Using JavaFX to Implement Multi-touch with Java* on Windows* 8

August 9

2013

This document covers the use of JavaFX to easily add multi-touch support to Windows 8 Desktop applications written in Java*. It starts by presenting an overview of JavaFX before covering two approaches to using JavaFX: using Java APIs only and using Java in conjunction with FXML. The examples are displayed in the context of a simple application tool for manipulating an image. The application is written in Java and is designed for Windows 8 devices.

Documentation for developers interested in multi-touch support in Java applications running on Windows 8 systems

Contents

1. Introduction

Multi-touch, gesture-based user interface support is inherent in Windows 8 applications written in languages that use the native Windows libraries; however, this option is not available to Java-based applications. In this case, an alternative development framework must be used. This white paper and accompanying sample applications cover JavaFX, an open source solution for adding multi-touch support to Windows 8 Desktop applications written in Java.

JavaFX is a set of Java packages designed to support the development of rich, cross-platform applications written in Java. These packages cover user interface controls, media streaming, embedded web content, and a hardware-accelerated graphics pipeline. JavaFX also includes multi-touch support, which we will discuss in this paper. JavaFX is a Java library, meaning it can be called directly from any Java code, but it also supports a declarative markup language called FXML, which can be used to construct a JavaFX user interface. We demonstrate both methods in this paper. More information on JavaFX can be found at http://www.oracle.com/technetwork/java/javafx/documentation/index.html.

2. Example Applications

The code samples used in this document are from two Java applications: one demonstrates how to use JavaFX's Java APIs, while the second uses Java in conjunction with FXML. To maintain readability of the source code, the demonstration applications are very simple, consisting of an image displayed in a main window that users can manipulate using common single and multi-touch gestures. At any point a user can reset the image to its starting condition. You can download the full source for the demo applications and then try it yourself, or use it as reference material to create your own application.

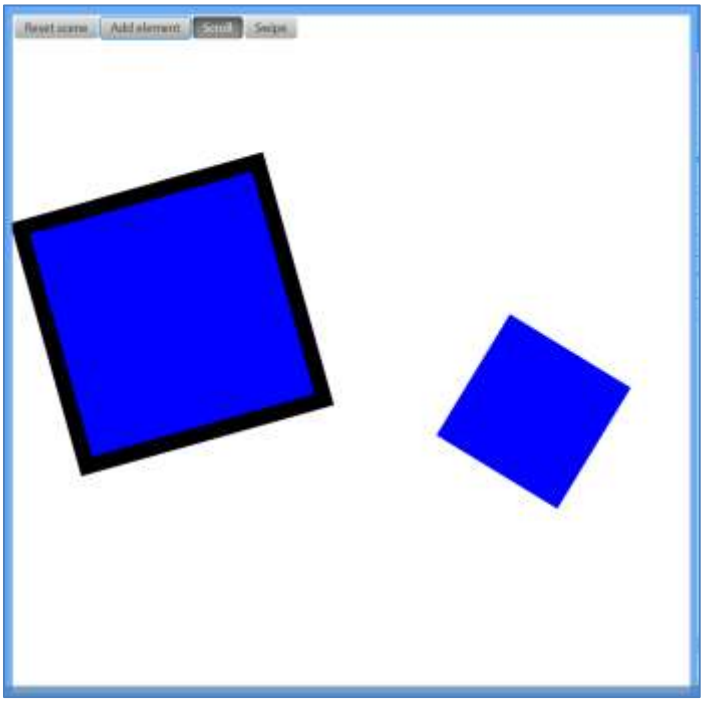

Figure 1: Sample application with multi-touch interaction

The following table describes the user interactions for the sample application.

Table 1: Supported Actions

Action	Result
Touch and drag image	Move image to a new location
Pinch on image	Decrease size of image
Spread on image	Increase size of image
Two-finger rotation on image	Rotate image
Touch [Reset] button	Restore image back to original location, size, and orientation

3. Development Environment

The example applications are Windows 8 Desktop applications, suitable for any tablet, convertible, or Ultrabook™ device with multi-touch support. JavaFX is fully integrated in the Java 7 Runtime Environment.

