# Advanced JavaFX 16

**Outcomes**:

*By the end of this chapter you should be able to*:

- *program components to respond to* ***mouse events*** *and* ***key events***;
- *explain the term* ***property*** *as applied to JavaFX components*;
- ***bind*** *properties of two or more components*;
- *use a* ***slider*** *to change the value of a variable*;
- *embed images, videos and webpages in applications*;
- *format applications by using* ***cascading style sheets***.

## 16.1 Introduction

In this chapter we will explore the more advanced aspects of programming with JavaFX. We will begin by looking at the ways in which applications can be programmed to respond to input events such as keystrokes and mouse movements, rather than just simple mouse clicks. We will then look at the unique nature of the attributes of JavaFX classes and explain how we can add event handlers to the individual attributes of a component, rather than to the component itself.

We will then go on to explore the way in which we can turn our programs into multimedia applications, and finally we will introduce you to cascading style sheets so that you can keep your formatting information separate from the rest of the program, and apply different "skins" to an application.

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_16

## 16.2 Input Events

In Chap. 10 we introduced you to the idea of events and event handling. The only event that we came across was an `ActionEvent`, the event that occurs when the mouse is clicked on a component. There are other events that are defined as subtypes of the `Event` class, the most important of which is the `InputEvent`. Two subtypes of `InputEvent` are discussed here—`MouseEvent` and `KeyEvent`. These occur, respectively, when a mouse button is pressed or the mouse is moved or dragged, and when a key is pressed on the keyboard.

### 16.2.1 Mouse Events

We saw in Chaps. 10 and 13 that a JavaFX `ActionEvent` has only one `EventType` defined (`ACTION`). In the case of `MouseEvent`, however, there are a number of `EventTypes`, the most common of which we have summarised in Table 16.1.

As you will see in a moment, there are convenience methods for all of these events—and in order to demonstrate some of these, we have developed a little game with which you can amuse your friends. Figure 16.1 shows how it looks when it runs. We have called it—rather unimaginatively—the `RedCircle` game; a red circle always moves away from the cursor so you can never click on it, despite being told to do so! And if in desperation you start to click the mouse, the words "Keep Trying" flash onto the screen!

**Table 16.1** Types of mouse event

| | |
|---|---|
| MOUSE_CLICKED | This event occurs when the mouse has been clicked (pressed and released) |
| MOUSE_PRESSED | This event occurs when the mouse button is pressed |
| MOUSE_RELEASED | This event occurs when the mouse button is released |
| MOUSE_MOVED | This event occurs when the mouse moves within a node and no buttons are pressed |
| MOUSE_DRAGGED | This event occurs when the mouse moves within a node with a pressed button |
| MOUSE_ENTERED | This event occurs when the mouse enters a node |
| MOUSE_EXITED | This event occurs when the mouse exits a node |

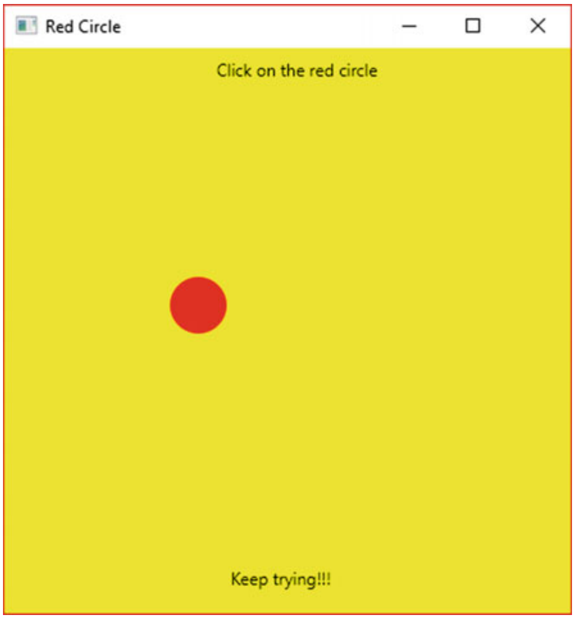


**Fig. 16.1** The *RedCircle* application

Here is the code for the class:

***RedCircle***

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.stage.Stage;
import javafx.scene.text.Text;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;

public class RedCircle extends Application
{
    @Override
    public void start(Stage stage)
    {
        final double WIDTH = 400;
        final double HEIGHT = 400;

        // circle starts in centre of screen
        Circle circle = new Circle(WIDTH/2, HEIGHT/2, 20, Color.RED);

        Text heading = new Text(WIDTH/2 - 50, 20, "Click on the red circle");
        Text message = new Text(WIDTH/2 - 40, HEIGHT - 20 , "");

        Group root = new Group(heading, circle, message);
        Scene scene = new Scene(root, WIDTH, HEIGHT, Color.YELLOW);

        /* when the mouse is move or dragged the centre of the circle is
           repositioned so that it is always 50 pixels to the left and
           50 pixels above the cursor */

        scene.setOnMouseMoved(e -> {
                                        circle.setCenterX(e.getX()-50);
                                        circle.setCenterY(e.getY()-50);
                                    }
                              );

        scene.setOnMouseDragged(e -> {
                                        circle.setCenterX(e.getX()-50);
                                        circle.setCenterY(e.getY()-50);
                                      }
                                );
```

```
            // a message is displayed when the mouse button is depressed
            scene.setOnMousePressed(e -> message.setText("Keep trying!!!"));

            // the message is blanked when the mouse button is released
            scene.setOnMouseReleased(e -> message.setText(""));

            stage.setScene(scene);
            stage.setTitle("Red Circle");
            stage.show();
        }

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

We have begun by declaring two constants `WIDTH` and `HEIGHT` to represent the dimensions of the scene, and then gone on to define a circle of radius 20 pixels that will start off at the centre of the graphic:

```
Circle circle = new Circle(WIDTH/2, HEIGHT/2, 20, Color.RED);
```

Next we create two instances of the `Text` class. The first is the heading, the second will hold the message that appears when the mouse button is depressed—this one starts off empty.

```
Text heading = new Text(WIDTH/2 - 50, 20, "Click on the red circle");
Text message = new Text(WIDTH/2 - 40, HEIGHT - 20 , "");
```

Next we group these three items together and add them to the scene.

```
Group root = new Group(heading, circle, message);
Scene scene = new Scene(root, WIDTH, HEIGHT, Color.YELLOW);
```

Now we come to the code that allows the graphic to respond to the mouse being moved.

```
scene.setOnMouseMoved(e -> {
                                 circle.setCenterX(e.getX()-50);
                                 circle.setCenterY(e.getY()-50);
                            }
                     );
```

Notice that we attach this to the scene itself. We use the convenience method `setOnMouseMoved` for this purpose. The idea is that whenever the mouse moves the circle also moves so that it is always 50 pixels above and 50 pixels to the left of the cursor.

