JavaFX

Handling JavaFX Events

Release 2.2

E24178-06

October 2013

This document describes how event handlers and event filters can be used to handle events such as mouse events, keyboard events, drag-and-drop events, window events, action events, touch events and others that are generated by your JavaFX application.

ORACLE®

JavaFX /Handling JavaFX Events, Release 2.2

E24178-06

Primary Author: Joni Gordon

Contributor: Lubomír Nerád

Contents

5 Working with Events from Touch-Enabled Devices

6 Working with Touch Events

Part I

About This Tutorial

In JavaFX applications, events are notifications that something has happened. As a user clicks a button, presses a key, moves a mouse, or performs other actions, events are dispatched. Registered event filters and event handlers within the application receive the event and provide a response. This tutorial describes how events are processed and provides examples of handling events.

This tutorial contains the following topics:

- Processing Events

  Describes the underlying architecture of how events are processed within JavaFX applications.

- Working with Convenience Methods

  Explains the simplest way to provide event handlers to handle the events generated as users interact with your application.

- Working with Event Filters

  Provides a sample of how event filters can be used to handle events.

- Working with Event Handlers

  Provides a sample of how event handlers can be used to handle events.

- Working with Events from Touch-Enabled Devices

  Describes the events generated for user gestures on touch-enabled devices and provides a sample that logs the events from gestures.

- Working with Touch Events

  Describes the events and touch points that are generated when a user touches a touch screen and provides a sample that shows how touch events can be used in a JavaFX application.

Expand the Table of Contents in the sidebar for a more detailed list of topics.

1

Processing Events

This topic describes events and the handling of events in JavaFX applications. Learn about event types, event targets, event capturing, event bubbling, and the underlying architecture of the event processing system.

Events are used to notify your application of actions taken by the user and enable the application to respond to the event. The JavaFX platform provides the structure for capturing an event, routing the event to its target, and enabling the application to handle the event as needed.

Events

An event represents an occurrence of something of interest to the application, such as a mouse being moved or a key being pressed. In JavaFX, an event is an instance of the `javafx.event.Event` class or any subclass of `Event`. JavaFX provides several events, including `DragEvent`, `KeyEvent`, `MouseEvent`, `ScrollEvent`, and others. You can define your own event by extending the `Event` class.

Every event includes the information described in Table 1–1.

Table 1–1 Event Properties

Property	Description
Event type	Type of event that occurred.
Source	Origin of the event, with respect to the location of the event in the event dispatch chain. The source changes as the event is passed along the chain.
Target	Node on which the action occurred and the end node in the event dispatch chain. The target does not change, however if an event filter consumes the event during the event capturing phase, the target will not receive the event.

Event subclasses provide additional information that is specific to the type of event. For example, the `MouseEvent` class includes information such as which button was pushed, the number of times the button was pushed, and the position of the mouse.

Event Types

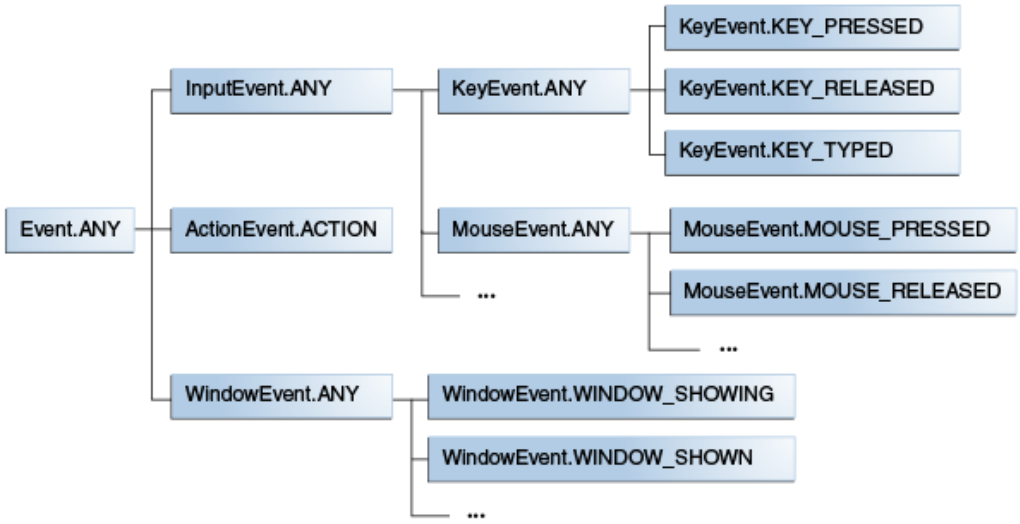
An event type is an instance of the `EventType` class. Event types further classify the events of a single event class. For example, the `KeyEvent` class contains the following event types:

- `KEY_PRESSED`

- KEY_RELEASED
- KEY_TYPED

Event types are hierarchical. Every event type has a name and a super type. For example, the name of the event for a key being pressed is KEY_PRESSED, and the super type is KeyEvent.ANY. The super type of the top-level event type is null. Figure 1–1 shows a subset of the hierarchy.

Figure 1–1 Event Type Hierarchy



The top-level event type in the hierarchy is Event.ROOT, which is equivalent to Event.ANY. In the subtypes, the event type ANY is used to mean any event type in the event class. For example, to provide the same response to any type of key event, use KeyEvent.ANY as the event type for the event filter or event handler. To respond only when a key is released, use the KeyEvent.KEY_RELEASED event type for the filter or handler.

Event Targets

The target of an event can be an instance of any class that implements the EventTarget interface. The implementation of the buildEventDispatchChain creates the event dispatch chain that the event must travel to reach the target.

The Window, Scene, and Node classes implement the EventTarget interface and subclasses of those classes inherit the implementation. Therefore, most of the elements in your user interface have their dispatch chain defined, enabling you to focus on responding to the events and not be concerned with creating the event dispatch chain.

If you create a custom UI control that responds to user actions and that control is a subclass of Window, Scene, or Node, your control is an event target through inheritance. If your control or an element of your control is not a subclass of Window, Scene, or Node, you must implement the EventTarget interface for that control or element. For example, the MenuBar control is a target through inheritance, but the MenuItem element of a menu bar must implement the EventTarget interface so that it can receive events.

Event Delivery Process

The event delivery process contains the following steps:

1. Target selection
2. Route construction
3. Event capturing
4. Event bubbling

Target Selection

When an action occurs, the system determines which node is the target based on internal rules:

- For key events, the target is the node that has focus.
- For mouse events, the target is the node at the location of the cursor. For synthesized mouse events, the touch point is considered the location of the cursor.
- For continuous gesture events that are generated by a gesture on a touch screen, the target is the node at the center point of all touches at the beginning of the gesture. For indirect gesture events that are generated by a gesture on something other than a touch screen, such as a trackpad, the target is the node at the location of the cursor.
- For swipe events that are generated by a swipe on a touch screen, the target is the node at the center of the entire path of all of the fingers. For indirect swipe events, the target is the node at the location of the cursor.
- For touch events, the default target for each touch point is the node at the location first pressed. A different target can be specified using the `ungrab()`, `grab()`, or `grab(`*node*`)` methods for a touch point in an event filter or event handler.

If more than one node is located at the cursor or touch, the topmost node is considered the target. For example, if a user clicks or touches the triangle shown in Figure 1–2, the triangle is the target, not the rectangle that contains the circle and the triangle.

Figure 1–2 Sample User Interface Event Targets

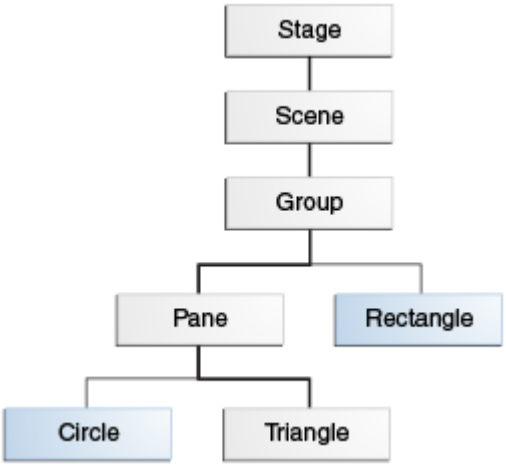
When a mouse button is pressed and the target is selected, all subsequent mouse events are delivered to the same target until the button is released. Similarly for gesture events, from the start of the gesture to the completion of the gesture, gesture events are delivered to the target identified at the beginning of the gesture. The default for touch events is to deliver the events to the initial target node that was identified for each touch point, unless the target is modified using the `ungrab()`, `grab()`, or `grab(`*node*`)` methods.

Route Construction

The initial event route is determined by the event dispatch chain that was created in the implementation of the `buildEventDispatchChain()` method of the selected event target. For example, if a user clicks the triangle shown in Figure 1–2, the initial route is shown by the gray nodes in Figure 1–3. When a scene graph node is selected as an

event target, the initial event route set in the default implementation of the `buildEventDispatchChain()` method in the `Node` class is a path from the stage to itself.