	Windows* 8
Language	Java* 7 or newer
Multi-touch library	JavaFX 2.2
IDE	Netbeans 7.3.1

Figure 2: Development environment

4. JavaFX Scene Structure

To display a user interface, JavaFX uses a scene graph design, an approach based on the hierarchical parent-child relationship of the elements that compose a scene. The base class for the JavaFX scene graph API is **javafx.scene**, and the base class for defining a single scene entity is **javafx.scene.Node**.

Looking deeper into the class structure, the path splits in two major directions:

- **javafx.scene.Parent** – An abstract class that can contain children, it further leads into implementation of controls, like Button or Label classes, or container constructs, like *javafx.scene.Region*.
- **javafx.scene.Shape** – This class is the starting point for various geometrical shape implementations, eventually leading to leaf nodes in the scene of type *element*.

Internally, the scene is composed of classes derived from **Node** and organized in a tree data structure. The tree is traversed by the painting engine when it is triggered by an event called a Pulse. Pulse events are emitted because of a trigger from the user or operating system or due to a time-triggered paint event. The painting engine in JavaFX is called Prism, and it abstracts OpenGL*, DirectX*, or, as a failsafe, Java2D painting engines.

In most cases, when building an application using the JavaFX framework, we will use concrete implementations of classes coming from the inheritance paths listed above, and not from the sub-classing

Node directly. However, Node is the class that provides the API for registering event handlers for touch and gesture events, therefore it will be the main focus of this article.

5. Options for Building User Interfaces Using JavaFX

JavaFX provides two options for creating user interfaces. Depending on our needs, we can synthesize the user interface by creating instances of the user interface component classes, or we can take a more visual approach by using Scene Builder to generate FXML templates. These two methods can be combined to form a flexible approach for designing the parts of the user interface. For more in depth information on creating user interfaces with JavaFX, please refer to the JavaFX tutorial (http://docs.oracle.com/javafx/).

6. Gestures in a Scene Defined Using Only Java API

The Node class provides an API to register callback event handlers. There are two ways of registering event handlers. The first is to use the `setEventHandler` function as shown in the following example:

```
setEventHandler(SwipeEvent.ANY, new EventHandler<SwipeEvent>() {
       @Override
             public void handle(SwipeEvent t) {
            …
            <event handler logic>
            …

             }
             t.consume();
       }
 });
```

The `setEventHandler` function takes two parameters:

1. The type of the event, which is a concrete instance of `EventType<T>`. The convention is that the event types are defined as static fields of the class to which they belong; for example, the SwipeEvent class would have:

```
public class SwipeEvent extends GestureEvent {
public static final EventType<SwipeEvent> ANY;
public static final EventType<SwipeEvent> SWIPE_LEFT;
public static final EventType<SwipeEvent> SWIPE_RIGHT;
public static final EventType<SwipeEvent> SWIPE_UP;
public static final EventType<SwipeEvent> SWIPE_DOWN;
...
```

2. The second parameter is a pointer to an instance of a class implementing a concrete version of the `EventHandler<T>` interface. In this example, an instance of the anonymous class was used, but the origin of the pointer is a design decision.

The second option for registering an event or gesture handler is through a set of convenience functions, also provided by the Node class. The naming convention follows `setOn<name_of_the_event>`, for example `setOnRotate(...)`. Since a convenience function corresponds to a single event, only one parameter is required, a pointer to an instance of a class that implements a concrete version of the `EventHandler<T>` interface. For example:

```
setOnTouchPressed(new EventHandler<TouchEvent>() {
        @Override
        public void handle(TouchEvent t) {
              …
              event handler logic
              …
              t.consume();
        }
});
```

The selection and use of these two approaches depend on the developer. In our case, `setEventHandler` was used for **SwipeEvent** while convenience functions were used for the rest of the gestures. For **SwipeEvent**, we passed `SwipeEvent.ANY` as the triggering event type and then detected the actual SwipeEvent type. This allowed us to keep the response logic in a single function instead of four almost identical convenience functions. On the other hand, implementing the logic for the other gestures using convenience functions produced code that was easier to study.

