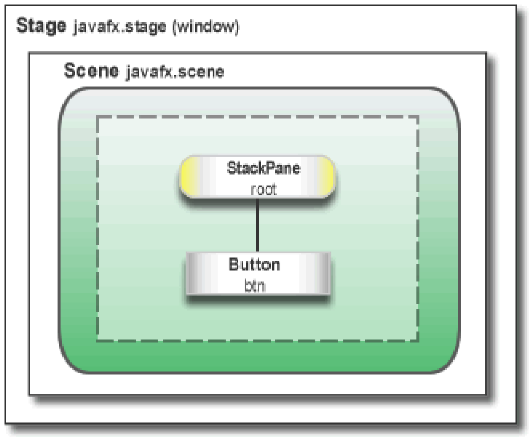
```
package helloworld;

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

public class HelloWorld extends Application {

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

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

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

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

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

Here are the important things to know about the basic structure of a JavaFX application:

- The main class for a JavaFX application extends the `javafx.application.Application` class. The `start()` method is the main entry point for all JavaFX applications.
- A JavaFX application defines the user interface container by means of a stage and a scene. The JavaFX `Stage` class is the top-level JavaFX container. The JavaFX `Scene` class is the container for all content. Example 3–1 creates the stage and scene and makes the scene visible in a given pixel size.
- In JavaFX, the content of the scene is represented as a hierarchical scene graph of nodes. In this example, the root node is a `StackPane` object, which is a resizable layout node. This means that the root node's size tracks the scene's size and changes when the stage is resized by a user.
- The root node contains one child node, a button control with text, plus an event handler to print a message when the button is pressed.
- The `main()` method is not required for JavaFX applications when the JAR file for the application is created with the JavaFX Packager tool, which embeds the JavaFX Launcher in the JAR file. However, it is useful to include the `main()` method so you can run JAR files that were created without the JavaFX Launcher, such as when using an IDE in which the JavaFX tools are not fully integrated. Also, Swing applications that embed JavaFX code require the `main()` method.

Figure 3–1 shows the scene graph for the Hello World application. For more information on scene graphs see Working with the JavaFX Scene Graph.

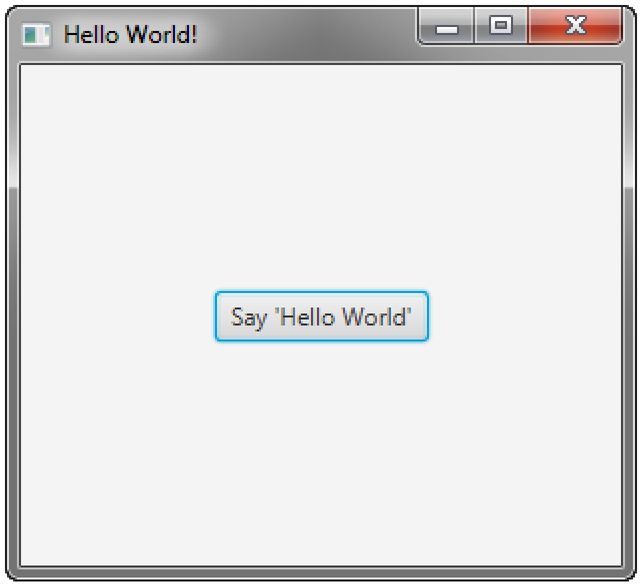
Figure 3–1 Hello World Scene Graph



Run the Application

1. In the Projects window, right-click the **HelloWorld** project node and choose **Run**.
2. Click the Say Hello World button.
3. Verify that the text "Hello World!" is printed to the NetBeans output window. Figure 3–2 shows the Hello World application, JavaFX style.

Figure 3–2 Hello World, JavaFX style



Where to Go Next

This concludes the basic Hello World tutorial, but continue reading for more lessons on developing JavaFX applications:

- Creating a Form in JavaFX teaches the basics of screen layout, how to add controls to a layout, and how to create input events.

- Fancy Forms with JavaFX CSS provides simple style tricks for enhancing your application, including adding a background image and styling buttons and text.
- Using FXML to Create a User Interface shows an alternate method for creating the login user interface. FXML is an XML-based language that provides the structure for building a user interface separate from the application logic of your code.
- Animation and Visual Effects in JavaFX shows how to bring an application to life by adding timeline animation and blend effects.

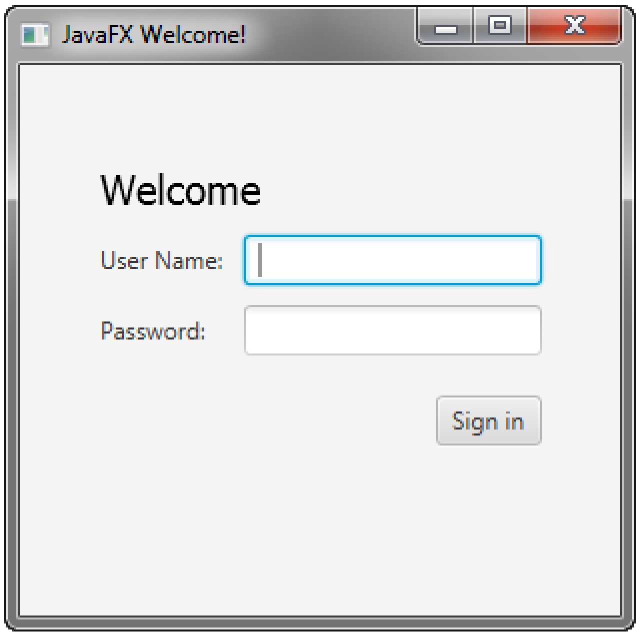
4

Creating a Form in JavaFX

Creating a form is a common activity when developing an application. This tutorial teaches you the basics of screen layout, how to add controls to a layout pane, and how to create input events.

In this tutorial, you will use JavaFX to build the login form shown in Figure 4–1.

Figure 4–1 Login Form



The tool used in this Getting Started tutorial is NetBeans IDE. Before you begin, ensure that the version of NetBeans IDE that you are using supports JavaFX 8. See the Certified System Configurations page of the Java SE Downloads page for details.

Create the Project

Your first task is to create a JavaFX project in NetBeans IDE and name it Login:

1. From the **File** menu, choose **New Project**.
2. In the **JavaFX** application category, choose **JavaFX Application**. Click **Next**.
3. Name the project **Login** and click **Finish**.

When you create a JavaFX project, NetBeans IDE provides a Hello World application as a starting point, which you have already seen if you followed the Hello World tutorial.

4. Remove the `start()` method that NetBeans IDE generated and replace it with the code in Example 4–1.

Example 4–1   Application Stage

```
@Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX Welcome");

        primaryStage.show();
    }
```

Tip: After you add sample code into a NetBeans project, press Ctrl (or Cmd) + Shift + I to import the required packages. When there is a choice of import statements, choose the one that starts with `javafx`.

Create a GridPane Layout

For the login form, use a `GridPane` layout because it enables you to create a flexible grid of rows and columns in which to lay out controls. You can place controls in any cell in the grid, and you can make controls span cells as needed.

The code to create the `GridPane` layout is in Example 4–2. Add the code before the line `primaryStage.show();`

Example 4–2   GridPane with Gap and Padding Properties

```
GridPane grid = new GridPane();
grid.setAlignment(Pos.CENTER);
grid.setHgap(10);
grid.setVgap(10);
grid.setPadding(new Insets(25, 25, 25, 25));

Scene scene = new Scene(grid, 300, 275);
primaryStage.setScene(scene);
```

Example 4–2 creates a `GridPane` object and assigns it to the variable named `grid`. The alignment property changes the default position of the grid from the top left of the scene to the center. The gap properties manage the spacing between the rows and columns, while the padding property manages the space around the edges of the grid pane. The insets are in the order of top, right, bottom, and left. In this example, there are `25` pixels of padding on each side.

The scene is created with the grid pane as the root node, which is a common practice when working with layout containers. Thus, as the window is resized, the nodes within the grid pane are resized according to their layout constraints. In this example, the grid pane remains in the center when you grow or shrink the window. The padding properties ensure there is a padding around the grid pane when you make the window smaller.

This code sets the scene width and height to 300 by 275. If you do not set the scene dimensions, the scene defaults to the minimum size needed to display its contents.

Add Text, Labels, and Text Fields

Looking at Figure 4–1, you can see that the form requires the title “Welcome “and text and password fields for gathering information from the user. The code for creating these controls is in Example 4–3. Add this code after the line that sets the grid padding property.

Example 4–3 Controls

```
Text scenetitle = new Text("Welcome");
scenetitle.setFont(Font.font("Tahoma", FontWeight.NORMAL, 20));
grid.add(scenetitle, 0, 0, 2, 1);

Label userName = new Label("User Name:");
grid.add(userName, 0, 1);

TextField userTextField = new TextField();
grid.add(userTextField, 1, 1);