So how do we find out where the cursor is? For this, we have made use of the parameter that is received by the `handle` method of `EventHandler`, to which the lambda expression refers. This parameter, `e`, to the left of the arrow is of type `MouseEvent` and has methods `getX` and `getY` that return the current position of the cursor. We use these to set the centre of the circle to the desired position.

We want exactly the same thing to happen when the mouse is dragged (that is, it is moved with the button depressed), so the `setOnMouseDragged` method is coded in the same way. However, when the button is pressed, we also want the message at the bottom of the screen to change from blank to “Keep trying!!!”, and for this purpose we use the `setOnMousePressed` method of `Scene`.

```
scene.setOnMousePressed(e -> message.setText("Keep trying!!!"));
```

When the button is released, the message returns to a blank message:

```
scene.setOnMouseReleased(e -> message.setText(""));
```

The final lines of code add the scene to the stage, set the title and make the stage visible, as you have seen in other examples.

### 16.2.2 Key Events

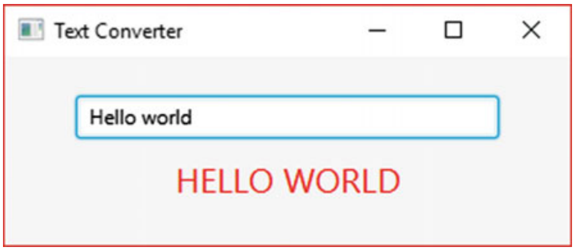
The other common input event is a key event. A key event occurs whenever a key is pressed or released. One of the common applications of this event is to check if the key pressed was the <Enter> key, which indicates that the entry is completed.

There are three common types of key event which are summarised in Table 16.2.

We have developed a couple of little applications to demonstrate these. The first one, which we have called `TextConverter`, allows the user to type something into a text field, and each time a character is entered, the content of the text field is displayed below in upper case. This is shown in Fig. 16.2—you can see we have used a similar interface to the `PushMe` class of Chap. 10, but this time there is no button, as the application responds each time a character is entered (as you will see, it actually responds when the key is released).

**Table 16.2** Types of key event

| | |
|---|---|
| KEY_TYPED | This event occurs when a key has been typed (pressed and released) |
| KEY_PRESSED | This event occurs when a key has been pressed |
| KEY_RELEASED | This event occurs when a key has been released |



Hello world

HELLO WORLD

**Fig. 16.2** The *TextConverter* application

Here is the code for the class:

***TextConverter***

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class TextConverter extends Application
{
    @Override
    public void start(Stage stage)
    {
        // create and configure a text field for user entry
        TextField textField = new TextField();
        textField.setMaxWidth(250);

        // create and configure a label to display the output
        Label label= new Label();
        label.setTextFill(Color.RED);
        label.setFont(Font.font("Ariel", 20));

        // display the contents of textField in upper case
        textField.setOnKeyReleased(e -> label.setText(textField.getText().toUpperCase()));

        // create and configure a VBox to hold the components
        VBox root = new VBox();
        root.setSpacing(10);
        root.setAlignment(Pos.CENTER);

        //add the components to the VBox
        root.getChildren().addAll(textField, label);


        // create a new scene
        Scene scene = new Scene(root);

        //add the scene to the stage, then configure the stage and make it visible
        stage.setScene(scene);
        stage.setTitle("Text Converter");
        stage.setHeight(150);
        stage.setWidth(350);
        stage.show();
    }

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

}
```

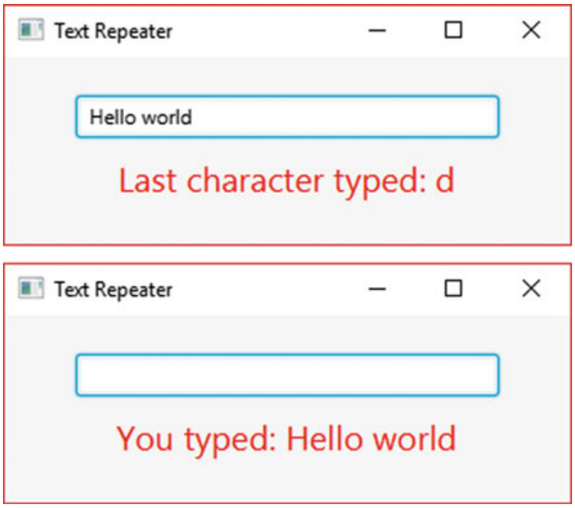


**Fig. 16.3** The *TextRepeater* application

The only thing we need to draw your attention to is this line:

```
textField.setOnKeyReleased(e -> label.setText(textField.getText().toUpperCase()));
```

We are using the convenience method `setOnKeyReleased` to program the response to a key stroke. The event is triggered when the key is released (as opposed to when it is pressed), and as you can see from the lambda expression, after each key press the entire content of the text field is copied to the label and converted to upper case.

Our next program—the `TextRepeater`—is slightly different. In this application, as soon as character is typed it is echoed on the label at the bottom—however, if it is the <Enter> key that is pressed then the text field is cleared, and the entire phrase that was typed is displayed. You can see how this works in Fig. 16.3.

The code for this application appears below:

***TextRepeater***

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class TextRepeater extends Application
{
    @Override
    public void start(Stage stage)
    {
        // create and configure a text field for user entry
        TextField textfield = new TextField();
        textfield.setMaxWidth(250);

        // create and configure a label to display the output
        Label repeatLabel= new Label();
        repeatLabel.setTextFill(Color.RED);
        repeatLabel.setFont(Font.font("Ariel", 20));


        textfield.setOnKeyTyped(e ->
                        {
                              if(e.getCharacter().equals("\r")) // check for the Enter key
                              {
                                   repeatLabel.setText("You typed: " + textfield.getText());
                                   textfield.setText("");
                              }
                              else
                              {
                                   repeatLabel.setText("Last character typed: " + e.getCharacter());
                              }
                        });

        // create and configure a VBox to hold our components
        VBox root = new VBox();
        root.setSpacing(10);
        root.setAlignment(Pos.CENTER);

        //add the components to the VBox
        root.getChildren().addAll(textfield, repeatLabel);


        // create a new scene
        Scene scene = new Scene(root);

        //add the scene to the stage, then configure the stage and make it visible
        stage.setScene(scene);
        stage.setTitle("Text Repeater");
        stage.setHeight(150);
        stage.setWidth(350);
        stage.show();
    }

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

}
```

Once again, the only thing we need to discuss is how we code the convenience method, in this case `setOnKeyTyped`:

```
textfield.setOnKeyTyped(e ->
                        {
                              if(e.getCharacter().equals("\r")) // check for the Enter key
                              {
                                   repeatLabel.setText("You typed: " + textfield.getText());
                                   textfield.setText("");
                              }
                              else
                              {
                                   repeatLabel.setText("Last character typed: " + e.getCharacter());
                              }
                        });
```

You can see that the lambda expression deals with the possibility that the <Enter> key has been pressed. In order to do this we have made use of the `getCharacter` method of `e`, the `KeyEvent` parameter that is sent into the `handle` method of `EventHandler` when the event concerned is a key event. After a `KEY_TYPED` event, the `getCharacter` method returns a string which holds the value of the character returned. We have checked to see if this is the special character "`\r`" which represents the <Enter> key (Unicode 13). If it was the <Enter> key, the entire string is copied from the text field to the label (appended to the message "You typed:"), and the text field is cleared.