Figure 1–3 Event Dispatch Chain



The route can be modified as event filters and event handlers along the route process the event. Also, if an event filter or event handler consumes the event at any point, some nodes on the initial route might not receive the event.

Event Capturing Phase

In the event capturing phase, the event is dispatched by the root node of your application and passed down the event dispatch chain to the target node. Using the event dispatch chain shown in Figure 1–3, the event travels from the Stage node to the Triangle node during the event capturing phase.

If any node in the chain has an event filter registered for the type of event that occurred, that filter is called. When the filter completes, the event is passed to the next node down the chain. If a filter is not registered for a node, the event is passed to the next node down the chain. If no filter consumes the event, the event target eventually receives and processes the event.

Event Bubbling Phase

After the event target is reached and all registered filters have processed the event, the event returns along the dispatch chain from the target to the root node. Using the event dispatch chain shown in Figure 1–3, the event travels from the Triangle node to the Stage node during the event bubbling phase.

If any node in the chain has a handler registered for the type of event encountered, that handler is called. When the handler completes, the event is returned to the next node up the chain. If a handler is not registered for a node, the event is returned to the next node up the chain. If no handler consumes the event, the root node eventually receives the event and processing is completed.

Event Handling

Event handling is provided by event filters and event handlers, which are implementations of the `EventHandler` interface. If you want an application to be

notified when an event occurs, register a filter or a handler for the event. The primary difference between a filter and a handler is when each one is executed.

Event Filters

An event filter is executed during the event capturing phase. An event filter for a parent node can provide common event processing for multiple child nodes and if desired, consume the event to prevent the child node from receiving the event. Filters that are registered for the type of event that occurred are executed as the event passes through the node that registered the filter.

A node can register more than one filter. The order in which each filter is called is based on the hierarchy of event types. Filters for a specific event type are executed before filters for generic event types. For example, a filter for the `MouseEvent.MOUSE_PRESSED` event is called before the filter for the `InputEvent.ANY` event. The order in which two filters at the same level are executed is not specified.

Event Handlers

An event handler is executed during the event bubbling phase. If an event handler for a child node does not consume the event, an event handler for a parent node can act on the event after a child node processes it and can provide common event processing for multiple child nodes. Handlers that are registered for the type of event that occurred are executed as the event returns through the node that registered the handler.

A node can register more than one handler. The order in which each handler is called is based on the hierarchy of event types. Handlers for a specific event type are executed before handlers for generic event types. For example, a handler for the `KeyEvent.KEY_TYPED` event is called before the handler for the `InputEvent.ANY` event. The order in which two handlers at the same level are executed is not specified, with the exception that handlers that are registered by the convenience methods described in Working with Convenience Methods are executed last.

Consuming of an Event

An event can be consumed by an event filter or an event handler at any point in the event dispatch chain by calling the `consume()` method. This method signals that processing of the event is complete and traversal of the event dispatch chain ends.

Consuming the event in an event filter prevents any child node on the event dispatch chain from acting on the event. Consuming the event in an event handler stops any further processing of the event by parent handlers on the event dispatch chain. However, if the node that consumes the event has more than one filter or handler registered for the event, the peer filters or handlers are still executed.

For example, using the event dispatch chain shown in Figure 1–3, assume that the Pane node has an event filter registered for the `KeyEvent.KEY_PRESSED` event and an event filter registered for the `InputEvent.ANY` event. If the filter for the key pressed event consumes the event, the filter for the input event is executed and the Triangle node does not receive the event.

Note that the default handlers for the JavaFX UI controls typically consume most of the input events.

Additional Resources

For more information on how events are processed, see the JavaFX API documentation for the `javafx.event` package.

2

Working with Convenience Methods

This topic describes convenience methods that you can use to register event handlers within your JavaFX application. Learn an easy way to create and register event handlers to respond to mouse events, keyboard events, action events, drag-and-drop events, window events, and others.

Some JavaFX classes define event handler properties, which provide a way to register event handlers. Setting an event handler property to a user-defined event handler automatically registers the handler to receive the corresponding event type. The setter methods for the event handler properties are convenience methods for registering event handlers.

Using Convenience Methods

Many of the convenience methods are defined in the `Node` class and are available to all of its subclasses. Other classes also contain convenience methods. Table 2–1 describes the events that convenience methods can be used to handle and identifies the classes in which the convenience methods are defined.

Table 2–1 Classes with Convenience Methods for Event Handling

User Action	Event Type	Class
Key on the keyboard is pressed.	`KeyEvent`	`Node`, `Scene`
Mouse is moved or a button on the mouse is pressed.	`MouseEvent`	`Node`, `Scene`
Full mouse press-drag-release action is performed.	`MouseDragEvent`	`Node`, `Scene`
Input from an alternate method for entering characters (typically for a foreign language) is generated, changed, removed, or committed.	`InputMethodEvent`	`Node`, `Scene`
Platform-supported drag and drop action is performed.	`DragEvent`	`Node`, `Scene`
Object is scrolled.	`ScrollEvent`	`Node`, `Scene`
Rotation gesture is performed on an object	`RotateEvent`	`Node`, `Scene`
Swipe gesture is performed on an object	`SwipeEvent`	`Node`, `Scene`
An object is touched	`TouchEvent`	`Node`, `Scene`
Zoom gesture is performed on an object	`ZoomEvent`	`Node`, `Scene`

Table 2–1 (Cont.) Classes with Convenience Methods for Event Handling

User Action	Event Type	Class
Context menu is requested	`ContextMenuEvent`	`Node`, `Scene`
Button is pressed, combo box is shown or hidden, or a menu item is selected.	`ActionEvent`	`ButtonBase`, `ComboBoxBase`, `ContextMenu`, `MenuItem`,`TextField`
Item in a list, table, or tree is edited.	■ `ListView.EditEvent` ■ `TableColumn.CellEditEvent` ■ `TreeView.EditEvent`	■ `ListView` ■ `TableColumn` ■ `TreeView`
Media player encounters an error.	`MediaErrorEvent`	`MediaView`
Menu is either shown or hidden.	`Event`	`Menu`
Popup window is hidden.	`Event`	`PopupWindow`
Tab is selected or closed.	`Event`	`Tab`
Window is closed, shown, or hidden.	`WindowEvent`	`Window`

Convenience methods for registering event handlers have the following format:

```
setOnEvent-type(EventHandler<? super event-class> value)
```

`Event-type` is the type of event that the handler processes, for example, `setOnKeyTyped` for `KEY_TYPED` events or `setOnMouseClicked` for `MOUSE_CLICKED` events. *`event-class`* is the class that defines the event type, for example, `KeyEvent` for events related to keyboard input or `MouseEvent` for events related to mouse input. The string `<? super` *`event-class`*`>` indicates that the method accepts an event handler for *`event-class`* or an event handler for one of its super classes as the argument. For example, an event handler for `InputEvent` could be used when the event is either a keyboard event or a mouse event.

The following statement shows the definition for the method that registers an event handler to handle the events that are generated when a key is typed, that is, when a key is pressed and released:

```
setOnKeyTyped(EventHandler<? super KeyEvent> value)
```

You can create and register your event handler in a single step by defining the handler as an anonymous class in the call to the convenience method. The event handler must implement the `handle()` method to provide the code needed to process the event.

A example of the use of a convenience method is shown in the code that is generated when you use the NetBeans IDE to create a JavaFX application. If you select the Create Application Class option when you create your JavaFX application, the main class that is created contains a "Hello World" application. The generated code is shown in Example 2–1.

Example 2–1 Hello World Example

```
package yourapplication;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Group;
import javafx.scene.Scene;
```

```
import javafx.scene.control.Button;
import javafx.stage.Stage;

public class YourApplication extends Application {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");
        Group root = new Group();
        Scene scene = new Scene(root, 300, 250);
        Button btn = new Button();
        btn.setLayoutX(100);
        btn.setLayoutY(80);
        btn.setText("Hello World");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            public void handle(ActionEvent event) {
                System.out.println("Hello World");
            }
        });
        root.getChildren().add(btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

The "Hello World" code creates a window with a single button. The `setOnAction()` method is used to register an event handler that handles the action events that are dispatched when the button is clicked. The `handle()` method in the event handler handles the event by printing the string "Hello World" to the console.

Examples for Mouse Events

Convenience methods for registering event handlers for mouse events include `setOnMouseEntered`, `setOnMouseExited`, and `setOnMousePressed`. Example 2–2 shows samples of these event handlers.

Example 2–2 Sample Event Handlers for Mouse Events

```
final Circle circle = new Circle(radius, Color.RED);

circle.setOnMouseEntered(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        System.out.println("Mouse entered");
    }
});

circle.setOnMouseExited(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        System.out.println("Mouse exited");
    }
});
```

```
circle.setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        System.out.println("Mouse pressed");
    }
});
```

To see how similar event handlers are used, run the Ensemble sample, which is available in the JavaFX samples that can be downloaded from the JDK Demos and Samples section of the Java SE Downloads page. The Ensemble sample also provides the source code for the event handlers.

Examples for Keyboard Events

Convenience methods for registering event handlers for keyboard events include `setOnKeyPressed` and `setOnKeyReleased`. Example 2–3 shows samples of these event handlers.

Example 2–3 Sample Event Handlers for Keyboard Events