Label pw = new Label("Password:");
grid.add(pw, 0, 2);

PasswordField pwBox = new PasswordField();
grid.add(pwBox, 1, 2);
```

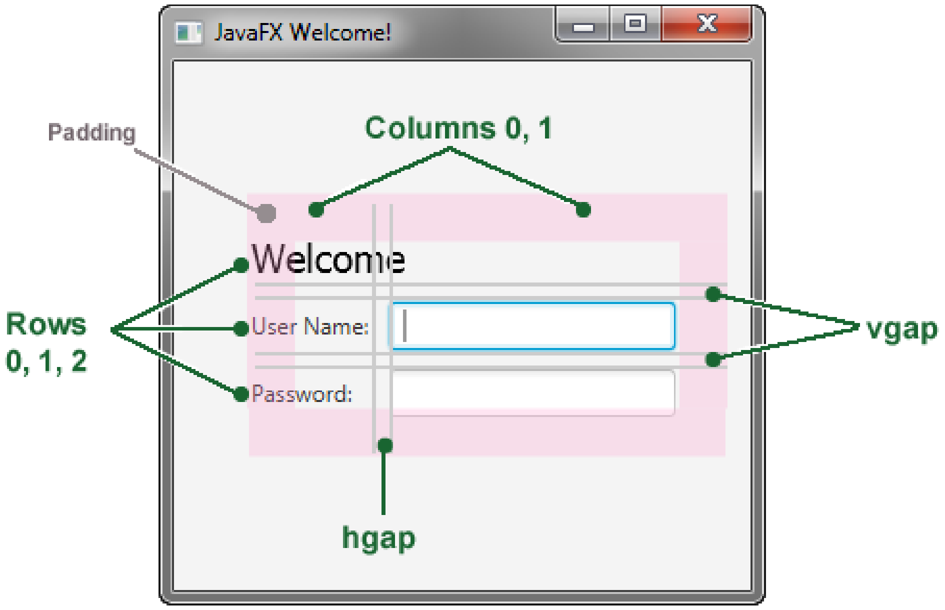
The first line creates a `Text` object that cannot be edited, sets the text to `Welcome`, and assigns it to a variable named `scenetitle`. The next line uses the `setFont()` method to set the font family, weight, and size of the `scenetitle` variable. Using an inline style is appropriate where the style is bound to a variable, but a better technique for styling the elements of your user interface is by using a style sheet. In the next tutorial, Fancy Forms with JavaFX CSS, you will replace the inline style with a style sheet.

The `grid.add()` method adds the `scenetitle` variable to the layout `grid`. The numbering for columns and rows in the grid starts at zero, and `scenetitle` is added in column 0, row 0. The last two arguments of the `grid.add()` method set the column span to 2 and the row span to 1.

The next lines create a `Label` object with text `User Name` at column 0, row 1 and a `Text Field` object that can be edited. The text field is added to the grid pane at column 1, row 1. A password field and label are created and added to the grid pane in a similar fashion.

When working with a grid pane, you can display the grid lines, which is useful for debugging purposes. In this case, you can add `grid.setGridLinesVisible(true)` after the line that adds the password field. Then, when you run the application, you see the lines for the grid columns and rows as well as the gap properties, as shown in Figure 4–2.

Figure 4–2 Login Form with Grid Lines



Add a Button and Text

The final two controls required for the application are a `Button` control for submitting the data and a `Text` control for displaying a message when the user presses the button.

First, create the button and position it on the bottom right, which is a common placement for buttons that perform an action affecting the entire form. The code is in Example 4–4. Add this code before the code for the scene.

Example 4–4 Button

```
Button btn = new Button("Sign in");
HBox hbBtn = new HBox(10);
hbBtn.setAlignment(Pos.BOTTOM_RIGHT);
hbBtn.getChildren().add(btn);
grid.add(hbBtn, 1, 4);
```

The first line creates a button named `btn` with the label `Sign in`, and the second line creates an `HBox` layout pane named `hbBtn` with spacing of `10` pixels. The `HBox` pane sets an alignment for the button that is different from the alignment applied to the other controls in the grid pane. The `alignment` property has a value of `Pos.BOTTOM_RIGHT`, which positions a node at the bottom of the space vertically and at the right edge of the space horizontally. The button is added as a child of the `HBox` pane, and the `HBox` pane is added to the grid in column 1, row 4.

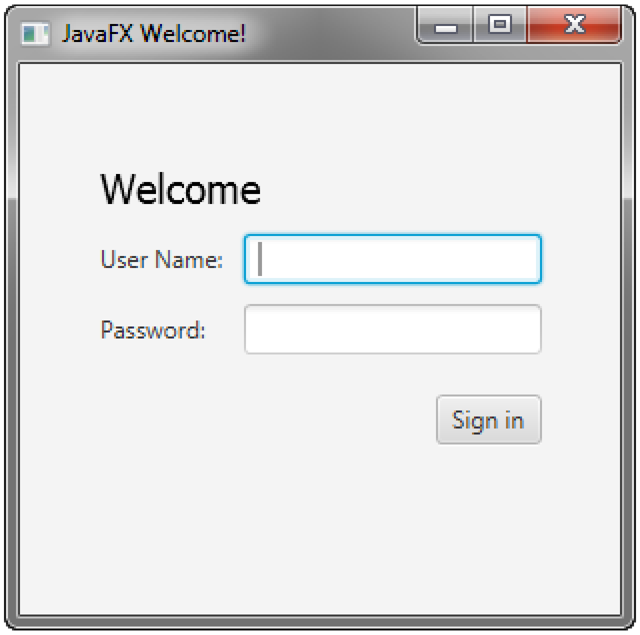
Now, add a `Text` control for displaying the message, as shown in Example 4–5. Add this code before the code for the scene.

Example 4–5 Text

