22.6 Playing Video with `Media`, `MediaPlayer` and `MediaViewer`

Many of today's most popular apps are multimedia intensive. JavaFX provides audio and video multimedia capabilities via the classes of package `javafx.scene.media`:

- For simple audio playback you can use class `AudioClip`.
- For audio playback with more playback controls and for video playback you can use classes `Media`, `MediaPlayer` and `MediaView`.

In this section, you'll build a basic video player. We'll explain classes `Media`, `MediaPlayer` and `MediaView` as we encounter them in the project's controller class (Section 22.6.2). The video used in this example is from NASA's multimedia library[3] and was downloaded from

3. For NASA's terms of use, visit `http://www.nasa.gov/multimedia/guidelines/`.

```
http://www.nasa.gov/centers/kennedy/multimedia/HD-ind
```

The video file `sts117.mp4` is provided in the `video` folder with this chapter's examples. When building the app from scratch, copy the video onto the app's folder.

Media Formats

For video, JavaFX supports MPEG-4 (also called MP4) and Flash Video formats. We downloaded a Windows WMV version of the video file used in this example, then converted it to MP4 via a free online video converter.[4]

4. There are many free online and downloadable video-format conversion tools. We used the one at `https://convertio.co/video-converter/`.

ControlsFX Library's `ExceptionDialog`

`ExceptionDialog` is one of many additional JavaFX controls available through the open-source project ControlsFX at

```
http://controlsfx.org
```

We use an `ExceptionDialog` in this app to display a message to the user if an error occurs during media playback.

You can download the latest version of ControlsFX from the preceding web page, then extract the contents of the ZIP file. Place the extracted ControlsFX JAR file (named `controlsfx-8.40.12.jar` at the time of this writing) in your project's folder—a JAR file is a compressed archive like a ZIP file, but contains Java class files and their corresponding

resources. We included a copy of the JAR file with the final example.

Compiling and Running the App with ControlsFX

To compile this app, you must specify the JAR file as part of the app's classpath. To do so, use the `javac` command's `-classpath` option, as in:

```
javac -classpath .;controlsfx-8.40.12.jar *.java
```

Similary, to run the app, use the `java` command's `-cp` option, as in

```
java -cp .;controlsfx-8.40.12.jar VideoPlayer
```

In the preceding commands, Linux and macOS users should use a colon (`:`) rather than a semicolon(`;`). The classpath in each command specifies the current folder containing the app's files—this is represented by the dot (`.`)—and the name of the JAR file containing the ControlsFX classes (including `ExceptionDialog`).

22.6.1 VideoPlayer GUI

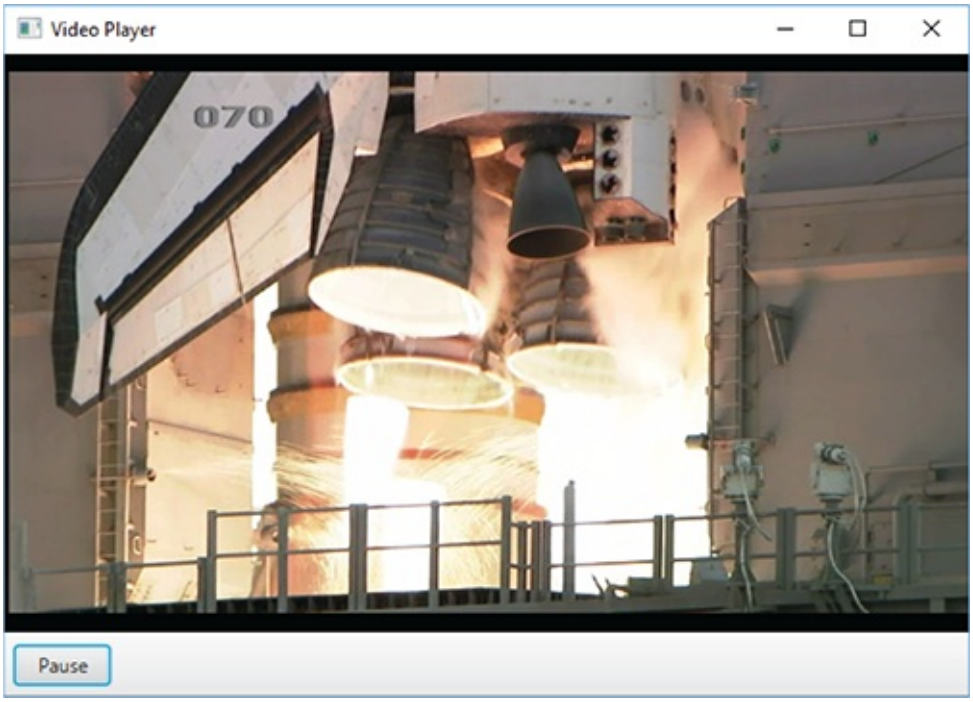
Figure 22.8 shows the completed `VideoPlayer.fxml` file and two sample screen captures of the final running `VideoPlayer` app. The GUI's layout is a `BorderPane` consisting of