```
final TextField textBox = new TextField();
textBox.setPromptText("Write here");

textBox.setOnKeyPressed(new EventHandler<KeyEvent>() {
    public void handle(KeyEvent ke) {
        System.out.println("Key Pressed: " + ke.getText());
    }
});

textBox.setOnKeyReleased(new EventHandler<KeyEvent>() {
    public void handle(KeyEvent ke) {
        System.out.println("Key Released: " + ke.getText());
    }
});
```

To see how similar event handlers are used, run the Ensemble sample, which is available in the JavaFX samples that can be downloaded from the JDK Demos and Samples section of the Java SE Downloads page. The Ensemble sample also provides the source code for the event handlers.

Additional Resources

For information on the available convenience methods, see the JavaFX API documentation.

3

Working with Event Filters

This topic describes event filters in JavaFX applications. Learn how event filters can be used to process the events generated by keyboard actions, mouse actions, scroll actions, and other user interactions with your application.

Event filters enable you to handle an event during the event capturing phase of event processing. A node can have one or more filters for handling an event. A single filter can be used for more than one node and more than one event type. Event filters enable the parent node to provide common processing for its child nodes or to intercept an event and prevent child nodes from acting on the event.

Registering and Removing an Event Filter

To process an event during the event capturing phase, a node must register an event filter. An event filter is an implementation of the `EventHandler` interface. The `handle()` method of this interface provides the code that is executed when the event that is associated with the filter is received by the node that registered the filter.

To register a filter, use the `addEventFilter()` method. This method takes the event type and the filter as arguments. In Example 3–1, the first filter is added to a single node and processes a specific event type. A second filter for handling input events is defined and registered by two different nodes. The same filter is also registered for two different types of events.

Example 3–1 Register a Filter

```
// Register an event filter for a single node and a specific event type
node.addEventFilter(MouseEvent.MOUSE_CLICKED,
                    new EventHandler<MouseEvent>() {
                        public void handle(MouseEvent) { ... };
                    });

// Define an event filter
EventHandler filter = new EventHandler(<InputEvent>() {
    public void handle(InputEvent event) {
        System.out.println("Filtering out event " + event.getEventType());
        event.consume();
    }

// Register the same filter for two different nodes
myNode1.addEventFilter(MouseEvent.MOUSE_PRESSED, filter);
myNode2.addEventFilter(MouseEvent.MOUSE_PRESSED, filter);

// Register the filter for another event type
myNode1.addEventFilter(KeyEvent.KEY_PRESSED, filter);
```

Note that an event filter that is defined for one type of event can also be used for any subtypes of that event. See Event Types for information on the hierarchy of event types.

When you no longer want an event filter to process events for a node or for an event type, remove the filter using the `removeEventFilter()` method. This method takes the event type and the filter as arguments. In Example 3–2, the filter defined in Example 3–1 is removed from the `MouseEvent.MOUSE_PRESSED` event for `myNode1`. The filter is still executed by `myNode2` and by `myNode1` for the `KeyEvent.KEY_PRESSED` event.

Example 3–2 Remove a Filter

```
// Remove an event filter
myNode1.removeEventFilter(MouseEvent.MOUSE_PRESSED, filter);
```

Using Event Filters

Event filters are typically used on a branch node of the event dispatch chain and are called during the event capturing phase of event handling. Use a filter to perform actions such as overriding an event response or blocking an event from reaching its destination.

To see an example of how filters can be used, download the `DraggablePanelsExample.zip` file. Extract the NetBeans project and open it in the NetBeans IDE. The following sections describe the filters that are used by this example.

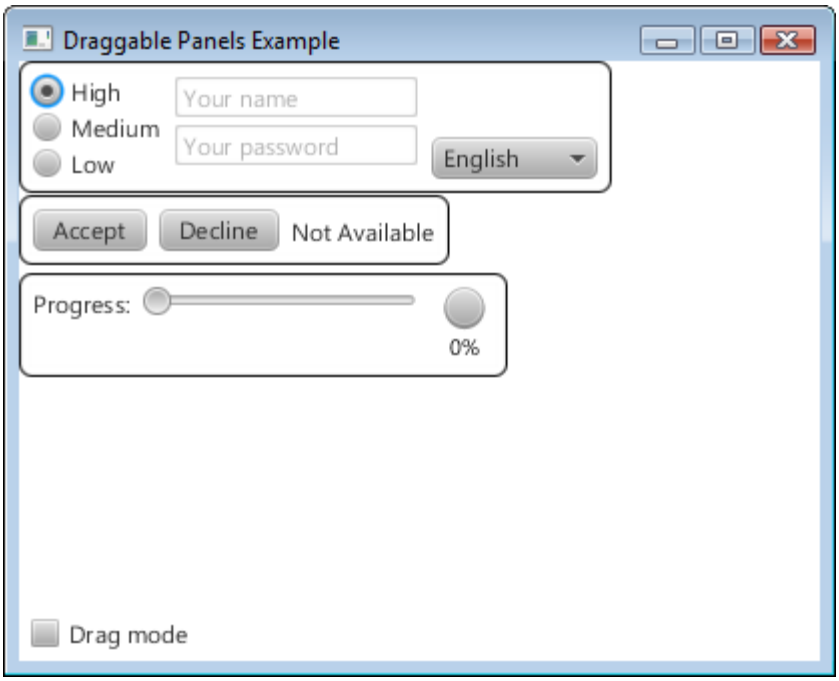
Draggable Panels Example

The Draggable Panels example demonstrates the following uses of filters:

- Registering a filter for a super-type event to provide common handling for subtype events
- Consuming an event to prevent a child node from acting on it

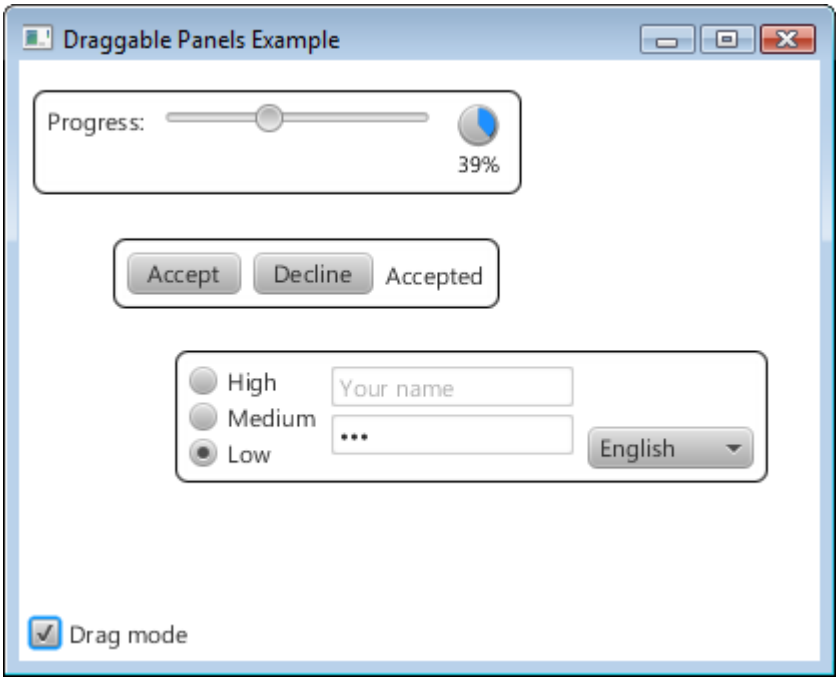
Figure 3–1 is the screen that is shown when the Draggable Panels example is started. The user interface consists of three panels. Each panel contains different UI controls. At the bottom of the screen is a check box that controls whether the panels can be dragged.

Figure 3–1 Initial Screen for the Draggable Panels Example



If the check box is not selected, clicking any of the controls in the panels generates a response from the control. If the check box is selected, the individual controls do not respond to mouse clicks. Instead, clicking anywhere within a panel and dragging the mouse moves the entire panel, enabling you to change the position of the panels as shown in Figure 3–2.

Figure 3–2 Screen with Repositioned Panels



Filters for the Draggable Panels Example

In the Draggable Panels example, the `makeDraggable()` method is used in the creation of the three panels to make each panel movable. This method and the filter definitions are shown in Example 3–3.

Example 3–3 Filter Definitions in makeDraggable()