6.1.Touch Events

In our example, we use touch events to allow a user to drag the selected rectangle around the scene. The application first waits for the user to touch and hold within the borders of the rectangle and then to move her finger around the scene.

To implement such application behavior we have used three event types related to touch, which are:

- *Touch begin* – Setup the initial state that consists of the current position of the touch and a flag to indicate the event is in motion.
- *Touch move* – Retrieve the current position of the touch and calculate the translation of the rectangle position.
- *Touch end* – When the user lifts her finger, we clear the event-in-motion flag.

The *touch begin* event is implemented as follows:

```
setOnTouchPressed(new EventHandler<TouchEvent>() {
      @Override
```

```
            if (moveInProgress == false) {
                  if (m_container.getRegisterredItem() !=
                    MovableRectangle.this) {
                        m_container.unregisterItem();
                        m_container.registerItem(MovableRectangle.this);
                  }

                   moveInProgress = true;
                   touchPointId = t.getTouchPoint().getId();

                   prevPos = new Point2D(t.getTouchPoint().getSceneX(),
                   t.getTouchPoint().getSceneY());
                   System.out.println("TOUCH BEGIN " + t.toString());
             }

             t.consume();
      }
});
```

The two key points to notice are:

- The call to the consume function – Event handlers follow a phase of event processing called bubbling, where an event object is passed up to the parent object if it is not consumed by the current event handler. To prevent the parent container of our rectangle from processing this event (which would be undesirable, in this case), we must call the `consume` method of the event object.
- Proper handling of multiple concurrent touch events – On multi-touch devices we can get input from multiple touch events occurring at the same time. In our case we do not want our rectangle's drag behavior to be interrupted by other touch events occurring at the same time. To distinguish touch events, JavaFX uses a unique touch event ID, which is a component of the touch event object. In our application, we save this ID when the touch begins and react only to *touch move* events with a similar ID.

The second part of our implementation is the handler that responds to the touch's move event. Its implementation is shown below:

```
setOnTouchMoved(new EventHandler<TouchEvent>() {
       @Override
       public void handle(TouchEvent t) {
             if (moveInProgress == true &&
               t.getTouchPoint().getId() == touchPointId) {
                  Point2D currPos = new Point2D(
                    t.getTouchPoint().getSceneX(),
                    t.getTouchPoint().getSceneY();
```

```
                    double[] translationVector = new double[2];
                    translationVector[0] = currPos.getX() - prevPos.getX();
                    translationVector[1] = currPos.getY() - prevPos.getY();

                    setTranslateX(getTranslateX() + translationVector[0]);
                    setTranslateY(getTranslateY() + translationVector[1]);

                    prevPos = currPos;
              }
              t.consume();
        }
});
```

Similar to the previous case, we consume the event after processing it. As mentioned before, we only calculate position translations for touch events with an ID equal to the ID value of the original touch (saved by the *touch begin* handler).

The final part is the *touch end* handler. The purpose of this handler is to clear the move-in-progress flag, completing the touch event. The next *touch begin* event will register another touch ID and start the process over.

```
setOnTouchReleased(new EventHandler<TouchEvent>() {
        @Override
              public void handle(TouchEvent t) {
              if (t.getTouchPoint().getId() == touchPointId) {
                    moveInProgress = false;
              }
              t.consume();
        }
});
```

6.2.Gestures

There are currently four types of gesture events supported in JavaFX:

1. Rotate
2. Scroll
3. Swipe
4. Zoom

Architectural Considerations

When dealing with these types of gestures, it is important to carefully choose the node on the scene that will register the event handlers. Let's look at the **Zoom** gesture.