- a `MediaView` (located in the Scene Builder **Library**'s **Controls** section) with the `fx:id mediaView` and
- a `ToolBar` (located in the Scene Builder **Library**'s **Containers** section) containing one `Button` with the `fx:id playPauseButton` and the text `"Play"`. The controller method `playPauseButtonPressed` responds when the `Button` is pressed.

We placed the `MediaView` in the `BorderPane`'s center region (lines 25–27) so that it occupies all available space in the `BorderPane`, and we placed the `ToolBar` in the `BorderPane`'s bottom region (lines 15–24). By default, Scene Builder adds one `Button` to the `ToolBar` when you drag the `ToolBar` onto your layout. You can then add other controls to the `ToolBar` as necessary. We set the controller class to `VideoPlayerController`.

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <!-- Fig. 22.8: VideoPlayer.fxml -->
 3  <!-- VideoPlayer GUI with a MediaView and a Butt
 4
 5  <?import javafx.scene.control.Button?>
 6  <?import javafx.scene.control.ToolBar?>
 7  <?import javafx.scene.layout.BorderPane?>
 8  <?import javafx.scene.media.MediaView?>
 9
10  <BorderPane prefHeight="400.0" prefWidth="600.0"
11     style="-fx-background-color: black;"
12     xmlns="http://javafx.com/javafx/8.0.60"
13     xmlns:fx="http://javafx.com/fxml/1"
```

```
14      fx:controller="VideoPlayerController">
15          <bottom>
16          <ToolBar prefHeight="40.0" prefWidth="200.
17              BorderPane.alignment="CENTER">
18              <items>
19                  <Button fx:id="playPauseButton"
20                      onAction="#playPauseButtonPressed"
21                      prefWidth="60.0" text="Play" />
22              </items>
23          </ToolBar>
24      </bottom>
25      <center>
26          <MediaView fx:id="mediaView" BorderPane.al
27      </center>
28  </BorderPane>
```



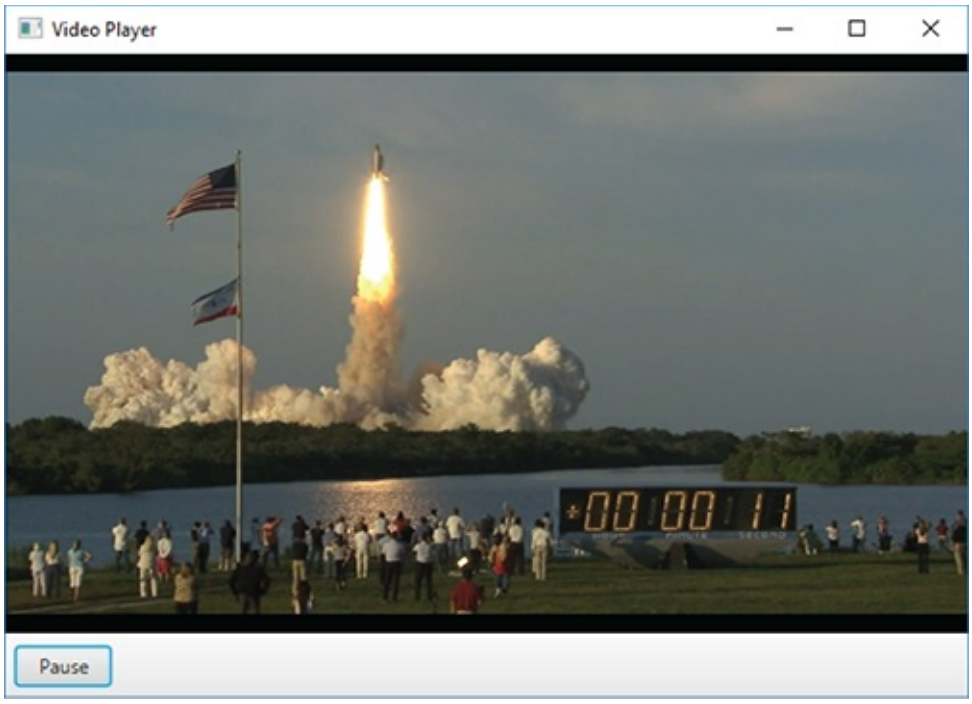


Fig. 22.8

`VideoPlayer` GUI with a `MediaView` and a `Button`. Video courtesy of NASA—see `http://www.nasa.gov/multimedia/guidelines/` for usage guidelines.

Description

22.6.2 `VideoPlayerController` Class

Figure 22.9 shows the completed `VideoPlayerController` class, which configures video playback and responds to state changes from the `MediaPlayer` and the events when the user presses the `playPauseButton`. The controller uses classes `Media`, `MediaPlayer` and `MediaView` as follows:

- A `Media` object specifies the location of the media to play and provides access to various information about the media, such as its duration, dimensions and more.
- A `MediaPlayer` object loads a `Media` object and controls playback. In addition, a `MediaPlayer` transitions through its various states (*ready*, *playing*, *paused*, etc.) during media loading and playback. As you'll see, you can provide `Runnable`s that execute in response to these state transitions.
- A `MediaView` object displays the `Media` being played by a given `MediaPlayer` object.