```
private Node makeDraggable(final Node node) {
    final DragContext dragContext = new DragContext();
    final Group wrapGroup = new Group(node);

    wrapGroup.addEventFilter(
        MouseEvent.ANY,
        new EventHandler<MouseEvent>() {
            public void handle(final MouseEvent mouseEvent) {
                if (dragModeActiveProperty.get()) {
                    // disable mouse events for all children
                    mouseEvent.consume();
                }
             }
        });

    wrapGroup.addEventFilter(
        MouseEvent.MOUSE_PRESSED,
        new EventHandler<MouseEvent>() {
            public void handle(final MouseEvent mouseEvent) {
                if (dragModeActiveProperty.get()) {
                    // remember initial mouse cursor coordinates
                    // and node position
                    dragContext.mouseAnchorX = mouseEvent.getX();
                    dragContext.mouseAnchorY = mouseEvent.getY();
                    dragContext.initialTranslateX =
                        node.getTranslateX();
                    dragContext.initialTranslateY =
                        node.getTranslateY();
                }
            }
        });

    wrapGroup.addEventFilter(
        MouseEvent.MOUSE_DRAGGED,
        new EventHandler<MouseEvent>() {
            public void handle(final MouseEvent mouseEvent) {
                if (dragModeActiveProperty.get()) {
                    // shift node from its initial position by delta
                    // calculated from mouse cursor movement
                    node.setTranslateX(
                        dragContext.initialTranslateX
                            + mouseEvent.getX()
                            - dragContext.mouseAnchorX);
                    node.setTranslateY(
                        dragContext.initialTranslateY
                            + mouseEvent.getY()
                            - dragContext.mouseAnchorY);
                }
            }
        });

    return wrapGroup;
```

```
}
```

Filters for the following events are defined and registered for each panel:

- `MouseEvent.ANY`. This filter processes all mouse events for the panel. If the Drag Mode check box is selected, the filter consumes the event, and the child nodes, which are the UI controls within the panel, do not receive the event. If the check box is not selected, the control at the location of the mouse cursor processes the event.
- `MouseEvent.MOUSE_PRESSED`. This filter processes only mouse-pressed events for the panel. If the Drag Mode check box is selected, the current location of the mouse is stored.
- `MouseEvent.MOUSE_DRAGGED`. This filter processes only mouse-dragged events for the panel. If the Drag Mode check box is selected, the panel is moved.

Note that a panel has three registered filters. Filters for specific event types are invoked before super-type events, so the filters for `MouseEvent.MOUSE_PRESSED` and `MouseEvent.MOUSE_DRAGGED` are invoked before the filter for `MouseEvent.ANY`.

Additional Resources

For information on event filters, see the JavaFX API documentation.

4

Working with Event Handlers

This topic describes event handlers in JavaFX applications. Learn how event handlers can be used to process the events generated by keyboard actions, mouse actions, scroll actions, and other user interactions with your application.

Event handlers enable you to handle events during the event bubbling phase. A node can have one or more handlers for handling an event. A single handler can be used for more than one node and more than one event type. If an event handler for a child node does not consume the event, an event handler for a parent node enables the parent node to act on the event after a child node processes it and to provide common event processing for multiple child nodes.

Registering and Removing an Event Handler

To process an event during the event bubbling phase, a node must register an event handler. An event handler is an implementation of the `EventHandler` interface. The `handle()` method of this interface provides the code that is executed when the event that is associated with the handler is received by the node that registered the handler.

To register a handler, use the `addEventHandler()` method. This method takes the event type and the handler as arguments. In Example 4–1, the first handler is added to a single node and processes a specific event type. A second handler for handling input events is defined and registered by two different nodes. The same handler is also registered for two different types of events.

Example 4–1 Register a Handler

```
// Register an event handler for a single node and a specific event type
node.addEventHandler(DragEvent.DRAG_ENTERED,
                     new EventHandler<DragEvent>() {
                         public void handle(DragEvent) { ... };
                     });

// Define an event handler
EventHandler handler = new EventHandler(<InputEvent>() {
    public void handle(InputEvent event) {
        System.out.println("Handling event " + event.getEventType());
        event.consume();
    }

// Register the same handler for two different nodes
myNode1.addEventHandler(DragEvent.DRAG_EXITED, handler);
myNode2.addEventHandler(DragEvent.DRAG_EXITED, handler);

// Register the handler for another event type
```

```
myNode1.addEventHandler(MouseEvent.MOUSE_DRAGGED, handler);
```

Note that an event handler that is defined for one type of event can also be used for any subtypes of that event. See Event Types for information on the hierarchy of event types.

When you no longer want an event handler to process events for a node or for an event type, remove the handler using the `removeEventHandler()` method. This method takes the event type and the handler as arguments. In Example 4–2, the handler defined in Example 4–1 is removed from the `DragEvent.DRAG_EXITED` event for `myNode1`. The handler is still executed by `myNode2` and by `myNode1` for the `MouseEvent.MOUSE_DRAGGED` event.

Example 4–2 Remove a Handler

```
// Remove an event handler
myNode1.removeEventHandler(DragEvent.DRAG_EXITED, handler);
```

> **Tip:** To remove an event handler that was registered by a convenience method, pass null to the convenience method, for example, `node1.setOnMouseDragged(null)`.

Using Event Handlers

Event handlers are typically used on a the leaf nodes or on a branch node of the event dispatch chain and are called during the event bubbling phase of event handling. Use a handler on a branch node to perform actions such as defining a default response for all child nodes.

To see an example of how handlers can be used, download the `KeyboardExample.zip` file. Extract the NetBeans project and open it in the NetBeans IDE. The following sections describe the handlers that are used by this example.

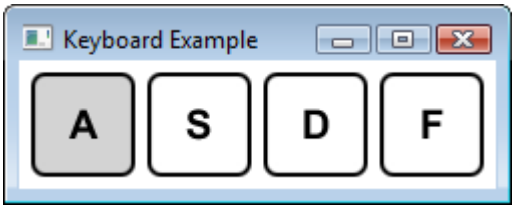
Keyboard Example

The Keyboard example demonstrates the following uses of handlers:

- Registering a single handler for two different event types
- Providing common event processing for child nodes in a parent node

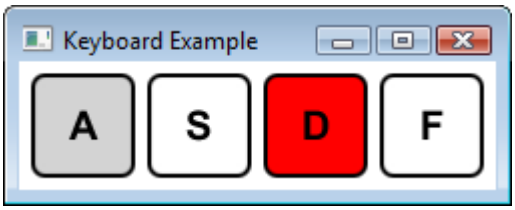
Figure 4–1 is the screen that is shown when the Keyboard Example is started. The user interface consists of four letters, each in its own square, which represent the corresponding keyboard key. The first key on the screen is highlighted, which indicates that it has the focus. Use the left and right arrow keys on the keyboard to move the focus to a different key on the screen.

Figure 4–1 Initial Screen for Keyboard Example



When the Enter key is pressed, the key on the screen with the focus changes to red. When the Enter key is released, the key on the screen returns to its previous color. When the key for a letter that matches one of the keys on the screen is pressed, the matching key on the screen changes to red, and returns to its previous color when the key is released. When a key that does not match any key on the screen is pressed, nothing happens. Figure 4–2 shows the screen when the A key has focus and the D key on the keyboard is pressed.

Figure 4–2 Key Pressed Screen



Handlers for the Keyboard Example

In the Keyboard example, internally each key shown on the screen is represented by a key node. All key nodes are contained in a single keyboard node. Each key node has a handler that receives key events when the key has focus. The handler responds to the key-pressed and key-released events for the Enter key by changing the color of the key on the screen. The event is then consumed so that the keyboard node, which is the parent node, does not receive the event.

Example 4–3 shows the `installEventHandler()` method that defines the handler for the key nodes.

Example 4–3 Handler for the Key Nodes

```
private void installEventHandler(final Node keyNode) {
    // handler for enter key press / release events, other keys are
    // handled by the parent (keyboard) node handler
    final EventHandler<KeyEvent> keyEventHandler =
        new EventHandler<KeyEvent>() {
            public void handle(final KeyEvent keyEvent) {
                if (keyEvent.getCode() == KeyCode.ENTER) {
                    setPressed(keyEvent.getEventType()
                        == KeyEvent.KEY_PRESSED);

                    keyEvent.consume();
                }
            }
        };

    keyNode.setOnKeyPressed(keyEventHandler);
    keyNode.setOnKeyReleased(keyEventHandler);
}
```

The keyboard node has two handlers that handle key events that are not consumed by a key node handler. The first handler changes the color of the key node that matches the key pressed. The second handler responds to the left and right arrow keys and moves the focus.

Example 4–4 shows `the installEventHandler()` method that defines the handlers for the keyboard node.

Example 4–4 Handlers for the Keyboard Node