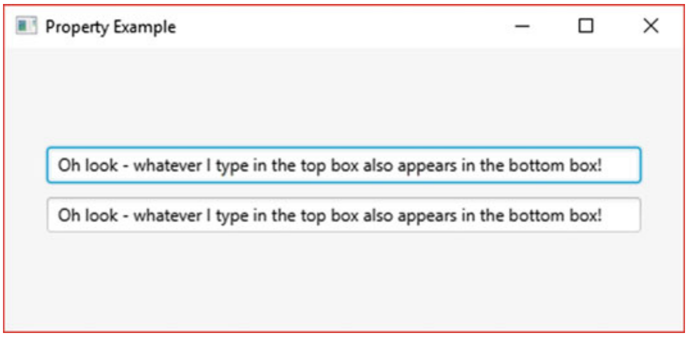
If the key pressed was any other key, the character is echoed on the label, again appended to a message.

## 16.3 Binding Properties

When we refer to the classes in the JavaFX package, we tend not to talk about the attributes of a class, but rather its **properties**.

In order to understand why we do this, let's consider the JavaFX `TextField` class for a moment. You have seen how we use the `setText` and `getText` methods of this class—these methods accept and return `Strings`, and you would therefore be forgiven for assuming that the `TextField` class has a `text` attribute which is of type `String`. But you would be wrong. The attribute in question (which is inherited from a higher level class) is actually of type `StringProperty`. This class, along with many other similar classes such as `DoubleProperty`, `IntegerProperty`, `BooleanProperty` and so on, is a wrapper class —similar to classes such as `Double` and `Integer` that you came across in the first semester.

Methods of control classes such as `TextField` hide the details of its properties because they are set up to deal with the more familiar types such as `String` and **`double`**. The interesting thing about these property classes is that they have some useful methods. Some of these methods enable you to add event handlers to a property rather than an object itself—you will see this in action in the next section. But two particularly interesting methods are `bind` and `bindBidirectional`.



Oh look - whatever I type in the top box also appears in the bottom box!

Oh look - whatever I type in the top box also appears in the bottom box!

**Fig. 16.4** Binding properties

These allow us to bind the properties of two different objects so that they act in unison.

To demonstrate this we have created a very simple application indeed, consisting essentially of two `textFields`. We have bound the text property of each of these with the result that the bottom one is frozen, while whatever is typed in the top one is applied immediately to the bottom one.

You can see this in Fig. 16.4.

The code for this class is shown below:

***PropertyExample***

```
import javafx.application.Application;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.geometry.Pos;
import javafx.scene.Scene;

public class PropertyExample extends Application
{
    @Override
    public void start(Stage stage)
    {
        TextField top = new TextField();
        TextField bottom = new TextField();
        top.setMaxWidth(420);
        bottom.setMaxWidth(420);

        // bind the text proerty of the bottom text field to that of the top
        bottom.textProperty().bind(top.textProperty());

        VBox root = new VBox(10);
        root.getChildren().addAll(top, bottom);
        root.setAlignment(Pos.CENTER);
        Scene scene = new Scene(root, 480, 200);
        stage.setScene(scene);
        stage.setTitle("Property Example");
        stage.show();
    }

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

All that we are interested in here is the line of code that binds the properties:

```
bottom.textProperty().bind(top.textProperty());
```

You have seen previously that the `getText` method of `TextField` returns a `String`. However, the `text` property itself is retrieved by using the method `textProperty` (inherited from the parent class). This is the general pattern for properties of JavaFX classes: for example, you will see below that a `Slider` class has a property called `value` of type `DoubleProperty`; this is retrieved with the `valueProperty` method.

As we have said, properties have a method called `bind`. We have used this in our example, calling the `bind` method of the `text` property of the bottom field. The property that is sent into this `bind` method—in our case the `text` property of the top field—is the one that can change and whose changes are mirrored in the bound property.

We could have replaced the above line of code with this:

```
bottom.textProperty().bindBidirectional(top.textProperty());
```

Using the `bindBidirectional` method means that we can change either property and it is mirrored in the other one.

We could, of course, "chain" properties so that we could have three or more properties working in unison. This is left for you to try in the end of chapter exercises.

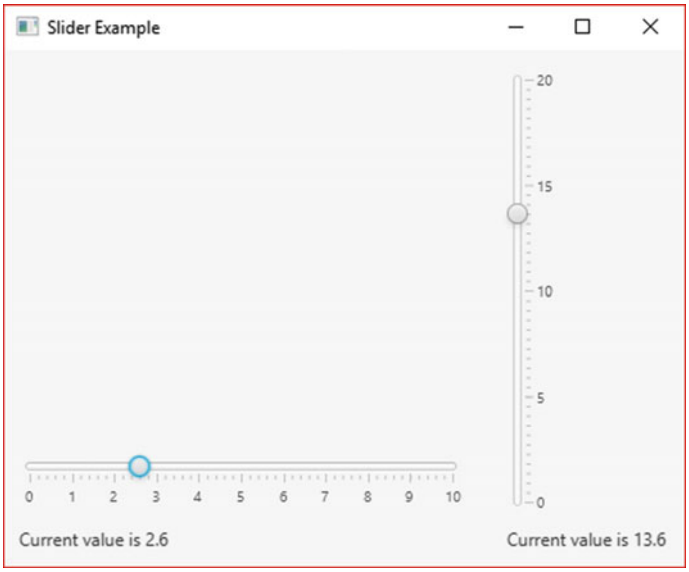
## 16.4 The *Slider* Class

A slider allows us to control the value of a variable by moving a sliding bar which is used to vary the value within a particular range. The application we have developed in Fig. 16.5 shows two sliders, one horizontal and one vertical. The current value of the movable "thumb" on each slider is displayed below the slider.

The complete code is shown below. Once you have had a look at it we will go through it with you.

***SliderDemo***

```
import java.text.DecimalFormat;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Orientation;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
public class SliderDemo extends Application
{
    @Override
    public void start(Stage stage)
    {
        final double horizSliderWidth = 300;
        final double vertSliderHeight = 300;

