

JavaFX

JavaFX for Swing Developers

Release 2.2

E20483-09

October 2013

JavaFX /JavaFX for Swing Developers, Release 2.2

E20483-09

Copyright © 2011, 2013 Oracle and/or its affiliates. All rights reserved.

Primary Author: Irina Fedortsova

Contributor: Artem Ananiev, Alexander Zvegintsev, Alexander Kouznetsov

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

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

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

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

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

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

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark licensed through X/Open Company, Ltd.

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

Contents

1	The JavaFX Advantage for Swing Developers	
	Using FXML.....	1-1
	JavaFX Scene Builder.....	1-1
	CSS Support.....	1-1
	JavaFX Media Support.....	1-2
	Animation.....	1-2
	HTML Content.....	1-2
2	Integrating JavaFX into Swing Applications	
	Adding JavaFX Content to a Swing Component.....	2-1
	Swing–JavaFX Interoperability and Threads.....	2-2
	Changing JavaFX Data in Response to a Change in Swing Data.....	2-2
	Changing Swing Data in Response to a Change in JavaFX Data.....	2-3
	Introducing the SimpleSwingBrowser Application.....	2-3
	Initializing Swing Data.....	2-3
	Loading JavaFX Content.....	2-5
	Updating Swing Data.....	2-6
	Deploying Swing-JavaFX Applications.....	2-7
	View and Download Application Files.....	2-7
3	Enriching Swing Applications with JavaFX Functionality	
	Sample Swing Application.....	3-1
	Integrating JavaFX Bar Chart.....	3-2
	View and Download Application Files.....	3-5
4	Leveraging Applications with Media Features	
	About Media Integration.....	4-1
	Building the Media Player Application.....	4-1
	Skinning the Application with CSS.....	4-2
	Adding a New Control to the Control Bar.....	4-3
	View and Download Application Files.....	4-4
5	Implementing a Swing Application in JavaFX	
	Analyzing the Converter Application Developed in Swing.....	5-1

Planning the Converter Application in JavaFX	5-2
Creating the Converter Application in JavaFX	5-2
Standard JavaFX Pattern to Create the GUI.....	5-2
Containers and Layouts	5-3
UI Controls	5-3
Usage of the Builder Classes.....	5-4
Mechanism of Getting Notifications on User Actions and Binding	5-4
Creating the ConversionPanel Class	5-4
Creating Instance Variables for UI Controls.....	5-5
Creating DoubleProperty and NumberFormat Objects.....	5-5
Laying Out the Components	5-5
Creating InvalidationListener Objects	5-6
Adding Change Listeners to Controls and Ensuring Synchronization	5-7
Creating the Converter Class	5-7
Defining Instance Variables.....	5-7
Creating the Constructor for the Converter Class	5-8
Creating the Graphical Scene	5-8
View and Download Application Files	5-9

Part I

About This Tutorial

This tutorial provides an overview of JavaFX benefits available to GUI developers, illustrates the JavaFX–Swing interoperability, shows how to enrich an existing Swing application by taking advantage of JavaFX functionality, and how to implement a typical Swing application in JavaFX.

This tutorial contains the following chapters:

- [The JavaFX Advantage for Swing Developers](#)
- [Integrating JavaFX into Swing Applications](#)
- [Enriching Swing Applications with JavaFX Functionality](#)
- [Leveraging Applications with Media Features](#)
- [Implementing a Swing Application in JavaFX](#)

The JavaFX Advantage for Swing Developers

JavaFX is designed to provide applications with such sophisticated GUI features as smooth animation, web views, audio and video playback, and styles based on Cascading Style Sheets (CSS).

For more than 10 years, application developers have found Swing to be a highly effective toolkit for building graphical user interfaces (GUIs) and adding interactivity to Java applications. However, some of today's most popular GUI features cannot be easily implemented by using Swing. These features and others described in the following sections can help application developers to meet the full range of modern requirements. Later chapters in this document explain how to use Swing and JavaFX together.

Using FXML

FXML is an XML-based markup language that enables developers to create a user interface (UI) in a JavaFX application separately from implementing the application logic. Swing has never offered a declarative approach to building a user interface. The declarative method for creating a UI is particularly suitable for the scene graph, because the scene graph is more transparent in FXML. Using FXML enables developers to more easily maintain complex user interfaces.

To learn more about the benefits of using FXML, see the [Getting Started with FXML](#) document.

JavaFX Scene Builder

To help developers build the layout of their applications, JavaFX provides a design tool called the JavaFX Scene Builder. You drag and drop UI components to a JavaFX Content pane, and the tool generates the FXML code that can be used in an IDE such as NetBeans or Eclipse.

For more information, see [Getting Started with JavaFX Scene Builder](#) and the [Scene Builder User Guide](#).

CSS Support

Cascading style sheets contain style definitions that control the look of UI elements. The usage of CSS in JavaFX applications is similar to the usage of CSS in HTML. With CSS, you can easily customize and develop themes for JavaFX controls and scene graph objects.

Using CSS as opposed to setting inline styles enables you to separate the logic of the application from setting its visual appearance. Using CSS also simplifies further maintenance of how your application looks and provides some performance benefits.

For more information about CSS, see [Skinning JavaFX Applications with CSS and JavaFX CSS Reference Guide](#).

JavaFX Media Support

With the media support provided by the JavaFX platform, you can leverage your desktop application by adding media functionality such as playback of audio and video files. Media functionality is available on all platforms where JavaFX is supported. For the list of supported media codecs, see [Introduction to JavaFX Media](#).

For more details, see [Leveraging Applications with Media Features](#) chapter.

Animation

Animation brings dynamics and a modern look to the interface of your applications. Animating objects in a Swing application is possible but is not straightforward. In the Swing rendering model, painting happens on a double buffer. All alterations of object properties and positions with time are rendered on a double buffer. Only when the painting is completed, is the final result actually painted onto the screen. To show time-based alterations of objects requires significant efforts from a developer using Swing. In contrast, JavaFX enables developers to animate graphical objects in their applications more easily because of the scene graph underlying the platform and the particular APIs that are specifically created for that purpose.

For more details about animation in JavaFX, see [Creating Transitions and Timeline Animation in JavaFX](#). Be sure to check the [Tree animation example](#).

