Chapter 18

Targeting Touch Devices

In This Chapter

- Discovering gestures on touch devices
- Adding listeners that respond to gesture events
- Creating a program that lets the user manipulate a shape with gestures

Touch devices are everywhere nowadays. Tablets and smartphones are the most common types of touch devices, but even some desktop users have installed touch-capable monitors. At one time, touch devices ran primarily non-Microsoft operating systems, such as Apple's iOS or Google's Android. But now, with Windows 8, many touch devices even run Windows.

Fortunately, JavaFX can run on all those platforms. In this chapter, you discover how to develop JavaFX applications that take advantage of the unique user interactions that are possible with touch devices, including basic touch and swiping and scrolling as well as multi-touch gestures such as zooming and rotating.

Introducing Gestures and Touch Events

Before I get into the details of working with gestures and touch events, I want to point out that the support for basic touch devices for JavaFX controls is already built-in and requires no programming to enable. For example, `Button` controls respond to taps on a touch device just as they respond to mouse clicks. So no special programming is needed to enable the basic operation of JavaFX controls on a touch device.

The good news is that the programming required to handle more advanced touch gestures, such as zooming or rotating, is pretty straightforward. JavaFX handles the difficult tasks of figuring out what gestures the user is making when he touches the screen. For example, when JavaFX sees that the user has touched the screen with two fingers and then rotated the fingers in

opposite directions, the user is attempting to rotate an object on the screen. JavaFX figures out which object is the target of the rotation, figures out how much the user has rotated the object, and then sends a `ROTATE` event to the target. All you have to do is provide an event listener for the `ROTATE` event.

Actually, when the user does a rotate gesture, JavaFX will likely send dozens of separate `ROTATE` events to the target of the rotation. Each of those events represents an incremental step along the way to the complete rotation. Your program will respond to those events by rotating the target object by the amount indicated by the `ROTATE` event; the result will be that the object smoothly follows the user's fingers as the user continues the rotation gesture.

JavaFX provides event handling for four distinct types of touch gestures:

- **Rotate:** A *rotate* gesture is recognized when the user places two fingers on the screen and rotates them in opposite directions. Three distinct events are generated when a rotate gesture is recognized:
  - `ROTATE_STARTED`: This event occurs once, as soon as the rotation gesture is recognized.
  - `ROTATE`: This event can occur multiple times throughout the rotation gesture. It is usually the event you'll want to handle.
  - `ROTATE_FINISHED`: This event occurs once, after the end of the rotation gesture is recognized.

  All rotation events are represented by a `RotateEvent` object, which defines two key methods:
  - `getAngle`: Returns the angle of rotation for this particular event.
  - `getTotalAngle`: Returns the cumulative angle for the entire rotation gesture.
- **Zoom:** A *zoom* gesture is recognized when the user places two fingers on the screen and then spreads them apart or brings them together. As with rotate gestures, zoom gestures create three types of events:
  - `ZOOM_STARTED`: This event occurs when the zoom gesture begins.
  - `ZOOM`: This event occurs multiple times throughout the zoom gesture.
  - `ZOOM_FINISHED`: This event occurs when the zoom gesture ends.

  Zoom events are defined by the `ZoomEvent` class, which has two important methods:
  - `getZoomFactor`: Returns the amount of the zoom for this particular event.
  - `getTotalZoomFactor`: Returns the accumulated zoom factor for the entire zoom gesture.

The zoom factor is a multiplier you can use to determine the new size of the zoomed object. For example, if the user doubles the size of the target object, the zoom factor will be 2.0.

- **Scroll:** A *scroll* gesture is recognized when the user places one finger on the screen and drags it to another location. Once again, three types of events are generated for scroll gestures:
  - `SCROLL_STARTED`: Occurs at the start of the scroll gesture.
  - `SCROLL`: Occurs multiple times throughout the scroll gesture.
  - `SCROLL_FINISHED`: Occurs when the scroll gesture ends.

  Scroll events are represented by the `ScrollEvent` class, which provides several methods to retrieve the scroll amount:
  - `getDeltaX`: Returns the horizontal change for this scroll event.
  - `getDeltaY`: Returns the vertical change for this scroll event.
  - `getTotalDeltaX`: Returns the total horizontal change for this scroll gesture.
  - `getTotalDeltaY`: Returns the total vertical change for this scroll gesture.
- **Swipe:** A *swipe* gesture is recognized when the user places one finger on the screen and quickly swipes it either horizontally or vertically. Unlike the other gestures, swipe gestures do not generate events to indicate the start and end of the gesture. Instead, one of the following events is generated to indicate the direction of the swipe:
  - `SWIPE_LEFT`
  - `SWIPE_RIGHT`
  - `SWIPE_UP`
  - `SWIPE_DOWN`

  Swipe events are represented by the `SwipeEvent` class.

Swipe events and scroll events are difficult to distinguish because a swipe is simply a fast scroll in either a horizontal or vertical direction. Thus, when the user performs a swipe gesture, scroll events are generated in addition to the swipe event.