        // numbers will be formatted to one decimal place
        DecimalFormat df = new DecimalFormat("0.0");

        // create and configure the vertical slider
        Slider vertSlider = new Slider(0, 20, 0);
        vertSlider.setMinHeight(vertSliderHeight);
        vertSlider.setShowTickMarks(true);
        vertSlider.setShowTickLabels(true);
        vertSlider.setSnapToTicks(true);
        vertSlider.setMajorTickUnit(5.0);
        vertSlider.setMinorTickCount(10);
        vertSlider.setOrientation(Orientation.VERTICAL); // default is horizontal

        // create and configure the horizontal slider
        Slider horizSlider = new Slider(0, 10, 0);
        horizSlider.setMinWidth(horizSliderWidth);
        horizSlider.setShowTickMarks(true);
        horizSlider.setShowTickLabels(true);
        horizSlider.setSnapToTicks(true);
        horizSlider.setMajorTickUnit(1.0);
        horizSlider.setMinorTickCount(4);

        // create two lables to keep track of each slider position
        Label horizLabel = new Label("Current value is 0.0");
        Label vertLabel = new Label("Current value is 0.0");

        // add a listener to the vertical slider
        vertSlider.valueProperty().addListener((observable, oldValue, newValue)  ->
                                    vertLabel.setText("Current value is " + df.format(newValue)));

        // add a listener to the horizonal slider
        horizSlider.valueProperty().addListener((obsValue, oldValue, newValue)  ->
                                    horizLabel.setText("Current value is " + df.format(newValue)));

        // create and configure a VBox to hold the vertical slider and label
        VBox vertBox = new VBox(10);
        vertBox.setAlignment(Pos.BOTTOM_LEFT);
        vertBox.setMinWidth(horizSliderWidth/3);
        vertBox.getChildren().addAll(vertSlider, vertLabel);

        // create and configure a VBox to hold the horizontal slider and label
        VBox horizBox = new VBox(10);
        horizBox.setAlignment(Pos.BOTTOM_LEFT);
        horizBox.getChildren().addAll(horizSlider, horizLabel);

        // create and configure an HBox as root
        HBox root = new HBox(30);
        root.setPadding(new Insets(10, 10, 10, 10));
        root.getChildren().addAll(horizBox, vertBox);

        // create and configure the scene and stage
        Scene scene = new Scene(root, 460, 350);
        stage.setScene(scene);
        stage.setTitle("Slider Example");
        stage.show();
    }

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



Current value is 2.6

Current value is 13.6

**Fig. 16.5** A horizontal and a vertical slider

After declaring some constants for the preferred width and height of the sliders, and defining a decimal format for the values, we create our sliders. First the vertical slider:

```
Slider vertSlider = new Slider(0, 20, 0);
vertSlider.setMinHeight(vertSliderHeight);
vertSlider.setShowTickMarks(true);
vertSlider.setShowTickLabels(true);
vertSlider.setSnapToTicks(true);
vertSlider.setMajorTickUnit(5.0);
vertSlider.setMinorTickCount(10);
vertSlider.setOrientation(Orientation.VERTICAL);
```

The constructor takes three **`double`** values, which represent, respectively, the minimum, maximum and initial values of the movable thumb. In our case the slider will have a range of zero to 20, and start at zero.

The other methods that you see should be self-explanatory—there are still more methods that you can explore, and as usual the best approach is to experiment with these for yourself.

The horizontal slider is created and configured in the same way, but in this case it is not necessary to specify the orientation, as horizontal is the default.

We go on to create the labels where the values will be displayed;

```
Label horizLabel = new Label("Current value is 0.0");
Label vertLabel = new Label("Current value is 0.0");
```

Next we define what happens when the value of the slider is moved, starting with the vertical slider. In the case of a slider we add the listener not to the slider itself but to one of its properties, `valueProperty`, which is of type `DoubleProperty`, and which holds the current position of the movable thumb:

```
vertSlider.valueProperty().addListener((observable, oldValue, newValue)  ->
                                      vertLabel.setText("Current value is " + df.format(newValue)));
```

We are not able to use a convenience method for this, but instead use the `addListener` method. The kind of listener we are adding is a `ChangeListener`, which has an abstract method named `changed`. You can see that we are using a lambda expression for this purpose. The JavaFX documentation tells us that the `changed` method is specified as follows:

```
changed(ObservableValue<? extends T> observable, T oldValue, T newValue)
```

When the value of the slider position changes, the `changed` method is called. It receives three parameters. The first of these represents the current value, and is of type `ObservableValue`—this is yet another wrapper class. As you can see it is a generic class. You will notice that the method is specified using a wildcard so that it will take objects of a particular type or subtypes of that type. In the case of a slider it will handle values of type `Double`. The next two parameters are the old value before it was changed, and the new value after the change.

Fortunately, in our lambda expression, we do not have to worry about specifying the types of the parameters, because the compiler does this for us—another example of the very useful ability of Java compilers to utilize type inference.

You can see that in our lambda expression we have used the new value parameter to display the position of the slider on the correct label.

The behaviour of the horizontal slider is specified in exactly the same way.

The rest of the code is all to do with placing our sliders on the scene. There is not too much to say about this because there is nothing new here—notice however that the box holding the vertical slider has been specified to be one-third the width of the box containing the horizontal slider.

## 16.5 Multimedia Nodes

It is very common that we want to embed such things as images or videos into our applications. Here we will briefly explore three classes that help us to do this—`ImageView`, `MediaView` and `WebView`.

### 16.5.1 Embedding Images

We will begin by looking at `ImageView`, a class that, as its name suggests, allows us to embed images. In Fig. 16.6 we see a very simple application in which an image is embedded into a scene graphic.

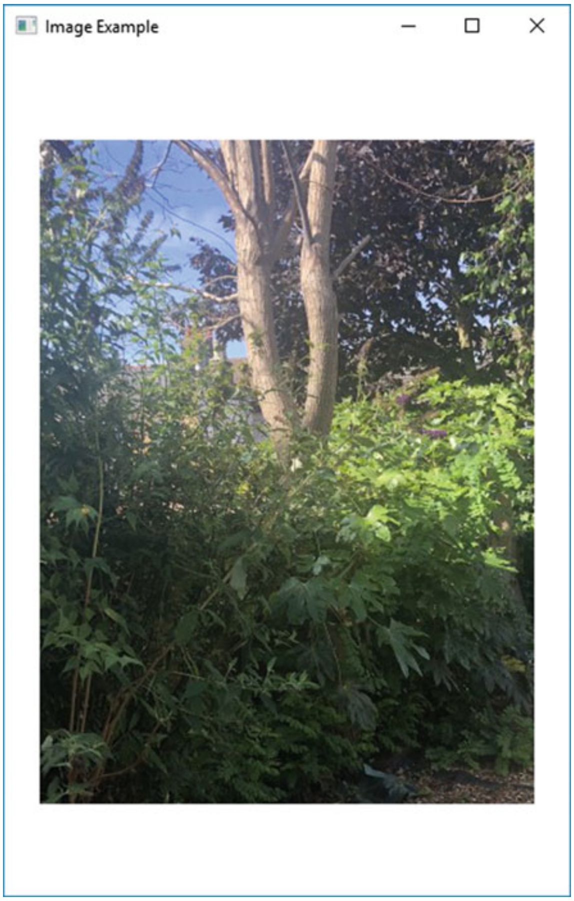


**Fig. 16.6** Embedding an image

The application code below shows how we did this:

***ImageHolder***

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.FlowPane;


public class ImageHolder extends Application
{
    @Override
    public void start(Stage stage)
    {
       Image image = new Image("Trees.jpg");  // create an image from a file
       ImageView imageView = new ImageView(image); // wrap the image in an ImageView node

       FlowPane root = new FlowPane();
       root.setAlignment(Pos.CENTER);
       root.getChildren().add(imageView); // add the ImageView object to the container

       Scene scene = new Scene(root, 400, 600);
       stage.setScene(scene);
       stage.setTitle("Image Example");

       stage.show();
    }

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

You can see that we created an object of the `Image` class from a file, `Trees.jpg`.[1]

```
Image image = new Image("Trees.jpg");
```

In order to be able to add this image to a container, we need create an object of the `ImageView` class:

```
ImageView imageView = new ImageView(image);
```

`ImageView` extends `Node`, and therefore, once we have wrapped it around an image, it can be added to a container (in this case a `FlowPane`) in the usual way:

```
root.getChildren().add(imageView);
```

Figure 16.7 shows a variation of this application.

[1]The program needs to have access to this file when it is run—normally it would be in the same directory as the `main` class.

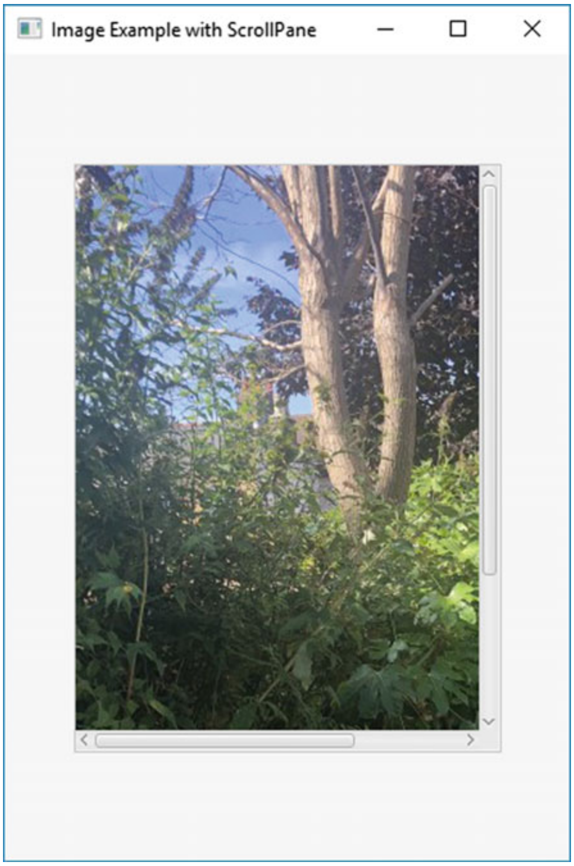


**Fig. 16.7** Embedding an image with a scroll pane

In this case, we have added our `imageView` object to a scroll pane, which we have then configured with a particular width and height:

```
ScrollPane scrollPane = new ScrollPane(imageView);
scrollPane.setPrefViewportWidth(250);
scrollPane.setPrefViewportHeight(350);
```

Then, instead of adding the `imageView` object directly to the `FlowPane`, we have added the `ScrollPane` object instead:

```
root.getChildren().add(scrollPane);
```

We can also add images to controls such as buttons. Figure 16.8 shows an image, a simple orange square, embedded in a button. In the first version the button was created without a caption, in the second with the caption "Quit".

Once the button was created, the image was added as follows:

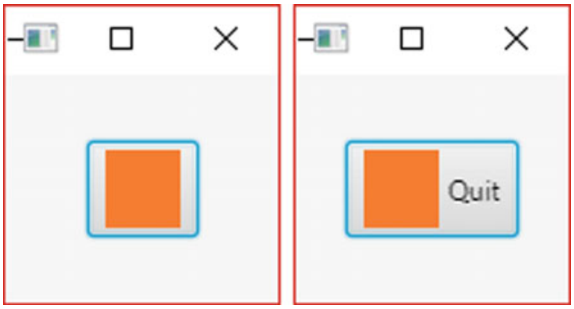


**Fig. 16.8** Adding an image to a button

```
Image image = new Image("OrangeSquare.png");
ImageView imageView = new ImageView(image);
button.setGraphic(imageView);
```

### 16.5.2 Embedding Videos

In the case of embedding a video there are some additional concepts that we need to show you, but the process is similar. We create a `Media` object from a file from which we then create a `MediaPlayer` object. This is then wrapped in a `MediaView` object. `MediaView` is an extension of `Node`, so can be added to a container. The video formats that are supported are MPEG-4 and FLV. MP3 audio is also supported.

Our application is shown in Fig. 16.9.

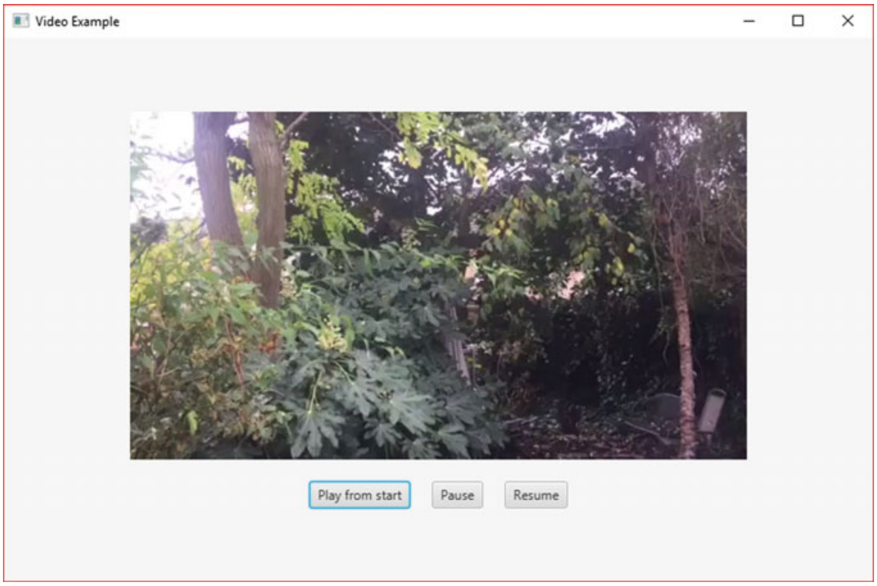


**Fig. 16.9** Embedding a video

As you can see, we have provided options for playing the video from the start as well as pausing and resuming. The code below shows how we have achieved this:

***VideoPlayer***

```
import java.io.File;
import javafx.util.Duration;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaView;
import javafx.stage.Stage;


public class VideoPlayer extends Application
{
    @Override
    public void start(Stage stage)
    {
        // create a File object from the video file
        File file = new File("Trees.mp4");

        // create a Media object from the File Object
        Media media = new Media(file.toURI().toString());

        // create a MediaPlayer object from the Media Object
        MediaPlayer mp = new MediaPlayer(media);

        // create a MediaView object from the MediaPlayer Object
        MediaView mv = new MediaView(mp);

        // create the buttons
        Button playFromStartButton = new Button("Play from start");
        Button pauseButton = new Button("Pause");
        Button resumeButton = new Button("Resume");

        // create and configure an HBox to hold the buttons
        HBox buttonBox = new HBox(20);
        buttonBox.setAlignment(Pos.CENTER);
        buttonBox.getChildren().addAll(playFromStartButton, pauseButton, resumeButton);

        // add event handlers to the buttons
        playFromStartButton.setOnAction(e -> {
                                        mp.seek(Duration.millis(0));
                                        mp.play();
                                       }
                                 );

        pauseButton.setOnAction(e -> mp.pause());

        resumeButton.setOnAction(e-> mp.play());

        // create the root container, and add it to the scene and stage
        VBox root = new VBox(20);
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(mv, buttonBox);


        Scene scene = new Scene(root,800,500);
        stage.setScene(scene);
        stage.setTitle("Video Example");

        stage.show();

    }

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

}
```

The Media object that we need to create requires an absolute reference to the source file. This is referred to as a **Uniform Resource Identifier (URI)**. In order to provide this we need first to create an instance of the File class which is provided in the java.io package.

```
File file = new File("Trees.mp4");
```

File has a method called toURI which retrieves the file's URI and allows us to send a String representation to the new Media object in the following way:

```
Media media = new Media(file.toURI().toString());
```

Now we are able to create our MediaPlayer object and hence our Media-View object

```
MediaPlayer mp = new MediaPlayer(media);
MediaView mv = new MediaView(mp);
```

Next we create our buttons and add them to an HBox, which will later be added to a VBox along with the MediaView object. Having done that, we need to add event handlers to hold the code that must be executed when the buttons are pressed.

The MediaPlayer class has a progress counter to keep track of the progress of the video. This value is held in an object of the Duration class, which has methods toMillis, toSeconds, toMinutes and toHours, which express the duration in milliseconds, seconds, minutes and hours respectively. It also has **static** methods called millis and minutes, each of which accepts a **double** and returns a Duration object representing the specified number of milliseconds or minutes respectively.

MediaPlayer has a method called seek which accepts a Duration object and moves the progress counter to the specified point. It also has a pause method and a play method, the second of which plays the video from the point indicated by the progress counter (so it resumes after a pause).

With this information in mind we can look at the code for the "Play from Start" button.

```
playFromStartButton.setOnAction(e -> {
                                          mp.seek(Duration.millis(0));
                                          mp.play();
                                     }
                                );
```

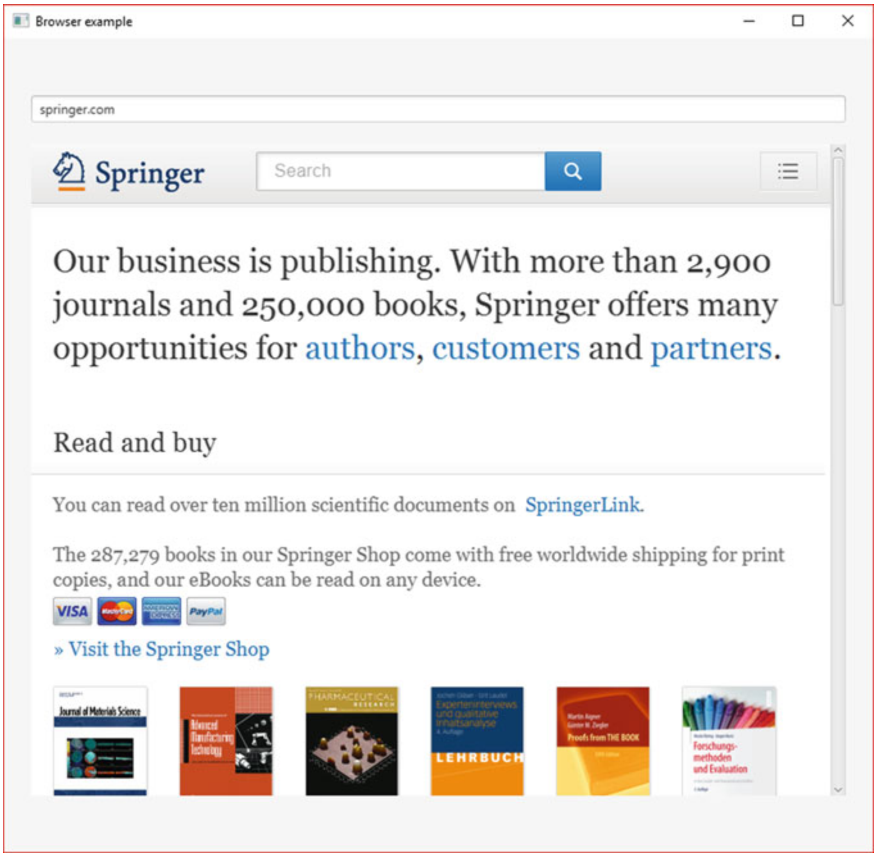
You can see how we use the `seek` method to set the counter back to the beginning, and then call the `play` method.

The "pause" and "resume" buttons simply invoke `pause` and `play` respectively.

### 16.5.3 Embedding Web Pages

To embed a webpage we need to create a WebView object that we can add to a container. This class incorporates a `WebEngine`, which is a class that is capable of managing web pages.

In Fig. 16.10 you can see our simple application. It provides a field for the user to type in the website address (the URL), and once the <Enter> key has been pressed the web page is retrieved and displayed.



Our business is publishing. With more than 2,900 journals and 250,000 books, Springer offers many opportunities for authors, customers and partners.