HTML Content

For a long time, Swing developers have wanted the ability to render HTML content in Java applications. JavaFX brought this feature to life by providing a user interface component that has web view and full browsing functionality.

For more details, see [Adding HTML Content to JavaFX Applications](#).

Integrating JavaFX into Swing Applications

This chapter describes how to add JavaFX content into a Swing application and how to use threads correctly when both Swing and JavaFX content operate within a single application.

JavaFX SDK provides the `JFXPanel` class, which is located in the `javafx.embed.swing` package and enables you to embed JavaFX content into Swing applications.

Adding JavaFX Content to a Swing Component

For the purpose of this chapter, you create a `JFrame` component, add a `JFXPanel` object to it, and set the graphical scene of the `JFXPanel` component that contains JavaFX content.

As in any Swing application, you create the graphical user interface (GUI) on an event dispatch thread (EDT). [Example 2-1](#) shows the `initAndShowGUI` method, which creates a `JFrame` component and adds a `JFXPanel` object to it. Creating an instance of the `JFXPanel` class implicitly starts the JavaFX runtime. After the GUI is created, call the `initFX` method to create the JavaFX scene on the JavaFX application thread.

Example 2-1

```
import javafx.application.Platform;
import javafx.embed.swing.JFXPanel;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class Test {

    private static void initAndShowGUI() {
        // This method is invoked on the EDT thread
        JFrame frame = new JFrame("Swing and JavaFX");
        final JFXPanel fxPanel = new JFXPanel();
        frame.add(fxPanel);
        frame.setSize(300, 200);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        Platform.runLater(new Runnable() {
            @Override
            public void run() {
```

```
        initFX(fxPanel);
    }
});
}

private static void initFX(JFXPanel fxPanel) {
    // This method is invoked on the JavaFX thread
    Scene scene = createScene();
    fxPanel.setScene(scene);
}

private static Scene createScene() {
    Group root = new Group();
    Scene scene = new Scene(root, Color.ALICEBLUE);
    Text text = new Text();

    text.setX(40);
    text.setY(100);
    text.setFont(new Font(25));
    text.setText("Welcome JavaFX!");

    root.getChildren().add(text);

    return (scene);
}

public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        @Override
        public void run() {
            initAndShowGUI();
        }
    });
}
}
```

Swing–JavaFX Interoperability and Threads

With JavaFX and Swing data coexisting in a single application, you may encounter the following interoperability situations:

- A JavaFX data change is triggered by a change in Swing data.
- A Swing data change is triggered by a change in JavaFX data.

Changing JavaFX Data in Response to a Change in Swing Data

JavaFX data should be accessed only on the JavaFX User thread. Whenever you must change JavaFX data, wrap your code into a `Runnable` object and call the `Platform.runLater` method as shown in [Example 2–2](#).

Example 2–2

```
jbutton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        Platform.runLater(new Runnable() {
            @Override
            public void run() {
                fxlabel.setText("Swing button clicked!");
            }
        });
    }
});
```

```

    }
  });
}
});

```

Changing Swing Data in Response to a Change in JavaFX Data

Swing data should be changed only on the EDT. To ensure that your code is implemented on the EDT, wrap it into a Runnable object and call the `SwingUtilities.invokeLater` method as shown in [Example 2-3](#).

Example 2-3

```

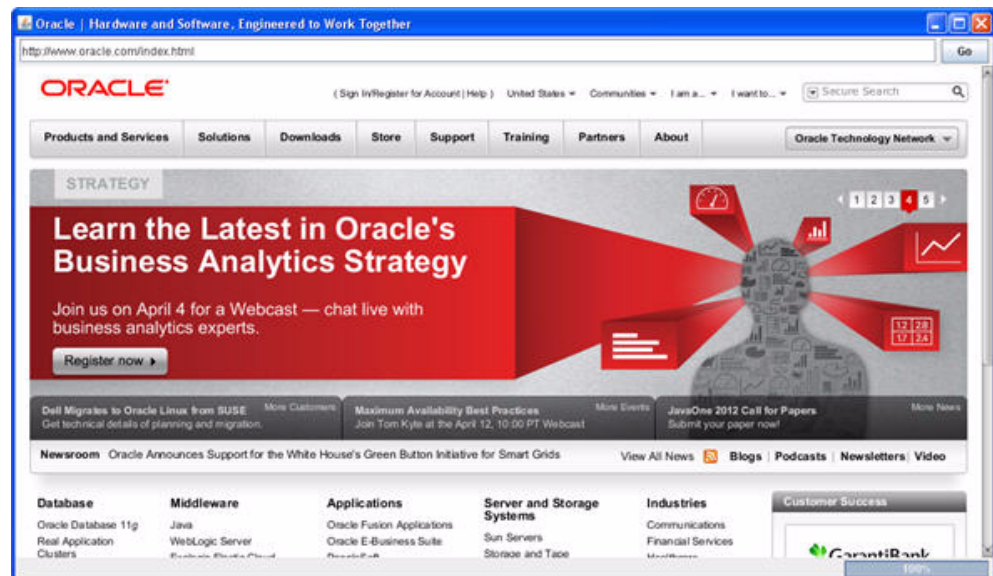
SwingUtilities.invokeLater(new Runnable() {
    @Override
    public void run() {
        //Code to change Swing data.
    }
});

```

Introducing the SimpleSwingBrowser Application

To see how Swing–JavaFX interoperability works, consider the `SimpleSwingBrowser` application. This is a Swing application with an integrated JavaFX component intended to view Web pages. You can type a URL in an address bar and view the page loaded in the application window. The `SimpleSwingBrowser` application window is shown in [Figure 2-1](#).

Figure 2-1 The `SimpleSwingBrowser` Application Window



Initializing Swing Data

You can view the `SimpleSwingBrowser.java` file or download the `SimpleSwingBrowser.zip` file with a NetBeans project. Extract files from the zip file to a directory on your local file system and run the project in your NetBeans IDE.

As of version 7.2, the NetBeans IDE provides support for Swing applications with the embedded JavaFX content. When creating a new project, in the **JavaFX** category choose **JavaFX in Swing Application**.

Note: To run this application from behind a firewall, you must specify proxy settings in order for the application to access a remote resource.