```
private void installEventHandler(final Parent keyboardNode) {
    // handler for key pressed / released events not handled by
    // key nodes
    final EventHandler<KeyEvent> keyEventHandler =
        new EventHandler<KeyEvent>() {
            public void handle(final KeyEvent keyEvent) {
                final Key key = lookupKey(keyEvent.getCode());
                if (key != null) {
                    key.setPressed(keyEvent.getEventType()
                                   == KeyEvent.KEY_PRESSED);

                    keyEvent.consume();
                }
            }
        };

    keyboardNode.setOnKeyPressed(keyEventHandler);
    keyboardNode.setOnKeyReleased(keyEventHandler);

    keyboardNode.addEventHandler(KeyEvent.KEY_PRESSED,
                                 new EventHandler<KeyEvent>() {
                                     public void handle(
                                         final KeyEvent keyEvent) {
                                               handleFocusTraversal(
                                                   keyboardNode,
                                                   keyEvent);
                                     }
                                 });
}
```

The two handlers for the key-pressed event are considered peer handlers. Therefore, even though each handler consumes the event, the other handler is still invoked.

Additional Resources

For information on event handlers, see the JavaFX API documentation.

5

Working with Events from Touch-Enabled Devices

This topic describes the events that are generated by the different types of gestures that are recognized by touch-enabled devices, such as touch events, zoom events, rotate events, and swipe events. This topic shows you how to work with these types of events in your JavaFX application.

Starting with JavaFX 2.2, users can interact with your JavaFX applications using touches and gestures on touch-enabled devices. Touches and gestures can involve a single point or multiple points of contact. The type of event that is generated is determined by the touch or type of gesture that the user makes.

Touch and gesture events are processed the same way that other events are processed. See Processing Events for a description of this process. Convenience methods for registering event handlers for touch and gesture events are available. See Working with Convenience Methods for more information.

Gesture and Touch Events

JavaFX applications generate gesture events when an application is running on a device with a touch screen or a trackpad that recognizes gestures. For platforms that recognize gestures, the native recognition is used to identify the gesture performed. Table 5–1 describes the gestures that are supported and the corresponding event types that are generated.

Table 5–1 Supported Gestures and Generated Event Types

Gesture	Description	Events Generated
Rotate	Two-finger turning movement where one finger moves clockwise around the other finger to rotate an object clockwise and one finger moves counterclockwise around the other finger to rotate an object counterclockwise.	■ ROTATION_STARTED ■ ROTATE ■ ROTATION_FINISHED
Scroll	Sliding movement, up or down for vertical scrolling, left or right for horizontal scrolling.	■ SCROLL_STARTED ■ SCROLL ■ SCROLL_FINISHED If a mouse wheel is used for scrolling, only the events of type SCROLL are generated.
Swipe	Sweeping movement across the screen or trackpad to the right, left, up, or down. Diagonal movement is not recognized as a swipe.	■ SWIPE_LEFT ■ SWIPE_RIGHT ■ SWIPE_UP ■ SWIPE_DOWN A single swipe event is generated for each swiping gesture. SCROLL_STARTED, SCROLL, and SCROLL_FINISHED events are also generated.
Zoom	Two-finger pinching motion where fingers are brought together to zoom out and fingers are moved apart to zoom in.	■ ZOOM_STARTED ■ ZOOM ■ ZOOM_FINISHED

Touch events are generated when the application is running on a device with a touch screen and the user touches one or more fingers to the screen. These events can be used to provide lower level tracking for the individual touch points that are part of a touch or gesture. For more information on touch events, see Working with Touch Events.

Targets of Gestures

The target of most gestures is the node at the center point of all touches at the beginning of the gesture. The target of a swipe gesture is the node at the center of the entire path of all of the fingers.

If more than one node is located at the target point, the topmost node is considered the target. All of the events generated from a single, continuous gesture, including inertia from the gesture, are delivered to the node that was selected when the gesture started. For more information on targets of events, see Target Selection.

Other Events Generated

Gestures and touches can generate other types of events in addition to the events for the gesture or touch performed. A swipe gesture generates scroll events in addition to the swipe event. Depending on the length of the swipe, it is possible that the swipe and scroll events have different targets. The target of a scroll event is the node at the point where the gesture started. The target of a swipe event is the node at the center of the entire path of the gesture.

Touches on a touch screen also generate corresponding mouse events. For example, touching a point on the screen generates TOUCH_PRESSED and MOUSE_PRESSED events. Moving a single point on the screen generates scroll events and drag events. Even if

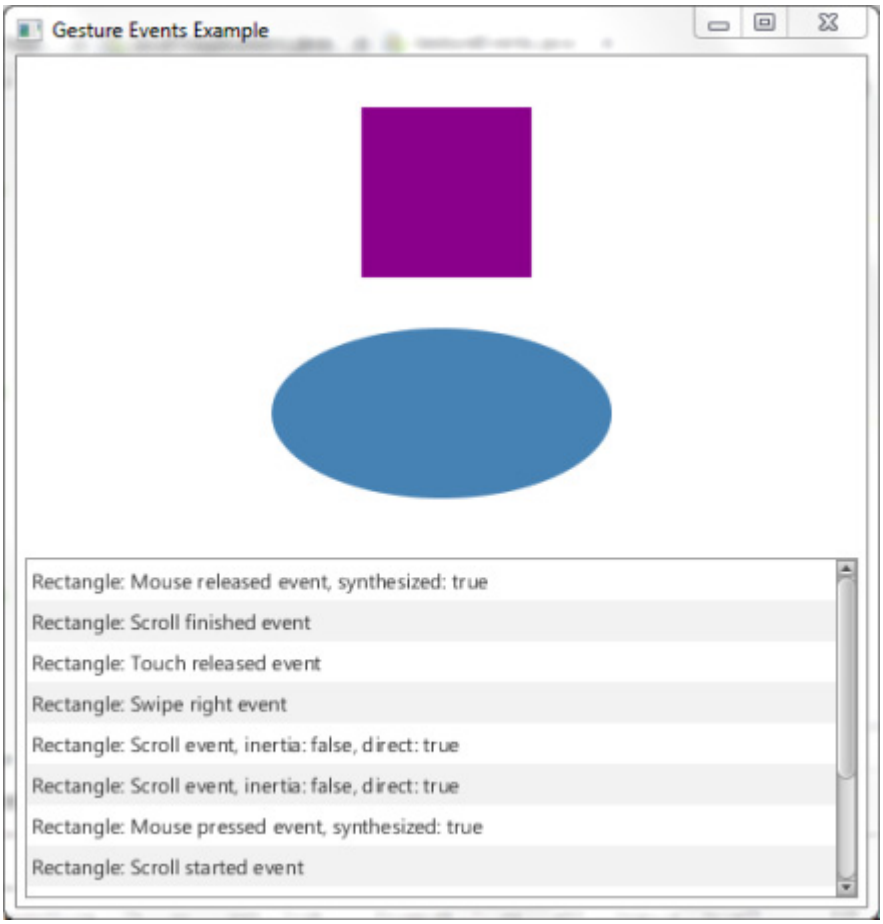
your application does not handle touch or gesture events directly, it can run on a touch-enabled device with minimal changes by responding to the mouse events that are generated in response to touches.

If your application handles touches, gestures, and mouse events, make sure that you do not handle a single action multiple times. For example, if a gesture generates scroll events and drag events and you provide the same processing for handlers of both types of events, the movement on the screen could be twice the amount expected. You can use the `isSynthesized()` method for mouse events to determine if the event was generated by mouse movement or movement on a touch screen and only handle the event once.

Gesture Events Example

The Gesture Events example shows a rectangle, an ellipse, and an event log. Figure 5–1 shows the example.

Figure 5–1 Gesture Events Example



The log contains a record of the events that were handled. This example enables you to try different gestures and see what events are generated for each.

The Gesture Events example is available in the GestureEventsExample.zip file. Extract the NetBeans project and open it in the NetBeans IDE.

To generate gesture events, you must run the example on a device with a touch screen or a trackpad that supports gestures. To generate touch events, you must run the example on a device with a touch screen.

Creating the Shapes

The Gesture Events example shows a rectangle and an ellipse. Example 5–1 shows the code used to create each shape and the layout pane that contains the shapes.

Example 5–1 Set Up the Shapes

```
// Create the shapes that respond to gestures and use a VBox to
// organize them
VBox shapes = new VBox();
shapes.setAlignment(Pos.CENTER);
shapes.setPadding(new Insets(15.0));
shapes.setSpacing(30.0);
shapes.setPrefWidth(500);
shapes.getChildren().addAll(createRectangle(), createEllipse());
...
private Rectangle createRectangle() {

    final Rectangle rect = new Rectangle(100, 100, 100, 100);
    rect.setFill(Color.DARKMAGENTA);
...
    return rect;
}

private Ellipse createEllipse() {

    final Ellipse oval = new Ellipse(100, 50);
    oval.setFill(Color.STEELBLUE);
...
    return oval;
}
```

You can use gestures to move, rotate, and zoom in and out of these objects.

Handling the Events

In general, event handlers for the shape objects in the Gesture Events example perform similar operations for each type of event that is handled. For all types of events, an entry is posted to the log of events.

On platforms that support inertia for gestures, additional events might be generated after the *event-type*`_FINISHED` event. For example, `SCROLL` events might be generated after the `SCROLL_FINISHED` event if there is any inertia associated with the scroll gesture. Use the `isInertia()` method to identify the events that are generated based on the inertia of the gesture. If the method returns `true`, the event was generated after the gesture was completed.

Events are generated by gestures on a touch screen or on a trackpad. `SCROLL` events are also generated by the mouse wheel. Use the `isDirect()` method to identify the source of the event. If the method returns `true`, the event was generated by a gesture on a touch screen. Otherwise, the method returns `false`. You can use this information to provide different behavior based on the source of the event.

Touches on a touch screen also generate corresponding mouse events. For example, touching an object generates both `TOUCH_PRESSED` and `MOUSE_PRESSED` events. Use the

isSynthesized() method to determine the source of the mouse event. If the method returns true, the event was generated by a touch instead of by a mouse.

The inc() and dec() methods in the Gesture Events example are used to provide a visual cue that an object is the target of a gesture. The number of gestures in progress is tracked, and the appearance of the target object is changed when the number of active gestures changes from 0 to 1 or drops to 0.

In the Gesture Events example, the handlers for the rectangle and ellipse are similar. Therefore, the code examples in the following sections show the handlers for the rectangle. See GestureEvents.java for the handlers for the ellipse.

Handling Scroll Events

When a scroll gesture is performed, SCROLL_STARTED, SCROLL, and SCROLL_FINISHED events are generated. When a mouse wheel is moved, only a SCROLL event is generated. Example 5–2 shows the rectangle's handlers for scroll events in the Gesture Events example. Handlers for the ellipse are similar.

Example 5–2 Define the Handlers for Scroll Events

