1 Stages, Scenes, and Layout

During the last decade, user interfaces have evolved beyond the capabilities of the old Java technologies. Modern users want to work with visually appealing applications and are used to the rich user interfaces brought by Web 2.0 and smartphones.

To address that, JavaFX was envisioned and added to Java a few releases ago. It was created from scratch to avoid any backward compatibility issues, and with a great understanding of the needs of modern user interfaces.

In this book, we will review the most important JavaFX APIs and will look into resolving some of the most common problems that JavaFX developers face, based on my development experience and over 500 questions I've answered in the JavaFX section of `stackoverflow.com`.

In the first chapter, we will start with the backstage of a JavaFX application, including its windows and content area, and see which API is responsible for each of these main building blocks:

- `Application`: This handles the application workflow, initialization, and command-line parameters
- `Stage`: The JavaFX term for the window
- `Scene`: This is the place for the window's content
- SceneGraph: The content of the `Scene`

At the end of the chapter, we will create a clock demo that will demonstrate the concepts from this chapter.

Application and JavaFX subsystems

The very first API, `javafx.application.Application`, represents the program itself. It prepares everything for us to start using JavaFX and is an entry point for all standalone JavaFX applications. It does the following:

- Initializes JavaFX toolkit (subsystems and native libraries required to run JavaFX)
- Starts JavaFX Application Thread (a thread where all UI work happens) and all working threads
- Constructs the `Application` instance (which provides a starting point for your program) and calls the user-overridden methods
- Handles application command line parameters
- Handles all cleanup and shutdown once the application ends

Let's look closely at each of these steps.

Components of the JavaFX toolkit

JavaFX toolkit is the stuff hidden under the hood of the JavaFX. It's a set of native and Java libraries that handles all the complexity of the drawing UI objects, managing events, and working with various hardware. Luckily, they are well-shielded by the API from the user. We will have a brief overview of the major components. It can be useful, for example, during debugging your application; by knowing these component names, you will be able to identify potential problems from stack traces or error messages.

Glass toolkit

This toolkit is responsible for low-level interaction with operating systems. It uses native system calls to manage windows, handle system events, timers, and other components.

Note that Glass is written from scratch; it doesn't use AWT or Swing libraries. So, it's better to not mix old Swing/AWT components and JavaFX ones for the sake of performance.

Prism and Quantum Toolkit

Prism renders things. It was optimized a lot over the course of JavaFX releases. Now, it uses hardware acceleration and software libraries available in the system such as DirectX or OpenGL. Also, Prism renders concurrently and also can render upcoming frames in advance while current frames are being shown, which gives a large performance advantage.

Quantum Toolkit manages the preceding Prism, Glass, and JavaFX API and handles events and rendering threads.

Media

This framework is responsible for video and audio data. In addition to playback functionality, JavaFX Media provides advanced functionality—for example, buffering, seeking, and progressive downloading.

For better performance, Media uses the separate thread, which is synchronized with frames, prepared by Prism, to show/play relevant media data using the correct framerate.

WebView/WebEngine

WebView is a web rendering engine based on the OpenSource WebKit engine, it supports the majority of modern HTML features.

Using WebView, you can incorporate any web resources or even whole sites into your JavaFX applications and integrate them with modern web tools, such as Google Maps.

Working with JavaFX Application Thread

Despite the development of the technology, building a thread-safe UI toolkit is still an enormous challenge due to the complexity of the events and state handling. JavaFX developers decided to follow Swing pattern, and instead of fighting endless deadlocks proclaimed that everything in the UI should be updated from and only from a special thread. It's called **JavaFX Application Thread**.

For simple programs, you don't notice this requirement, as common JavaFX program entry points are already run on this thread.

Once you started adding multithreading to your application, you will need to take care of the thread you use to update the UI. For example, it's a common approach to run a lengthy operation on the separate thread:

```
new Thread(() -> {
    //read myData from file
    root.getChildren().add(new Text(myData));
}).start();
```

This code tries to access JavaFX UI from a common `Thread`, and it will lead to:

```
java.lang.IllegalStateException: Not on FX application thread
```

To address that, you need to wrap your JavaFX code in the next construction:

```
Platform.runLater(()-> {
    root.getChildren().add(new Text("new data"));
});
```

Note that you can not construct UI objects on JavaFX Application Thread. But, once you have showed them to the user you need to follow the JavaFX UI Thread rule.

Also, note that having one thread for the update UI means that while you run your code on this thread nothing is being updated, and the application looks frozen for the user. So, any long computational, network, or file-handling tasks should be run on a regular thread.

If you need to check in your code which thread you are on, you can use the following API:

```
boolean Platform.isFxApplicationThread();
```

Application class

The most common way to use the JavaFX API is to subclass your application from the `javafx.application.Application` class. There are three overridable methods in there:

- `public void init()`: Overriding this method allows you to run code before the window is created. Usually, this method is used for loading resources, handling command-line parameters, and validating environments. If something is wrong at this stage, you can exit the program with a friendly command-line message without wasting resources on the window's creation.

Note this method is not called on *JavaFX Application Thread*, so you shouldn't construct any objects that are sensitive to it, such as `Stage` or `Scene`.

- `public abstract void start(Stage stage)`: This is the main entry point and the only method that is abstract and has to be overridden. The first window of the application has been already prepared and is passed as a parameter.
- `public void stop()`: This is the last user code called before the application exits. You can free external resources here, update logs, or save the application state.

The following JavaFX code sample shows the workflow for all these methods:

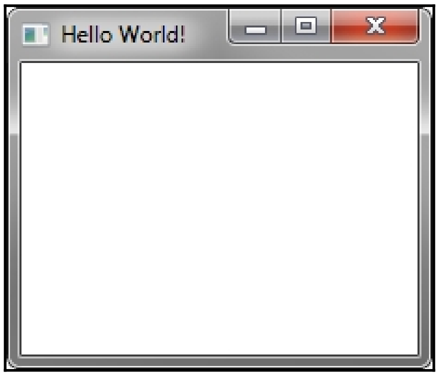
Note the comment in the first line—it shows the relative location of this code sample in our book's GitHub repository. The same comment will accompany all future code samples, for your convenience.

```
// chapter1/HelloFX.java
import javafx.application.Application;
import javafx.scene.*;
import javafx.stage.Stage;

public class FXApplication extends Application {

    @Override
    public void init() {
        System.out.println("Before");
    }

    @Override
    public void start(Stage stage) {
        Scene scene = new Scene(new Group(), 300, 250);
        stage.setTitle("Hello World!");
        stage.setScene(scene);
        stage.show();
    }

    public void stop() {
        System.out.println("After");
    }
}
```

Note that you don't need the `main()` method to run JavaFX. For example, this code can be compiled and run from the command line:

```
> javac FXApplication.java
> java FXApplication
```

It shows a small empty window:



Using the Application.launch() method

If you need to have control over the moment JavaFX starts, you can use the `Application.launch()` method:

```
public static void main(String[] args) {
    // you custom code
    Application.launch(MyApplication.class, args);
}
```

Here, `MyApplication` should extend `javafx.application.Application`.

Managing command-line parameters

Unlike the regular Java programs, which receive all parameters in the `main(String[] args)` method, JavaFX provides an extra API to get them: `Application.getParameters()`. Then, you can access them next categories:

- raw format: without any changes

- parsed named pairs: only parameters which were formatted as Java options: `--name=value`. They will be automatically built into a name-value map.
- unnamed: parameters which didn't fit into the previous category.

Let's compile and run a program with next demo parameters (run these commands from `Chapter1/src` folder of book's GitHub repository):

```
javac FXParams.java
java FXParams --param1=value1 uparam2 --param3=value3
```

This will run next code, see the corresponding API calls in bold:

```
// FXParams.java
System.out.println("== Raw ==");
getParameters().getRaw().forEach(System.out::println);
System.out.println("== Unnamed ==");
getParameters().getUnnamed().forEach(System.out::println);
System.out.println("== Named ==");
getParameters().getNamed().forEach((p, v) -> { System.out.println(p + "="
+v);});
```

JavaFX will parse these parameters and allocated them into categories:

```
== Raw ==
--param1=value1
uparam
--param3=value 3
== Unnamed ==
uparam
== Named ==
param3=value 3
param1=value1
```

Closing the JavaFX application

Usually, the JavaFX application closes once all of its windows (Stages) are closed.

You can close the application at any moment by
calling `javafx.application.Platform.exit()`.

Don't call `System.exit()` as you may be used to doing in Java programs. By doing that you break the JavaFX application workflow and may not call important logic written in on close handlers such as `Application.stop()`.

If you don't want your application to automatically close, add the following code at the beginning of your program:

```
javafx.application.Platform.setImplicitExit(false);
```

Stage – a JavaFX term for the window

Every UI app needs a window. In JavaFX, the `javafx.stage.Stage` class is responsible for that. The very first stage/windows are prepared for you by Application and your usual entry point for the app is method start, which has `Stage` as a parameter.

If you want to have more windows, just create a new `Stage`:

```
Stage anotherStage = new Stage();
stage2.show();
```

Working with Stage modality options

Modality determines whether events (for example, mouse clicks) will pass to an other application's windows. This is important as you would then need to show the user a modal dialog style window or a warning, which should be interacted with before any other action with the program.

`Stage` supports three options for modality:

- `Modality.NONE`: The new `Stage` won't block any events. This is the default.
- `Modality.APPLICATION_MODAL`: The new `Stage` will block events to all other application's windows.
- `Modality.WINDOW_MODAL`: The new `Stage` will block only events to hierarchy set by `initOwner()` methods.

These options can be set by calling the `Stage.initModality()` method.

The following sample shows how it works. Try to run it and close each window to check events handling, and see the comments inline:

```
// chapter1/FXModality.java
public class FXModality extends Application {

    @Override
    public void start(Stage stage1) {
        // here we create a regular window
        Scene scene = new Scene(new Group(), 300, 250);
        stage1.setTitle("Main Window");
        stage1.setScene(scene);
        stage1.show();

        // this window doesn't block mouse and keyboard events
        Stage stage2 = new Stage();
        stage2.setTitle("I don't block anything");
        stage2.initModality(Modality.NONE);
        stage2.show();

        // this window blocks everything - you can't interact
        // with other windows while it's open
        Stage stage3 = new Stage();
        stage3.setTitle("I block everything");
        stage3.initModality(Modality.APPLICATION_MODAL);
        stage3.show();

        // this window blocks only interaction with it's owner window
(stage1)
        Stage stage4 = new Stage();
        stage4.setTitle("I block only clicks to main window");
        stage4.initOwner(stage1);
        stage4.initModality(Modality.WINDOW_MODAL);
        stage4.show();
    }
}
```

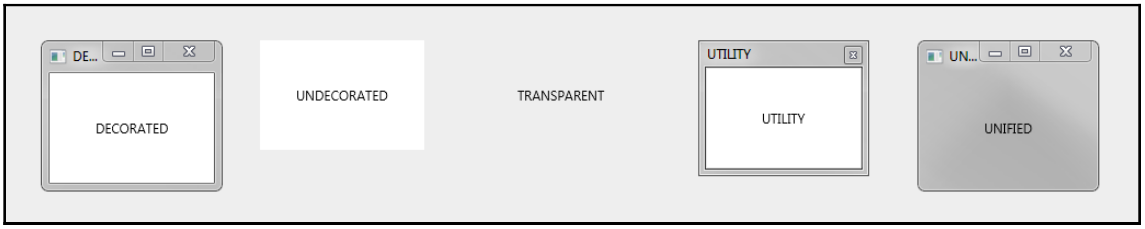
Using Stage styles

`Stage` style is the way your window is decorated outside of the `Scene`.

You can control how `Stage` will look using `StageStyle` enum values. It can be passed to the constructor or through the `initStyle()` method:

```
// chapter1/StageStylesDemo
Stage stage = new Stage(StageStyle.UNDECORATED)
// or
stage.initStyle(StageStyle.TRANSPARENT)
```

See the existing options in the following figure. Note that a Windows screenshot was used here, but decorations will look different on other operating systems because they are a part of the OS user interface and not drawn by Java or JavaFX.



Setting fullscreen and other window options

There are several other options to manipulate `Stage` that are self-explanatory, like in the following examples:

```
// chapter1.StageFullScreen.java
stage.setFullScreen(true);
stage.setIconified(true);
stage.setMaxWidth(100);
//...
```

The only unusual thing about this API is the extra fullscreen options—you can set up a warning message and key combination to exit fullscreen using the following methods:

```
 primaryStage.setFullScreenExitHint("Exit code is Ctrl+B");
primaryStage.setFullScreenExitKeyCombination(KeyCombination.valueOf("Ctrl+B
"));
```

Note the convenient `KeyCombination` class, which can parse names of shortcuts. If you prefer more strict methods, you can use `KeyCodeCombination` instead:

```
KeyCodeCombination kc = new KeyCodeCombination(KeyCode.B,
KeyCombination.CONTROL_DOWN);
```

Scene and SceneGraph

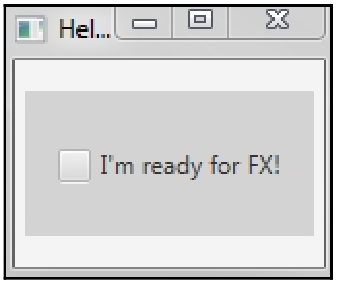
Every element of the JavaFX `Scene` is a part of the large graph (or a tree, strictly speaking) that starts from the root element of the `Scene`. All these elements are represented by the class `Node` and its subclasses.