In the NetBeans IDE, right-click the **SimpleSwingBrowser** project in the Projects window, select **Properties**, and in the Projects Properties dialog, select **Run**.

In the VM Options field, set the proxy in the following format:

```
-Dhttp.proxyHost=webcache.mydomain.com -Dhttp.proxyPort=8080
```

The GUI of the SimpleSwingBrowser application is created on the EDT when the application starts. The main method is implemented as shown in [Example 2-4](#).

Example 2-4

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {

        @Override
        public void run() {
            SimpleSwingBrowser browser = new SimpleSwingBrowser();
            browser.setVisible(true);
            browser.loadURL("http://oracle.com");
        }
    });
}
```

The SimpleSwingBrowser class initializes Swing objects and calls the `initComponents` method to create the GUI as shown in [Example 2-5](#).

Example 2-5

```
public class SimpleSwingBrowser extends JFrame {

    private final JFXPanel jfxPanel = new JFXPanel();
    private WebEngine engine;

    private final JPanel panel = new JPanel(new BorderLayout());
    private final JLabel lblStatus = new JLabel();

    private final JButton btnGo = new JButton("Go");
    private final JTextField txtURL = new JTextField();
    private final JProgressBar progressBar = new JProgressBar();

    public SimpleSwingBrowser() {
        super();
        initComponents();
    }

    private void initComponents() {
        createScene();

        ActionListener al = new ActionListener() {
```

```

        @Override
        public void actionPerformed(ActionEvent e) {
            loadURL(txtURL.getText());
        }
    };

    btnGo.addActionListener(al);
    txtURL.addActionListener(al);

    progressBar.setPreferredSize(new Dimension(150, 18));
    progressBar.setStringPainted(true);

    JPanel topBar = new JPanel(new BorderLayout(5, 0));
    topBar.setBorder(BorderFactory.createEmptyBorder(3, 5, 3, 5));
    topBar.add(txtURL, BorderLayout.CENTER);
    topBar.add(btnGo, BorderLayout.EAST);

    JPanel statusBar = new JPanel(new BorderLayout(5, 0));
    statusBar.setBorder(BorderFactory.createEmptyBorder(3, 5, 3, 5));
    statusBar.add(lblStatus, BorderLayout.CENTER);
    statusBar.add(progressBar, BorderLayout.EAST);

    panel.add(topBar, BorderLayout.NORTH);
    panel.add(jfxPanel, BorderLayout.CENTER);
    panel.add(statusBar, BorderLayout.SOUTH);

    getContentPane().add(panel);

    setPreferredSize(new Dimension(1024, 600));
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    pack();
    }
}

```

The topmost window of this application is a `JFrame` object, which contains various Swing components such as a text field, a button, a progress bar, and a JFX panel intended to display JavaFX content.

Loading JavaFX Content

On the first run, the web page at `http://oracle.com` is loaded into a `WebView` object. As a new URL is entered in the address bar, the action listener, which is attached to the `txtURL` text field in the `initComponents` method, initiates the loading of a page as shown in [Example 2-6](#).

Example 2-6

```

ActionListener al = new ActionListener() {
    @Override public void actionPerformed(ActionEvent e) {
        loadURL(txtURL.getText());
    }
};

```

JavaFX data should only be accessed on the JavaFX application thread. The `loadURL` method wraps the code into a `Runnable` object and calls the `Platform.runLater` method as shown in [Example 2-7](#).

Example 2-7

```
public void loadURL(final String url) {
    Platform.runLater(new Runnable() {
        @Override public void run() {
            String tmp = toURL(url);

            if (url == null) {
                tmp = toURL("http://" + url);
            }

            engine.load(tmp);
        }
    });
}

private static String toURL(String str) {
    try {
        return new URL(str).toExternalForm();
    } catch (MalformedURLException exception) {
        return null;
    }
}
```

Updating Swing Data

As a new page is loaded into the `WebView` component, the title of the page is retrieved from the JavaFX data and passed to the Swing GUI to be placed on the application window as a title. This behavior is implemented in the `createScene` method as shown in [Example 2-8](#).

Example 2-8

```
private void createScene() {

    Platform.runLater(new Runnable() {
        @Override
        public void run() {

            WebView view = new WebView();
            engine = view.getEngine();

            engine.titleProperty().addListener(new ChangeListener<String>() {
                @Override
                public void changed(ObservableValue<? extends String> observable,
                String oldValue, final String newValue) {
                    SwingUtilities.invokeLater(new Runnable() {
                        @Override
                        public void run() {
                            SimpleSwingBrowser.this.setTitle(newValue);
                        }
                    });
                }
            });
        }
    });
}
```

Deploying Swing-JavaFX Applications

As of JavaFX 2.2, you can use the same deployment approach to package Swing-JavaFX applications that you would for pure JavaFX applications. Using JavaFX Ant tasks, you only need to set an attribute that the application's primary UI toolkit is Swing. The resulting package provides support for the same set of execution modes as a package for a JavaFX application; in other words, the application can be run standalone, using Web Start, embedded into a web page, or distributed as a self-contained application bundle. For more information, see the Deploying JavaFX Applications guide.

View and Download Application Files

View Source Code

<http://docs.oracle.com/javafx/2/swing/SimpleSwingBrowser.java.htm>

Download Source Code

<http://docs.oracle.com/javafx/2/swing/SimpleSwingBrowser.zip>

Enriching Swing Applications with JavaFX Functionality

In this chapter you learn how to intermix a Swing table and JavaFX bar chart in a single application.

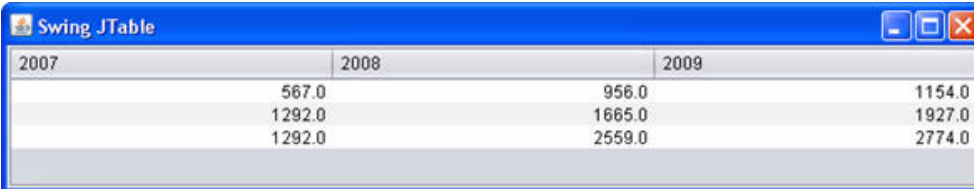
This chapter starts with a Swing application and provides an example of how to enrich the Swing application by adding JavaFX functionality.

Sample Swing Application

Many real-world projects employ Swing applications that deal with tables. You can continue using the existing code and still take an advantage of JavaFX APIs. For example, you can add a JavaFX bar chart to provide a colorful illustration of the tabular data. This chapter provides the `SwingInterop` example that handles a Swing table and a JavaFX bar chart. As you change the data in a table cell, the bar chart immediately updates.

Start with the sample application that has only the Swing table shown in [Figure 3-1](#).

Figure 3-1 *Swing JTable Application Window*



2007	2008	2009
567.0	956.0	1154.0
1292.0	1665.0	1927.0
1292.0	2559.0	2774.0

This application consists of two classes:

- `SampleTableModel.java`
- `SwingInterop.java`

The `SampleTableModel` class inherits from the `AbstractTableModel` class and defines the table.

The `SwingInterop` class inherits from the `JApplet` class and is the basic class of the application. Its `main` method calls the `run` method on the Event Dispatch Thread (EDT) to create the graphical user interface (GUI). The `run` method creates a `JFrame` object and a `JApplet` object, and initializes the `JApplet` object with an instance of the `SwingInterop` class. Then it calls the `init` method, which creates the table and adds the table to the content pane of the applet.

You can see the implementation of both classes by using the links in the sidebar.

Integrating JavaFX Bar Chart

To provide data for a bar chart, modify the `SampleTableModel` class by adding a new class variable (`bcData`) and a method that retrieves data from the table and returns the data in the format appropriate for the bar chart. The implementation of the `getBarChartData` method is shown in [Example 3-1](#).

Example 3-1