```
rect.setOnScroll(new EventHandler<ScrollEvent>() {
        @Override public void handle(ScrollEvent event) {
            if (!event.isInertia()) {
                rect.setTranslateX(rect.getTranslateX() + event.getDeltaX());
                rect.setTranslateY(rect.getTranslateY() + event.getDeltaY());
            }
            log("Rectangle: Scroll event" +
                ", inertia: " + event.isInertia() +
                ", direct: " + event.isDirect());
            event.consume();
        }
});

rect.setOnScrollStarted(new EventHandler<ScrollEvent>() {
        @Override public void handle(ScrollEvent event) {
            inc(rect);
            log("Rectangle: Scroll started event");
            event.consume();
        }
});

rect.setOnScrollFinished(new EventHandler<ScrollEvent>() {
        @Override public void handle(ScrollEvent event) {
            dec(rect);
            log("Rectangle: Scroll finished event");
            event.consume();
        }
});
```

In addition to the common handling described in Handling the Events, SCROLL events are handled by moving the object in the direction of the scroll gesture. If the scroll gesture ends outside of the window, the shape is moved out of the window. SCROLL events that are generated based on inertia are ignored by the handler for the rectangle. The handler for the ellipse continues to move the ellipse in response to SCROLL events that are generated from inertia and could result in the ellipse moving out of the window even if the gesture ends within the window.

Handling Zoom Events

When a zoom gesture is performed, ZOOM_STARTED, ZOOM, and ZOOM_FINISHED events are generated. Example 5–3 shows the rectangle's handlers for zoom events in the Gesture Events example. Handlers for the ellipse are similar.

Example 5–3 Define the Handlers for Zoom Events

```
rect.setOnZoom(new EventHandler<ZoomEvent>() {
        @Override public void handle(ZoomEvent event) {
            rect.setScaleX(rect.getScaleX() * event.getZoomFactor());
            rect.setScaleY(rect.getScaleY() * event.getZoomFactor());
            log("Rectangle: Zoom event" +
                ", inertia: " + event.isInertia() +
                ", direct: " + event.isDirect());

            event.consume();
        }
});

rect.setOnZoomStarted(new EventHandler<ZoomEvent>() {
        @Override public void handle(ZoomEvent event) {
            inc(rect);
            log("Rectangle: Zoom event started");
            event.consume();
        }
});

rect.setOnZoomFinished(new EventHandler<ZoomEvent>() {
        @Override public void handle(ZoomEvent event) {
            dec(rect);
            log("Rectangle: Zoom event finished");
            event.consume();
        }
});
```

In addition to the common handling described in Handling the Events, ZOOM events are handled by scaling the object according to the movement of the gesture. The handlers for the rectangle and ellipse handle all ZOOM events the same, regardless of inertia or the source of the event.

Handling Rotate Events

When a rotate gesture is performed, ROTATE_STARTED, ROTATE, and ROTATE_FINISHED events are generated. Example 5–4 shows the rectangle's handlers for rotate events in the Gesture Events example. Handlers for the ellipse are similar.

Example 5–4 Define the Handlers for Rotate Events

```
rect.setOnRotate(new EventHandler<RotateEvent>() {
        @Override public void handle(RotateEvent event) {
            rect.setRotate(rect.getRotate() + event.getAngle());
            log("Rectangle: Rotate event" +
                ", inertia: " + event.isInertia() +
                ", direct: " + event.isDirect());
            event.consume();
        }
});

rect.setOnRotationStarted(new EventHandler<RotateEvent>() {
        @Override public void handle(RotateEvent event) {
```

```
            inc(rect);
            log("Rectangle: Rotate event started");
            event.consume();
        }
});

rect.setOnRotationFinished(new EventHandler<RotateEvent>() {
        @Override public void handle(RotateEvent event) {
            dec(rect);
            log("Rectangle: Rotate event finished");
            event.consume();
        }
});
```

In addition to the common handling described in Handling the Events, `ROTATE` events are handled by rotating the object according to the movement of the gesture. The handlers for the rectangle and ellipse handle all `ROTATE` events the same, regardless of inertia or the source of the event.

Handling Swipe Events

When a swipe gesture is performed, either a `SWIPE_DOWN`, `SWIPE_LEFT`, `SWIPE_RIGHT`, or `SWIPE_UP` event is generated, depending on the direction of the swipe. Example 5–5 shows the rectangle's handlers for `SWIPE_RIGHT` and `SWIPE_LEFT` events in the Gesture Events example. The ellipse does not handle swipe events.

Example 5–5 Define the Handlers for Swipe Events

```
rect.setOnSwipeRight(new EventHandler<SwipeEvent>() {
        @Override public void handle(SwipeEvent event) {
            log("Rectangle: Swipe right event");
            event.consume();
        }
});

rect.setOnSwipeLeft(new EventHandler<SwipeEvent>() {
        @Override public void handle(SwipeEvent event) {
            log("Rectangle: Swipe left event");
            event.consume();
        }
});
```

The only action performed for swipe events is to record the event in the log. However, swipe gestures also generate scroll events. The target of the swipe event is the topmost node at the center of the path of the gesture. This target could be different than the target of the scroll events, which is the topmost node at the point where the gesture started. The rectangle and ellipse respond to scroll events that are generated by a swipe gesture when they are the target of the scroll events.

Handling Touch Events

When a touch screen is touched, a `TOUCH_MOVED`, `TOUCH_PRESSED`, `TOUCH_RELEASED`, or `TOUCH_STATIONARY` event is generated for each touch point. The touch event contains information for every touch point that is part of the touch action. Example 5–6 shows the rectangle's handlers for touch pressed and touch released events in the Gesture Events example. The ellipse does not handle touch events.

Example 5–6 Define the Handlers for Touch Events

```
rect.setOnTouchPressed(new EventHandler<TouchEvent>() {
        @Override public void handle(TouchEvent event) {
            log("Rectangle: Touch pressed event");
            event.consume();
        }
});

rect.setOnTouchReleased(new EventHandler<TouchEvent>() {
        @Override public void handle(TouchEvent event) {
            log("Rectangle: Touch released event");
            event.consume();
        }
});
```

The only action performed for touch events is to record the event in the log. Touch events can be used to provide lower level tracking for the individual touch points that are part of a touch or gesture. See Working with Touch Events for more information and an example.

Handling Mouse Events

Mouse events are generated by actions with the mouse and by touches on a touch screen. Example 5–7 shows the ellipse's handlers for `MOUSE_PRESSED` and `MOUSE_RELEASED` events in the Gesture Events example.

Example 5–7 Define Handlers for Mouse Events

```
oval.setOnMousePressed(new EventHandler<MouseEvent>() {
        @Override public void handle(MouseEvent event) {
            if (event.isSynthesized()) {
                log("Ellipse: Mouse pressed event from touch" +
                    ", synthesized: " + event.isSynthesized());
            }
            event.consume();
        }
});