All SceneGraph elements are split into two categories: `Node` and `Parent`. Parent is Node as well, but it can have children `Node` objects. Thus, `Node` objects are always leaves (endpoints of the SceneGraph), but `Parent` objects can be both leaves and vertices depending on whether they have children or not.

`Parent` objects generally have no idea what kind of `Node` objects their children are. They manage only direct children and delegate all further logic down the graph.

This way, you can build complex interfaces from smaller blocks step by step, organize UI elements in any way, and quickly change the configuration on the higher levels without modifying the lower ones.

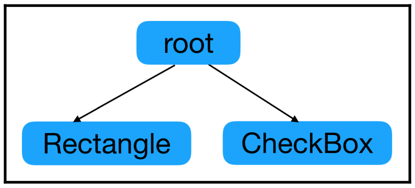
Let's take a look at the next short JavaFX application, which shows a window with a checkbox and a gray background:



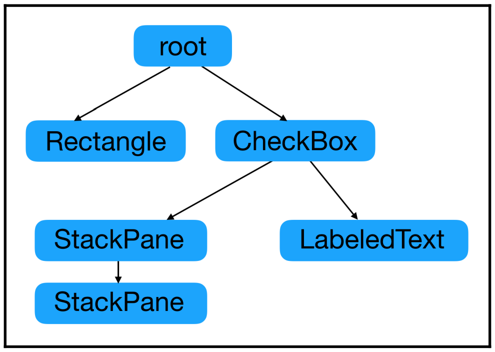
Take a look at the following code snippet:

```
public class HelloFX extends Application {
   @Override
   public void start(Stage stage) {
       StackPane root = new StackPane();
       CheckBox node = new CheckBox("I'm ready for FX!");
       Rectangle rect = new Rectangle(70, 70, Color.GREEN);
       root.getChildren().addAll(rect, node);
       Scene scene = new Scene(root, 150, 100);
       stage.setScene(scene);
       stage.setTitle("Hello FX!");
       stage.show();
   }
}
```

From the code, the scenegraph here looks like this:



But, `CheckBox` itself consists of several nodes, and by digging deeper you can see that it looks like this:



You can always check the scenegraph structure by traversing the graph, starting from the `Scene` root. Here is a convenient method that prints the scenegraph and indents each parent:

```
public void traverse(Node node, int level) {
 for (int i = 0; i < level; i++) {
  System.out.print(" ");
 }
 System.out.println(node.getClass());
 if (node instanceof Parent) {
  Parent parent = (Parent) node;
  parent.getChildrenUnmodifiable().forEach(n->traverse(n, level +1));
 }
}
```

For our HelloFX example, it will provide the following output:

```
class javafx.scene.layout.StackPane
 class javafx.scene.shape.Rectangle
 class javafx.scene.control.CheckBox
  class com.sun.javafx.scene.control.skin.LabeledText
  class javafx.scene.layout.StackPane
   class javafx.scene.layout.StackPane
```

Organizing the Scene content with Layout Managers

In this section, we will review various Layout Managers that control how your nodes are organized on a `Scene`.

Layout Managers don't have much UI by themselves; usually, only the background and borders are visible and customizable. Their main role is to manage their children nodes.

Free layout

The following managers don't relocate or resize your nodes at all: `Pane`, `Region`, and `Group`. You set coordinates for each of your nodes manually. You can use these layout managers when you want to set absolute positions for each element, or when you want to write your own layout logic.

Let's review the difference between these free layout managers.

The most basic layout manager – Group

`Group` is a very lightweight layout manager. It doesn't support a lot of customizing options (for example, background color) and doesn't have any size control—Group's size is a combination of child sizes, and anything too large will be trimmed.

So, unless you need to have a huge amount of components and care a lot about performance, consider using another manager.

Region and Pane layout managers

`Region` and `Pane` support the whole range of styles and effects. They are used as a basis for almost all JavaFX UI components.

The only difference between them is an access level to their children's list.

`Pane` gives the **public** access to the `getChildren()` method. So, it's used as an ancestor to layout managers and controls which API allows the manipulating of children.

`Region`, on the other hand, doesn't allow changing its children list. `getChildren()` is a private method, so the only way to access them is `Region.getChildrenUnmodifiable()`, which doesn't allow you to change the list. This approach is used when a component is not meant to have new children. For example, all `Controls` and `Charts` extend `Region`.

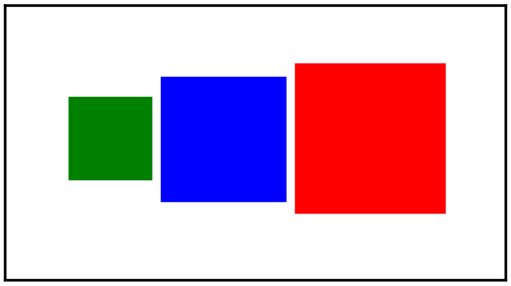
Behavioral layout

For these layout managers, you choose the behavior for layouting of your nodes, and they will do the following tasks for you:

- Calculate child nodes' sizes
- Initial positioning of the child nodes
- Reposition nodes if they change their sizes or the layout manager changes its size

The first manager to look at is `HBox`. It arranges its children in simple rows:

```
HBox root = new HBox(5);
root.getChildren().addAll(
new Rectangle(50, 50, Color.GREEN),
new Rectangle(75, 75, Color.BLUE),
new Rectangle(90, 90, Color.RED));
```



The corresponding `VBox` does the same for columns.

`StackPane` positions nodes in its center.

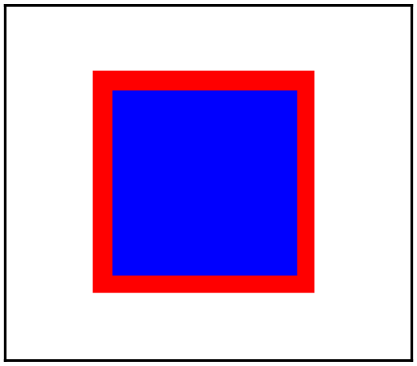
As nodes will overlap here, note that you can control their Z-order using the following APIs:

- `Node.toBack()` will push it further from the user
- `Note.toFront()` will bring it to the top position

Take a look at the following example code:

```
Pane root = new StackPane();
Rectangle red;
root.getChildren().addAll(
   new Rectangle(50, 50, Color.GREEN), // stays behind blue and red
   new Rectangle(75, 75, Color.BLUE),
   red = new Rectangle(90, 90, Color.RED));

red.toBack();
```

This is the image that it produces:



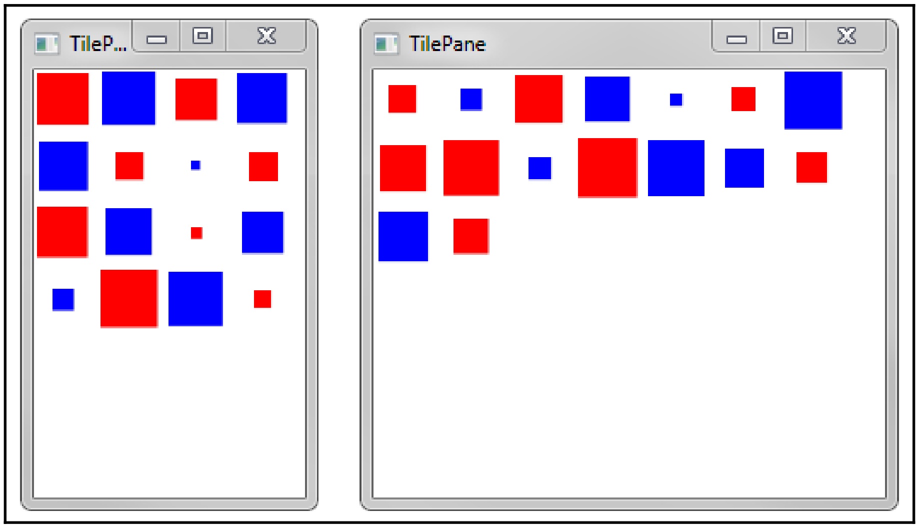
Positional layout

This group of managers allows you to choose a more precise location for each component, and they do their best to keep the node there. Each manager provides a distinct way to select where you want to have your component. Let's go through examples and screenshots depicting that.

TilePane and FlowPane

`TilePane` places nodes in the grid of the same-sized tiles. You can set preferable column and row counts, but `TilePane` will rearrange them as space allows.

In the following example, you can see different rectangles being located in the same-sized tiles:

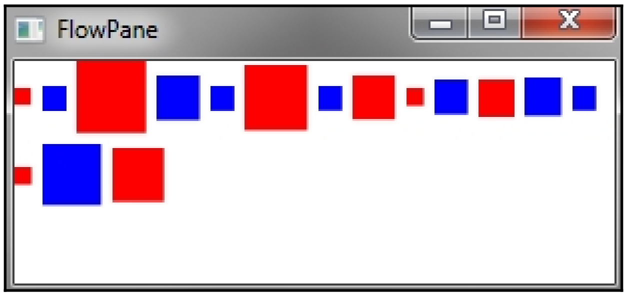


Refer to the following code:

```
// chapter1/layoutmanagers/TilePaneDemo.java
public class TilePaneDemo extends Application {
    @Override
    public void start(Stage primaryStage) {
        TilePane root = new TilePane(5,5);
        root.setPrefColumns(4);
        root.setPrefRows(4);
        // compare to
        // FlowPane root = new FlowPane(5, 5);
        for (int i = 0; i < 4; i++) {
            for (int j = 0; j < 4; j++) {
                double size = 5 + 30 * Math.random();
                Rectangle rect = new Rectangle(size, size,
                              (i+j)%2 == 0 ? Color.RED : Color.BLUE);
                root.getChildren().add(rect);
            }
        }
```

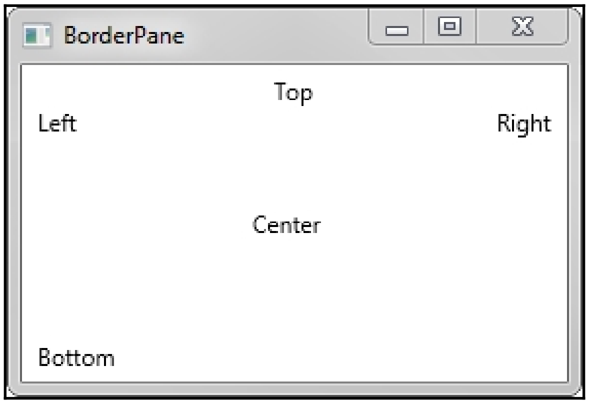
```
            Scene scene = new Scene(root, 300, 250);
            primaryStage.setTitle(root.getClass().getSimpleName());
            primaryStage.setScene(scene);
            primaryStage.show();
        }
    }
```

If you don't need tiles to have the same size, you can use `FlowPane` instead. It tries to squeeze as many elements in the line as their sizes allow. The corresponding `FlowPaneDemo.java` code sample differs from the last one only by the layout manager name, and produces the following layout:



BorderPane layout manager

`BorderPane` suggests several positions to align each subnode: top, bottom, left, right, or center:



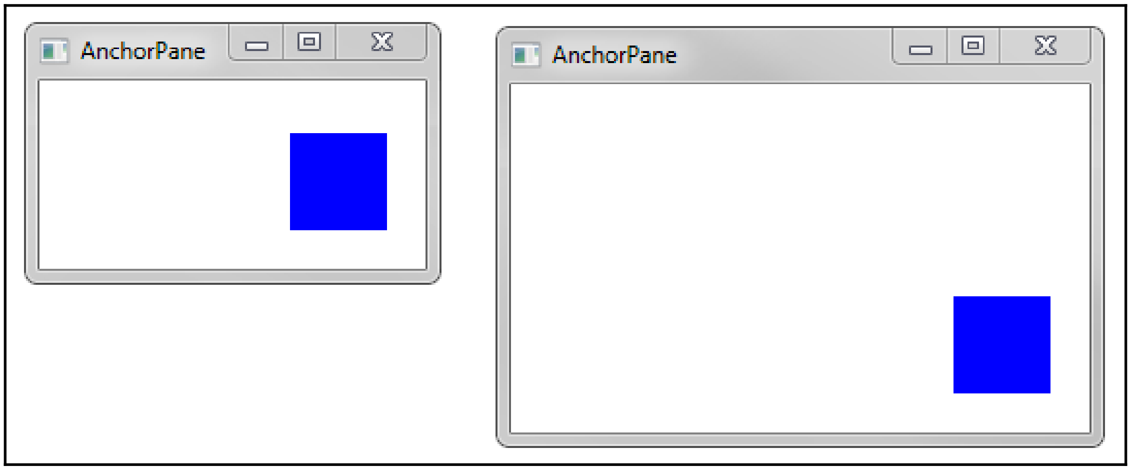
Refer to the following code:

```
BorderPane root = new BorderPane();
root.setRight(new Text("Right "));
root.setCenter(new Text("Center"));
root.setBottom(new Text(" Bottom"));
root.setLeft(new Text(" Left"));

Text top = new Text("Top");
root.setTop(top);

BorderPane.setAlignment(top, Pos.CENTER);
```

Note the last line, where the static method is used to adjust top-element horizontal alignment. This is a JavaFX-specific approach to set Pane **constraints**.