```
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.chart.BarChart;

public class SampleTableModel extends AbstractTableModel {
    private static ObservableList<BarChart.Series> bcData;

    public ObservableList<BarChart.Series> getBarChartData() {
        if (bcData == null) {
            bcData = FXCollections.observableArrayList();
            for (int row = 0; row < getRowCount(); row++) {
                ObservableList<BarChart.Data> series =
FXCollections.<BarChart.Data>observableArrayList();
                for (int column = 0; column < getColumnCount(); column++) {
                    series.add(new BarChart.Data(getColumnName(column),
getValueAt(row, column)));
                }
                bcData.add(new BarChart.Series(series));
            }
        }
        return bcData;
    }
}
//rest of the SampleTableModel class code
```

The `SwingInterop` class overrides the `JApplet.init` method to create the content pane of the application. Modify the `init` method to create a `JFXPanel` object to hold the JavaFX bar chart and a `JSplitPane` object to hold both the JavaFX chart and the table. The required changes to the `init` method are shown in bold in [Example 3-2](#).

Example 3-2

```
@Override
public void init() {
    tableModel = new SampleTableModel();
    // create javafx panel for charts
    chartFxPanel = new JFXPanel();
    chartFxPanel.setPreferredSize(new Dimension(PANEL_WIDTH_INT, PANEL_HEIGHT_
INT));

    //create JTable
    JTable table = new JTable(tableModel);
    table.setAutoCreateRowSorter(true);
    table.setGridColor(Color.DARK_GRAY);
    SwingInterop.DecimalFormatRenderer renderer =
new SwingInterop.DecimalFormatRenderer();
    renderer.setHorizontalAlignment(JLabel.RIGHT);
    for (int i = 0; i < table.getColumnCount(); i++) {
        table.getColumnModel().getColumn(i).setCellRenderer(renderer);
    }
}
```

```

    }
    JScrollPane tablePanel = new JScrollPane(table);
    tablePanel.setPreferredSize(new Dimension(PANEL_WIDTH_INT,
TABLE_PANEL_HEIGHT_INT));
    JPanel chartTablePanel = new JPanel();
    chartTablePanel.setLayout(new BorderLayout());

    //Create split pane that holds both the bar chart and table
    JSplitPane jsplitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
    jsplitPane.setTopComponent(chartTablePanel);
    jsplitPane.setBottomComponent(tablePanel);
    jsplitPane.setDividerLocation(410);
    chartTablePanel.add(chartFxPanel, BorderLayout.CENTER);

    //Add the split pane to the content pane of the application
    add(jsplitPane, BorderLayout.CENTER);
}

```

To get rid of a syntax error, add import statements and the definition of the `chartFxPanel` class variable to the `SwingInterop` class as shown in [Example 3-3](#).

Example 3-3

```

import javafx.embed.swing.JFXPanel;
import javax.swing.*;

public class SwingInterop extends JApplet {
    private static JFXPanel chartFxPanel;
    // rest of the SwingInterop class code here
}

```

You prepared the UI of your Swing application to render JavaFX data. The next step is creating the JavaFX scene. Because the JavaFX scene must be created on the JavaFX Application thread, wrap your code into a `Runnable` object as shown in [Example 3-4](#). Add this code at the end of the `init` method.

Example 3-4

```

Platform.runLater(new Runnable() {
    @Override
    public void run() {
        createScene();
    }
});

```

Add the import statement shown in [Example 3-5](#) to the `SwingInterop` class.

Example 3-5

```

import javafx.application.Platform;

```

Implement the `createScene` method of the `SwingInterop` class as shown in [Example 3-6](#). Add the import statements and define the instance variable `chart`.

Example 3-6

```

import javafx.scene.Scene;
import javafx.scene.chart.Chart;

private void createScene() {

```

```

        chart = createBarChart();
        chartFxPanel.setScene(new Scene(chart));
    }

```

The `createBarChart` method creates the chart diagram and adds a change listener to the table. Note that any change of JavaFX data must happen on the JavaFX thread. For this reason, wrap the code in the event handler, which updates the JavaFX chart, into a `Runnable` object and pass it to the `Platform.runLater` method. The implementation of the `createBarChart` method is shown in [Example 3-7](#).

Example 3-7

```

private BarChart createBarChart() {
    CategoryAxis xAxis = new CategoryAxis();
    xAxis.setCategories(FXCollections.<String>observableArrayList(tableModel.
getColumnNames()));
    xAxis.setLabel("Year");
    double tickUnit = tableModel.getTickUnit();