Read and buy

You can read over ten million scientific documents on SpringerLink.

The 287,279 books in our Springer Shop come with free worldwide shipping for print copies, and our eBooks can be read on any device.

» Visit the Springer Shop

**Fig. 16.10** Embedding a web page

Here is the code:

***WebBrowser***

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.scene.web.WebView;
import javafx.stage.Stage;

public class WebBrowser extends Application
{
  @Override
  public void start(Stage stage)
  {
        // create and configure a WebView node
        WebView wv = new WebView();
        wv.setMaxSize(750, 700);

        // create the text field where the URL will be entered
        TextField entry = new TextField();
        entry.setMaxWidth(750);

        /* define the behaviour that occurs when the URL has been entered and
           the Enter key has been pressed */
        entry.setOnKeyTyped(e -> {
                             String url;
                             if(e.getCharacter().equals("\r"))
                             {
                                       url = entry.getText();
                                       if(!url.startsWith("http"))
                                       {
                                           url = "http://" + url;
                                       }
                                       wv.getEngine().load(url);
                         }

                   });


        VBox root = new VBox(20);
        root.setAlignment(Pos.CENTER);

        root.setMaxSize(750,700);
        root.getChildren().addAll(entry, wv);

        Scene scene = new Scene(root, 800, 750);

        stage.setScene(scene);
        stage.setTitle("Browser example");
        stage.show();
  }

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

You can see that having created both the `WebView` and the `TextField`, we go on to add an event handler to the `TextField` in order to provide the code that will be executed when the URL is entered:

```
entry.setOnKeyTyped(e -> {
                              String url;
                              if(e.getCharacter().equals("\r"))
                              {
                                        url = entry.getText();
                                        if(!url.startsWith("http"))
                                        {
                                            url = "http://" + url;
                                        }
                                        wv.getEngine().load(url);
                         }

                   });
```

After declaring a variable to hold the URL, we check to see if the <Enter> key has been pressed (special character "`\r`"). We then check to see if the string entered starts with "http". This is because the method we are going to use to load the page requires the full URL, which includes the protocol—normally this would be "http" or "https". Usually people leave this out went entering a URL, and let the browser deal with it. That is what we have done here—if the protocol has not been included we prefix the address with "http://".

Next we retrieve the engine from the `WebView` object with the `getEngine` method, and use the `load` method of this object to load the URL.

## 16.6 Cascading Style Sheets

Those of you who have had experience in website development will certainly be familiar with cascading style sheets (CSS). These enable a webpage, written in HTML, to be free from any formatting detail. The HTML code is on the whole restricted to providing the basic components and functionality, whereas the appearance of the page is placed in a separate file with the extension `.css`.

JavaFX provides a similar capability, although as we shall see the syntax is slightly different, with the properties requiring the prefix `-fx-`.

This topic of style sheets is in fact a vast one, and all that it is possible to do here is to whet your appetite by explaining the basic principles, which will enable you to go on and study more detail if you are interested.

In Fig. 16.11 we have created a simple registration form (the buttons have not been activated), but we have left out any formatting information, so the application is simply formatted according to the JavaFX defaults.

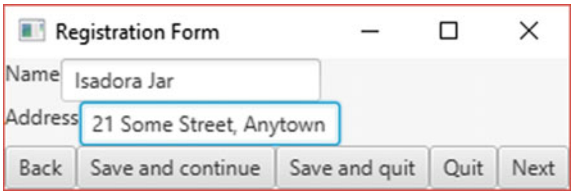


**Fig. 16.11** An unformatted application

Take a look at the code:

***CSSDemo***

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class CSSDemo extends Application
{
  @Override
  public void start(final Stage stage)
  {
      HBox row1 = new HBox();
      HBox row2 = new HBox();
      HBox row3 = new HBox();

      Label nameLabel = new Label("Name");
      TextField nameField = new TextField();

      Label addressLabel = new Label("Address");
      TextField addressField = new TextField();

      Button backButton = new Button("Back");
      Button saveAndContinueButton = new Button("Save and continue");
      Button saveAndQuitButton = new Button("Save and quit");
      Button quitButton = new Button("Quit");
      Button nextButton = new Button("Next");

      row1.getChildren().addAll(nameLabel, nameField);
      row2.getChildren().addAll(addressLabel, addressField);
      row3.getChildren().addAll(backButton, saveAndContinueButton,
                                              saveAndQuitButton, quitButton, nextButton);

      VBox root = new VBox();
      root.getChildren().addAll(row1, row2, row3);

      Scene scene = new Scene(root);
      stage.setScene(scene);
      stage.setTitle("Registration Form");
      stage.show();
  }

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

As you can see, without the formatting information the code looks very uncluttered and easy to read. This is not only something that is very useful when developing and maintaining an application, but it also has other important benefits. By placing the formatting information in a separate file it means that we can provide a similar look and feel—a corporate look—across different applications, by using the same style sheet. It also means that if we want to change the appearance of the application we can simply replace one style sheet with another. We often use the word **skin** to refer to a particular look that an application has—so to replace one skin with another, we simply change the style sheet associated with the application.

So, we added a style sheet to the above code. The result is shown in Fig. 16.12.

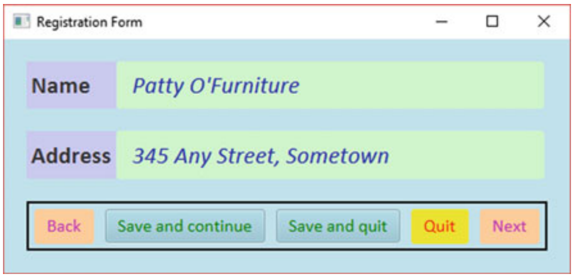


**Fig. 16.12** The registration form with a style sheet

We called this style sheet `Example.css`. To load it into our application it is necessary to add the following line of code:

```
scene.getStylesheets().add("Example.css");
```

The file needs to be located in the same directory as the `main` class.
The CSS file looks like this:

***Example.css***

```
.root
{
    -fx-font-size: 16pt;
    -fx-font-family: "Calibri";
    -fx-base: #add8e6; /* lightblue */
    -fx-spacing: 20px;
    -fx-padding: 20px;
}

.button
{
    -fx-text-fill: #008000;  /* green */
    -fx-font-size: 12pt;
}

.label
{
    -fx-background-color: #ccccff ;
    -fx-min-width : 80px;
    -fx-min-height: 42px;
    -fx-font-weight: bold;
    -fx-padding: 4px;
}

.text-field
{
    -fx-min-width: 380px;
    -fx-text-fill: #1100cc;
    -fx-background-color: #ccffcc ;
    -fx-font-style: italic;

}

.button1
{
    -fx-text-fill: #ff00ff; /* magenta */
    -fx-background-color: #ffcc99;
}

#quit
{
    -fx-text-fill: red;
    -fx-background-color: #ffff00; /* yellow */
}

#row3
{

    -fx-spacing: 10px;
    -fx-border-color: #000000; /* black */
    -fx-border-width: 2px;
    -fx-padding: 5px;

}
```

The style sheet consists of a number of **styles** or **selectors**, introduced by a full-stop or a hash. These styles refer to particular components. Many style names are provided by the system and are the same or similar to the node to which they refer: for example button, label and text-field (note the hyphen—style names tend to have hyphens when they consist of two words joined together). Styles corresponding to classes are referred to as class styles. Styles have **properties**, which refer to the properties of the particular node. The properties are set by rules which are placed between braces.