oval.setOnMouseReleased(new EventHandler<MouseEvent>() {
        @Override public void handle(MouseEvent event) {
            if (event.isSynthesized()) {
                log("Ellipse: Mouse released event from touch" +
                    ", synthesized: " + event.isSynthesized());
            }
            event.consume();
        }
});
```

Mouse pressed and mouse released events are handled by the ellipse only when the events are generated by touches on a touch screen. Handlers for the rectangle record all mouse pressed and mouse released events in the log.

Managing the Log

The Gesture Events example shows a log of the events that were handled by the shapes on the screen. An `ObservableList` object is used to record the events for each shape, and a `ListView` object is used to display the list of events. The log is limited to 50 entries. The newest entry is added to the top of the list and the oldest entry is removed from the bottom. See GestureEvents.java for the code that manages the log.

Work with the shapes in the application and notice what events are generated for each gesture that you perform.

Additional Resources

See the JavaFX API documentation for more information on gesture events, touch events, and touch points.

6

Working with Touch Events

This topic describes the touch events that enable users to interact with your JavaFX application using a touch screen. Touch points identify each point of contact for a touch. This topic shows you how to identify the touch points and handle touch events to provide sophisticated responses to touch actions.

A touch action consists of one or more points of contact on a touch screen. The action can be a simple press and release, or a more complicated series of holds and moves between the press and release. A series of events is generated for each point of contact for the duration of the action. In addition to the touch events, mouse events and gesture events are generated. If your JavaFX application does not require a complex response to a touch action, you might prefer to handle the mouse or gesture event instead of the touch event. For more information about handling gesture events, see Working with Events from Touch-Enabled Devices.

Touch events are introduced in JavaFX 2.2 and require a touch screen and the Windows 7 operating system.

Overview of Touch Actions

The term touch action refers to the entire scope of a user's touch from the time that contact is made with the touch screen to the time that the touch screen is released by all points of contact. The types of touch events that are generated during a touch action are `TOUCH_PRESSED`, `TOUCH_MOVED`, `TOUCH_STATIONARY`, and `TOUCH_RELEASED`.

Each point of contact with the screen is considered a touch point. For each touch point, a touch event is generated. When a touch action contains multiple points of contact, a set of events, which is one event for each touch point, is generated for each state in the touch action.

See the sections Touch Points, Touch Events, and Event Sets for more information about these elements. See Touch Events Example for an example of how touch events can be used in a JavaFX application.

Touch Points

When a user touches a touch screen, a touch point is created for each individual point of contact. A touch point is represented by an instance of the `TouchPoint` class, and contains information about the location, the state, and the target of the point of contact. The states of a touch point are pressed, moved, stationary, and released.

> **Tip:** The number of touch points that are generated might be limited by the touch screen. For example, if the touch screen supports only two points of contact and the user touches the screen with three fingers, only two touch points are generated. For the purposes of this article, it is assumed that the touch screen recognizes all points of contact.

Each touch point has an ID, which is assigned sequentially as touch points are added to the touch action. The ID of a touch point remains the same from the time that contact is made with the touch screen to the time that contact is released. When a point of contact is released, the associated touch point is no longer part of the touch action. For example, if the touch screen is touched with two fingers, the ID assigned to the first touch point is 1 and the ID assigned to the second touch point is 2. If the second finger is removed from the touch screen, only touch point 1 remains as part of the touch action. If another finger is then added to the touch action, the ID assigned to the new touch point is 3, and the touch action has touch points 1 and 3.

Touch Events

Touch events are generated to track the actions of touch points. A touch event is represented by an instance of the `TouchEvent` class. Touch events are generated only from touches on a touch screen. Touch events are not generated from a trackpad.

Touch events are similar to other events, which have a source, target, and event types that further define the action that occurs. The types of touch events are `TOUCH_PRESSED`, `TOUCH_MOVED`, `TOUCH_STATIONARY`, and `TOUCH_RELEASED`. Multiple `TOUCH_MOVED` and `TOUCH_STATIONARY` events can be generated for a touch point, depending on the distance moved and the time that a touch point is held in place. See Processing Events for basic information about events and how events are processed.

Touch events also have the following items:

- Touch point

  Main touch point that is associated with this event

- Touch count

  The number of touch points currently associated with the touch action

- List of touch points

  The set of the touch points currently associated with the touch action

- Event set ID

  ID of the event set that contains this event

Event Sets

When a touch action has a single point of contact, a single touch event is generated for each state of the action. When a touch action has multiple points of contact, a set of touch events is generated for each state of the action. Each touch event in the set is associated with a different one of the touch points.

Each set of events has an event set ID. The event set ID increments by one for each set that is generated in response to the touch action. The events in the set can have different event types, depending on the state of the touch point with which it is associated. As points of contact are added or removed during the touch action, the number of events in the event set changes. For example, Table 6–1 describes the event

sets that are generated when a user touches the touch screen with two fingers, moves both fingers, touches the touch screen with a third finger, moves all fingers, and then removes all fingers from the screen.

Table 6–1 Event Sets for a Single Touch Action

Event Set ID	Number of Touch Events	Event Type for Each Event
1	1	`TOUCH_PRESSED`
2	2	`TOUCH_STATIONARY`, `TOUCH_PRESSED`
3	2	`TOUCH_MOVED`, `TOUCH_MOVED`
4	3	`TOUCH_STATIONARY`, `TOUCH_STATIONARY`, `TOUCH_PRESSED`
5	3	`TOUCH_MOVED`, `TOUCH_MOVED`, `TOUCH_MOVED`
6	3	`TOUCH_MOVED`, `TOUCH_MOVED`, `TOUCH_MOVED`
7	3	`TOUCH_MOVED`, `TOUCH_MOVED`, `TOUCH_MOVED`
8	3	`TOUCH_RELEASED`, `TOUCH_STATIONARY`, `TOUCH_STATIONARY`
9	2	`TOUCH_RELEASED`, `TOUCH_STATIONARY`
10	1	`TOUCH_RELEASED`

Touch Point Targets and Touch Event Targets

The target of a touch event is the target of the touch point that is associated with the event. The initial target of the touch point is the topmost node at the initial point of contact with the touch screen. If a touch action has multiple points of contact, it is possible for each touch point, and therefore each touch event, to have a different target. This feature enables you to handle each touch point independently of the other touch points. See Handling Concurrent Touch Points Independently for an example.

Typically, all of the events for one touch point are delivered to the same target. However, you can alter the target of subsequent events using the `grab()` and `ungrab()` methods for the touch point.

The `grab()` method enables the node that is currently processing the event to make itself the target of the touch point. The `grab(`*target*`)` method enables another node to be made the target of the touch point. Because events in the event set have access to all of the touch points for the set, it is possible to use the `grab()` method to direct all subsequent events for the touch action to the same node. The `grab()` method can also be used to reset the target of a touch point, as shown in Changing the Target of a Touch Point.

The `ungrab()` method is used to release the touch point from the current target. Subsequent events for the touch action are then sent to the topmost node at the current location of the touch point.

Additional Events Generated from Touches

When a user touches a touch screen, other types of events are generated in addition to touch events:

- Mouse events

  Simulated mouse events enable an application to run on a device with a touch screen even if touch events are not handled by the application. Use the

`isSynthesized()` method to determine if the mouse event is from a touch action. See Handling Mouse Events for an example.

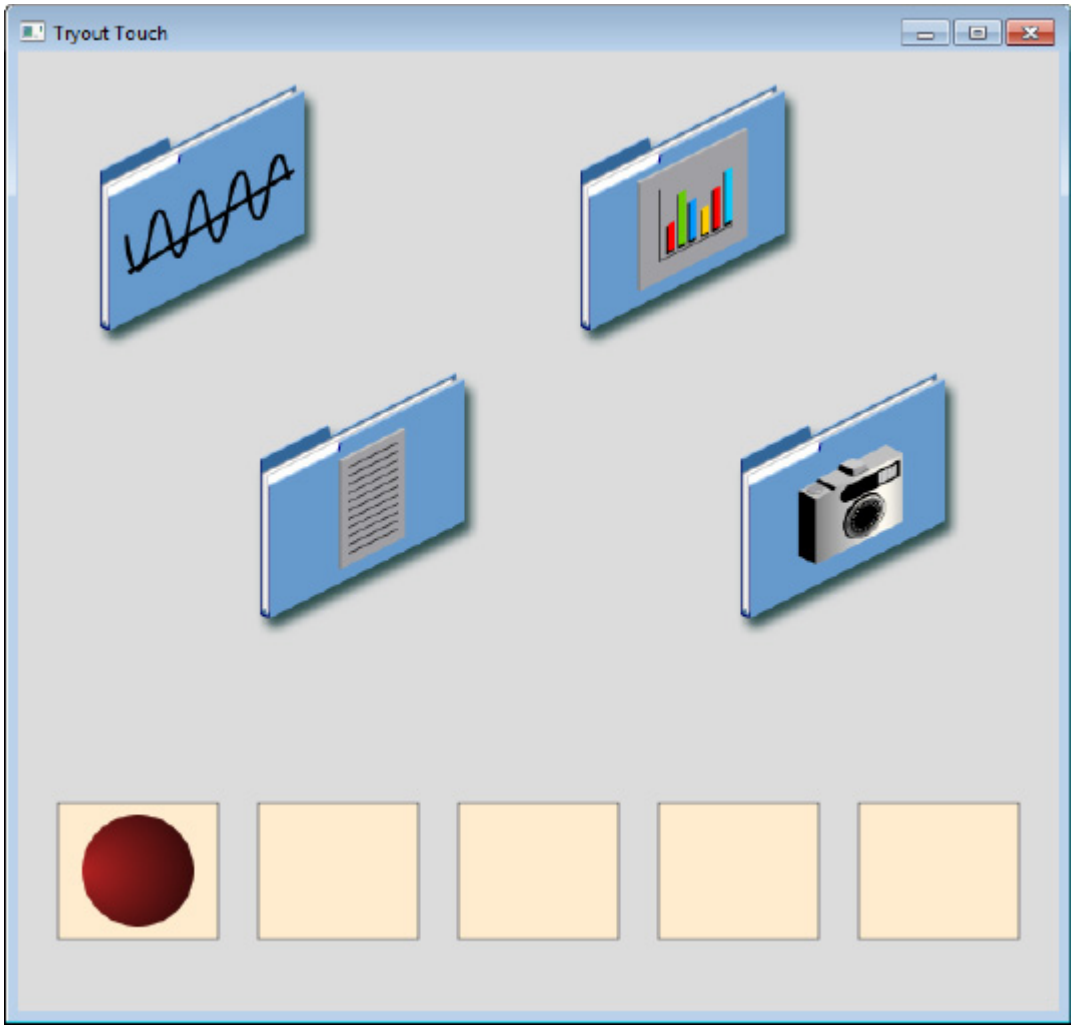
- Gesture events

  Gesture events are generated for the commonly recognized touch actions of scrolling, swiping, rotating, and zooming. If these are the only types of touch actions that your application must handle, you can handle these gesture events instead of the touch events. See Working with Events from Touch-Enabled Devices for information on gesture events.

Touch Events Example

The Touch Events example uses four folders to demonstrate the ability to independently handle each touch point in a set. The example also shows how the `grab()` method can be used to repeatedly jump the circle from one rectangle to another. Figure 6–1 shows the user interface for the example.

Figure 6–1 Touch Events Example



The Touch Events example is available in the TouchEventsExample.zip file. Extract the NetBeans project and open it in the NetBeans IDE. To generate touch events, you must run the example on a device with a touch screen.

Handling Concurrent Touch Points Independently

In a typical gesture, the target is the node at the center of all of the points of contact, and only one node is affected by the response to the gesture. By handling each touch point separately, you can affect all of the nodes that are touched.

In the Touch Events example, you can move each folder by touching the folder and moving your finger. You can move multiple folders at once by touching each folder with a separate finger and moving all fingers.

Each folder is an instance of the `TouchImage` class. The `TouchImage` class creates an image view and adds event handlers for `TOUCH_PRESSED`, `TOUCH_RELEASED`, and `TOUCH_MOVED` events. Example 6–1 shows the definition of this class.

Example 6–1 TouchImage Class Definition