    NumberAxis yAxis = new NumberAxis();
    yAxis.setTickUnit(tickUnit);
    yAxis.setLabel("Units Sold");

    final BarChart chart = new BarChart(xAxis, yAxis,
tableModel.getBarChartData());
    tableModel.addTableModelListener(new TableModelListener() {

        public void tableChanged(TableModelEvent e) {
            if (e.getType() == TableModelEvent.UPDATE) {
                final int row = e.getFirstRow();
                final int column = e.getColumn();
                final Object value =
((SampleTableModel) e.getSource()).getValueAt(row, column);

                Platform.runLater(new Runnable() {
                    public void run() {
                        XYChart.Series<String, Number> s =
(XYChart.Series<String, Number>) chart.getData().get(row);
                        BarChart.Data data = s.getData().get(column);
                        data.setYValue(value);
                    }
                });
            }
        }
    });
    return chart;
}

```

Add the import statements shown in [Example 3-8](#).

Example 3-8

```

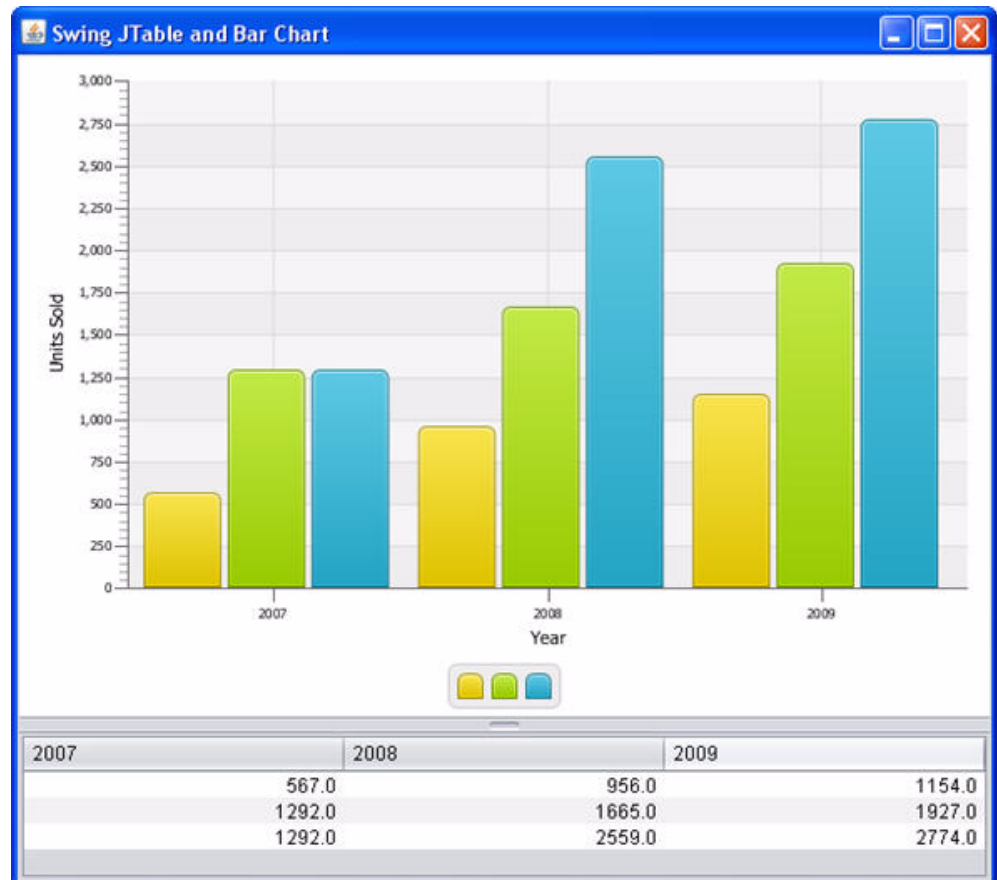
import javafx.collections.FXCollections;
import javafx.scene.chart.BarChart;
import javafx.scene.chart.CategoryAxis;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javax.swing.event.TableModelEvent;
import javax.swing.event.TableModelListener;

```

Rename the title of the frame to "Swing jTable and Bar Chart" and run the SwingInterop application.

The application window is shown in [Figure 3-2](#).

Figure 3-2 SwingInterop Application Window



View and Download Application Files

View Source Code

<http://docs.oracle.com/javafx/2/swing/SwingInterop.java.htm>

<http://docs.oracle.com/javafx/2/swing/SampleTableModel.java.htm>

Download Source Code

<http://docs.oracle.com/javafx/2/swing/SwingInterop.zip>

Leveraging Applications with Media Features

In this chapter you review a Media Player application that plays a video file and has controls typical for a video player such as a start/pause button, sliders to show playback progress and adjust volume, and a check box that turns repeat on.

For the purpose of this chapter, get familiar with the `javafx.scene.media` package that enables developers to create media applications.

About Media Integration

Any JavaFX media application can be built using the following key classes:

- `Media` class: Represents a media resource
- `MediaPlayer` class: Provides the controls for playing the specified resource
- `MediaView` class: Provides a view of the media resource played by a `MediaPlayer` object

Because the `MediaView` class is a subclass of the `Node` class, the `MediaView` object can be added to a JavaFX scene. This is the principal factor that provides a foundation for integration of the JavaFX media functionality into desktop and web applications. Now that you know how to embed the JavaFX scene into Swing applications, you can further leverage your applications by integrating the Media Player component. You can animate the `MediaView` object, transform it, and apply effects to it, just as you can with any other node. In this way, you can support numerous creative tasks.

Building the Media Player Application

The *Incorporating Media Assets Into JavaFX Applications* document provides step-by-step instructions on how to create the `EmbeddedMediaPlayer` application. It also provides the Netbeans project source. Follow the detailed instructions to build the application or download the source project using the link on the sidebar.

The `MediaPlayer` application discussed in this chapter is based on the `EmbeddedMediaPlayer` application but is slightly improved as follows:

- As a best programming practice, the application uses an external CSS file.
- The control bar contains the `Loop` check box to turn repeat on.

The application window is shown in [Figure 4-1](#).

Figure 4-1 Media Player Application Window

You can modify the EmbeddeMediaPlayer project or save its copy with a different name and modify the new project.

Skinning the Application with CSS

To skin the application with CSS, first create the `mediaplayer.css` file and save it in the folder with the source files of your application. Add the style rules shown in [Example 4-1](#).

Example 4-1