```
final Text actiontarget = new Text();
        grid.add(actiontarget, 1, 6);
```

Figure 4–3 shows the form now. You will not see the text message until you work through the next section of the tutorial, Add Code to Handle an Event.

Figure 4–3 Login Form with Button



Add Code to Handle an Event

Finally, make the button display the text message when the user presses it. Add the code in Example 4–6 before the code for the scene.

Example 4–6 Button Event

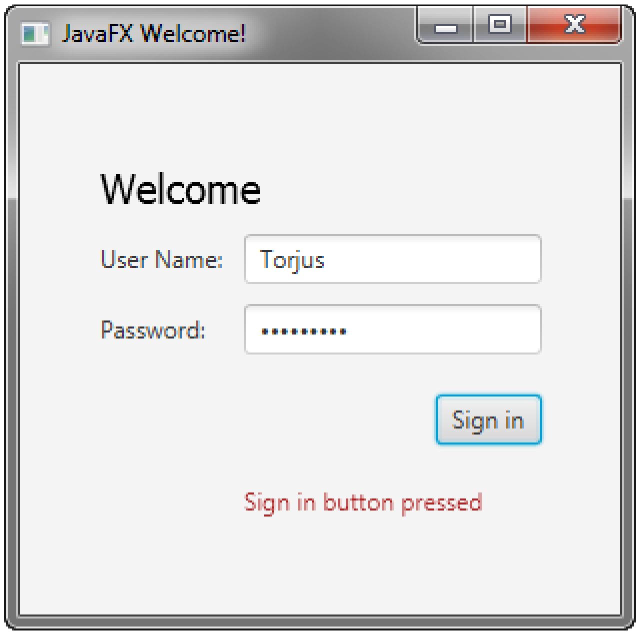
```
btn.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent e) {
        actiontarget.setFill(Color.FIREBRICK);
        actiontarget.setText("Sign in button pressed");
    }
});
```

The `setOnAction()` method is used to register an event handler that sets the `actiontarget` object to `Sign in button pressed` when the user presses the button. The color of the `actiontarget` object is set to firebrick red.

Run the Application

Right-click the **Login** project node in the Projects window, choose **Run**, and then click the Sign in button. Figure 4–4 shows the results. If you run into problems, then take a look at the code in the `Login.java` file that is included in the downloadable Login.zip file.

Figure 4–4 Final Login Form



Where to Go from Here

This concludes the basic form tutorial, but you can continue reading the following tutorials on developing JavaFX applications.

- Fancy Forms with JavaFX CSS provides tips on how to add a background image and radically change the style of the text, label, and button in the login form.
- Using FXML to Create a User Interface shows an alternate method for creating the login user interface. FXML is an XML-based language that provides the structure for building a user interface separate from the application logic of your code.
- Working With Layouts in JavaFX explains the built-in JavaFX layout panes, and tips and tricks for using them.

Also try out the JavaFX samples, which you can download from the JDK Demos and Samples section of the Java SE Downloads page at `http://www.oracle.com/technetwork/java/javase/downloads/`. The Ensemble sample contains examples of layouts and their source code.

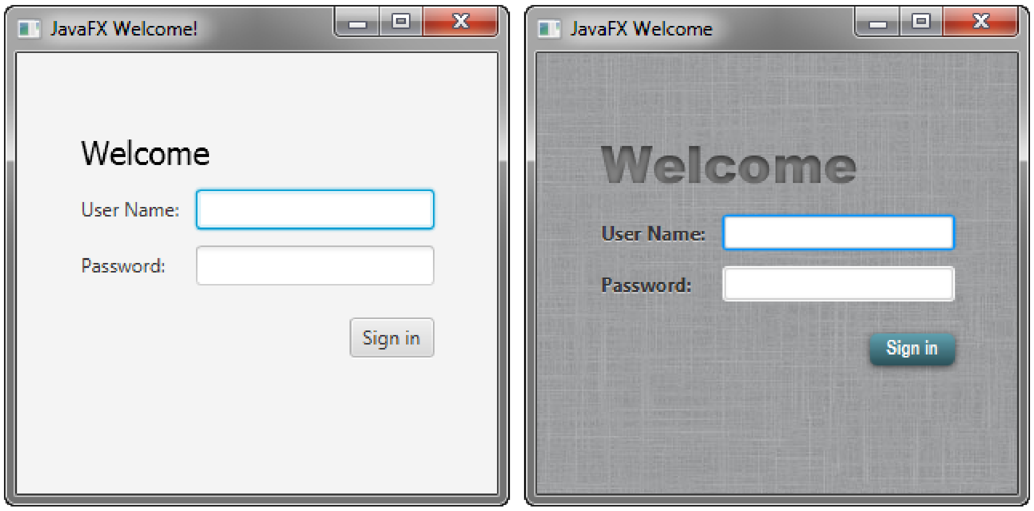
5

Fancy Forms with JavaFX CSS

This tutorial is about making your JavaFX application look attractive by adding a Cascading Style Sheet (CSS). You develop a design, create a `.css` file, and apply the new styles.

In this tutorial, you will take a Login form that uses default styles for labels, buttons, and background color, and, with a few simple CSS modifications, turn it into a stylized application, as shown in Figure 5–1.

Figure 5–1 Login Form With and Without CSS



The tool used in this Getting Started tutorial is NetBeans IDE. Before you begin, ensure that the version of NetBeans IDE that you are using supports JavaFX 8. See the Certified System Configurations page of the Java SE Downloads page for details.

Create the Project

If you followed the Getting Started guide from the start, then you already created the Login project required for this tutorial. If not, download the Login project by right-clicking Login.zip and saving it to your file system. Extract the files from the zip file, and then open the project in NetBeans IDE.

Create the CSS File

Your first task is to create a new CSS file and save it in the same directory as the main class of your application. After that, you must make the JavaFX application aware of the newly added Cascading Style Sheet.

1. In the NetBeans IDE Projects window, expand the **Login** project node and then the **Source Packages** directory node.
2. Right-click the **login** folder under the Source Packages directory and choose **New**, then **Other**.
3. In the New File dialog box, choose **Other**, then **Cascading Style Sheet**, and click **Next**.
4. Enter **Login** for the File Name text field and ensure the Folder text field value is `src\login`.
5. Click **Finish**.
6. In the `Login.java` file, initialize the `style sheets` variable of the `Scene` class to point to the Cascading Style Sheet by including the line of code shown in bold below so that it appears as shown in Example 5–1.

Example 5–1 Initialize the stylesheets Variable

```
Scene scene = new Scene(grid, 300, 275);
primaryStage.setScene(scene);
scene.getStylesheets().add
 (Login.class.getResource("Login.css").toExternalForm());
primaryStage.show();
```

This code looks for the style sheet in the `src\login` directory in the NetBeans project.

Add a Background Image

A background image helps make your form more attractive. For this tutorial, you add a gray background with a linen-like texture.

First, download the background image by right-clicking the background.jpg image and saving it into the `src\login` folder in the Login NetBeans project.

Now, add the code for the `background-image` property to the CSS file. Remember that the path is relative to the style sheet. So, in the code in Example 5–2, the `background.jpg` image is in the same directory as the `Login.css` file.

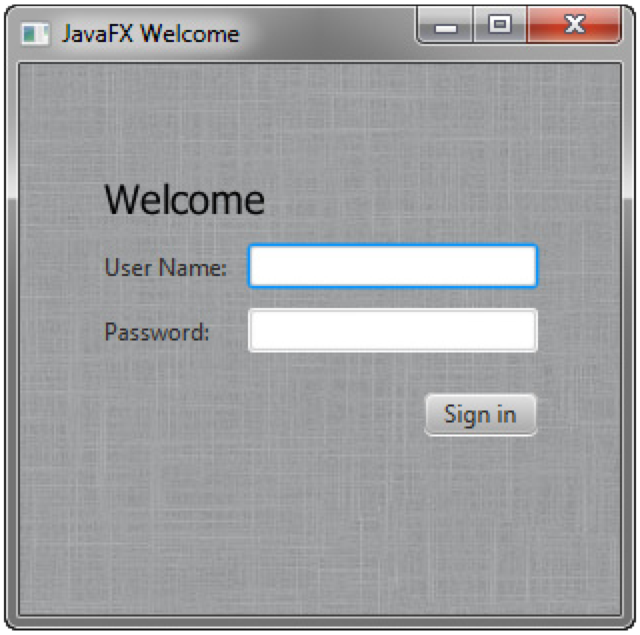
Example 5–2 Background Image

```
.root {
     -fx-background-image: url("background.jpg");
}
```

The background image is applied to the `.root` style, which means it is applied to the root node of the `Scene` instance. The style definition consists of the name of the property (`-fx-background-image`) and the value for the property (`url("background.jpg")`).

Figure 5–2 shows the login form with the new gray background.

Figure 5–2 Gray Linen Background



Style the Labels

The next controls to enhance are the labels. You will use the `.label` style class, which means the styles will affect all labels in the form. The code is in Example 5–3.

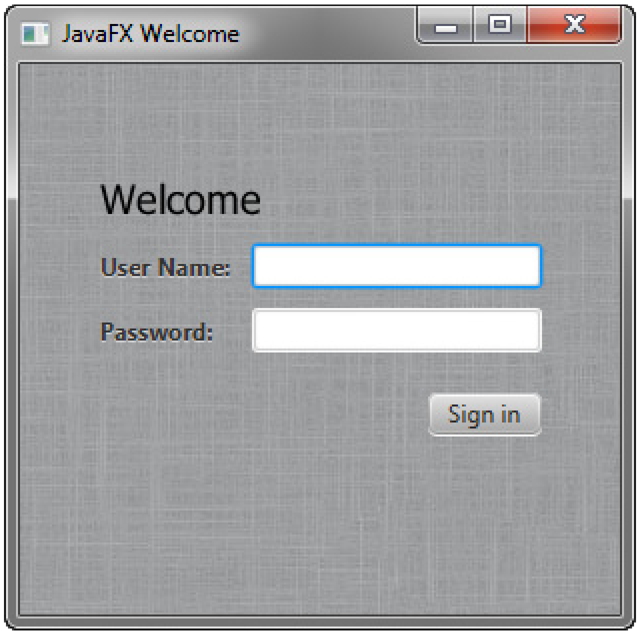
Example 5–3 Font Size, Fill, Weight, and Effect on Labels

```
.label {
    -fx-font-size: 12px;
    -fx-font-weight: bold;
    -fx-text-fill: #333333;
    -fx-effect: dropshadow( gaussian , rgba(255,255,255,0.5) , 0,0,0,1 );
}
```

This example increases the font size and weight and applies a drop shadow of a gray color (#333333). The purpose of the drop shadow is to add contrast between the dark gray text and the light gray background. See the section on effects in the JavaFX CSS Reference Guide for details on the parameters of the drop shadow property.

The enhanced User Name and Password labels are shown in Figure 5–3.

Figure 5–3 Bigger, Bolder Labels with Drop Shadow



Style Text

Now, create some special effects on the two `Text` objects in the form: `scenetitle`, which includes the text `Welcome`, and `actiontarget`, which is the text that is returned when the user presses the Sign in button. You can apply different styles to `Text` objects used in such diverse ways.

1. In the `Login.java` file, remove the following lines of code that define the inline styles currently set for the text objects:

   ```
   scenetitle.setFont(Font.font("Tahoma", FontWeight.NORMAL, 20));
   ```

   ```
   actiontarget.setFill(Color.FIREBRICK);
   ```

   By switching to CSS over inline styles, you separate the design from the content. This approach makes it easier for a designer to have control over the style without having to modify content.

2. Create an ID for each text node by using the `setID()` method of the Node class: Add the following lines in bold so that they appear as shown in Example 5–4.

Example 5–4 Create ID for Text Nodes