```
1  // Fig. 22.9: VideoPlayerController.java
2  // Using Media, MediaPlayer and MediaView to pla
3  import java.net.URL;
4  import javafx.beans.binding.Bindings;
5  import javafx.beans.property.DoubleProperty;
6  import javafx.event.ActionEvent;
7  import javafx.fxml.FXML;
8  import javafx.scene.control.Button;
9  import javafx.scene.media.Media;
10 import javafx.scene.media.MediaPlayer;
11 import javafx.scene.media.MediaView;
12 import javafx.util.Duration;
13 import org.controlsfx.dialog.ExceptionDialog;
14
15 public class VideoPlayerController {
16    @FXML private MediaView mediaView;
17    @FXML private Button playPauseButton;
18    private MediaPlayer mediaPlayer;
```

```
19      private boolean playing = false;
20
21      public void initialize() {
22          // get URL of the video file
23          URL url = VideoPlayerController.class.getR
24
25          // create a Media object for the specified
26          Media media = new Media(url.toExternalForm
27
28          // create a MediaPlayer to control Media p
29          mediaPlayer = new MediaPlayer(media);
30
31          // specify which MediaPlayer to display in
32          mediaView.setMediaPlayer(mediaPlayer);
33
34          // set handler to be called when the video
35          mediaPlayer.setOnEndOfMedia(
36                  new Runnable() {
37                  public void run() {
38                      playing = false;
39                  playPauseButton.setText("Play");
40                  mediaPlayer.seek(Duration.ZERO);
41                      mediaPlayer.pause();
42                  }
43              }
44          );
45
46          // set handler that displays an ExceptionD
47          mediaPlayer.setOnError(
48                  new Runnable() {
49                  public void run() {
50                  ExceptionDialog dialog =
51                  new ExceptionDialog(mediaPlaye
52                      dialog.showAndWait();
53                  }
54              }
55          );
56
57          // set handler that resizes window to vide
58          mediaPlayer.setOnReady(
```

Fig. 22.9

Using `Media`, `MediaPlayer` and `MediaView` to play a video.

Instance Variables

Lines 16–19 declare the controller's instance variables. When the app loads, the `mediaView` variable (line 16) is assigned a reference to the `MediaView` object declared in the app's FXML. The `mediaPlayer` variable (line 18) is configured in method `initialize` to load the video specified by a `Media` object and used by method `playPauseButtonPressed` (lines 73–85) to play and pause the video.

Creating a `Media` Object Representing the Video to Play

Method `initialize` configures media playback and registers event handlers for `MediaPlayer` events. Line 23 gets a `URL` representing the location of the `sts117.mp4` video file. The notation