```
#mediaControl {
    -fx-background-color: #bfc2c7;
}
#mediaViewPane {
    -fx-background-color: black;;
}
```

Next, open the `MediaControl.java` file and remove from the `MediaControl` constructor the lines shown in [Example 4-2](#).

Example 4-2

```
setStyle("-fx-background-color: #bfc2c7;");
mvPane.setStyle("-fx-background-color: black;");
```

Then modify the `MediaControl` constructor by adding the lines shown in bold in [Example 4-3](#).

Example 4-3

```
public MediaControl(final MediaPlayer mp) {
    this.mp = mp;
    setId("mediaControl");

    mediaView = new MediaView(mp);
    Pane mvPane = new Pane();
    mvPane.getChildren().add(mediaView);
}
```



```
mvPane.setId("mediaViewPane");
setCenter(mvPane);
```

Adding a New Control to the Control Bar

Adding a new control to the control bar requires only a few steps. In the section where you define the `MediaControl` class instance variables, remove the definition of the `repeat` variable shown in [Example 4-4](#).

Example 4-4

```
private final boolean repeat = false;
```

In the `MediaControl` class, remove the code that used the `repeat` instance variable shown in [Example 4-5](#).

Example 4-5

```
mp.setCycleCount(repeat ? MediaPlayer.INDEFINITE : 1);
```

Now add the class variable `repeatBox` as shown in [Example 4-6](#).

Example 4-6

```
private CheckBox repeatBox;
```

Add a label and the check box to the control bar of your Media Player. Place the following code in the `MediaControl` constructor after the lines that added the `volumeSlider` to the bar, as shown in [Example 4-7](#).

Example 4-7

```
mediaBar.getChildren().add(volumeSlider);

Label repeatLabel = new Label(" Loop: ");
repeatLabel.setPrefWidth(50);
repeatLabel.setMinWidth(25);
mediaBar.getChildren().add(repeatLabel);

repeatBox = new CheckBox();
repeatBox.setSelected(true);
mediaBar.getChildren().add(repeatBox);

setBottom(mediaBar);
```

Implement the logic of using the check box in the `setOnEndOfMedia` method, as shown in [Example 4-8](#).

Example 4-8

```
mp.setOnEndOfMedia(new Runnable() {

    public void run() {
        if (repeatBox.isSelected()) {
            mp.seek(mp.getStartTime());
        } else {
            playButton.setText(">");
            stopRequested = true;
            atEndOfMedia = true;
        }
    }
});
```

```
        }  
    }  
});
```

You can view the source files and download the `MediaPlayer.zip` file with the NetBeans project.

To enable the Media Player access a remote media resource when running from behind a firewall, provide the proxy settings in the following format:

`-Dhttp.proxyHost=yourproxyhost.com -Dhttp.proxyPort=portNumber`. In this example, `yourproxyhost.com` is your proxy and `portNumber` is a port number to use.

View and Download Application Files

View Source Code

<http://docs.oracle.com/javafx/2/swing/MediaPlayer.java.htm>

<http://docs.oracle.com/javafx/2/swing/MediaControl.java.htm>

Download Source Code

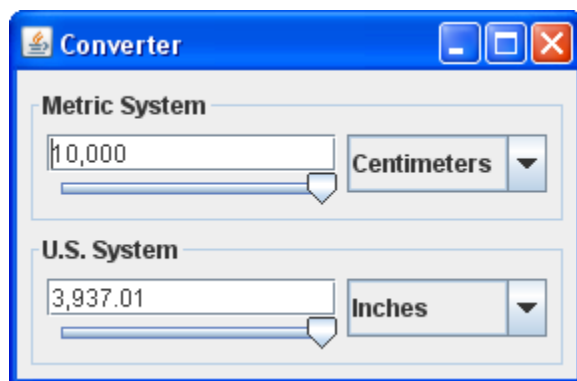
<http://docs.oracle.com/javafx/2/swing/MediaPlayer.zip>

Implementing a Swing Application in JavaFX

In this chapter, you consider a Swing application and learn how to implement it in JavaFX.

For the purpose of this chapter, get familiar with the Converter application shown in [Figure 5-1](#). This application converts distance measurements between metric and U.S. units.

Figure 5-1 Converter Application in Java



Analyzing the Converter Application Developed in Swing

For more information about the implementation of this example in the Java programming language, see [How to Use Panels](#) and [Using Models](#) trails in the [Swing tutorial](#). In particular, the graphical user interface (GUI) is discussed in the trail about the panels.

To learn the code of the Converter application, download its NetBeans project or the source files available at the [example index](#).

Swing components use models. If you look at the contents of the project, you notice the `ConverterRangeModel` and `FollowerRangeModel` classes that define models for the Converter application.

The Converter application consists of the following files:

- `ConversionPanel.java` — contains a custom `JPanel` subclass to hold components
- `Converter.java` — contains the main application class
- `ConverterRangeModel.java` — defines the top slider's model

- `FollowerRangeModel.java` — defines the bottom slider's model
- `Units.java` — creates `Unit` objects

Note that the synchronization between each text field and its slider is implemented by event handlers that listen for changes in values.

Planning the Converter Application in JavaFX

The Converter application contains two similar panels that hold components such as a text field, slider, and combo box. The panels have titles. The `TitlePane` class from the `javafx.scene.control` package ideally suits the GUI of the Converter application.

In what follows, you will implement the `ConversionPanel` class and add two instances of this class to the graphical scene of the Converter application.

First, note that the components within a single `ConversionPanel` object should be synchronized as follows. Whenever you move the knob on the slider, you must update the value in the text field and vice versa: Whenever you change the value in the text field, you must adjust the position of the knob on the slider.

As soon as you choose another value from the combo box, you must update the value of the text field and, hence, the position of the knob on the slider.

Second, note that both `ConversionPanel` objects should be synchronized. As soon as changes happen on one panel, the corresponding components on another panel must be updated.

It is suggested that you implement synchronization between the panels using the `DoubleProperty` object, called `meters`, and listen to changes in the properties of the text fields and combo boxes by creating and registering two `InvalidationListener` objects: `fromMeters` and `toMeters`. Whenever the property of the text field on one panel changes, the `invalidated` method of the attached `InvalidationListener` object is called, which updates the `meters` property. Because the `meters` property changes, the `invalidated` method of the `InvalidationListener` object, attached to the `meters` property, is called, which updates the corresponding text field on another panel.

Similarly, whenever the property of the combo box on one panel changes, the `invalidated` method of the attached `InvalidationListener` object is called, which updates the text field on this panel.

To provide synchronization between the value of the slider and the value of the `meters` object, use bidirectional binding.

For more information about JavaFX properties and binding, see [Using JavaFX Properties and Binding](#).

Creating the Converter Application in JavaFX

Create a new JavaFX project in NetBeans IDE and name it Converter. Copy the `Unit.java` file from the Swing application to the Converter project. Add a new `java` class to this project and name it `ConversionPanel.java`.

Standard JavaFX Pattern to Create the GUI

Before you start creating the GUI of the Converter application in JavaFX, see the standard pattern of GUI creation in Swing applications, as shown in [Example 5-1](#).

Example 5-1