```
...
Text scenetitle = new Text("Welcome");
scenetitle.setId("welcome-text");
...
...
grid.add(actiontarget, 1, 6);
actiontarget.setId("actiontarget");
..
```

3. In the `Login.css` file, define the style properties for the `welcome-text` and `actiontarget` IDs. For the style name, use the ID preceded by a number sign (#), as shown in Example 5–5.

Example 5–5 Text Effect

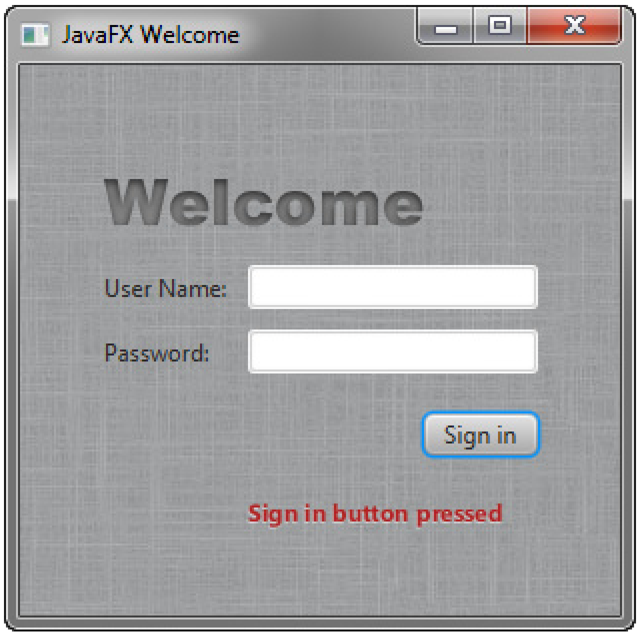
```
#welcome-text {
   -fx-font-size: 32px;
   -fx-font-family: "Arial Black";
   -fx-fill: #818181;
   -fx-effect: innershadow( three-pass-box , rgba(0,0,0,0.7) , 6, 0.0 , 0 , 2 );
}
#actiontarget {
  -fx-fill: FIREBRICK;
  -fx-font-weight: bold;
  -fx-effect: dropshadow( gaussian , rgba(255,255,255,0.5) , 0,0,0,1 );
}
```

The size of the Welcome text is increased to 32 points and the font is changed to Arial Black. The text fill color is set to a dark gray color (#818181) and an inner shadow effect is applied, creating an embossing effect. You can apply an inner shadow to any color by changing the text fill color to be a darker version of the background. See the section on effects in the JavaFX CSS Reference Guide for details about the parameters of inner shadow property.

The style definition for `actiontarget` is similar to what you have seen before.

Figure 5–4 shows the font changes and shadow effects on the two `Text` objects.

Figure 5–4 Text with Shadow Effects



Style the Button

The next step is to style the button, making it change style when the user hovers the mouse over it. This change will give users an indication that the button is interactive, a standard design practice.

First, create the style for the initial state of the button by adding the code in Example 5–6. This code uses the `.button` style class selector, such that if you add a button to the form at a later date, then the new button will also use this style.

Example 5–6 Drop Shadow for Button

```
.button {
    -fx-text-fill: white;
    -fx-font-family: "Arial Narrow";
    -fx-font-weight: bold;
    -fx-background-color: linear-gradient(#61a2b1, #2A5058);
    -fx-effect: dropshadow( three-pass-box , rgba(0,0,0,0.6) , 5, 0.0 , 0 , 1 );
}
```

Now, create a slightly different look for when the user hovers the mouse over the button. You do this with the hover pseudo-class. A pseudo-class includes the selector for the class and the name for the state separated by a colon (:), as shown in Example 5–7.

Example 5–7 Button Hover Style

```
.button:hover {
    -fx-background-color: linear-gradient(#2A5058, #61a2b1);
}
```

Figure 5–5 shows the initial and hover states of the button with its new blue-gray background and white bold text.

Figure 5–5 Initial and Hover Button States

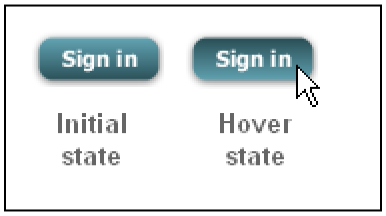
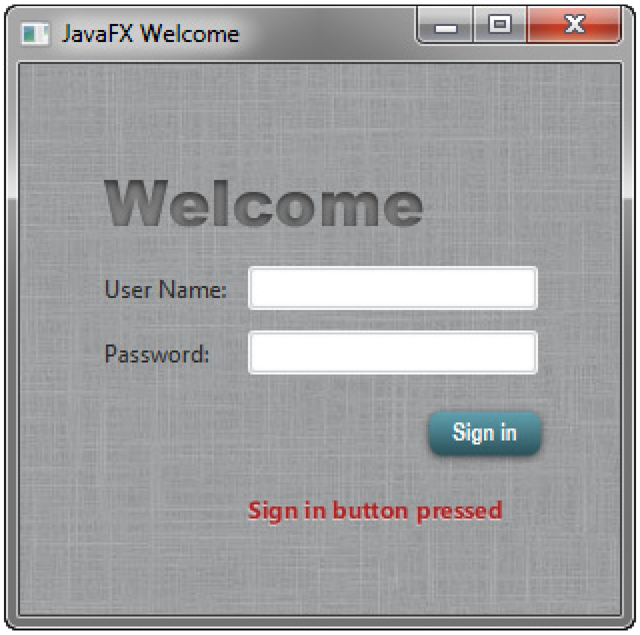


Figure 5–6 shows the final application.

Figure 5–6 Final Stylized Application



Where to Go from Here

Here are some things for you to try next:

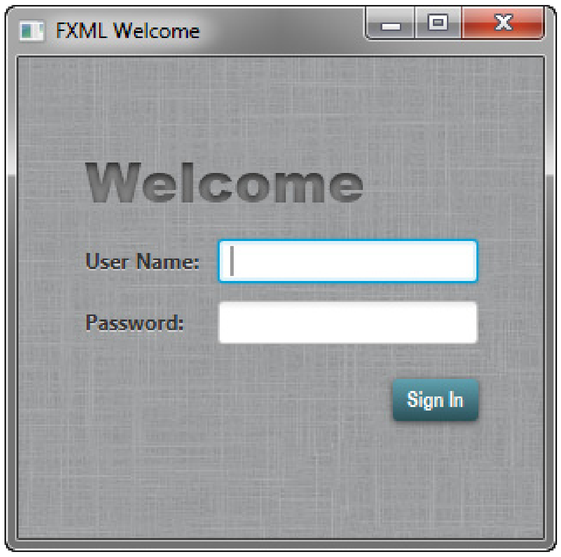
- See what you can create using CSS. Some documents that can help you are Skinning JavaFX Applications with CSS, Styling Charts with CSS, and the JavaFX CSS Reference Guide. The Skinning with CSS and the CSS Analyzer section of the JavaFX Scene Builder User Guide also provides information on how you can use the JavaFX Scene Builder tool to skin your JavaFX FXML layout.
- See Styling FX Buttons with CSS for examples of how to create common button styles using CSS.

6
Using FXML to Create a User Interface

This tutorial shows the benefits of using JavaFX FXML, which is an XML-based language that provides the structure for building a user interface separate from the application logic of your code.

If you started this document from the beginning, then you have seen how to create a login application using just JavaFX. Here, you use FXML to create the same login user interface, separating the application design from the application logic, thereby making the code easier to maintain. The login user interface you build in this tutorial is shown in Figure 6–1.

Figure 6–1   Login User Interface



This tutorial uses NetBeans IDE. Ensure that the version of NetBeans IDE that you are using supports JavaFX 8. See the Certified System Configurations section of the Java SE Downloads page for details.

Set Up the Project

Your first task is to set up a JavaFX FXML project in NetBeans IDE:

1. From the **File** menu, choose **New Project**.
2. In the **JavaFX** application category, choose **JavaFX FXML Application**. Click **Next**.

3. Name the project **FXMLExample** and click **Finish**.

   NetBeans IDE opens an FXML project that includes the code for a basic Hello World application. The application includes three files:

   - **`FXMLExample.java.`** This file takes care of the standard Java code required for an FXML application.
   - **`FXMLDocument.fxml.`** This is the FXML source file in which you define the user interface.
   - **`FXMLDocumentController.java.`** This is the controller file for handling the mouse and keyboard input.

4. Rename `FXMLDocumentController.java` to F`XMLExampleController.java` so that the name is more meaningful for this application.

   a. In the Projects window, right-click **FXMLDocumentController.java** and choose **Refactor** then **Rename**.

   b. Enter **FXMLExampleController**, and click **Refactor**.

5. Rename `FXMLDocument.fxml` to `fxml_example.fxml.`

   a. Right-click **FXMLDocument.fxml** and choose **Rename**.

   b. Enter **fxml_example** and click **OK**.

Load the FXML Source File

The first file you edit is the `FXMLExample.java` file. This file includes the code for setting up the application main class and for defining the stage and scene. More specific to FXML, the file uses the `FXMLLoader` class, which is responsible for loading the FXML source file and returning the resulting object graph.

Make the changes shown in bold in Example 6–1.

Example 6–1 FXMLExample.java

```
@Override
public void start(Stage stage) throws Exception {
   Parent root = FXMLLoader.load(getClass().getResource("fxml_example.fxml"));

    Scene scene = new Scene(root, 300, 275);

    stage.setTitle("FXML Welcome");
    stage.setScene(scene);
    stage.show();
}
```

A good practice is to set the height and width of the scene when you create it, in this case 300 by 275; otherwise the scene defaults to the minimum size needed to display its contents.

Modify the Import Statements

Next, edit the `fxml_example.fxml` file. This file specifies the user interface that is displayed when the application starts. The first task is to modify the import statements so your code looks like Example 6–2.

Example 6–2 XML Declaration and Import Statements

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

<?import java.net.*?>
<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.text.*?>
```

As in Java, class names can be fully qualified (including the package name), or they can be imported using the import statement, as shown in Example 6–2. If you prefer, you can use specific import statements that refer to classes.