```
public static class TouchImage extends ImageView {
    private long touchId = -1;
    double touchx, touchy;

    public TouchImage(int x, int y, Image img) {
        super(img);
        setTranslateX(x);
        setTranslateY(y);
        setEffect(new DropShadow(8.0, 4.5, 6.5, Color.DARKSLATEGRAY));

        setOnTouchPressed(new EventHandler<TouchEvent>() {
            @Override public void handle(TouchEvent event) {
                if (touchId == -1) {
                    touchId = event.getTouchPoint().getId();
                    touchx = event.getTouchPoint().getSceneX() - getTranslateX();
                    touchy = event.getTouchPoint().getSceneY() - getTranslateY();
                }
                event.consume();
            }
        });

        setOnTouchReleased(new EventHandler<TouchEvent>() {
            @Override public void handle(TouchEvent event) {
                if (event.getTouchPoint().getId() == touchId) {
                    touchId = -1;
                }
                event.consume();
            }
        });

        setOnTouchMoved(new EventHandler<TouchEvent>() {
            @Override public void handle(TouchEvent event) {
                if (event.getTouchPoint().getId() == touchId) {
                    setTranslateX(event.getTouchPoint().getSceneX() - touchx);
                    setTranslateY(event.getTouchPoint().getSceneY() - touchy);
                }
                event.consume();
            }
        });
    }
}
```

When a folder is touched, a touch point is created for each point of contact and touch events are sent to the folder. The touch ID is used to ensure that a folder responds only once when multiple points of contact are on the folder.

When a TOUCH_PRESSED event is received, the touch ID is checked to determine if it is a new touch for this folder. If so, the touch ID is set to the ID of the touch point and the location of the touch point is saved.

When a TOUCH_RELEASED event is received, the touch ID is checked to ensure that it matches the touch point that is being processed. If so, the touch ID is reset to indicate that processing is complete.

When a TOUCH_MOVED event is received, the touch ID is checked to ensure that it matches the touch point that is being processed. If so, the folder is moved to the new location for the touch point. If the touch ID does not match the touch point, then more than one point of contact is likely on the folder. To avoid responding to multiple movements of the same folder, the event is ignored.

Changing the Target of a Touch Point

The target of a touch point is typically the same node for the duration of the touch action. However, in some situations, you might want to change the target of a touch point during the touch action.

In the Touch Events example, the circle moves from one rectangle to another by touching the circle with one finger and a rectangle with a second finger. While the second finger remains on the circle after the jump, lift the first finger and touch a different rectangle to cause the circle to jump again. This action is possible only if you change the target of the second touch point.

The circle is an instance of the Ball class. The Ball class creates a circle and adds event handlers for the TOUCH_PRESSED, TOUCH_RELEASED, TOUCH_MOVED, and TOUCH_STATIONARY events. The same handler is used for the TOUCH_MOVED and TOUCH_STATIONARY events. Example 6–2 shows the definition of this class.

Example 6–2 Ball Class Definition

```
private static class Ball extends Circle {
    double touchx, touchy;

    public Ball(int x, int y) {
        super(35);

        RadialGradient gradient = new RadialGradient(0.8, -0.5, 0.5, 0.5, 1,
            true, CycleMethod.NO_CYCLE, new Stop [] {
                new Stop(0, Color.FIREBRICK),
                new Stop(1, Color.BLACK)
            });

        setFill(gradient);
        setTranslateX(x);
        setTranslateY(y);

        setOnTouchPressed(new EventHandler<TouchEvent>() {
            @Override public void handle(TouchEvent event) {
                if (event.getTouchCount() == 1) {
                    touchx = event.getTouchPoint().getSceneX() - getTranslateX();
                    touchy = event.getTouchPoint().getSceneY() - getTranslateY();
                    setEffect(new Lighting());
                }
                event.consume();
            }
        });
```

```
        setOnTouchReleased(new EventHandler<TouchEvent>() {
            @Override public void handle(TouchEvent event) {
                setEffect(null);
                event.consume();
            }
        });

        // Jump if the first finger touched the ball and is either
        // moving or still, and the second finger touches a rectangle
        EventHandler<TouchEvent> jumpHandler = new EventHandler<TouchEvent>() {
            @Override public void handle(TouchEvent event) {

                if (event.getTouchCount() != 2) {
                    // Ignore if this is not a two-finger touch
                    return;
                }

                TouchPoint main = event.getTouchPoint();
                TouchPoint other = event.getTouchPoints().get(1);

                if (other.getId() == main.getId()) {
                    // Ignore if the second finger is in the ball and
                    // the first finger is anywhere else
                    return;
                }

                if (other.getState() != TouchPoint.State.PRESSED ||
                    other.belongsTo(Ball.this) ||
                    !(other.getTarget() instanceof Rectangle) ){
                    // Jump only if the second finger was just
                    // pressed in a rectangle
                    return;
                }

                // Jump now
                setTranslateX(other.getSceneX() - touchx);
                setTranslateY(other.getSceneY() - touchy);

                // Grab the destination touch point, which is now inside
                // the ball, so that jumping can continue without
                // releasing the finger
                other.grab();

                // The original touch point is no longer of interest so
                // call ungrab() to release the target
                main.ungrab();

                event.consume();
            }
        };

        setOnTouchStationary(jumpHandler);
        setOnTouchMoved(jumpHandler);
    }
}
```

When a `TOUCH_PRESSED` event is received, the number of touch points is checked to ensure that only the instance of the `Ball` class is being touched. If so, the location of the touch point is saved, and a lighting effect is added to show that the circle is selected.

When a TOUCH_RELEASED event is received, the lighting effect is removed to show that the circle is no longer selected.

When a TOUCH_MOVED or TOUCH_STATIONARY event is received, the following conditions that are required for a jump are checked:

- The touch count must be two.

  The touch point that is associated with this event is considered the start point of the jump. The event has access to all of the touch points for the touch action. The second touch point in the set of touch points is considered the end point of the jump.

- The state of the second touch point is PRESSED.

  The circle is moved only when the second point of contact is made. Any other state for the second touch point is ignored.

- The target of the second touch point is a rectangle.

  The circle can jump only from rectangle to rectangle, or within a rectangle. If the target of the second touch point is anything else, the circle is not moved.

If the conditions for a jump are met, the circle is jumped to the location of the second touch point. To jump again, the first point of contact is released and a third location is touched, with the expectation that the circle will jump to the third location. However, when the first point of contact is released, the touch point whose target was the circle goes away and now the circle no longer gets touch events. A second jump is not possible without lifting both fingers and starting a new jump.

To make a second jump possible while keeping the second finger on the circle and touching a new location, the grab() method is used to make the circle the target of the second touch point. After the grab, events for the second touch point are sent to the circle instead of the rectangle that was the original target. The circle can then watch for a new touch point and jump again.

Additional Resources

See the JavaFX API documentation for more information on touch events and touch points.