Let's take a look at the first style in our style sheet:

```
.root
{
    -fx-font-size: 16pt;
    -fx-font-family: "Calibri";
    -fx-base: #add8e6; /* lightblue */
    -fx-spacing: 20px;
    -fx-padding: 20px;
}
```

As you can see, class styles are introduced by a full-stop. Names of properties all begin with `-fx-`.

`root` is a style that refers to the root node in the scene, and all descendant nodes will have this style unless these definitions are overridden.

The first two lines set the size of the font (in points), and the font family.

The next line sets the base colour of the node. The colour here is expressed as a hexadecimal number, which corresponds to the RGB values we explained in Sect. 16.3. The first two digits (starting at the left) represent red, the next two green and the final two blue. So here we have red with an intensity of 173 (AD in hexadecimal), green with 216 (D8 in hex), and blue with 230 (E6 in hex). This actually corresponds to the pre-defined colour `lightblue`, and we could have used this constant instead. We could also have written `rgb(173, 216, 230).`

The last two lines set the spacing and padding, measured in pixels (`px`). We could have used `cm` or `in` (centimetres or inches), or we could have used `em`. An `em` is the size of the font that is currently in use—so here, 1 `em` would represent 16`pt`.

We go on to set the style properties for the `.button` class, the `.label` class and the `.text-field` class. These properties will take precedence over any properties set in the `.root` class.

After this we see the following:

```
.button1
{
    -fx-text-fill: #ff00ff; /* magenta */
    -fx-background-color: #ffcc99;
}
```

What we have done here is to set the properties for our own named style `button1`. This can then be applied to components of our choice. We set this style specifically for our "back" and "next" buttons. We apply this style to those components with the following code:

```
backButton.getStyleClass().add("button1");
nextButton.getStyleClass().add("button1");
```

Finally we see two styles introduced not with a full stop, but with a hash (#):

```
#quit
{
    -fx-text-fill: red;
    -fx-background-color: #ffff00; /* yellow */
}

#row3
{

    -fx-spacing: 10px;
    -fx-border-color: #000000; /* black */
    -fx-border-width: 2px;
    -fx-padding: 5px;
}
```

A hash introduces a style for a particular named component. In our code we gave two components an id, with the `setId` method:

```
row3.setId("row3");
quitButton.setId("quit");
```

Once we have assigned an id to a node we can then set its style with the hash as shown.

Just to summarise, the additional lines of code that we needed to include in our program were as follows:

```
scene.getStylesheets().add("Example.css");
backButton.getStyleClass().add("button1");
nextButton.getStyleClass().add("button1");
row3.setId("row3");
quitButton.setId("quit");
```

As you can imagine, we have only scratched the surface of what is available when using style sheets, but we hope it will be enough for you to do some additional reading and experiments, and of course to check out the Oracle™ website for further help and information. Those of you who are interested in JavaFX style sheets will get an opportunity to apply them as part of an end of chapter programming exercise in the advanced case study of Chap. 21.

## 16.7 Self-test Questions

1. Distinguish between a `MouseEvent` and a `KeyEvent`.
2. The following application draws a small rectangle on a scene graphic.

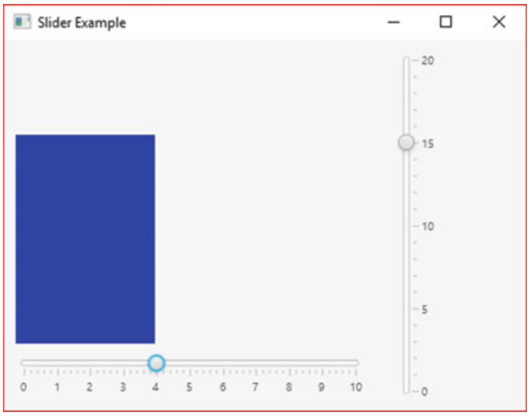
```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.geometry.Pos;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;

public class DrawRectangle extends Application
{
    @Override
    public void start(Stage stage)
    {
        final double WIDTH = 400;
        final double HEIGHT = 400;

        Rectangle rect = new Rectangle(10, 10);
        rect.setFill(Color.RED);

        VBox root = new VBox();
        root.getChildren().add(rect);
        root.setAlignment(Pos.TOP_LEFT);

        Scene scene = new Scene(root, WIDTH, HEIGHT);

        // method goes here

        stage.setScene(scene);
        stage.setTitle("Draw Rectangle");
        stage.show();
    }

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

   Replace the comment in the program with some code that would allow the user to change the width and height of the rectangle by dragging the mouse. You should look at the `RedCircle` class to get some ideas for how to go about this.

3. Explain the term *property* in the context of JavaFX components.
4. Explain what is meant by *binding* properties, and describe how this could be achieved in the case of a slider.
5. Describe how different types of *multi-media nodes* that can be incorporated into your JavaFX applications.
6. Explain the reasons for using *cascading style sheets* to separate formatting information from the rest of the application.

## 16.8 Programming Exercises

1. Implement a few of the programs that we have developed in this chapter, and experiment with different settings in order to change some the features.
2. Adapt the `PropertyExample` application of Sect. 16.3 so that three or more `TextFields` operate in unison.
3. Implement the program you adapted in question 2 of the self-test questions. Try this with other shapes such as circle and ellipse.
4. In the `RedCircle` class of Sect. 16.2.1 we used convenience methods to program responses to a `MOUSE_PRESSED` event, a `MOUSE_RELEASED` event, a `MOUSE_MOVED` event and a `MOUSE_DRAGGED` event.
   Experiment with a the `MOUSE_ENTERED` and `MOUSE_EXITED` events. One idea might be to change the colour of a button when the cursor is moved over it.
5. Adapt the `SliderDemo` from Sect. 16.4 so that instead of printing an integer value, it draws an expanding rectangle as the slider is moved. This is demonstrated in the following diagram:



6. Adapt the `SliderDemo` program so that the properties of the two sliders are bound, and when one slider moves, the other moves accordingly.
7. Design your own skin for the registration form in Sect. 16.6 by creating a new style sheet.