Create a GridPane Layout

The Hello World application generated by NetBeans uses an `AnchorPane` layout. For the login form, you will use a `GridPane` layout because it enables you to create a flexible grid of rows and columns in which to lay out controls.

Remove the `AnchorPane` layout and its children and replace it with the `GridPane` layout in Example 6–3.

Example 6–3 GridPane Layout

```
<GridPane fx:controller="fxmlexample.FXMLExampleController"
    xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10" vgap="10">
<padding><Insets top="25" right="25" bottom="10" left="25"/></padding>

</GridPane>
```

In this application, the `GridPane` layout is the root element of the FXML document and as such has two attributes. The `fx:controller` attribute is required when you specify controller-based event handlers in your markup. The `xmlns:fx` attribute is always required and specifies the `fx` namespace.

The remainder of the code controls the alignment and spacing of the grid pane. The alignment property changes the default position of the grid from the top left of the scene to the center. The `gap` properties manage the spacing between the rows and columns, while the `padding` property manages the space around the edges of the grid pane.

As the window is resized, the nodes within the grid pane are resized according to their layout constraints. In this example, the grid remains in the center when you grow or shrink the window. The padding properties ensure there is a padding around the grid when you make the window smaller.

Add Text and Password Fields

Looking back at Figure 6–1, you can see that the login form requires the title “Welcome” and text and password fields for gathering information from the user. The code in Example 6–4 is part of the `GridPane` layout and must be placed above the `</GridPane>` statement.

Example 6–4 Text, Label, TextField, and Password Field Controls

```
 <Text text="Welcome"
       GridPane.columnIndex="0" GridPane.rowIndex="0"
```

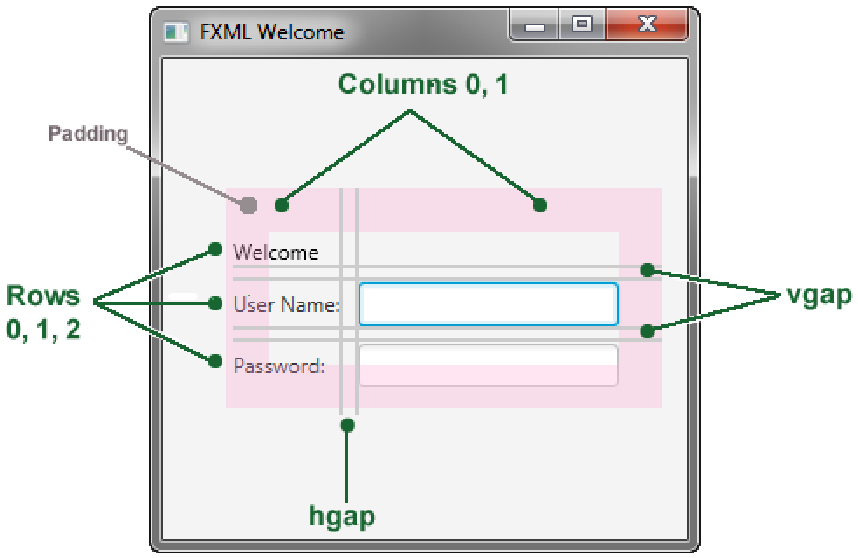
```
        GridPane.columnSpan="2"/>

    <Label text="User Name:"
        GridPane.columnIndex="0" GridPane.rowIndex="1"/>

    <TextField
        GridPane.columnIndex="1" GridPane.rowIndex="1"/>

    <Label text="Password:"
        GridPane.columnIndex="0" GridPane.rowIndex="2"/>

    <PasswordField fx:id="passwordField"
        GridPane.columnIndex="1" GridPane.rowIndex="2"/>
```

The first line creates a `Text` object and sets its text value to `Welcome`. The `GridPane.columnIndex` and `GridPane.rowIndex` attributes correspond to the placement of the `Text` control in the grid. The numbering for rows and columns in the grid starts at zero, and the location of the `Text` control is set to (0,0), meaning it is in the first column of the first row. The `GridPane.columnSpan` attribute is set to 2, making the Welcome title span two columns in the grid. You will need this extra width later in the tutorial when you add a style sheet to increase the font size of the text to 32 points.

The next lines create a `Label` object with text `User Name` at column 0, row 1 and a `TextField` object to the right of it at column 1, row 1. Another `Label` and `PasswordField` object are created and added to the grid in a similar fashion.

When working with a grid layout, you can display the grid lines, which is useful for debugging purposes. In this case, set the `gridLinesVisible` property to `true` by adding the statement `<gridLinesVisible>true</gridLinesVisible>` right after the `<padding></padding>` statement. Then, when you run the application, you see the lines for the grid columns and rows as well as the gap properties, as shown in Figure 6–2.

Figure 6–2 Login Form with Grid Lines



Add a Button and Text

The final two controls required for the application are a `Button` control for submitting the data and a `Text` control for displaying a message when the user presses the button. The code is in Example 6–5. Add this code before `</GridPane>`.

Example 6–5 HBox, Button, and Text

```
<HBox spacing="10" alignment="bottom_right"
        GridPane.columnIndex="1" GridPane.rowIndex="4">
        <Button text="Sign In"
        onAction="#handleSubmitButtonAction"/>
</HBox>

<Text fx:id="actiontarget"
       GridPane.columnIndex="0" GridPane.columnSpan="2"
 GridPane.halignment="RIGHT" GridPane.rowIndex="6"/>
```

An `HBox` pane is needed to set an alignment for the button that is different from the default alignment applied to the other controls in the `GridPane` layout. The `alignment` property is set to `bottom_right`, which positions a node at the bottom of the space vertically and at the right edge of the space horizontally. The `HBox` pane is added to the grid in column 1, row 4.

The `HBox` pane has one child, a `Button` with `text` property set to `Sign in` and an `onAction` property set to `handleSubmitButtonAction()`. While FXML is a convenient way to define the structure of an application's user interface, it does not provide a way to implement an application's behavior. You implement the behavior for the `handleSubmitButtonAction()` method in Java code in the next section of this tutorial, Add Code to Handle an Event.

Assigning an `fx:id` value to an element, as shown in the code for the `Text` control, creates a variable in the document's namespace, which you can refer to from elsewhere in the code. While not required, defining a controller field helps clarify how the controller and markup are associated.

Add Code to Handle an Event

Now make the `Text` control display a message when the user presses the button. You do this in the `FXMLExampleController.java` file. Delete the code that NetBeans IDE generated and replace it with the code in Example 6–6.

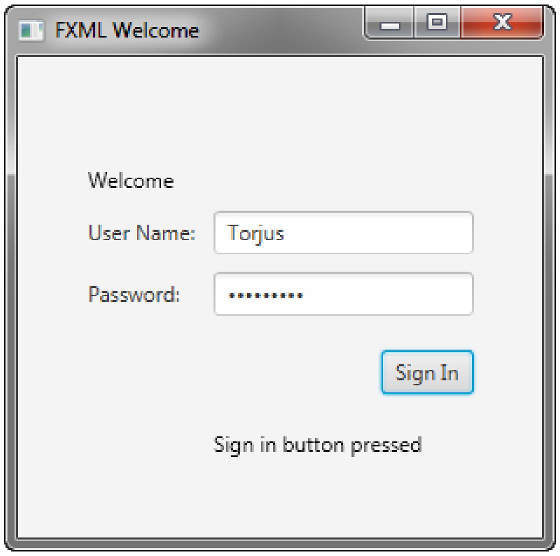
Example 6–6 FXMLExampleController.java

```
package fxmlexample;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.text.Text;

public class FXMLExampleController {
    @FXML private Text actiontarget;

    @FXML protected void handleSubmitButtonAction(ActionEvent event) {
        actiontarget.setText("Sign in button pressed");
    }

}
```

The `@FXML` annotation is used to tag nonpublic controller member fields and handler methods for use by FXML markup. The `handleSubmtButtonAction` method sets the `actiontarget` variable to `Sign in button pressed` when the user presses the button.