AnchorPane layout manager

This manager allows you to *anchor* any child Node to its sides to keep them in place during resizing:



Refer to the following code:

```
Rectangle rect = new Rectangle(50, 50, Color.BLUE);
Pane root = new AnchorPane(rect);
AnchorPane.setRightAnchor(rect, 20.);
AnchorPane.setBottomAnchor(rect, 20.);
```

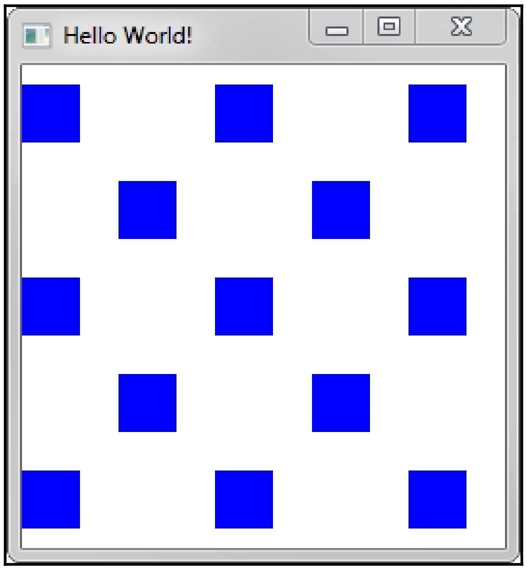
GridPane layout manager

`GridPane` is most complex layout manager; it allows users to sets rows and columns where child Nodes can be placed.

You can control a lot of constraints through the API: grow strategy, the relative and absolute sizes of columns and rows, resize policy, and so on. I won't go through all of them to avoid repeating JavaDoc, but will show only a short sample—let's make a small *chessboard pattern* using `GridPane`:

```
GridPane root = new GridPane();
for (int i = 0; i < 5; i++) {
    root.getColumnConstraints().add(new ColumnConstraints(50));
    root.getRowConstraints().add(new RowConstraints(50));
}
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 5; j++) {
        if ((i+j)%2 == 0)
            root.add(new Rectangle(30, 30, Color.BLUE), i, j);
    }
}
```

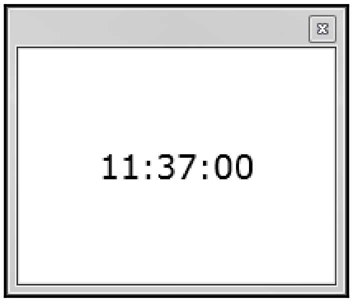
We get the following output:



Clock demo

To demonstrate the topics covered in this chapter, I have written a small clock application.

It will become more complex with each upcoming chapter; for the first *release* it just shows a current local time in text form and updates it every second, demonstrating `Stage`/`Scene` usage, one of the layout managers, and the Application FX Thread workflow:


11:37:00

See the inline comments for details about the program:

```
// chapter1/clock/ClockOne.java
public class ClockOne extends Application {
    // we are allowed to create UI objects on non-UI thread
    private final Text txtTime = new Text();
    private volatile boolean enough = false;
    // this is timer thread which will update out time view every second
    Thread timer = new Thread(() -> {
        SimpleDateFormat dt = new SimpleDateFormat("hh:mm:ss");
        while(!enough) {
            try {
                // running "long" operation not on UI thread
                Thread.sleep(1000);
            } catch (InterruptedException ex) {}
            final String time = dt.format(new Date());
            Platform.runLater(()-> {
                // updating live UI object requires JavaFX App Thread
                txtTime.setText(time);
            });
        }
    });
```

```
        @Override
        public void start(Stage stage) {
            // Layout Manager
            BorderPane root = new BorderPane();
            root.setCenter(txtTime);
            // creating a scene and configuring the stage
            Scene scene = new Scene(root, 200, 150);
            stage.initStyle(StageStyle.UTILITY);
            stage.setScene(scene);
            timer.start();
            stage.show();
        }
        // stop() method of the Application API
        @Override
        public void stop() {
            // we need to stop our working thread after closing a window
            // or our program will not exit
            enough = true;
        }

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

Summary

In this chapter, we studied the main JavaFX concepts: SceneGraph, Application, Stages, and layout. Also, we provided an overview of the main layout managers and wrote our first few JavaFX programs.

A clock demo was presented to demonstrate a JavaFX application lifecycle and working with the threads.

In the next chapter, we'll look into Scene content: Shapes, Text, and basic controls.

2
Building Blocks – Shapes, Text, and Controls

To construct a rich user interface, you need the building blocks. JavaFX provides a large range of very customizable graphical instruments. We'll start from the smallest building blocks—shapes, text, and simple controls—and will use them to understand how JavaFX works with graphical elements.

In this chapter, we will cover the following topics

- Creating and customizing the shapes
- Working with text
- Coordinates and bounds
- Basic controls

By combining and customizing these Nodes and arranging them using the layout managers we reviewed in the previous chapter, you can already build a sophisticated UI. At the end of the chapter, we'll revisit the Clock application from the previous chapter to demonstrate the topics we learned in a more complex application.

Shapes and their properties

Everything you see on the screen of your computer can be split into three large categories:

- Shapes
- Text
- Images

Thus, by being able to create each of these, you can build any UI in reasonable time. Let's start with JavaFX shapes.

JavaFX shapes overview

The simplest object you can put on a scene is a shape. Under the `javafx.scene.shape package`, the JavaFX API supports a great range of shapes, from circles and rectangles to polygons and SVG paths. Most shapes can be divided then into two categories—lines and closed shapes. The properties that are shared among all shapes will be covered in the next section. After, we will review each shape's specific APIs.

Closed shapes

There are just four options here—**Rectangle**, **Circle**, **Ellipse**, and **Polygon**. They mostly don't have any special API, just a minimum required by basic math to describe their form.

The only small difference is Rectangle, which can have rounded corners, controlled by the `setArcHeight()` and `setArcWidth()` methods.

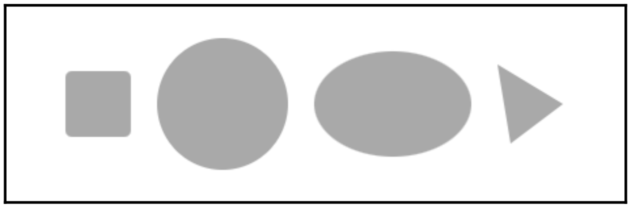
For the polygon, you need to provide the coordinates of each vertex through the `getPoints()` method.

For example, take a look at the following code:

```
    // chapter2/shapes/ClosedShapes.java
    Rectangle rect = new Rectangle(50,50);
    rect.setArcHeight(10);
    rect.setArcWidth(10);
    rect.setFill(Color.DARKGREY);

    Circle circle = new Circle(50);
    circle.setFill(Color.DARKGREY);

    Ellipse ellipse = new Ellipse();
    ellipse.setRadiusX(60);
    ellipse.setRadiusY(40);
    ellipse.setFill(Color.DARKGREY);

    Polygon polygon = new Polygon();
    polygon.setFill(Color.DARKGREY);
    polygon.getPoints().addAll(
               0.0, 0.0,
               50.0, 30.0,
               10.0, 60.0);

   // adding 4 shapes to the scene
   HBox hbox = new HBox(20);
```

```
hbox.setPadding(new Insets(20));
hbox.setAlignment(Pos.CENTER);
hbox.getChildren().addAll(rect, circle, ellipse, polygon);
primaryStage.setScene(new Scene(hbox, 500, 150));
```

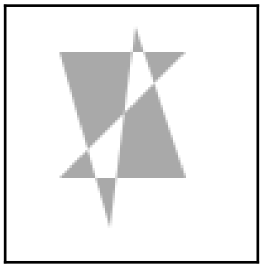
You can match all four shapes on this screenshot:



Note that you can have crossing edges for the polygon. JavaFX will do its best to determine which parts of such polygons are internal, judging by the starting point:

```
polygon.getPoints().addAll(
        0., 0.,
        50., 0.,
        0., 50.,
        50., 50.,
        30., -10.,
        20.,70.);
```

The preceding code draws the following shape:



Lines

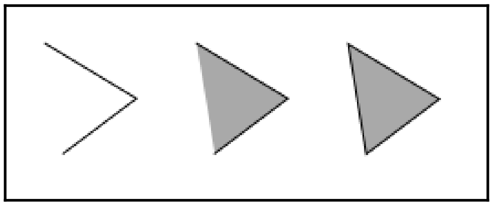
`Line` is as simple as it sounds. You write start and end coordinates, and they get connected:

```
Line line = new Line(10, 10, 100, 50);
```

`Polyline` is a set of consecutive lines. You need to provide several pairs of coordinates where the end of each line is the start of the next one. Make sure you are providing an even number of parameters:

```
// chapter2/shapes/Polylines.java
Polyline polyline = new Polyline();
polyline.getPoints().addAll(
            0.0, 0.0,
            50.0, 30.0,
            10.0, 60.0);
```

Note that despite not always having a full border, line-type shapes can have a background. If you assign it using the `setFill()` method, these shapes will use invisible edge, connecting the first and last points of the line. Here is an example of the same polylines with and without a background:



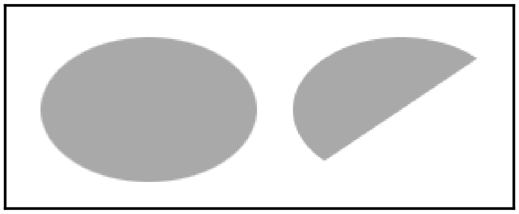
The third shape is a `Polygon` with the same points set. The only difference between `Polygon` and `Polyline` is that the former automatically adds a line between the first and the last points to create a closed figure.

Curves

`Arc` is a piece of the ellipse. It has the extra `startAngle` and `length` properties compared to `Ellipse`. Both these parameters are measured in degrees, ranging from 0 to 360.

The following is an example of an ellipse and a similar arc with a length of 180 degrees:

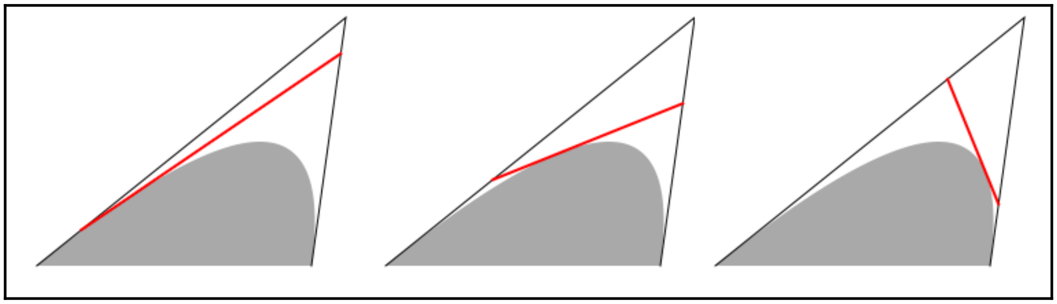
```
// chapter2/shapes/ArcAndEllipse.java
Ellipse ellipse = new Ellipse();
ellipse.setRadiusX(60);
ellipse.setRadiusY(40);
ellipse.setFill(Color.DARKGREY);

Arc arc = new Arc();
arc.setRadiusX(60);
arc.setRadiusY(40);
arc.setFill(Color.DARKGREY);
arc.setStartAngle(45);
arc.setLength(180);
```



`QuadCurve` and `CubicCurve` represent quadratic and cubic Bezier parametric curves. This is a popular method for modeling a smooth curve.

To draw a `QuadCurve`, you need to set a start point, an end point, and a control point for the curve. After that, JavaFX will draw a curve by shifting tangent with vertexes on the lines from the start point to the control point, and from the control point to the end point.

It's easier than it sounds—luckily, we have a powerful JavaFX API, so I've created a small animation, demonstrating how it works. In the next screenshot, the gray area is a `QuadCurve`, the two black lines connect the start, end, and control points, and the red line is a moving tangent:



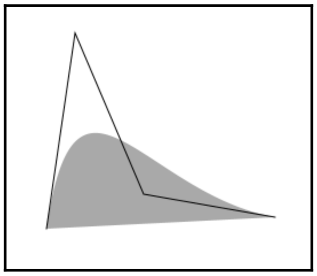
The actual sample is animated, but I'll provide only the curve code for it; see `Chapter 5`, *Animation*, for the details about the animation API. Take a look at the following code snippet:

```
// chapter2/shapes/AnimatedQuadCurve.java
QuadCurve quad = new QuadCurve();
quad.setStartX(50);
quad.setStartY(200);
quad.setEndX(250);
quad.setEndY(200);
quad.setControlX(275);
quad.setControlY(20);
quad.setFill(Color.DARKGRAY);

// two lines connecting start, end and control points
Polyline lines = new Polyline(
        quad.getStartX(), quad.getStartY(),
        quad.getControlX(), quad.getControlY(),
        quad.getEndX(), quad.getEndY());

// bold tangent line
Line tangent = new Line(quad.getStartX(), quad.getStartY(),
quad.getControlX(), quad.getControlY());
tangent.setStroke(Color.RED);
tangent.setStrokeWidth(2);
```

`CubicCurve` is a step up from `QuadCurve` and has two control points:

```
CubicCurve cubic = new CubicCurve();
cubic.setStartX(50.0);
cubic.setStartY(200.0);
cubic.setControlX1(75.0);
cubic.setControlY1(30.0);
cubic.setControlX2(135.0);
cubic.setControlY2(170.0);
cubic.setEndX(250.0);
cubic.setEndY(190.0);
cubic.setFill(Color.DARKGRAY);