```
public class Converter {
    private void initAndShowGUI() {
        ...
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                initAndShowGUI();
            }
        });
    }
}
```

To map this pattern to JavaFX, you extend the `javafx.application.Application` class, override the `start` method, and call the `main` method, as shown in [Example 5-2](#).

Example 5-2

```
import javafx.application.Application;
import javafx.stage.Stage;

public class Converter extends Application {
    @Override
    public void start(Stage t) {
        ...
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

When you create a new JavaFX project in the NetBeans IDE, this pattern is automatically generated for you. However, it is important that you understand the basic approach to GUI creation in JavaFX, especially if you use a text editor.

Containers and Layouts

In Swing, containers and layout managers are different entities. You create a container, such as a `JPanel` or `JComponent` object, and set a layout manager for this container. You can assign a specific layout manager and write `.add()` in your code or assign none of the layout managers.

In JavaFX, the container itself takes care of laying out its child nodes. You create a specific layout pane, such as a `Vbox`, `FlowPane`, or `TitledPane` object, and then add content to the list of its child nodes using the `.getChildren().add()` methods.

There are several layout container classes in JavaFX, called panes, some of which have their counterparts in Swing, such as the `FlowPane` class in JavaFX and `FlowLayout` class in Swing.

For more information, see [Working With Layouts in JavaFX](#).

UI Controls

JavaFX SDK provides a set of standard UI controls. Some of the UI controls have their counterparts in Swing such as the `Button` class in JavaFX and `JButton` in Swing;

`Slider` in JavaFX and `JSlider` in Swing; and `TextField` in JavaFX and `JTextField` in Swing.

To implement the Converter application in JavaFX, you can use the standard UI controls provided by the `TextField`, `Slider`, and `ComboBox` classes.

For more information, see [Using JavaFX UI Controls](#).

Usage of the Builder Classes

The JavaFX SDK provides a set of builder classes that can be used to create objects. For example, the `SliderBuilder` class is used to create objects of the `Slider` class. The builder classes and the classes whose objects they build reside within the same packages. To create an object using the corresponding builder class, see the code pattern shown in [Example 5-3](#).

Example 5-3

```
slider=SliderBuilder.create().max(MAX).build();
```

Note that the usage of builders is not compulsory. You might use builders for your convenience, or you might not. An alternative way of creating the same object without using the builder class is shown in [Example 5-4](#).

Example 5-4

```
slider = new Slider();  
slider.setMax(MAX);
```

Mechanism of Getting Notifications on User Actions and Binding

In Swing, you can register a listener on any component and listen for changes in the component properties, such as size, position, or visibility; or listen for events, such as whether the component gained or lost the keyboard focus; or whether the mouse was clicked, pressed, or released over the component.

In JavaFX, each object has a set of properties for which you can register a listener. The listener is called whenever a value of the property changes.

Note that an object can be registered as a listener for changes in another object's properties. Thus, you can use the binding mechanism to synchronize some properties of two objects.

Creating the ConversionPanel Class

The `ConversionPanel` class is used to hold components: a text field, a slider, and a combo box. When creating the graphical scene of the Converter application, you add two instances of the `ConversionPanel` class to the graphical scene. Add the import statement for the `TitledPane` class and extend the `ConversionPanel` class as shown in [Example 5-5](#).

Example 5-5

```
import javafx.scene.control.TitledPane;  
  
public class ConversionPanel extends TitledPane {  
  
}
```

Creating Instance Variables for UI Controls

Add import statements for the `TextField`, `Slider`, `ComboBox` controls and define instance variables for the components as shown in [Example 5–6](#).

Example 5–6

```
import java.text.NumberFormat;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Slider;
import javafx.scene.control.TextField;

private ComboBox<Unit> comboBox;
private Slider slider;
private TextField textField;
```

Creating DoubleProperty and NumberFormat Objects

Add the import statement for the `DoubleProperty` class and create a `DoubleProperty` object named `meters` as shown in [Example 5–7](#). The `meters` object is used to ensure the synchronization between two `ConversionPanel` objects.

Example 5–7

```
import javafx.beans.property.DoubleProperty;

private DoubleProperty meters;
```

Add the import statement for the `NumberFormat` class and add the block of code after this import statement to define the text field format as shown in [Example 5–8](#).

Example 5–8

```
private NumberFormat numberFormat;

{
    numberFormat = NumberFormat.getNumberInstance();
    numberFormat.setMaximumFractionDigits(2);
}
```

Laying Out the Components

To lay out the text field and the slider, use the `VBox` class. To lay out both of these components and a combo box, use the `HBox` class. Add the import statements for the `ObservableList`, `TextFieldBuilder`, `SliderBuilder`, `ComboBoxBuilder`, `HBoxBuilder`, `VBoxBuilder` classes and implement the constructor of the `ConversionPanel` class as shown in [Example 5–9](#).