You can run the application now to see the complete user interface. Figure 6–3 shows the results when you type text in the two fields and click the Sign in button.   If you have any problems, then you can compare your code against the FXMLLogin example.

Figure 6–3   FXML Login Window



Use a Scripting Language to Handle Events

As an alternative to using Java code to create an event handler, you can create the handler with any language that provides a JSR 223-compatible scripting engine. Such languages include JavaScript, Groovy, Jython, and Clojure.

Optionally, you can try using JavaScript now.

1. In the file `fxml_example.fxml`, add the JavaScript declaration after the XML doctype declaration.

   ```
   <?language javascript?>
   ```

2. In the `Button` markup, change the name of the function so the call looks as follows:

   ```
   onAction="handleSubmitButtonAction(event);"
   ```

3. Remove the `fx:controller` attribute from the `GridPane` markup and add the JavaScript function in a `<script>` tag directly under it, as shown in Example 6–7.

Example 6–7   JavaScript in FXML

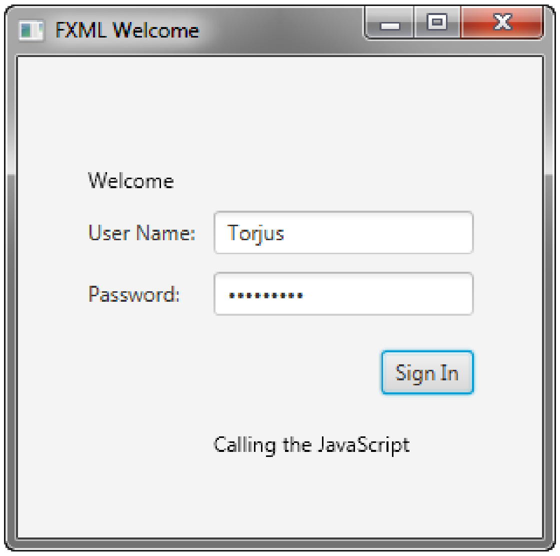
```
<GridPane xmlns:fx="http://javafx.com/fxml"
          alignment="center" hgap="10" vgap="10">
     <fx:script>
         function handleSubmitButtonAction() {
             actiontarget.setText("Calling the JavaScript");
         }
     </fx:script>
```

Alternatively, you can put the JavaScript functions in an external file (such as `fxml_example.js`) and include the script like this:

```
<fx:script source="fxml_example.js"/>
```

The result is in Figure 6–4.

Figure 6–4 Login Application Using JavaScript



If you are considering using a scripting language with FXML, then note that an IDE might not support stepping through script code during debugging.

Style the Application with CSS

The final task is to make the login application look attractive by adding a Cascading Style Sheet (CSS).

1. Create a style sheet.
   a. In the Project window, right-click the fxmlexample folder under Source Packages and choose **New**, then **Other**.
   b. In the New File dialog box, choose **Other**, then **Cascading Style Sheet** and click **Next**.
   c. Enter **Login** and click **Finish**.
   d. Copy the contents of the `Login.css` file into your CSS file. The `Login.css` file is included in the downloadable LoginCSS.zip file. For a description of the classes in the CSS file, see Fancy Forms with JavaFX CSS.
2. Download the gray, linen-like image for the background by right-clicking the background.jpg file and saving it to the `fxmlexample` folder.
3. Open the `fxml_example.fxml` file and add a stylesheets element before the end of the markup for the `GridPane` layout as shown in Example 6–8.

Example 6–8 Style Sheet

```
<stylesheets>
```

```
    <URL value="@Login.css" />
  </stylesheets>

</GridPane>
```

The @ symbol before the name of the style sheet in the URL indicates that the style sheet is in the same directory as the FXML file.

4. To use the root style for the grid pane, add a style class to the markup for the `GridPane` layout as shown in Example 6–9.

Example 6–9   Style the GridPane

```
<GridPane fx:controller="fxmlexample.FXMLExampleController"
    xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10" vgap="10"
    styleClass="root">
```

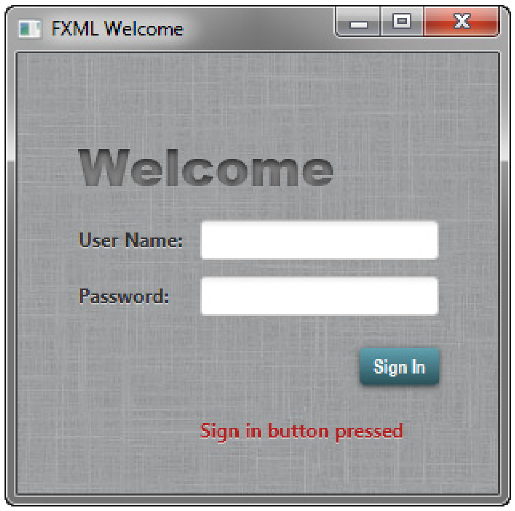
5. Create a `welcome-text` ID for the Welcome `Text` object so it uses the style `#welcome-text` defined in the CSS file, as shown in Example 6–10.

Example 6–10   Text ID

```
<Text id="welcome-text" text="Welcome"
        GridPane.columnIndex="0" GridPane.rowIndex="0"
        GridPane.columnSpan="2"/>
```

6. Run the application. Figure 6–5 shows the stylized application. If you run into problems, then take a look at the code that is included in the downloadable FXMLExample.zip file

Figure 6–5   Stylized Login Application



Where to Go from Here

Now that you are familiar with FXML, look at Introduction to FXML, which provides more information on the elements that make up the FXML language. The document is included in the `javafx.fxml` package in the API documentation.

You can also try out the JavaFX Scene Builder tool by opening the `fxml_example.fxml` file in Scene Builder and making modifications. This tool provides a visual layout environment for designing the UI for JavaFX applications and automatically generates the FXML code for the layout. Note that the FXML file might be reformatted when saved. See the Getting Started with JavaFX Scene Builder for more information on this tool. The Skinning with CSS and the CSS Analyzer section of the JavaFX Scene Builder User Guide also gives you information on how you can skin your FXML layout.

7 Animation and Visual Effects in JavaFX

You can use JavaFX to quickly develop applications with rich user experiences. In this Getting Started tutorial, you will learn to create animated objects and attain complex effects with very little coding.

Figure 7–1 shows the application to be created.

Figure 7–1 Colorful Circles Application

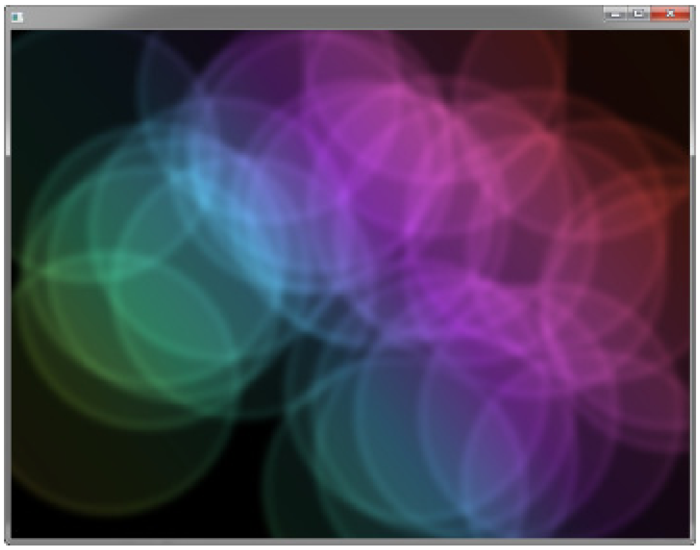
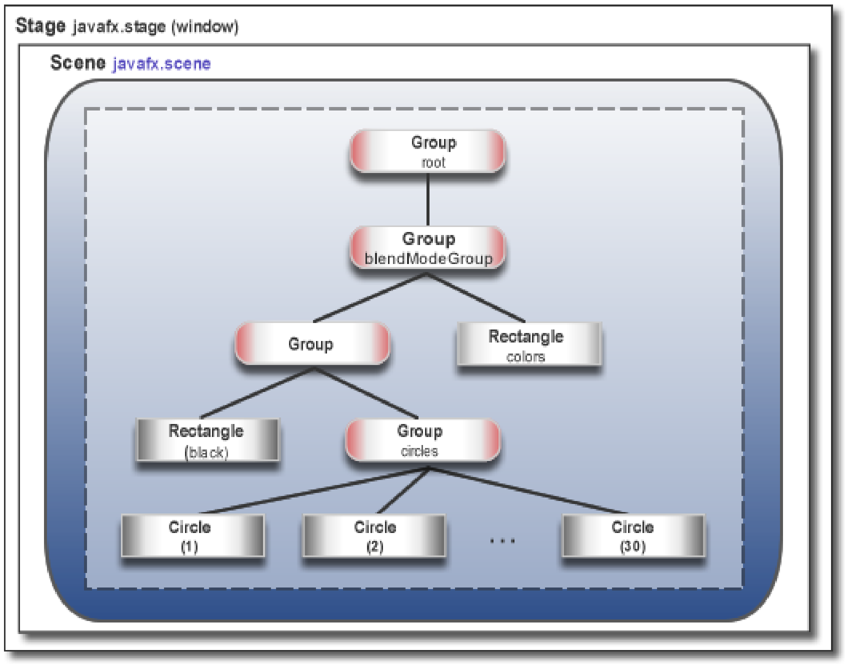


Figure 7–2 shows the scene graph for the `ColorfulCircles` application. Nodes that branch are instantiations of the `Group` class, and the nonbranching nodes, also known as leaf nodes, are instantiations of the `Rectangle` and `Circle` classes.