Polyline lines = new Polyline(
        cubic.getStartX(), cubic.getStartY(),
        cubic.getControlX1(), cubic.getControlY1(),
        cubic.getControlX2(), cubic.getControlY2(),
        cubic.getEndX(), cubic.getEndY());
```

By having two control points you can create a shape with an uneven curve:



Paths

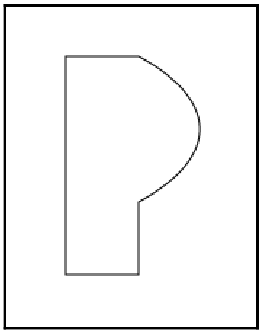
`Path` is a metashape. You can merge any combination of shape-like `PathElement` objects into one `Path` and use the result as a single shape.

The following `PathElement` classes are supported in JavaFX 9:

```
ArcTo, ClosePath, CubicCurveTo, HLineTo, LineTo, MoveTo, QuadCurveTo,
VLineTo
```

Their parameters mostly resemble corresponding shapes, except `ClosePath`, which is a marker of the path end.

Here is an example:



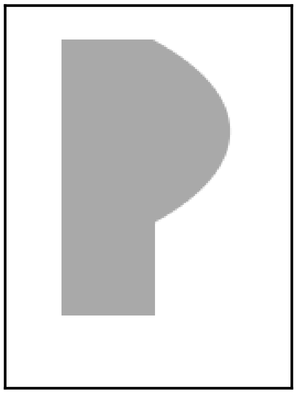
Take a look at the following code snippet:

```
// chapter2/shapes/PathDemo.java
ArcTo arcTo = new ArcTo();
arcTo.setRadiusX(250);
arcTo.setRadiusY(90);
arcTo.setX(50);
arcTo.setY(100);
arcTo.setSweepFlag(true);

Path path = new Path(
        new MoveTo(0, 0),
        new HLineTo(50),
        arcTo, // ArcTo is set separately due to its complexity
        new VLineTo(150),
        new HLineTo(0),
        new ClosePath()
);
```

`SVGPath` is similar to `Path`, but it uses text commands instead of code elements. These commands are not specific to JavaFX, they are part of Scalable Vector Graphics specification. So, you can reuse existing SVG shapes in JavaFX:

```
SVGPath svgPath = new SVGPath();
svgPath.setContent("M0,0 H50 A250,90 0 0,1 50,100 V150 H0 Z");
// SVG notation help:
// M - move, H - horizontal line, A - arc
// V - vertical line, Z - close path
svgPath.setFill(Color.DARKGREY);
```

This is the result:



By comparing Path and SVGPath, you can see a certain resemblance. It explains an odd, at first, choice of `PathElement` objects—they were mimicked after SVG ones.

Adding Text to the JavaFX scene

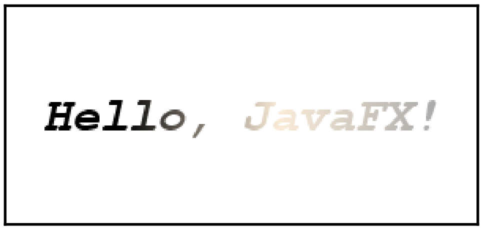
The last-but-not-least shape to cover is `Text`. `Text` draws letters in various fonts. Text weight, posture, and size are controlled by the Font API:

```
// chapter2/other/TextDemo.java
Text txt = new Text("Hello, JavaFX!");
txt.setFont(Font.font ("Courier New", FontWeight.BOLD, FontPosture.ITALIC,
20));
```

Text color is controlled through the standard `setFill()` method, which means we can use all functionality of `Paint` for text as well:

```
// gradient fill, see details few sections below
Stop[] stops = new Stop[]{new Stop(0, Color.BLACK), new Stop(1,
Color.DARKGRAY), new Stop(0.5, Color.ANTIQUEWHITE)};
LinearGradient gradient = new LinearGradient(50, 50, 250, 50, false,
CycleMethod.NO_CYCLE, stops);
txt.setFill(gradient);
```

The output will be as follows:

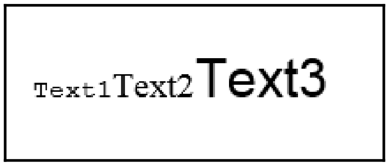


Text supports multiline strings with a `\n` delimiter.

For combining different text styles there is the `TextFlow` class, which takes multiple `Text` objects as children:

```
Text txt1 = new Text("Text1");
txt1.setFont(Font.font ("Courier New", 15));
Text txt2 = new Text("Text2");
txt2.setFont(Font.font ("Times New Roman", 20));
Text txt3 = new Text("Text3");
txt3.setFont(Font.font ("Arial", 30));
TextFlow textFlow = new TextFlow(txt1, txt2, txt3);
```

The output is as follows:



Note that `Text` and `TextFlow` support bi-directional text, which will be shown left-to-right when required.

Now we are done with the overview of shapes. Let's look into the common properties all shapes have.

Controlling Shape's color

`Shape` can have two colors: one for the interior (`setFill` method) and one for the border (`setStroke`). Color in JavaFX is handled by the Paint API, which is worth a deeper look:

Paint

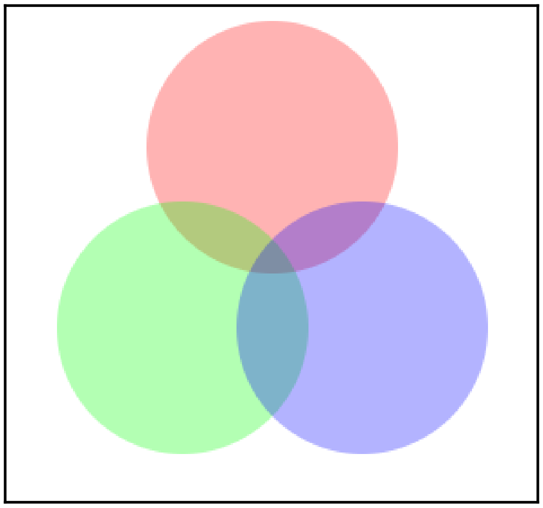
Basic paint is just a color. The `Color` class is derived from Paint and provides the following options:

- Predefined constants from `Color.BLACK` to a fancy `Color.ANTIQUEWHITE`
- Color-building methods and corresponding getters—`rgb()`, `hsb()`, `web()`
- Opacity through a parameter of the aforementioned methods
- Color-adjusting methods—`saturate()`, `darker()`, `deriveColor()`, and others

Here is a small example of semi-transparent color circles to show how colors can blend. I understand it's slightly harder to grasp in the black-and-white picture, so try it on your computer:

```
// chapter2/paint/ColorsDemo.java
Pane root = new Pane();
root.getChildren().addAll(
    // RED, opacity 0.3
    new Circle(150,80,70, Color.rgb(255, 0, 0, 0.3)),
    // GREEN, opacity 0.3
    new Circle(100,180,70, Color.hsb(120, 1.0, 1.0, 0.3)),
    // BLUE, opacity 0.3
    new Circle(200,180,70, Color.web("0x0000FF", 0.3))
);
```

The output is as follows:



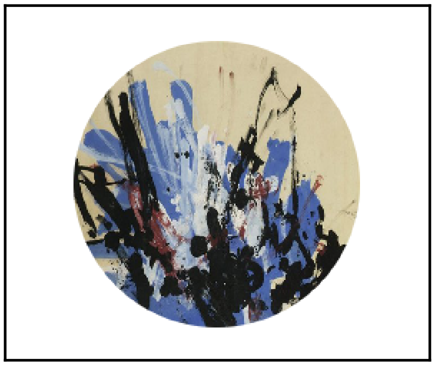
If you want to get rid of color at all, you can use the special constant `Color.TRANSPARENT`.

ImagePattern

`ImagePattern` paint allows us to have an image filling a shape:

```
// chapter2/paint/ImagePatternDemo.java
StackPane root = new StackPane();
root.getChildren().add(
     new Circle(100,
        new ImagePattern(
              new Image(
"https://upload.wikimedia.org/wikipedia/commons/3/3f/Chimpanzee_congo_paint
ing.jpg"
                   )));
```

The output is as follows:

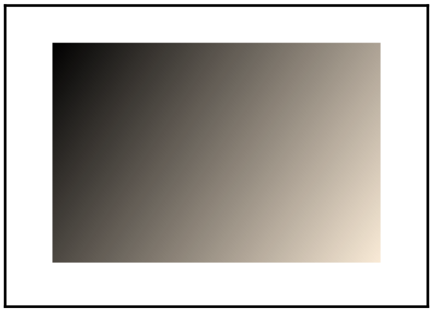


Gradients

Gradients are complex color sequences which change linearly from one to another.

In JavaFX API, each color break is defined by the `Stop` class. Also, by setting coordinates, you can adjust the gradient's direction. For example, in the following code the gradient goes from black to white and from the top-left corner to the bottom-right one:

```
// chapter2/paint/GradientDemo.java
Rectangle rect = new Rectangle(300, 200);
Stop[] stops = new Stop[]{
    new Stop(0, Color.BLACK),
    new Stop(1, Color.ANTIQUEWHITE)};
LinearGradient lg1 = new LinearGradient(0, 0, 300, 200, false,
CycleMethod.NO_CYCLE, stops);
rect.setFill(lg1);
```



Customizing lines with Stroke API

The next `Shape` element is a Stroke—the shape's border. Stroke API governs various attributes of the lines: color, width, location, and dashes.

Basic Stroke

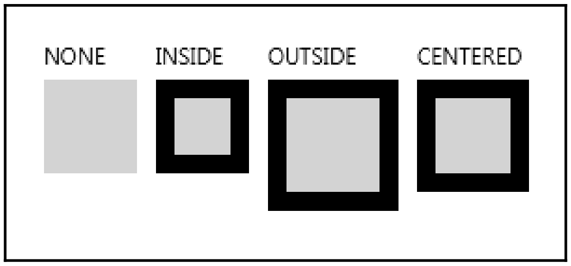
Note that the JavaFX developers decided to not introduce an extra class for strokes; all corresponding methods belong to the `Shape` class instead:

```
shape.setStroke(Color.BLACK);
shape.setStrokeWidth(10);
shape.setStrokeType(StrokeType.CENTERED);
```

The first two are pretty self-explanatory—color and width. Next up is a stroke type that controls positioning relative to shape's edge. See the following image and corresponding sample:

```
// chapter2/strokes/StrokeTypesDemo.java
hbox.getChildren().add(new VBox(5, new Text("NONE"), new Rectangle(50,50,
Color.LIGHTGRAY)));
for (StrokeType type : StrokeType.values()) {
    Rectangle rect = new Rectangle(50,50, Color.LIGHTGRAY);
    rect.setStrokeType(type);
    rect.setStroke(Color.BLACK);
    rect.setStrokeWidth(10);

    hbox.getChildren().add(new VBox(5, new Text(type.toString()), rect));
}
```

We get the following output:



TIP

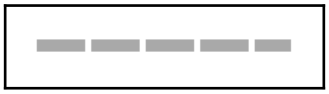
Some shapes may have no inner area at all, for example, `Line`. So, to change the color of such a shape you need to use `setStroke()` rather than `setFill()`.

Dashed lines

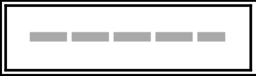
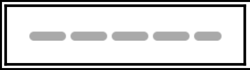
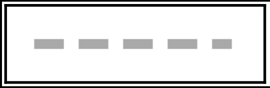
By using the `strokeDashArray` method, you can make dashed lines. Each dash's size, and the spaces between them, are set in Stroke Dash Array. Each odd position is the dash length, and each even position is the length of the space before the next dash. Refer to the following code snippet:

```
// chapter2/strokes/DashExamples.java
Line line = new Line(50, 0, 250, 0);
line.setStrokeWidth(10);
line.setStroke(Color.DARKGRAY);
line.getStrokeDashArray().addAll(30.0, 15.0);
```

This method will give us the following line:



Note that the gaps between dashes look smaller than the 15 pixels we set. This is because, by default, `StrokeLineCap` is set to `SQUARE,` which means each gap ends with half of a square shape with a size of the half of stroke width. Here is a comparison of all three line caps:

`line.setStrokeLineCap(StrokeLineCap.SQUARE);`	
`line.setStrokeLineCap(StrokeLineCap.ROUND);`	
`line.setStrokeLineCap(StrokeLineCap.BUTT);`	

And the last thing to note about dash is offset, which is a point in the dashed line that will be used as a start for drawing it:

```
line.setStrokeDashOffset(20);
```

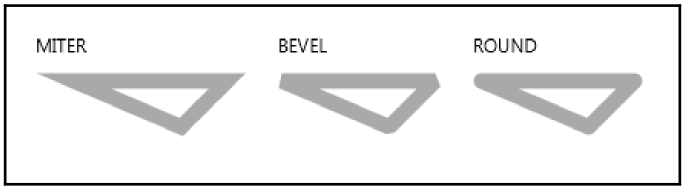
Connecting line designs using Line Join

Line Join describes how lines will look at intersections or angles. There are several options here:

- `StrokeLineJoin.MITER`: A sharp angle made from outer parts of the connecting lines
- `StrokeLineJoin.BEVEL`: A cut out angle
- `StrokeLineJoin.ROUND`: A rounded-up angle