Our first thought is that if we are going to modify the size of our rectangle object (zoom), then it should be the rectangle that registers the event handler and processes the event. But what about the situation where the rectangle is small enough that a proper zoom gesture cannot be performed? Having the rectangle handle the event in this case, leads to a situation where we cannot un-zoom our rectangle. A better choice is to use the Pane element, which is a background canvas stretched to the size of the screen. This class receives callback information from its children when they are selected, and if a gesture occurs, performs the proper transformation on the selected node.

Rotate Gesture

The *rotate* event handler is implemented as follows:

```
setOnRotate(new EventHandler<RotateEvent>() {
      @Override
      public void handle(RotateEvent t) {
            if (currentSelection != null) {
                  Node selNode = currentSelection.getCorrespondingNode();
                  selNode.setRotate(selNode.getRotate() + t.getAngle());
            }
      t.consume();
      }
});
```

The implementation follows the convention of using the convenience function to register the callback handler. The **RotateEvent** class provides all parameters necessary to describe the gesture. The **Node** class provides a set of helper functions for both 2-D and 3-D. An alternative method is also available to allow you to stack multiple transformations by using the `javafx.scene.transform.Transform` type and deriving from it classes that implement more specific types of transformations, like `javafx.scene.transform.Translate`.

Zoom and Scroll Gestures

The *zoom* and *scroll* gesture event handlers are implemented similarly. Here is the *zoom* event handler:

```
setOnZoom(new EventHandler<ZoomEvent>() {
      @Override
      public void handle(ZoomEvent t) {
            if (currentSelection != null) {
                  Node selNode = currentSelection.getCorrespondingNode();
                  selNode.setScaleX(selNode.getScaleX() * t.getZoomFactor());
                  selNode.setScaleY(selNode.getScaleY() * t.getZoomFactor());
            }
            t.consume();
      }
});
```

And the *scroll* gesture event handler:

```
setOnScroll(new EventHandler<ScrollEvent>() {
      @Override
      public void handle(ScrollEvent t) {
            if (selectedGesture == GestureSelection.SCROLL &&
                    currentSelection != null) {
                  Node selNode = currentSelection.getCorrespondingNode();
                  selNode.setTranslateX(selNode.getTranslateX() +
                    (t.getDeltaX() / 10.0));
                  selNode.setTranslateY(selNode.getTranslateY() +
                    (t.getDeltaY() / 10.0));
            }
            t.consume();
      }
});
```

Swipe Gestures

For the last gesture type on the list, *swipe*, the application took a slightly different approach. A single swipe event has four different convenience functions that apply to the possible directions of the swipe gesture: `setOnSwipeLeft`, `setOnSwipeRight`, `setOnSwipeUp`, and `setOnSwipeDown`. In our application the *translation direction* is defined by the direction of the swipe, which makes it reasonable to implement using `setEventHandler` instead of the convenience functions. This results in a very simple implementation, as you can see in the following code:

```
setEventHandler(SwipeEvent.ANY, new EventHandler<SwipeEvent>() {
      @Override
      public void handle(SwipeEvent t) {
            if (selectedGesture == GestureSelection.SWIPE &&
              currentSelection != null) {
                Node selNode =currentSelection.getCorrespondingNode();
                TranslateTransition transition = new
                TranslateTransition(Duration.millis(1000), selNode);
                if (t.getEventType() == SwipeEvent.SWIPE_DOWN) {
                      transition.setByY(100);
                } else if (t.getEventType() == SwipeEvent.SWIPE_UP) {
                      transition.setByY(-100);
                } else if (t.getEventType() ==SwipeEvent.SWIPE_LEFT) {
                      transition.setByX(-100);
                } else if (t.getEventType()==SwipeEvent.SWIPE_RIGHT) {
                      transition.setByX(100);
                }
                transition.play();

            }
            t.consume();
      }
});
```

6.3.Cautions When Working with Gestures

We want to conclude this section by discussing two cases related to gesture events of which developers should be aware.