Figure 7–2 Colorful Circles Scene Graph



The tool used in this Getting Started tutorial is NetBeans IDE. Before you begin, ensure that the version of NetBeans IDE that you are using supports JavaFX 8. See the Certified System Configurations section of the Java SE Downloads page for details.

Set Up the Application

Set up your JavaFX project in NetBeans IDE as follows:

1. From the **File** menu, choose **New Project**.
2. In the **JavaFX** application category, choose **JavaFX Application**. Click **Next**.
3. Name the project **ColorfulCircles** and click **Finish**.
4. Delete the import statements that NetBeans IDE generated.

   You can generate the import statements as you work your way through the tutorial by using either the code completion feature or the Fix Imports command from the Source menu in NetBeans IDE. When there is a choice of import statements, choose the one that starts with `javafx`.

Set Up the Project

Delete the `ColorfulCircles` class from the source file that NetBeans IDE generated and replace it with the code in Example 7–1.

Example 7–1 Basic Application

```
public class ColorfulCircles extends Application {

    @Override
    public void start(Stage primaryStage) {
        Group root = new Group();
        Scene scene = new Scene(root, 800, 600, Color.BLACK);
        primaryStage.setScene(scene);

        primaryStage.show();
    }

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

For the ColorfulCircles application, it is appropriate to use a group node as the root node for the scene. The size of the group is dictated by the size of the nodes within it. For most applications, however, you want the nodes to track the size of the scene and change when the stage is resized. In that case, you use a resizable layout node as the root, as described in Creating a Form in JavaFX.

You can compile and run the ColorfulCircles application now, and at each step of the tutorial, to see the intermediate results. If you run into problems, then take a look at the code in the `ColorfulCircles.java` file, which is included in the downloadable ColorfulCircles.zip file. At this point, the application is a simple black window.

Add Graphics

Next, create 30 circles by adding the code in Example 7–2 before the `primaryStage.show()` line.

Example 7–2 30 Circles

```
Group circles = new Group();
for (int i = 0; i < 30; i++) {
   Circle circle = new Circle(150, Color.web("white", 0.05));
   circle.setStrokeType(StrokeType.OUTSIDE);
   circle.setStroke(Color.web("white", 0.16));
   circle.setStrokeWidth(4);
   circles.getChildren().add(circle);
}
root.getChildren().add(circles);
```

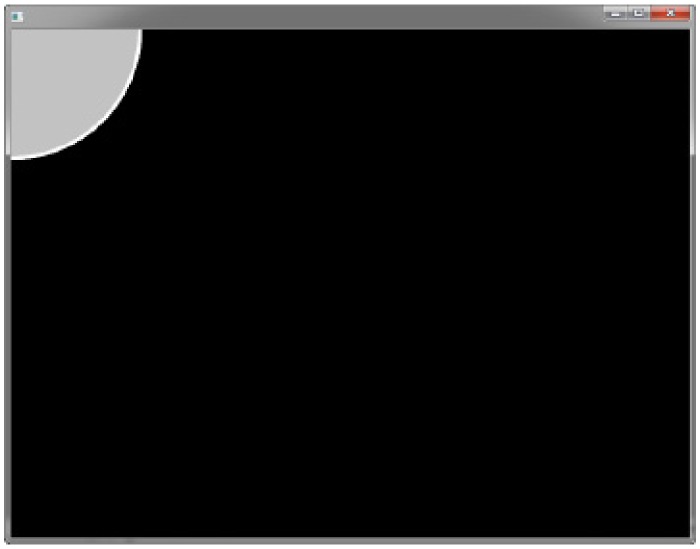
This code creates a group named `circles`, then uses a `for` loop to add 30 circles to the group. Each circle has a radius of `150`, fill color of `white`, and opacity level of `5` percent, meaning it is mostly transparent.

To create a border around the circles, the code includes the `StrokeType` class. A stroke type of `OUTSIDE` means the boundary of the circle is extended outside the interior by the `StrokeWidth` value, which is `4`. The color of the stroke is `white`, and the opacity level is `16` percent, making it brighter than the color of the circles.

The final line adds the `circles` group to the root node. This is a temporary structure. Later, you will modify this scene graph to match the one shown in Figure 7–2.

Figure 7–3 shows the application. Because the code does not yet specify a unique location for each circle, the circles are drawn on top of one another, with the upper left-hand corner of the window as the center point for the circles. The opacity of the overlaid circles interacts with the black background, producing the gray color of the circles.

Figure 7–3 Circles



Add a Visual Effect

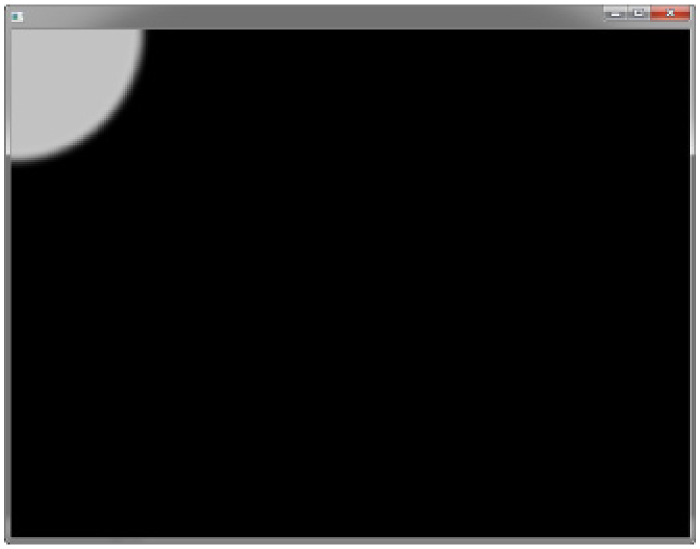
Continue by applying a box blur effect to the circles so that they appear slightly out of focus. The code is in Example 7–3. Add this code before the `primaryStage.show()` line.

Example 7–3 Box Blur Effect

```
circles.setEffect(new BoxBlur(10, 10, 3));
```

This code sets the blur radius to `10` pixels wide by `10` pixels high, and the blur iteration to `3`, making it approximate a Gaussian blur. This blurring technique produces a smoothing effect on the edge of the circles, as shown in Figure 7–4.

Figure 7–4 Box Blur on Circles



Create a Background Gradient

Now, create a rectangle and fill it with a linear gradient, as shown in Example 7–4.

Add the code before the `root.getChildren().add(circles)` line so that the gradient rectangle appears behind the circles.

Example 7–4 Linear Gradient