Listening for Gestures

You can install a listener for any gesture on any node object by calling one of the methods listed in Table 18-1. As with all JavaFX events, these events use functional interfaces, so you can easily implement the event listeners with Lambda expressions.

Table 18-1 Node Methods for Installing Gesture Event Listeners

Method	*Explanation*
`void setOnRotate(RotateEvent listener)`	Creates a listener for the `ROTATE` event.
`void setOnRotateStarted(RotateEvent listener)`	Creates a listener for the `ROTATE_STARTED` event.
`void setOnRotateFinished(RotateEvent listener)`	Creates a listener for the `ROTATE_FINISHED` event.
`void setOnZoom(ZoomEvent listener)`	Creates a listener for the `ZOOM` event.
`void setOnZoomStarted(ZoomEvent listener)`	Creates a listener for the `ZOOM_STARTED` event.
`void setOnZoomFinished(ZoomEvent listener)`	Creates a listener for the `ZOOM_FINISHED` event.
`void setOnScroll(ScrollEvent listener)`	Creates a listener for the `SCROLL` event.
`void setOnScrollStarted(ScrollEvent listener)`	Creates a listener for the `SCROLL_STARTED` event.
`void setOnScrollFinished(ScrollEvent listener)`	Creates a listener for the `SCROLL_FINISHED` event.
`void setOnSwipeLeft(SwipeEvent listener)`	Creates a listener for the `SWIPE_LEFT` event.
`void setOnSwipeRight(SwipeEvent listener)`	Creates a listener for the `SWIPE_RIGHT` event.
`void setOnSwipeUp(SwipeEvent listener)`	Creates a listener for the `SWIPE_UP` event.
`void setOnSwipeDown(SwipeEvent listener)`	Creates a listener for the `SWIPE_DOWN` event.

Here's an example that creates a rectangle, and then installs a listener for the rotate event and adjusts the rectangles rotation when the rotate event is fired:

```
Rectangle r = new Rectangle(150, 150, 200, 200);
r.setFill(Color.DARKGRAY);
r.setOnRotate(e ->
    {
        r.setRotate(r.getRotate() + e.getAngle());
        e.consume();
    });
```

Here, the rotation of the rectangle is modified by first retrieving the current rotation and then adding the angle retrieved from the rotate event.

This event handler calls the `consume` method to discard the event. This is important when dealing with gesture events. If you don't consume the event, the event will be passed up the chain and may be processed again by other objects.

One other detail you should know about is that gesture events have a characteristic know as *inertia,* which causes events to continue to be raised after the user completes the gesture. For example, when the user completes a scroll gesture, the `SCROLL_FINISHED` event occurs, but several `SCROLL` events are generated after the `SCROLL_FINISHED` event. This creates a more realistic experience for the user, but can complicate your programming.

You can call the gesture event's `isInertia` method to determine whether an event was created as a result of inertia. Then, if you want to suppress the effect of inertia, you can ignore the event if `isInertia` returns `true`. For example:

```
Rectangle r = new Rectangle(150, 150, 200, 200);
r.setFill(Color.DARKGRAY);
r.setOnRotate(e ->
    {
        if (!isInertia())
        {
            r.setRotate(r.getRotate() + e.getAngle());
            e.consume();
        }
    });
```

Here, the rectangle is rotated only if the event is not generated as a result of inertia.

Looking at an Example Program

In this section, you look at a sample program — the Gesturator. The Gesturator program displays a simple rectangle on the screen and allows the user to manipulate the rectangle using gestures. Figure 18-1 shows the scene displayed by the Gesturator program.

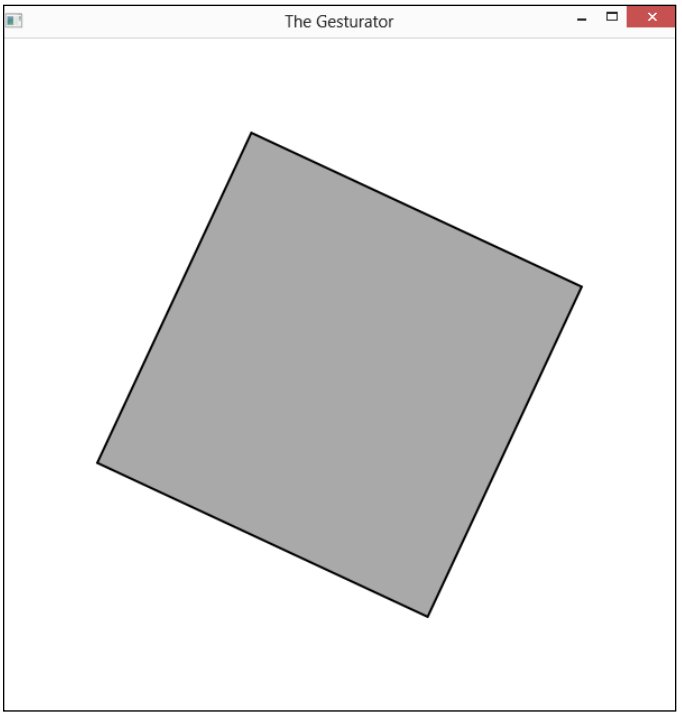

Figure 18-1: The Gesturator program in action.