Example 5–9

```
import javafx.collections.ObservableList;
import javafx.scene.control.ComboBoxBuilder;
import javafx.scene.control.SliderBuilder;
import javafx.scene.control.TextFieldBuilder;
import javafx.scene.layout.HBoxBuilder;
import javafx.scene.layout.VBoxBuilder;
```

```

public ConversionPanel(String title, ObservableList<Unit> units,
DoubleProperty meters) {
    setText(title);
    setCollapsible(false);
    setContent(HBoxBuilder.create()
        .children(
            VBoxBuilder.create()
                .children(
                    textField = TextFieldBuilder.create()
                        .build(),
                    slider = SliderBuilder.create()
                        .max(MAX)
                        .build()
                )
            .build(),
            comboBox = ComboBoxBuilder.<Unit>create()
                .items(units)
                .converter(new StringConverter<Unit>() {

                    @Override
                    public String toString(Unit t) {
                        return t.description;
                    }

                    @Override
                    public Unit fromString(String string) {
                        throw new UnsupportedOperationException("Not supported yet.");
                    }
                })
            .build()
        )
        .build());
    this.meters = meters;

    comboBox.getSelectionModel().select(0);
}

```

The last line of code selects a value in the `ComboBox` object.

Creating InvalidationListener Objects

To listen to changes in the properties of the text fields and combo boxes, create the `InvalidationListener` objects `fromMeters` and `toMeters` as shown in [Example 5-10](#).

Example 5-10

```

import javafx.beans.InvalidationListener;

private InvalidationListener fromMeters = new InvalidationListener() {

    @Override
    public void invalidated(Observable arg0) {
        if (!textField.isFocused()) {
            textField.setText(numberFormat.format(meters.get() /
getMultiplier()));
        }
    }
};

private InvalidationListener toMeters = new InvalidationListener() {

```



```

@Override
public void invalidated(Observable arg0) {
    if (!textField.isFocused()) {
        return;
    }
    try {
        meters.set(numberFormat.parse(textField.getText()).doubleValue() *
getMultiplier());
    } catch (Exception ignored) {
    }
}

```

Adding Change Listeners to Controls and Ensuring Synchronization

To provide the synchronization between the text fields and combo boxes, add change listeners as shown in [Example 5–11](#).

Example 5–11

```

meters.addListener(fromMeters);
comboBox.valueProperty().addListener(fromMeters);
textField.textProperty().addListener(toMeters);
fromMeters.invalidated(null);

```

Create a bidirectional binding between the value of the slider and the value of the meters object as shown in [Example 5–12](#).

Example 5–12

```

slider.valueProperty().bindBidirectional(meters);

```

When a new value is typed in the text field, the `invalidated` method of the `toMeters` listener is called, which updates the value of the meters object.

Creating the Converter Class

Open the `Converter.java` file that was automatically generated by the NetBeans IDE and remove all of the code except for the `main` method. Then, press `Ctrl` (or `Cmd`)+`Shift`+`I` to correct the import statements.

Defining Instance Variables

Add import statements for the `ObservableList`, `DoubleProperty`, and `SimpleDoubleProperty` classes and create `metricDistances`, `usaDistances`, and `meters` variables of the appropriate types as shown in [Example 5–13](#).

Example 5–13

```

import javafx.beans.property.DoubleProperty;
import javafx.collections.ObservableList;
import javafx.beans.property.SimpleDoubleProperty;

private ObservableList<Unit> metricDistances;
private ObservableList<Unit> usaDistances;
private DoubleProperty meters = new SimpleDoubleProperty(1);

```

Creating the Constructor for the Converter Class

In the constructor for the `Converter` class, create `Unit` objects for the metric and the U.S. distances as shown in [Example 5–14](#). Add the import statement for the `FXCollections` class. Later, you will instantiate two `ConversionPanel` objects with these units.

Example 5–14

```
import javafx.collections.FXCollections;

public Converter() {
    metricDistances = FXCollections.observableArrayList(
        new Unit("Centimeters", 0.01),
        new Unit("Meters", 1.0),
        new Unit("Kilometers", 1000.0));

    usaDistances = FXCollections.observableArrayList(
        new Unit("Inches", 0.0254),
        new Unit("Feet", 0.305),
        new Unit("Yards", 0.914),
        new Unit("Miles", 1613.0));
}
```

Creating the Graphical Scene

Override the `start` method to create the graphical scene for your `Converter` application. Add two `ConversionPanel` objects to the graphical scene and lay out them vertically. Note that two `ConversionPanel` objects are instantiated with the same `meters` object. Use the `VBoxBuilder` class as a root container for the graphical scene. Add import statements for the `SceneBuilder`, `VBoxBuilder`, and `StageBuilder` classes and instantiate two `ConversionPanel` objects as shown in [Example 5–15](#).

Example 5–15

```
import javafx.scene.SceneBuilder;
import javafx.scene.layout.VBoxBuilder;
import javafx.stage.StageBuilder;

@Override
public void start(Stage stage) {

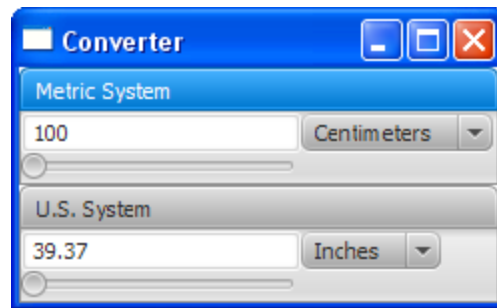
    StageBuilder.create()
        .scene(SceneBuilder.create()
            .root(VBoxBuilder.create()
                .children(
                    new ConversionPanel(
                        "Metric System", metricDistances, meters),
                    new ConversionPanel(
                        "U.S. System", usaDistances, meters))
                .build())
            .applyTo(stage);

    stage.show();
}
```

You can download the source code of the `Converter` application in [JavaFX](#).

The Converter application in JavaFX is shown in [Figure 5–2](#).

Figure 5–2 Converter Application in JavaFX



Compare the two applications with the same functionality implemented using the Swing library and JavaFX.

Not only does the application in JavaFX contain three files as compared with five files of the Swing application, but the code in JavaFX is cleaner. The applications also differ in look and feel.

View and Download Application Files

View Source Code

<http://docs.oracle.com/javafx/2/swing/Converter.java.htm>

<http://docs.oracle.com/javafx/2/swing/ConversionPanel.java.htm>

Download Source Code

<http://docs.oracle.com/javafx/2/swing/Converter.zip>