First, we note that swipe and scroll events are very similar in nature, and in fact, a swipe gesture will trigger the scroll event at the same time. In our example we give the user the option to choose which should be recognized, so both can be easily evaluated

Second, a situation that might produce undesirable effects is when a user drags the rectangle on the screen. Like a swipe event, this will also trigger a scroll event, which can generate unexpected motion of the rectangle. To prevent this situation from happening, the application consumes both the scroll and swipe events from the rectangle at the same time to prevent them from bubbling to the canvas.

```
setOnScroll(new EventHandler<ScrollEvent>() {
      @Override
      public void handle(ScrollEvent t) {
            t.consume();
```

```
      }
});

setEventHandler(SwipeEvent.ANY, new EventHandler<SwipeEvent>() {
      @Override
      public void handle(SwipeEvent t) {
            t.consume();
      }
});
```

7. Defining a Scene Using Java and FXML

In addition to the low-level approach when building user interfaces, JavaFX provides another option, one based on an XML-syntax language called FXML. FXML is a higher-level, declarative markup language used to describe the user interface for a JavaFX application. Developers can write FXML directly or use the JavaFX Scene Builder to create FXML markup. The advantages of using FXML are that it cleanly separates user interface design from application logic and it gives user interface designers an easier way to be involved in the development process.

When using FXML, the FXML file is dynamically loaded into the application, where it is converted in to a tree structure just as if you built it entirely using Java code. The resulting root Node element can then be connected directly onto a scene, or plugged in as another part of a bigger project.

7.1.Defining Scene Elements in JavaFX

There are different approaches when building user interfaces in FXML; this document will cover the following two:

- Building a component hierarchy from ready-to-use classes
- Building custom components using a root element defined in Java

The first approach, which can also use custom components, uses previously defined tags that implement a specific user interface element. For example:

```
<Pane id="StackPane" fx:id="touchPane" onRotate="#onRotate"
 onScroll="#onScroll" onSwipeDown="#onSwipe" onSwipeLeft="#onSwipe"
 onSwipeRight="#onSwipe" onSwipeUp="#onSwipe" onZoom="#onZoom"
 prefHeight="1000.0" prefWidth="1000.0" xmlns:fx=http://javafx.com/fxml
 fx:controller="jfxgestureexample2.TouchPaneController">
      <children>
            <HBox fx:id="buttons" fillHeight="false" prefHeight="30.0"
              prefWidth="451.0" spacing="3.0">
                <children>
                      ...CUT...
                </children>
```

```
                  <padding>
                        <Insets bottom="3.0" left="3.0" right="3.0"
                         top="3.0" />
                  </padding>
            </HBox>
      </children>
</Pane>
```

FXML files like this example are parsed using `FXMLLoader`, which creates a tree object structure and adds it to the scene:

```
Parent root = FXMLLoader.load(getClass().getResource("TouchPane.fxml"));

      Scene scene = new Scene(root);
      stage.setScene(scene);
      stage.show();
```

The second approach is useful when we want to create a custom user interface component or a user interface portion with a custom-class Root element. In this case, we use fx:root and specify the type of the root class. Note, the type does not have to be one of our classes, but it has to be a type within the inheritance path.

```
<fx:root type="jfxgestureexample2.MovableElementController"
  onScroll="#onScroll" onSwipeDown="#onSwipe" onSwipeLeft="#onSwipe"
  onSwipeRight="#onSwipe" onSwipeUp="#onSwipe" onTouchMoved="#onTouchMoved"
  onTouchPressed="#onTouchPressed" onTouchReleased="#onTouchReleased"
  prefHeight="50.0" prefWidth="50.0" styleClass="mainFxmlClassUnselected"
  xmlns:fx="http://javafx.com/fxml">
      <stylesheets>
            <URL value="@movableelement.css" />
      </stylesheets>
</fx:root>
```