Here are the key features of the Gesturator program:

- Initially, the rectangle is centered in the screen.
- The user can rotate the rectangle using a rotate gesture.
- The user can change the size of the rectangle by using a zoom gesture.
- The user can move the rectangle by dragging it with a single finger. The movement is constrained so that the entire rectangle stays within the bounds of the scene.

- The user can shove the rectangle to one edge of the screen by swiping the rectangle in the correct direction. The program responds to the swipe gesture by moving the rectangle all the way to edge of the screen.

Listing 18-1 provides the complete source code for the Gesturator program.

Listing 18-1: The Gesturator Program

```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.control.*;
import javafx.scene.paint.*;
import javafx.scene.shape.*;
import javafx.scene.input.*;
import javafx.event.*;

public class Gesturator extends Application
{

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

    private final double RECT_X = 200;                                    →18
    private final double RECT_Y = 200;
    private final double SCENE_X = 600;
    private final double SCENE_Y = 600;

    @Override public void start(Stage primaryStage)
    {
        Group root = new Group();

        Rectangle r =                                                     →27
            new Rectangle((SCENE_X - RECT_X)/2,
                          (SCENE_Y - RECT_Y)/2,
                          RECT_X,
                          RECT_Y);
        r.setFill(Color.DARKGRAY);
        r.setStroke(Color.BLACK);
        r.setStrokeWidth(2);

        r.setOnZoom(e ->                                                  →36
            {
                r.setScaleX(r.getScaleX() * e.getZoomFactor());
                r.setScaleY(r.getScaleY() * e.getZoomFactor());
                e.consume();
            });
```

(continued)

Listing 18-1 ***(continued)***

```
        r.setOnRotate(e ->                                              →43
            {
                r.setRotate(r.getRotate() + e.getAngle());
                e.consume();
            });

        r.setOnScroll(e ->                                              →49
            {
                if (!e.isInertia())                                     →51
                {
                    double newX = r.getX() + e.getDeltaX();             →53
                    if ( (newX >= 0) &&
                         (newX <= SCENE_X - r.getWidth()) )
                    {
                        r.setX(newX);
                    }
                    double newY = r.getY() + e.getDeltaY();             →59
                    if ( (newY >= 0) &&
                         (newY <= SCENE_Y - r.getHeight()) )
                    {
                        r.setY(newY);
                    }
                }
                e.consume();
            });

        r.setOnSwipeLeft(e ->                                           →69
            {
                r.setX(0);
                e.consume();
            });

        r.setOnSwipeRight(e ->                                          →75
            {
                r.setX(SCENE_X - r.getWidth());
                e.consume();
            });

        r.setOnSwipeUp(e ->                                             →81
            {
                r.setY(0);
                e.consume();
            });

        r.setOnSwipeDown(e ->                                           →87
            {
                r.setY(SCENE_Y - r.getHeight());
                e.consume();
            });
```

```
        root.getChildren().add(r);                                          →93
        Scene scene = new Scene(root, SCENE_X, SCENE_Y);
        primaryStage.setTitle("The Gesturator");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

The following paragraphs explain the key points of the Gesturator program:

→ **18:** Private fields are used to set the size of the rectangle and the size of the scene. These constants are referred to several times throughout the program.

→ **27:** The rectangle is created using the size specified by `RECT_X` and `RECT_Y`. The initial center position is calculated using the size of the scene and the size of the rectangle.

→ **36:** The zoom event listener adjusts the size of the rectangle by multiplying its current size by the zoom factors provided by the `ZoomEvent` object.

→ **43:** The rotate event listener rotates the rectangle by adding the rotation angle provided by the `RotateEvent` object to the current rotation value of the rectangle.

→ **49:** The scroll event handler is a little more complicated than the other event handlers because it imposes several constraints on the position of the rectangle.

→ **51:** The scroll event is ignored if it is the result of inertia. That way, the rectangle stops its movement immediately when the user stops the scroll gesture.

→ **53:** To determine the new `x` position, the scroll event listener first calculates the proposed new `x` position by adding the amount of horizontal movement (`getDeltaX`) to the current `x` position. Then, it sets the rectangles `x` position to the proposed position only if the proposed position is greater than zero and less than the width of the screen minus the current width of the rectangle. The `if` statement prevents the user from moving the rectangle past the left or right edge of the scene.

→ **59:** Similar logic is used to change the `y` position. First, the proposed `y` position is calculated; then, the proposed `y` position is applied only if it does not move the rectangle past the top or bottom edges of the scene.

→ `69:` The swipe left listener moves the rectangle to the left edge of the screen.

→ `75:` The swipe right listener moves the rectangle to the right edge of the screen.

→ `81:` The swipe up listener moves the rectangle to the top edge of the screen.

→ `87:` The swipe down listener moves the rectangle to the bottom edge of the screen.

→ `93:` The rectangle is added to the root node. Then, the scene is created and displayed on the stage.