```
Rectangle colors = new Rectangle(scene.getWidth(), scene.getHeight(),
     new LinearGradient(0f, 1f, 1f, 0f, true, CycleMethod.NO_CYCLE, new
         Stop[]{
            new Stop(0, Color.web("#f8bd55")),
            new Stop(0.14, Color.web("#c0fe56")),
            new Stop(0.28, Color.web("#5dfbc1")),
            new Stop(0.43, Color.web("#64c2f8")),
            new Stop(0.57, Color.web("#be4af7")),
            new Stop(0.71, Color.web("#ed5fc2")),
            new Stop(0.85, Color.web("#ef504c")),
            new Stop(1, Color.web("#f2660f")),}));
colors.widthProperty().bind(scene.widthProperty());
colors.heightProperty().bind(scene.heightProperty());
root.getChildren().add(colors);
```

This code creates a rectangle named `colors`. The rectangle is the same width and height as the scene and is filled with a linear gradient that starts in the lower left-hand corner (0, 1) and ends in the upper right-hand corner (1, 0). The value of `true` means the gradient is proportional to the rectangle, and `NO_CYCLE` indicates that the color cycle will not repeat. The `Stop[]` sequence indicates what the gradient color should be at a particular spot.

The next two lines of code make the linear gradient adjust as the size of the scene changes by binding the width and height of the rectangle to the width and height of the scene. See Using JavaFX Properties and Bindings for more information on binding.

The final line of code adds the `colors` rectangle to the root node.

The gray circles with the blurry edges now appear on top of a rainbow of colors, as shown in Figure 7–5.

Figure 7–5 Linear Gradient

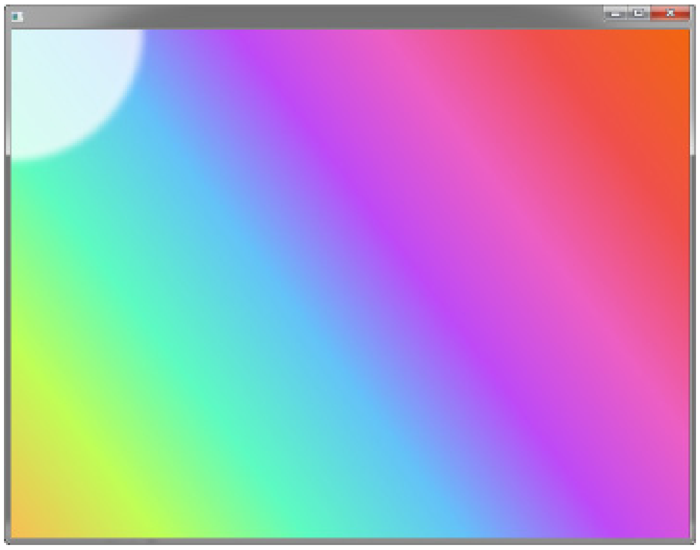
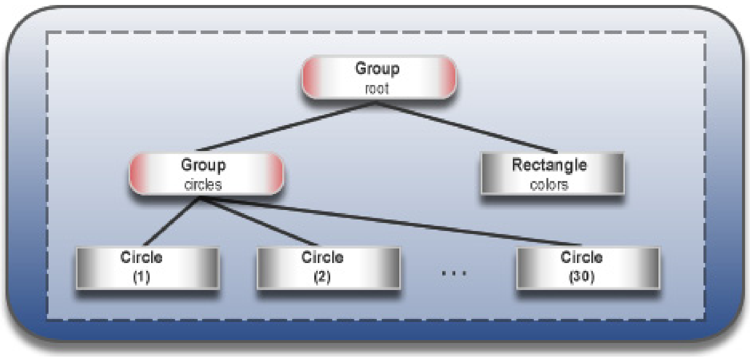


Figure 7–6 shows the intermediate scene graph. At this point, the `circles` group and `colors` rectangle are children of the root node.

Figure 7–6 Intermediate Scene Graph



Apply a Blend Mode

Next, add color to the circles and darken the scene by adding an overlay blend effect. You will remove the `circles` group and the linear gradient rectangle from the scene graph and add them to the new overlay blend group.

1. Locate the following two lines of code:

```
root.getChildren().add(colors);
root.getChildren().add(circles);
```

2. Replace the two lines of code from Step 1 with the code in Example 7–5.

Example 7–5 Blend Mode

```
Group blendModeGroup =
    new Group(new Group(new Rectangle(scene.getWidth(), scene.getHeight(),
        Color.BLACK), circles), colors);
```

```
colors.setBlendMode(BlendMode.OVERLAY);
root.getChildren().add(blendModeGroup);
```

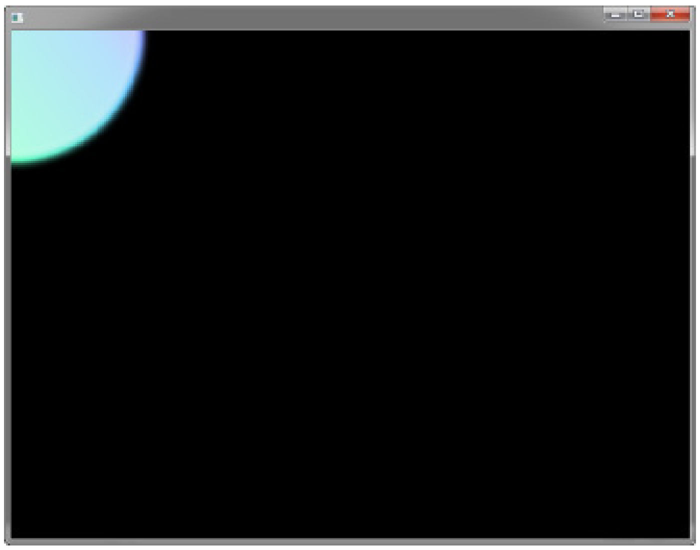
The group `blendModeGroup` sets up the structure for the overlay blend. The group contains two children. The first child is a new (unnamed) `Group` containing a new (unnamed) black rectangle and the previously created `circles` group. The second child is the previously created `colors` rectangle.

The `setBlendMode()` method applies the overlay blend to the `colors` rectangle. The final line of code adds the `blendModeGroup` to the scene graph as a child of the root node, as depicted in Figure 7–2.

An overlay blend is a common effect in graphic design applications. Such a blend can darken an image or add highlights or both, depending on the colors in the blend. In this case, the linear gradient rectangle is used as the overlay. The black rectangle serves to keep the background dark, while the nearly transparent circles pick up colors from the gradient, but are also darkened.

Figure 7–7 shows the results. You will see the full effect of the overlay blend when you animate the circles in the next step.

Figure 7–7 Overlay Blend



Add Animation

The final step is to use JavaFX animations to move the circles:

1. If you have not done so already, add `import static java.lang.Math.random;` to the list of import statements.
2. Add the animation code in Example 7–6 before the `primaryStage.show()` line.

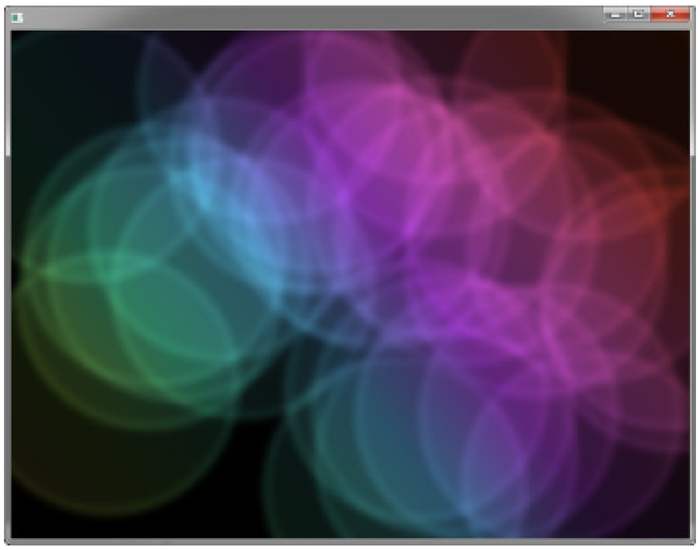
Example 7–6 Animation

```
Timeline timeline = new Timeline();
for (Node circle: circles.getChildren()) {
    timeline.getKeyFrames().addAll(
        new KeyFrame(Duration.ZERO, // set start position at 0
            new KeyValue(circle.translateXProperty(), random() * 800),
                new KeyValue(circle.translateYProperty(), random() * 600)
            ),
            new KeyFrame(new Duration(40000), // set end position at 40s
                new KeyValue(circle.translateXProperty(), random() * 800),
                new KeyValue(circle.translateYProperty(), random() * 600)
            )
        );
}
// play 40s of animation
timeline.play();
```

Animation is driven by a timeline, so this code creates a timeline, then uses a `for` loop to add two key frames to each of the 30 circles. The first key frame at 0 seconds uses the properties `translateXProperty` and `translateYProperty` to set a random position of the circles within the window. The second key frame at 40 seconds does the same. Thus, when the timeline is played, it animates all circles from one random position to another over a period of 40 seconds.

Figure 7–8 shows the 30 colorful circles in motion, which completes the application. For the complete source code, see the `ColorfulCircles.java` file, which is included in the downloadable ColorfulCircles.zip file..

Figure 7–8 Animated Circles



Where to Go from Here

Here are several suggestions about what to do next:

- Try the JavaFX samples, which you can download from the JDK Demos and Samples section of the Java SE Downloads page at `http://www.oracle.com/technetwork/java/javase/downloads/`. Especially take a look at the Ensemble application, which provides sample code for animations and effects.
- Read Creating Transitions and Timeline Animation in JavaFX. You will find more details on timeline animation as well as information on fade, path, parallel, and sequential transitions.

- See Creating Visual Effects in JavaFX for additional ways to enhance the look and design of your application, including reflection, lighting, and shadow effects.
- Try the JavaFX Scene Builder tool to create visually interesting applications. This tool provides a visual layout environment for designing the UI for JavaFX applications and generates FXML code. You can use the Properties panel or the Modify option of the menu bar to add effects to the UI elements. See the Properties Panel and the Menu Bar sections of the JavaFX Scene Builder User Guide for information.

A background.jpg

This appendix provides a graphical image used in the Using FXML to Create a User Interface.

Legal Terms and Copyright Notice

background.jpg