The actual setup of the root element, along with the controller class, occurs after the FXML file has been loaded but before it is instantiated. In our example application, this happens inside the `MovableElementController` class, which is both the controller and root object itself. In addition, it implements the rectangle that is visible on the scene.

```
public MovableElementController(ISelectableItemContainer container) {
      super();
```

```
      m_container = container;

      FXMLLoader loader = new
        FXMLLoader(getClass().getResource("MovableElement.fxml"));
      loader.setRoot(this);
      loader.setController(this);

      try {
            loader.load();
      } catch (IOException exception) {
            throw new RuntimeException(exception);
      }
}
```

7.2.Connecting FXML and Back-end Code

The next step is to connect events from FXML to the back-end code and to connect the back-end code to the user interface. FXML provides a very convenient API for exposing the elements present in the FXML document and connecting callback functions to those elements. It heavily relies on Java annotations and the reflection mechanism, combined with JavaFX properties and bindings.

First, on the FXML side, each type of tag exposes a set of properties to which we can assign handlers that are present in the controller class. The second part of the definition has either the controller class set directly in FXML using the `fx:controller` property or assigned dynamically, similar to the previous example.

```
<Pane id="StackPane" fx:id="touchPane" onRotate="#onRotate"
  onScroll="#onScroll" onSwipeDown="#onSwipe" onSwipeLeft="#onSwipe"
  onSwipeRight="#onSwipe" onSwipeUp="#onSwipe" onZoom="#onZoom"
  prefHeight="1000.0" prefWidth="1000.0" xmlns:fx=http://javafx.com/fxml
  fx:controller="jfxgestureexample2.TouchPaneController">
```

The name of the handlers must match methods found in the controller class and are marked with a hash (#) symbol preceding the name. On the Java code side, such methods are declared using the `@FXML` annotation:

```
@FXML
public void onZoom(ZoomEvent t) {
      if (currentSelection != null) {
            Node selNode = currentSelection.getCorrespondingNode();
            selNode.setScaleX(selNode.getScaleX() * t.getZoomFactor());
            selNode.setScaleY(selNode.getScaleY() * t.getZoomFactor());
```

```
        }
        t.consume();
}
```

The specified method has the same declaration structure as the *handle* method from the `EventHandler` interface and it accepts proper parameters.

To expose an element present in the FXML document to the controller class we have to assign it and its ID name, as in this example:

```
<ToggleButton id="setScrollBtn" fx:id="setSwipeBtn" mnemonicParsing="false"
  text="Swipe" toggleGroup="$gestureSelectionGroup" />
```

Plus we need to declare a property matching the type and name used in the controller class:

```
public class TouchPaneController implements Initializable,
  ISelectableItemContainer {

        ...
        @FXML private Pane touchPane;
        @FXML private HBox buttons;
        @FXML ToggleButton setScrollBtn;
        @FXML ToggleButton setSwipeBtn;
        @FXML ToggleGroup gestureSelectionGroup;
        ...
```

This is all that is required from the developer; the framework will handle the binding.