```
VideoPlayerController.class
```

creates a `Class` object representing the `VideoPlayerController` class. This is equivalent to calling inherited method `getClass()`. Next line 26 creates a `Media` object representing the video. The argument to the

`Media` constructor is a `String` representing the video's location, which we obtain with `URL` method `toExternalForm`. The URL `String` can represent a local file on your computer or can be a location on the web. The `Media` constructor throws various exceptions, including `MediaException`s if the media cannot be found or is not of a supported media format.

Creating a `MediaPlayer` Object to Load the Video and Control Playback

To load the video and prepare it for playback, you must associate it with a `MediaPlayer` object (line 29). Playing multiple videos requires a separate `MediaPlayer` for each `Media` object. However, a given `Media` object can be associated with multiple `MediaPlayer`s. The `MediaPlayer` constructor throws a `NullPointerException` if the `Media` is `null` or a `MediaException` if a problem occurs during construction of the `MediaPlayer` object.

Attaching the `MediaPlayer` Object to the

`MediaView` to Display the Video

A `MediaPlayer` does not provide a view in which to display video. For this purpose, you must associate a `MediaPlayer` with a `MediaView`. When the `MediaView` already exists—such as when it's created in FXML—you call the `MediaView`'s `setMediaPlayer` method (line 32) to perform this task. When creating a `MediaView` object programmatically, you can pass the `MediaPlayer` to the `MediaView`'s constructor. A `MediaView` is like any other `Node` in the scene graph, so you can apply CSS styles, transforms and animations (Sections 22.7–22.9) to it as well.

Configuring Event Handlers for `MediaPlayer` Events

A `MediaPlayer` transitions through various states. Some common states include *ready*, *playing* and *paused*. For these and other states, you can execute a task as the `MediaPlayer` enters the corresponding state. In addition, you can specify tasks that execute when the end of media playback is reached or when an error occurs during playback. To perform a task for a given state, you specify an object that implements the `Runnable` interface (package `java.lang`). This interface contains a no-parameter `run` method that returns `void`.

For example, lines 35–44 call the `MediaPlayer`'s `setOnEndOfMedia` method, passing an object of an anonymous inner class that implements interface `Runnable` to execute when video playback completes. Line 38 sets the `boolean` instance variable `playing` to `false` and line 39 changes the text on the `playPauseButton` to `"Play"` to indicate that the user can click the `Button` to play the video again. Line 40 calls `MediaPlayer` method `seek` to move to the beginning of the video and line 41 pauses the video.

Lines 47–55 call the `MediaPlayer`'s `setOnError` method to specify a task to perform if the `MediaPlayer` enters the *error* state, indicating that an error occurred during playback. In this case, we display an `ExceptionDialog` containing the `MediaPlayer`'s error message. Calling the `ExceptionDialog`'s `showAndWait` method indicates that the app must wait for the user to dismiss the dialog before continuing.

Binding the `MediaViewer`'s Size to the Scene's Size

Lines 58–69 call the `MediaPlayer`'s `setOnReady` method to specify a task to perform if the `MediaPlayer` enters the *ready* state. We use property bindings to bind the `MediaView`'s `width` and `height` properties to the scene's `width` and `height` properties so that the `MediaView` resizes with app's window. A `Node`'s `sceneProperty`

returns a `ReadOnlyObjectProperty<Scene>` that you can use to access to the `Scene` in which the `Node` is displayed. The `ReadOnlyObjectProperty<Scene>` represents an object that has many properties. To bind to a specific properties of that object, you can use the methods of class `Bindings` (package `javafx.beans.binding`) to select the corresponding properties. The `Scene`'s `width` and `height` are each `DoubleProperty` objects. `Bindings` method `selectDouble` gets a reference to a `DoubleProperty`. The method's first argument is the object that contains the property and the second argument is the name of the property to which you'd like to bind.

Method `playPauseButtonPressed`

The event handler `playPauseButtonPressed` (lines 73–85) toggles video playback. When `playing` is `true`, line 78 sets the `playPauseButton`'s text to `"Pause"` and line 79 calls the `MediaPlayer`'s `play` method; otherwise, line 82 sets the `playPauseButton`'s text to `"Play"` and line 83 calls the `MediaPlayer`'s `pause` method.

Using Java SE 8 Lambdas

to Implement the `Runnables`

8

Each of the anonymous inner classes in this controller's `initialize` method can be implemented more concisely using lambdas as shown in Section 17.16.