7.3.Defining Functions for Touch Events and Gestures

The application logic for the FXML example is the same as the direct Java API example, so the actual gestures and touch event handlers are almost identical. To keep you from referring back to the earlier sections we will present the code from the FXML implementation here. It is worth mentioning that the same callback function is assigned to all four swipe actions in the user interface declaration in the FXML file.

```
public class MovableElementController extends Pane implements ISelectableItem
{

        ... CUT...
```

```
@FXML
public void onTouchPressed(TouchEvent t) {
      if (moveInProgress == false) {
            if (m_container.getRegisterredItem() !=
              MovableElementController.this) {
                m_container.unregisterItem();

                m_container.registerItem(
                  MovableElementController.this);
            }

            moveInProgress = true;
            touchPointId = t.getTouchPoint().getId();

            prevPos = new Point2D(t.getTouchPoint().getSceneX(),
              t.getTouchPoint().getSceneY());
            System.out.println("TOUCH BEGIN " + t.toString());
      }

      t.consume();
}

@FXML
public void onTouchMoved(TouchEvent t) {
      if (moveInProgress == true && t.getTouchPoint().getId() ==
        touchPointId) {

            Point2D currPos = new
              Point2D(t.getTouchPoint().getSceneX(),
              t.getTouchPoint().getSceneY());
            double[] translationVector = new double[2];
            translationVector[0] = currPos.getX() - prevPos.getX();
            translationVector[1] = currPos.getY() - prevPos.getY();

            setTranslateX(getTranslateX() + translationVector[0]);
            setTranslateY(getTranslateY() + translationVector[1]);

            prevPos = currPos;
      }
      t.consume();
}
```

```
      @FXML
      public void onTouchReleased(TouchEvent t) {
            if (t.getTouchPoint().getId() == touchPointId) {
                  moveInProgress = false;
                  System.err.println("TOUCH RELEASED " + t.toString());
            }

            t.consume();
      }

      ... CUT...
}


public class TouchPaneController implements Initializable,
  ISelectableItemContainer {

      ... CUT...

      @FXML private Pane touchPane;
      @FXML private HBox buttons;
      @FXML ToggleButton setScrollBtn;
      @FXML ToggleButton setSwipeBtn;
      @FXML ToggleGroup gestureSelectionGroup;

      ... CUT...

      @FXML
      public void onScroll(ScrollEvent t) {
            if (selectedGesture == GestureSelection.SCROLL &&
              currentSelection != null) {
                  Node selNode = currentSelection.getCorrespondingNode();
                  selNode.setTranslateX(selNode.getTranslateX() +
                    (t.getDeltaX() / 10.0));
                  selNode.setTranslateY(selNode.getTranslateY() +
                    (t.getDeltaY() / 10.0));
            }

            t.consume();
      }
```

```
@FXML
public void onZoom(ZoomEvent t) {
      if (currentSelection != null) {
            Node selNode = currentSelection.getCorrespondingNode();
            selNode.setScaleX(selNode.getScaleX() * t.getZoomFactor());
            selNode.setScaleY(selNode.getScaleY() * t.getZoomFactor());
      }

      t.consume();
}

@FXML
public void onRotate(RotateEvent t) {
      if (currentSelection != null) {
            Node selNode = currentSelection.getCorrespondingNode();
            selNode.setRotate(selNode.getRotate() + t.getAngle());
      }

      t.consume();
}

@FXML
public void onSwipe(SwipeEvent t) {
      if (selectedGesture == GestureSelection.SWIPE &&
        currentSelection != null) {
            Node selNode = currentSelection.getCorrespondingNode();
            TranslateTransition transition = new
              TranslateTransition(Duration.millis(1000), selNode);
            if (t.getEventType() == SwipeEvent.SWIPE_DOWN) {
                  transition.setByY(100);
            } else if (t.getEventType() == SwipeEvent.SWIPE_UP) {
                  transition.setByY(-100);
            } else if (t.getEventType() == SwipeEvent.SWIPE_LEFT) {
                  transition.setByX(-100);
            } else if (t.getEventType() == SwipeEvent.SWIPE_RIGHT) {
                  transition.setByX(100);
            }
            transition.play();

      }
```

```
                t.consume();
        }
}
```

Closing

JavaFX provides a powerful and flexible, means of adding multi-touch and gesture support to a Java-based application. In addition to media streaming, embedded web content and a hardware-accelerated graphics pipeline, JavaFX includes user interface components and multi-touch events. Developers can create and access these components using JavaFX API calls directly from Java, building their application's user interface piece by piece. Alternatively, they can define the user interface using the FXML scripting language—by either writing FXML directly or using JavaFX Scene Builder.

Whichever approach a developer chooses, connecting the front-end user interface with the back-end logic is a straightforward process. With JavaFX it is easy to design a well-architected, modern user interface with multi-touch support for a Java application.

You can download the full source for the demo applications and then try it yourself or use it as reference material to create your own Java-based touch application.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.