```
// chapter2.strokes/LineJoins.java
shape.setStrokeLineJoin(StrokeLineJoin.MITER);
shape.setStrokeMiterLimit(3);
```

The output is as follows:



Working with the Shape operations

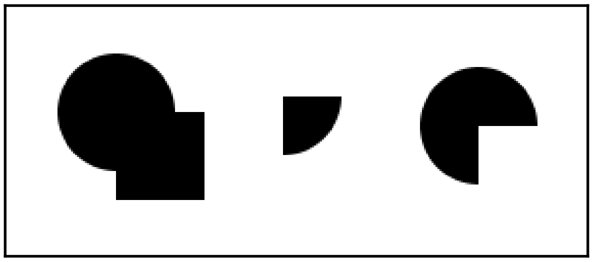
There are three operations that allow for the combining of two shapes into one:

- **Union**: Combines two shapes
- **Intersect**: Leaves only the shared part of two shapes
- **Subtract**: Removes the shared part from the first shape

These are static methods that can be applied to any two shapes:

```
Circle circle = new Circle(30);
Rectangle rect = new Rectangle(45, 45);
root.getChildren().addAll(
    Shape.union(circle, rect),
    Shape.intersect(circle, rect),
    Shape.subtract(circle, rect));
```

The output is as follows:



Transformations

JavaFX API supports basic transformations for every `Node` (and `Shape`, which extends `Node`).

Three basic transformations can be used through `Node` methods:

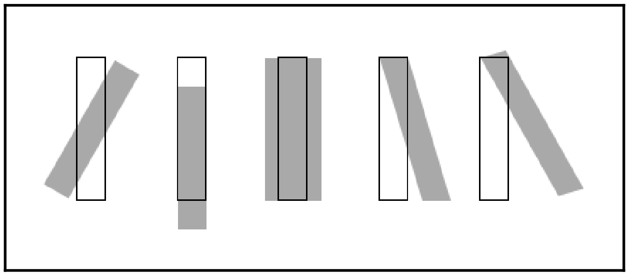
- `setRotate` (double angle): Rotates around the center of the `Node`
- `setTranslateX` (double pixels), `setTranslateY` (double pixels): Shifts the `Node` by a set amount of pixels
- `setScaleX` (double scale), `setScaleY` (double scale): Increases (or decreases) the `Node` by multiplying its horizontal or vertical dimensions by scale

For more complex transformations, the `Transform` class can be used. It allows us to work precisely with every parameter of the transformation. For example, you can concatenate two transformations into one combined and use two different nodes to save.

Note that through `Transform`, there are usually more options available. For example, the `setRotate()` method always uses the center of the shape as a pivot point, whereas for the rotate transformation you can set a deliberate pivot point inside the shape:

```
Rotate rotateTransform = new Rotate();
rotateTransform.setAngle(45);
rotateTransform.setPivotX(10);
rotateTransform.setPivotY(10);
node.getTransforms().add(rotateTransform);
```

The following demo shows rotate, translate, scale, and shear transforms. Additionally, there is a fifth transformation that is a combination of shear and rotate. On the following figure, the black border is an original rectangle and the gray shape is the transformed one:



The code for the preceding figure is a bit long because of the *double rectangle* functionality; the actual transformations are at the end. Take a look at the following code snippet:

```
// chapter2/other/Transformations.java
public class Transformations extends Application {
    // service method to make similar rectangles
    private Rectangle addRect() {
        // here we create two rectangles:
        // one for transformation
        Rectangle rect = new Rectangle(20, 100, Color.DARKGRAY);
        // and another to demonstrate original
        // untransformed rectangle bounds
        Rectangle rectOrig = new Rectangle(20, 100);
        rectOrig.setFill(Color.TRANSPARENT);
        rectOrig.setStroke(Color.BLACK);

        StackPane pane = new StackPane(rect, rectOrig);
        root.getChildren().add(pane);
        return rect;
    }
    TilePane root = new TilePane(50,50);
    @Override
    public void start(Stage primaryStage) {
        // rotate transformation
        Rectangle rect1 = addRect();
        rect1.setRotate(30);
        // translate transformation
        Rectangle rect2 = addRect();
        rect2.setTranslateY(20);
        // scale transformation
        Rectangle rect3 = addRect();
        rect3.setScaleX(2);
        // shear transformation
        Rectangle rect4 = addRect();
        rect4.getTransforms().add(new Shear(0.3, 0));
        // combining two transformations
        Rectangle rect5 = addRect();
        Transform t1 = new Shear(0.3, 0);
        Transform t2 = new Rotate(-15);
        rect5.getTransforms().add(t1.createConcatenation(t2));
        // adding all transformed rectangles to the scene
        root.setPadding(new Insets(50));
        primaryStage.setTitle("Hello World!");
        primaryStage.setScene(new Scene(root, 500, 250));
        primaryStage.show();
    }
}
```

Coordinates and bounds

Let's look into determining `Shape` and `Node` bounds in `Scene` and Scenegraph.

The simplest of all are `layoutBounds`. These rectangular bounds are used for all size and location calculations for this Node, and describe its basic shape size. They don't include any extra effects or transformations.

The next thing is `boundsInLocal`. These bounds include all information about effects. So, you can determine how large of an area is covered by your Node or Shape.

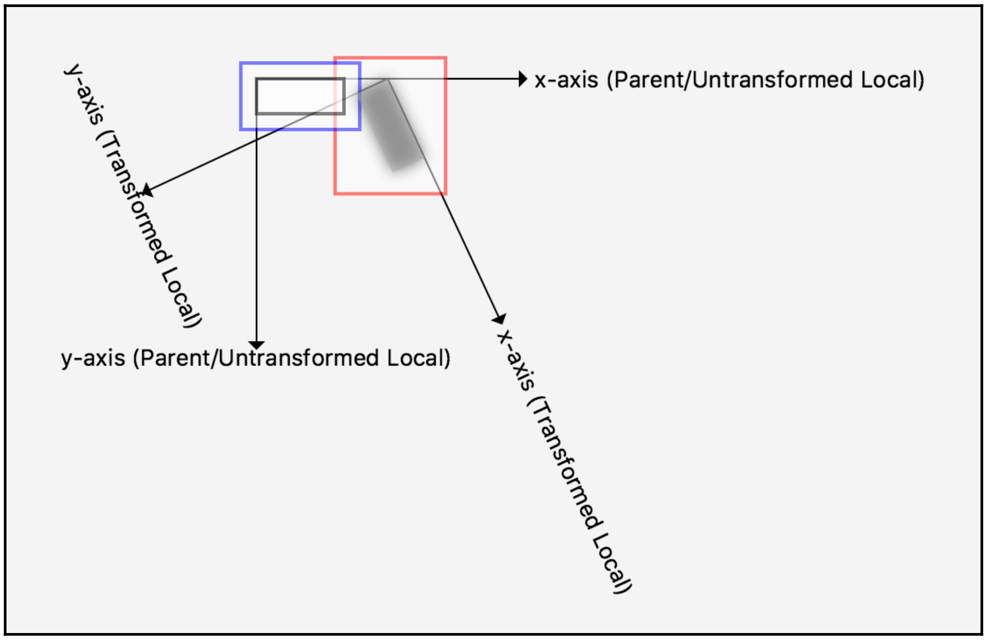
The last one is `boundsInParent`. These are bounds after all transformations as well and bounding rectangles uses their parents' coordinate system.

Working with Bounds Demo

There is a very nice public demo by *Kishori Sharan* that shows how bounds work: `http://www.java2s.com/Tutorials/Java/JavaFX_How_to/Node/Know_how_three_bounds_layoutBounds_boundsInLocal_and_boundsInParent_are_computed_for_a_node.htm`.

It uses a deprecated API, so I've fixed it and added it to our GitHub as `chapter2/other/BoundsDemo.java`.

In the following screenshot from this demo, there is a Rectangle with rotation and translate transformations, and a shadow effect:



x-axis (Parent/Untransformed Local)
y-axis (Transformed Local)
y-axis (Parent/Untransformed Local)
x-axis (Transformed Local)

The smallest rectangle represents `layoutBounds`—it's the position of original `Rectangle` before any other changes.

The bigger rectangle around it is `boundsInLocal`—the size of the `Rectangle` after the shadow effect, which spills a bit over the edges.

The blurred rotated rectangle is an actual image you will see after applying all described effects. The large square around it is `boundsInParent`—the actual size the `Rectangle` takes on the `Scene`.

I strongly suggest playing with that demo to get a grasp of how bounds work.

Using the ScenicView tool to study JavaFX scenegraph

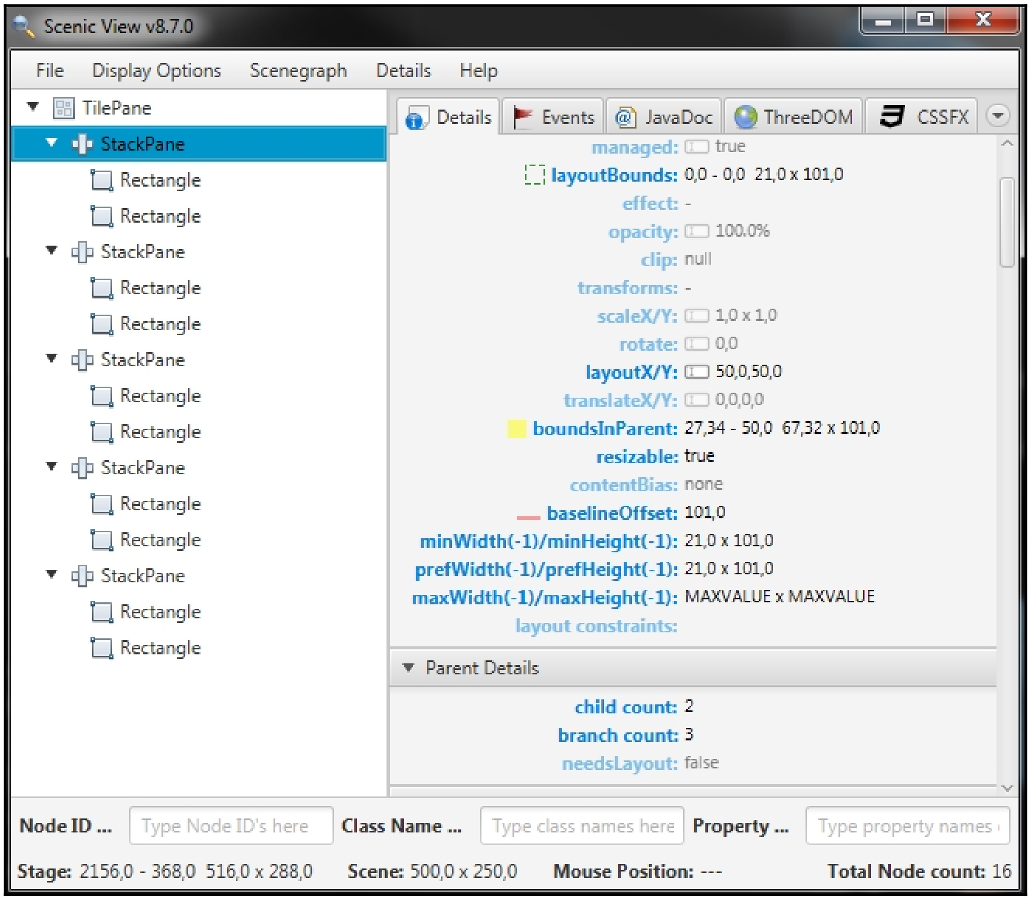
ScenicView is a great tool made and supported by one of the JavaFX developers, *Jonathan Giles*. It can be downloaded from `http://fxexperience.com/scenic-view/`.

ScenicView benefits from open JavaFX architecture and the SceneGraph paradigm. It allows the traversing SceneGraph of any JavaFX application to run on the same machine and check the properties of every Node.

Working with ScenicView is very simple:

- Run your JavaFX application
- Call `java -jar scenicView.jar`

ScenicView will automatically find your app and show its structure in a window like the following:



```
▼ TilePane
  ▼ StackPane
      Rectangle
      Rectangle
  ▼ StackPane
      Rectangle
      Rectangle
  ▼ StackPane
      Rectangle
      Rectangle
  ▼ StackPane
      Rectangle
      Rectangle
  ▼ StackPane
      Rectangle
      Rectangle
```

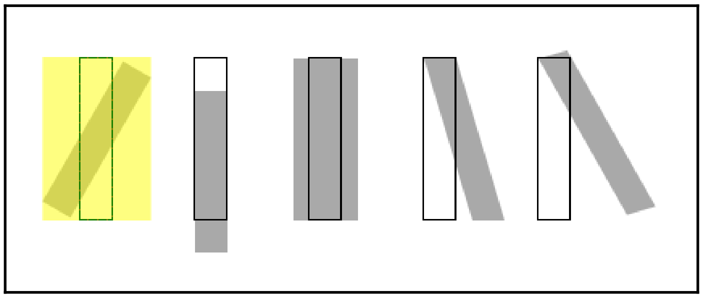
```
                      managed: true
                 layoutBounds: 0,0 - 0,0  21,0 x 101,0
                       effect: -
                      opacity: 100.0%
                         clip: null
                   transforms: -
                     scaleX/Y: 1,0 x 1,0
                       rotate: 0,0
                    layoutX/Y: 50,0,50,0
                 translateX/Y: 0,0,0,0
               boundsInParent: 27,34 - 50,0  67,32 x 101,0
                    resizable: true
                  contentBias: none
               baselineOffset: 101,0
  minWidth(-1)/minHeight(-1): 21,0 x 101,0
prefWidth(-1)/prefHeight(-1): 21,0 x 101,0
  maxWidth(-1)/maxHeight(-1): MAXVALUE x MAXVALUE
           layout constraints:
▼ Parent Details
                  child count: 2
                 branch count: 3
                  needsLayout: false
```

Node ID ... Type Node ID's here Class Name ... Type class names here Property ... Type property names

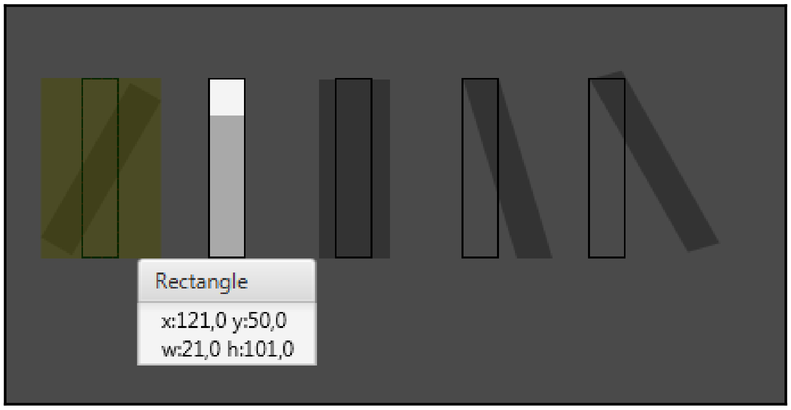
Stage: 2156,0 - 368,0 516,0 x 288,0 Scene: 500,0 x 250,0 Mouse Position: --- Total Node count: 16

On the left, you can see the scenegraph, and on the right, you can review all properties of the selected node.

Note how the `boundsInParent` and `layoutBounds` properties are additionally marked right inside your application:



The next feature is **node detection**; by clicking *Ctrl* + *Shift* + *S*, ScenicView will detect which node is located at the mouse coordinates, provide a brief description, and select it in the ScenicView on click:



This feature provides immense value while debugging complex JavaFX applications.

Basic Controls

Controls are a special subset of `Node` objects that were designed to handle user interaction. Most of them allow user input and support focus traversal.

Another difference from `Shape` is that `Control` objects are inherited not directly from `Node` but through the `Region` interface (like layout managers), which means their size and location are not fixed and can be managed by layout managers.

Button and Event Handlers

The first and most common control is button's family. `Button` has an `EventHandler` that is called when `Button` is clicked (or fired). All code in the event handler is always run on JavaFX Application Thread:

```
// chapter2/other/ButtonDemo.java
Button btn = new Button();
btn.setText("Say 'Hello World'");
btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
});
```

In addition to text, you can use any `Node` inside a button:

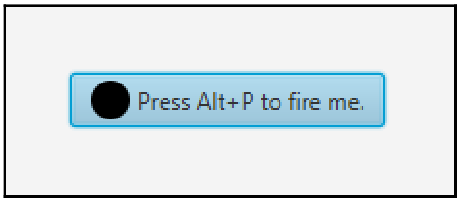
```
Button btn = new Button();
btn.setText("Say 'Hello World'");
btn.setGraphic(new Circle(10));
```

`Button` can be assigned the roles of default button or cancel button, responding to *Enter* or *Esc* correspondingly:

```
btn.setDefaultButton(true);
//or
btn.setCancelButton(true);
```

`Button` can be easily assigned with a mnemonic *(Alt* + letter) by using the _ sign:

```
btn.setText("_Press Alt+P to fire me."):
```



Note you can disable this behavior by calling `btn.setMnemonicParsing(false);`.

There are also 4 more classes that share a common ancestor `(ButtonBase)` and most functionality with `Button`:

- `ToggleButton` saves state clicked/unclicked and changes visuals accordingly. Its state can be retrieved by calling the `isSelected()` method.
- `CheckBox` has the same behavior as `ToggleButton` but different visuals and API. `CheckBox` state is controlled by the `getState()` method.
- `Hyperlink` is a button with no extra decorations, which also remembers if it was already clicked.
- `MenuButton` is a part of the Menu API that allows you to create user and context menus.

Size of the Controls

`Control` (or any `Region` derived class) has three size limitations:

- `minSize`: Minimal size
- `prefSize`: Preferred size
- `maxSize`: Maximal size

Layout managers, while handling their children, try to use their `prefSize`. If it's not possible, they are obliged to not shrink any child smaller than their `minSize` and not let them grow bigger than their `maxSize`.

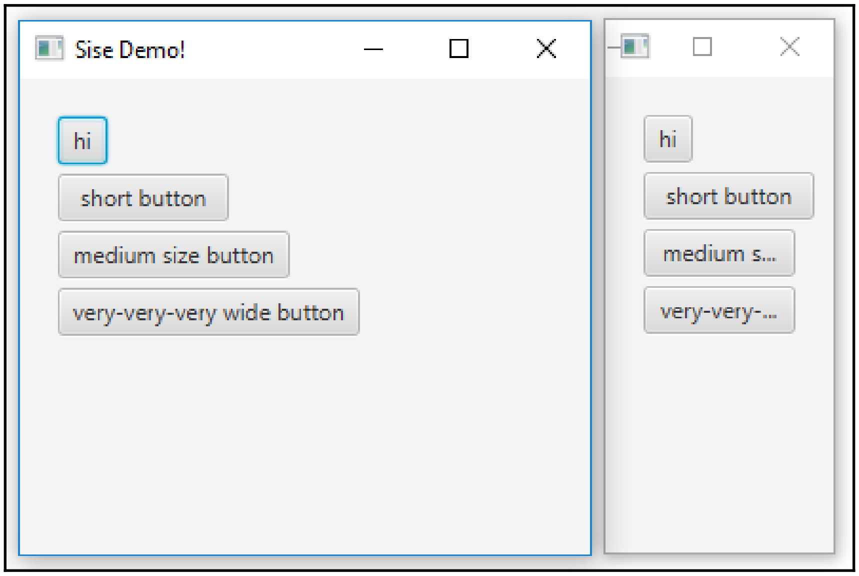
Controls usually set their `prefSize` by themselves based on their content.

For example, button prefSize is based on the length of the text inside, `minSize` is just enough to show ellipsis instead of text, and the maxSize of `Button` is similar to `prefSize`.

Thus, you don't need to care about a size of the `Button` in the following sample:

```
VBox root = new VBox(5);
root.setPadding(new Insets(20));
Button btnShort = new Button("short");
btnShort.setMinWidth(50);
root.getChildren().addAll(
        new Button("hi"),
        btnShort,
        new Button("mediocre"),
        new Button("wide-wide-wide")
);
```

And, if you try to resize the window, the button `btnShort` keeps its width, as shown in the following screenshots:



If you want your `Control` or other `Region` to always have a fixed size, you need to set all three sizes to the same value.

TIP If you want to use Control's own prefSize to be used as max or min you can use a special constant—`Control.USE_PREF_SIZE`. For example, `btn.setMinHeight(Control.USE_PREF_SIZE);`.

Clock demo

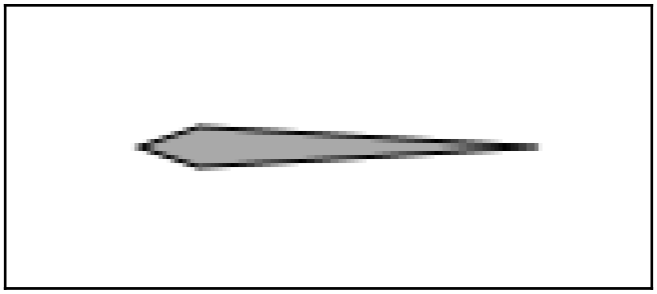
Let's apply some of the stuff we learned in this Chapter to the Clock demo we started in the previous chapter.

The digital clock is not as fun as an analog one, so let's add hands to it.

We need three hands—hours, minutes, and seconds. We'll use a simple line for the seconds hand and slightly more complex path shapes for the hour and minute ones. For example, here is the path for the minute hand:

```
Path minuteHand = new Path(
                new MoveTo(0, 0),
                new LineTo(15, -5),
                new LineTo(100,0),
                new LineTo(15,5),
                new ClosePath());
minuteHand.setFill(Color.DARKGRAY);
```

This code gives us not the prettiest but a conveniently simple hand. We'll work on making it nicer in following chapters:



To show time, the hand has to rotate. We'll use the `Rotate` transformation. We need to rotate the hand around the leftmost point, not the center, so we set the pivot point to zero coordinates:

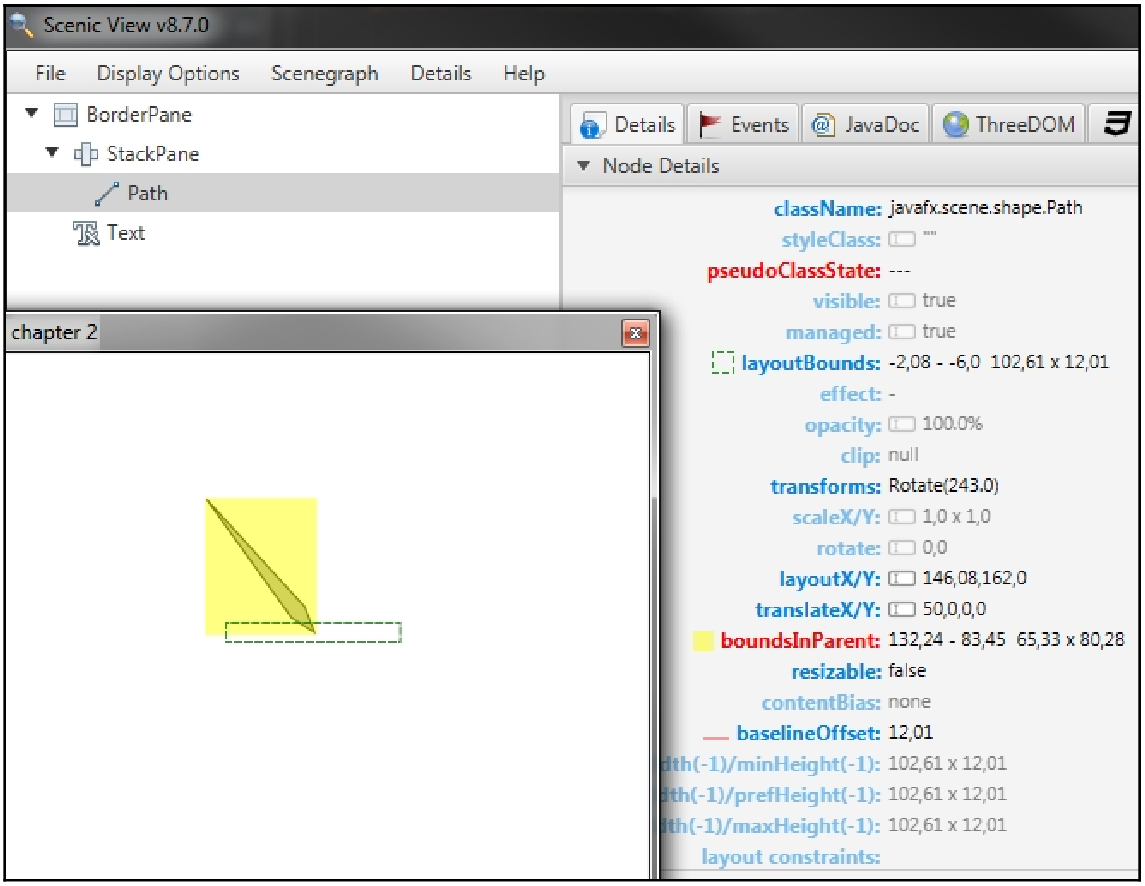
```
Rotate rotateMinutesHand = new Rotate();
rotateMinutesHand.setPivotX(0);
rotateMinutesHand.setPivotY(0);
minuteHand.getTransforms().add(rotateMinutesHand);
```

Now, we can control the time set by modifying the angle for this `Rotate` transformation.

Also, our hand is inside our layout manager, which tries to center it around `Path` central point as well. But, we want to have a center in the local coordinates (0,0). To achieve that, we will translate our hand left by half of its actual size:

```
minuteHand.setTranslateX( minuteHand.getBoundsInLocal().getWidth()/2 );
```

For a better understanding, take a look at the ScenicView screenshot for this code—the dashed box is `layoutBounds` and the colored rectangle is `boundsInParent`, which changes after every rotation:



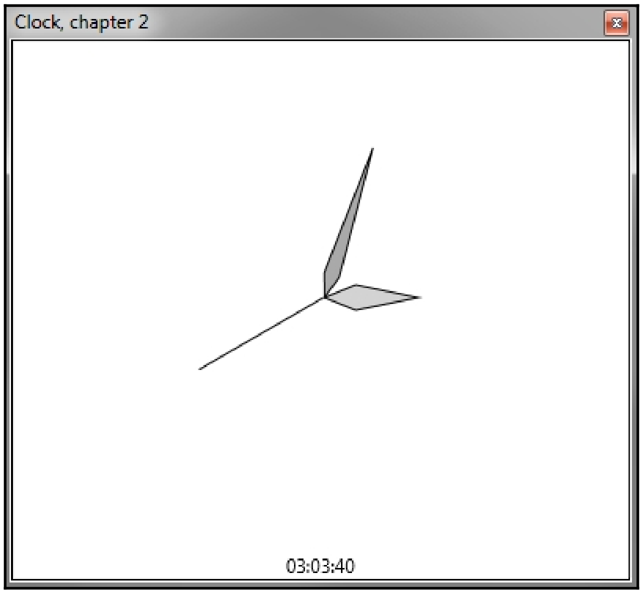
className: javafx.scene.shape.Path
styleClass: ""
pseudoClassState: ---
visible: true
managed: true
layoutBounds: -2,08 - -6,0 102,61 x 12,01
effect: -
opacity: 100.0%
clip: null
transforms: Rotate(243.0)
scaleX/Y: 1,0 x 1,0
rotate: 0,0
layoutX/Y: 146,08,162,0
translateX/Y: 50,0,0,0
boundsInParent: 132,24 - 83,45 65,33 x 80,28
resizable: false
contentBias: none
baselineOffset: 12,01
dth(-1)/minHeight(-1): 102,61 x 12,01
dth(-1)/prefHeight(-1): 102,61 x 12,01
dth(-1)/maxHeight(-1): 102,61 x 12,01
layout constraints:

Here is the full code:

```
public class ClockTwo extends Application {
    private final Text txtTime = new Text();

    private Rotate rotateSecondHand = new Rotate(0,0,0);
    private Rotate rotateMinuteHand = new Rotate(0,0,0);
    private Rotate rotateHourHand = new Rotate(0,0,0);
    private Thread timer = new Thread(() -> {
        SimpleDateFormat dt = new SimpleDateFormat("hh:mm:ss");
        Date now = new Date();
        String time = dt.format(now);
        Platform.runLater(()-> {
                // updating live UI object requires JavaFX App Thread
                rotateSecondHand.setAngle(now.getSeconds() * 6 - 90);
                rotateMinuteHand.setAngle(now.getMinutes()* 6 - 90);
                rotateHourHand.setAngle(now.getHours()* 30 - 90);
                txtTime.setText(time);
        });
        try {
                // running "long" operation not on UI thread
             Thread.sleep(1000);
        } catch (InterruptedException ex) {
        }
    });

    @Override
    public void start(Stage stage) {
        // create minutes hand
        Path minuteHand = new Path(
                new MoveTo(0, 0),
                new LineTo(15, -5),
                new LineTo(100,0),
                new LineTo(15,5),
                new ClosePath());
        minuteHand.setFill(Color.DARKGRAY);
        minuteHand.getTransforms().add(rotateMinuteHand);
minuteHand.setTranslateX(minuteHand.getBoundsInLocal().getWidth()/2);
        // create second hand
        Line secondHand = new Line(0,0, 90, 0);
        secondHand.getTransforms().add(rotateSecondHand);
secondHand.setTranslateX(secondHand.getBoundsInLocal().getWidth()/2);
        // create hour hand
        Path hourHand = new Path(
                new MoveTo(0, 0),
                new LineTo(20, -8),
                new LineTo(60,0),
                new LineTo(20,8),
                new ClosePath());
        hourHand.setFill(Color.LIGHTGRAY);
        hourHand.getTransforms().add(rotateHourHand);
        hourHand.setTranslateX(hourHand.getBoundsInLocal().getWidth()/2);
        BorderPane root = new BorderPane();
        root.setCenter(new StackPane(minuteHand, hourHand, secondHand));
        root.setBottom(txtTime);
        BorderPane.setAlignment(txtTime, Pos.CENTER);
        Scene scene = new Scene(root, 400, 350);
        stage.initStyle(StageStyle.UTILITY);
        stage.setScene(scene);
```

```
            stage.setTitle("Clock, chapter 2");
            timer.setDaemon(true);
            timer.start();
            stage.show();
            System.out.println(minuteHand.getBoundsInLocal().getWidth());
        }
        public static void main(String[] args) {
            launch(args);
        }
    }
```

The clock will appear as follows:



Summary

In this chapter, we've studied a lot of different building blocks for JavaFX applications: shapes, text, and simple controls. Also, we looked at various ways to customize them. With these tools, you can already build a visually rich JavaFX UI.

The only problem is that it will be static, which is not enough for a modern application. In the next two chapters, we will learn about Binding and Animation, which will allow us to develop more dynamic applications.

3 Connecting Pieces – Binding

Dynamically sharing information between UI elements is a big part of an application. We are all used to handling dynamic updates in Java through established means such as listeners or event queues, but JavaFX brought a new option, the Binding API, to directly bind components' properties together.

In this chapter, we will study how the JavaFX Binding API greatly simplifies communication between any JavaFX components—nodes, controls, FX collections, animation, media, and more.

In this chapter, we'll cover the following topics:

- Binding basics and its benefits over regular listeners
- Binding operations
- User-defined or custom bindings
- Binding collections

Working with the Property API

In addition to the regular getters and setters, JavaFX provides a Property API to almost all its classes' fields.

For example, there are the following methods to work with the `Stage` title:

```
String getTitle();              //getter
void setTitle(String title);    //setter
StringProperty titleProperty(); //property access
```

Technically, getters and setters are not required, and the `Property` value can be used all the time.

The `Property` class has the following two important APIs—**Observable** and **Binding**.

Using the Observable API

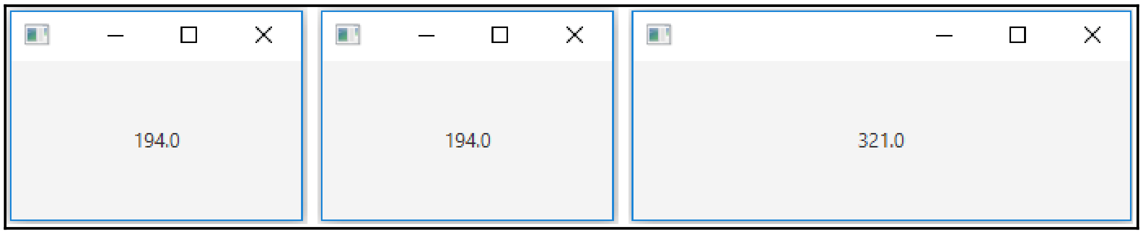
An `Observable` is an interface which allows you to subscribe to the change event of the corresponding property. All JavaFX properties are observable, thus you can track changes of every small parameter of every JavaFX node using the Observable API.

Let's take a look at the next simple example, which dynamically updates the width of the JavaFX application's window.

The window, represented by the `Stage` class, has a property called `widthProperty`, which we can listen to:

```
// chapter3/basics/WidthObservable.java
public void start(Stage stage) {
   Label lblWidth = new Label();
   // Note, we are not using any binding yet
   stage.widthProperty().addListener(new ChangeListener<Number>() {
     @Override
     public void changed(ObservableValue<? extends Number> o, Number
oldVal, Number newVal) {
                lblWidth.setText(newVal.toString());
            }
   });

   stage.setScene(new Scene(new StackPane(lblWidth), 200, 150));
   stage.show();
}
```

Here is a screenshot:



Every property in JavaFX is implementing the similar `ObservableValue` interface and can be listened to for changes.

TIP

Not clearing listeners properly is a common reason for Java memory leaks. To address that, JavaFX supports weak-reference for all listeners, so if you are working on a large application, consider using `WeakListener` descendants and `WeakEventHandler`.

Besides plain values, you can observe collection types through the following interfaces extending `Observable`:

- `ObservableList`: `java.util.List` with the Observable API
- `ObservableMap`: `java.util.Map` with the Observable API
- `ObservableArray`: A class that encapsulates a resizable array and adds the Observable API

Overall, there are over 100 classes implementing `Observable` in JavaFX.

Introducing the Binding API

The Binding API allows you to simplify listeners to just one line of code:

```
// chapter3/basics/WidthBinding.java
public void start(Stage stage) {
    Label lblWidth = new Label();

    lblWidth.textProperty().bind(stage.widthProperty().asString());

    stage.setScene(new Scene(new StackPane(lblWidth), 200, 150));
    stage.show();
}
```

Let's take a closer look at the binding code:

```
lblWidth               // object we want to change
  .textProperty()      // property of the object to be changed
  .bind(               // bind() call
     stage             // object we want to monitor
     .widthProperty()  // property of the monitored object we want to track
     .asString());     // assigning Double value to a String property
```

The `bind()` method comes from the base `Property` interface, which has the following API methods:

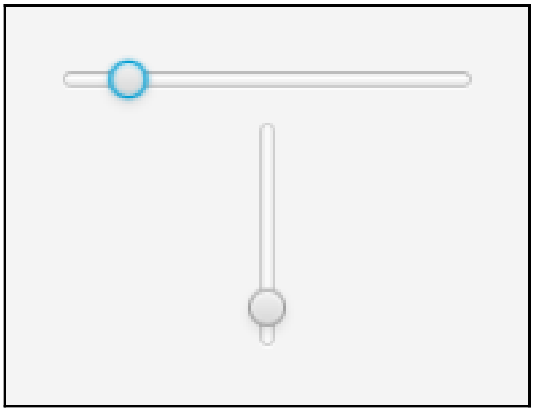
- `bind (ObservableValue<? extends T> observable)`: Binds a `Property` to `Observable`.
- `unbind ()`: Stops binding.
- `boolean isBound ()`: Checks whether a `Property` is already bound. It's important as only one binding connection can be set for a `Property`, although this connection can be made as complex as you need, as will be shown later in the chapter.
- `bindBidirectional (Property<T> other)`: Ties two properties together in both directions.
- `unbindBidirectional (Property<T> other)`: Stops the bidirectional binding.

Bidirectional binding connects two properties, essentially making them always have the same value.

Consider the following example:

```
// chapter3/basics/BidirectionalBindingDemo.java
public void start(Stage stage) {
    Slider s1 = new Slider(0, 100, 40);
    Slider s2 = new Slider(0, 100, 40);
    s2.setOrientation(Orientation.VERTICAL);
    s1.valueProperty().bindBidirectional(s2.valueProperty());

    VBox root = new VBox(5, s1, s2);
    root.setAlignment(Pos.CENTER);
    root.setPadding(new Insets(20));

    stage.setScene(new Scene(root, 200, 150));
    stage.show();
}
```

The preceding code creates two sliders. You can drag either of them and the other one will mimic the movement:



Rules of binding

Although binding looks very simple to use, there are several rules that you need to be aware of while using binding:

- Read-only properties can't be bound
- Only one bind can be active for a property, but several properties can be bound to another one
- Binding and setters do not work together
- Bidirectional bindings are less strict

Let's look at each of these points more closely in the next sections.

Read-only properties

All properties can be observed (or be used as a parameter for bind), but bind can't be called for some of them. The exception is read-only properties, which can't be changed and can't be bound.

All read-only properties implement `ReadOnlyProperty`, or they can be recognized by the class name starting from `ReadOnly` words. For example, `widthProperty` in the first example has the `ReadOnlyDoubleProperty` type.

Binding is a one-to-many relation

Any property can be observed as many times as you want:

```
Label lblWidth = new Label();
Label lblWidth2 = new Label();

lblWidth.textProperty().bind(stage.widthProperty().asString());
lblWidth2.textProperty().bind(stage.widthProperty().asString());
```

But, any consecutive call for the `bind()` method will override the previous one:

```
lblWidth.textProperty().bind(stage.widthProperty().asString());
lblWidth.textProperty().bind(stage.heightProperty().asString());
// now lblWidth listens for height and ignores width
```

Don't be disappointed by this rule. You can call `bind()` effectively only once, right. But, the parameter of `bind()` can be much more complex than just one property. We'll discuss this in more detail in the *Using binding operations* section.

Binding blocks setters

Once a property is bound, you can't set it directly. The following code will throw `RuntimeException`: `Label.text : A bound value cannot be set`. Take a look at the following code snippet:

```
lblWidth.textProperty().bind(stage.widthProperty().asString());
lblWidth.setText("hi"); // exception
```

If you want to do it, you need to call `unbind()` first.

Bidirectional binding

Bidirectional binding, meanwhile, has much fewer restrictions, because it ties properties harder, so there is no need to extra control them.

You can use setters:

```
label2.textProperty().bindBidirectional(label.textProperty());
label2.setText("hi");
```

You can combine bidirectional and regular bindings:

```
lblWidth.textProperty().bind(stage.widthProperty().asString());
lblWidth2.textProperty().bindBidirectional(lblWidth.textProperty());
```

And, you can bind one property bidirectionally several times:

```
label.textProperty().bindBidirectional(label2.textProperty());
label.textProperty().bindBidirectional(label3.textProperty());
label.setText("hi"); // or label2.setText("hi"); or label3.setText("hi");
                     // any of them will update all three labels at once
```

Note that the `isBound()` method works only for one-directional binding. Unfortunately, there is no API to check for bidirectional binding. Also, don't mix `unbind()`, which works only for regular bindings, and `unbindBidirectional()`, for bidirectional ones.

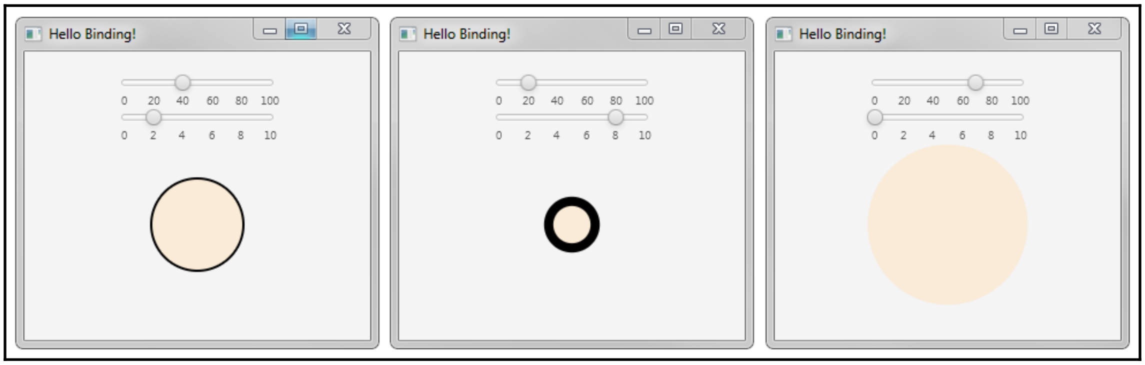
Using binding for visual help

While studying JavaFX classes, you can conveniently check how properties work using binding to a slider or checkbox—add them to your program, bind to the property you are interested in, and observe how different values of this property affect the corresponding node.

For example, let's look at this circle and bind a few sliders to its stroke and radius:

```
// chapter3/basics/CirclePropertiesDemo.java
public void start(Stage primaryStage) {
        Circle circle = new Circle(150, 150, 40, Color.ANTIQUEWHITE);
        circle.setStroke(Color.BLACK);

        Slider sliderRadius = new Slider(0, 100, 40);
        sliderRadius.relocate(80, 20);
        sliderRadius.setShowTickLabels(true);
        sliderRadius.setMajorTickUnit(20);
        circle.radiusProperty()
             .bind(sliderRadius.valueProperty());
        Slider sliderStrokeWidth = new Slider(0, 10, 2);
        sliderStrokeWidth.setShowTickLabels(true);
        sliderStrokeWidth.setMajorTickUnit(2);
        sliderStrokeWidth.relocate(80, 50);
        circle.strokeWidthProperty()
             .bind(sliderStrokeWidth.valueProperty());
        Pane root = new Pane();
        root.getChildren().addAll(sliderRadius, circle, sliderStrokeWidth);
        primaryStage.setTitle("Hello Binding!");
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
```

Here are the examples of this app with sliders moved to different positions:



The role of listeners

Note that binding is not a complete replacement for listeners. `EventListener` objects are still required to program a reaction to the user actions or system events. Binding plays its role, then you need to update one entity based on the state of another.

So, the rule of thumb is, if you are writing `ChangeListener` or `InvalidationListener`, consider using binding instead.

Using binding operations

Directly connecting properties is not always enough. The Binding API can build complex dependencies between properties.

String operations

Concatenation is the most common string operation. It's called by one property of the `String` type and takes another property of any type or a constant that will be used as a `String`, as shown in the following code snippet:

```
// chapter3/operations/ConcatBinding.java
label.textProperty().bind(
    stage.widthProperty().asString() // property one
    .concat(" : ")                   // concatting with a string constant
    .concat(stage.heightProperty())  // concatting with a property 2
);
```

You don't need to call `asString()` for a `concat()` parameter, as it's done by JavaFX.

Note two very important concepts here:

- Binding operations allow us to chain them one after another (as in the Builder pattern, or in the Java8 Streams API)
- We've bound one property to the two others here

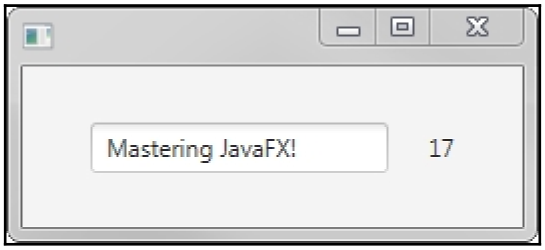
And, we are not limited by two properties: you can build a binding or almost any complexity by chaining method calls.

For strings, most of the binding utility is available in the `StringExpression` class, which allows using string-relevant functions as bindings. `StringExpression` implements `ObservableStringValue` and is an ancestor for `StringBinding`. So, any string-related binding will have it in a chain of inheritance.

For example, if you want to bind to string length, you can use the `StringExpression` method `length()`, which returns an `IntegerBinding` type. Here is an example:

```
// chapter3/operations/StringLengthBind.java
public void start(Stage stage) {
   TextField textField = new TextField();
   Label lblLength = new Label();

   lblLength.textProperty().bind(
      textField.textProperty()
         .length()   // this length returns IntegerBinding, not just an
integer
         .asString() // so you can keep observing it and use binding
methods
      );

   HBox root = new HBox(20, textField, lblLength);
   root.setAlignment(Pos.CENTER);
   stage.setScene(new Scene(root, 200, 100));
   stage.show();
}
```

And, this code gives us an automatic length counter for a text field, as shown in the following screenshot:


Mastering JavaFX!
17

For even more flexibility, you can use the `Bindings` utility class, which holds quite a few utility methods. Even the preceding `concat()` method was added only for convenience and is implemented through calls to `Bindings`.

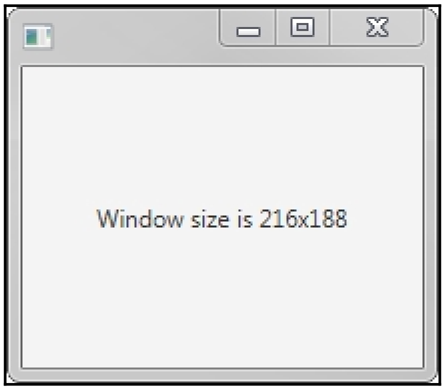
For example, it would be convenient to have the word `"Count:"` shown in front of the number. But, how one can add a String before a bindable object? There is nothing to call `concat()` from yet. `Bindings` comes to help here with the static `concat()` method:

```
lblLength.textProperty().bind(
        Bindings.concat("Count: ", textField.textProperty().length()));
```

In our first width example, we can use the `format()` output method, reusing syntax from the Java, `java.util.Formatter`:

```
Bindings.format("Window size is %1$.0fx%2$.0f", stage.widthProperty(),
stage.heightProperty());
```

The preceding code produces the following window, which is automatically updated on resizing:


Window size is 216x188

Arithmetic operations

JavaFX binding supports basic arithmetic operations such as add, subtract, divide, multiply, and negate.

You can find them in binding/expression classes related to `Number`:

```
DoubleExpression, FloatExpression, IntegerExpression, LongExpression
```

The base class is `NumberExpression`. There are several overloaded methods to support work with both `Observable` objects and constants.

The very basic add operation is defined in `NumberExpression` and takes `NumberExpression` as a parameter. Here comes a question of the exact type of the result such an operation will produce. The rule is the same as in Java—*data type with most precision and capacity takes precedence:*

```
Double > Float > Long > Integer
```

Both operands are checked according to this chain and, if any of them hits, it means the result will be of that type.

For example, the following operation will produce `DoubleBinding`:

```
IntegerProperty intProp = new SimpleIntegerProperty(5);
DoubleProperty doubleProp = new SimpleDoubleProperty(1.5);
NumberBinding addBinding = intProp.add(doubleProp); // DoubleBinding
```

Fortunately, almost all number properties in JavaFX are `DoubleProperty`, so you don't need to care that much about the correct type.

The `Bindings` utility class adds also the `min()` and `max()` binding methods, which work both for properties and for constants.

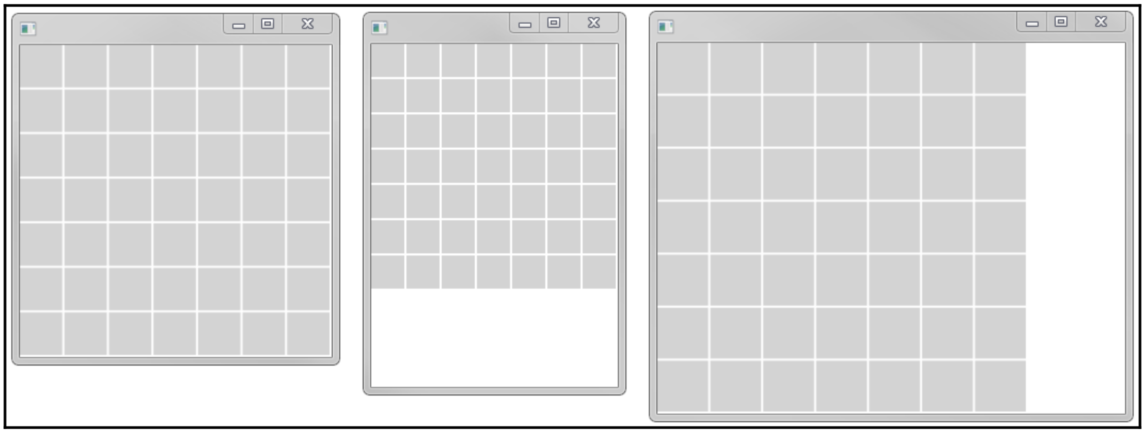
Let's look at a more complex example here. The following code will create a 7x7 grid of rectangles that will try to fill the maximum possible area of a stage during resizing. We need three bindings for that:

- One to calculate which side of the window is smaller now
- One, applied to all rectangles, to change the rectangles' sizes accordingly
- One, again applied to all rectangles, to choose the rectangles' positions correctly:

```
// chapter3/operations/Rectangles.java
public void start(Stage stage) {
    Pane root = new Pane();

    final int count = 7; //number of rectangles

    // this is binding to calculate rectangle size
    // based on their count and minimal of scene width and height
    NumberBinding minSide = Bindings
            .min(root.heightProperty(), root.widthProperty())
            .divide(count);
    for (int x = 0; x < count; x++) {
        for (int y = 0; y < count; y++) {
            Rectangle rectangle = new Rectangle(0, 0, Color.LIGHTGRAY);

            // binding rectangle location to it's side size
            rectangle.xProperty().bind(minSide.multiply(x));
            rectangle.yProperty().bind(minSide.multiply(y));

            // binding rectangle's width and height
            rectangle.heightProperty().bind(minSide.subtract(2));
            rectangle.widthProperty().bind(rectangle.heightProperty());

            root.getChildren().add(rectangle);
        }
    }
```

```
        stage.setScene(new Scene(root, 500, 500));
        stage.show();
    }
```

Here are a few examples of this application's window in different sizes:



Try to run this application and check the logic behind it to get a better grasp on bindings.

Boolean operations

Boolean operations allow you to incorporate conditions directly into the binding.

First of all, there are `BooleanExpression`/`BooleanBinding` classes to represent observable boolean values. Then, to make if-then-else condition handlers, there are special methods in the Binding API:

```
    when( CONDITION ) -> then( A ) -> otherwise( B )
```

`Here`, `when`, `then`, and `otherwise` are actual method names, not pseudocode. This construction means the following:

- Listen to `A` and `B`
- If `A` is changed and the `CONDITION` is true, update the bound value with `A`
- If `B` is changed and the `CONDITION` is false, update the bound value with `B`

As an example, let's forget for a minute we have the `min()` method from the rectangles grid example and re-implement it using boolean operations:

```
// original code
NumberBinding min = Bindings.min(root.heightProperty(),
root.widthProperty());

// boolean bindings version
NumberBinding minSide = Bindings
   .when( root.heightProperty().lessThan(root.widthProperty()) ) //
CONDITION
         .then( root.heightProperty() )      // option A
         .otherwise( root.widthProperty() ); // option B
```

Note the `lessThan()` call in the condition. It's one of the great range-of-conditional Boolean operations provided by JavaFX. Here is a self-explanatory list of them:

```
equal()
notEqual()
equalIgnoreCase()
notEqualIgnoreCase()

greaterThan()
greaterThanOrEqual()
lessThan()
lessThanOrEqual()

isEmpty()
isNotEmpty()
isNull()
isNotNull()

not()
or()
and()
```

I consciously dropped parameter types here as there is a huge list of overloaded methods for each possible type, for your convenience. All these methods are accessible from the `Bindings` class and the most common ones from `Expression` classes of corresponding types.

Working with bidirectional binding and converters

Bidirectional binding is very convenient for properties of the same type, but you are not restricted by that. You can bind bidirectionally properties of `String` and any other type by providing a corresponding converter.

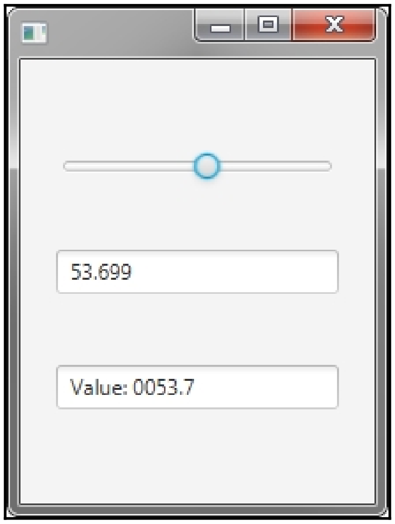
There is a large list of predefined converters for the common types, for example, binding a number and a string:

```
Bindings.bindBidirectional(
     textProperty, numberProperty, new NumberStringConverter());
```

This converter also supports `Locale` and patterns from the `java.text.DecimalFormat` class. See the following application as an example:

```
// chapter3/other/BidiConverters.java
public void start(Stage stage) {
    Slider s1 = new Slider(0, 100, 40);
    TextField tf1 = new TextField();
    tf1.textProperty().bindBidirectional(s1.valueProperty(),
                            new NumberStringConverter());
    TextField tf2 = new TextField();
    tf2.textProperty().bindBidirectional(s1.valueProperty(),
                            new NumberStringConverter(Locale.US, "Value:
0000.#"));
    VBox root = new VBox(40, s1, tf1, tf2);
    root.setAlignment(Pos.CENTER);
    root.setPadding(new Insets(20));

    stage.setScene(new Scene(root, 200, 250));
    stage.show();
}
```

This code produces the following application, where you can edit any field or move the slider to update the same value shown in three different formats. Refer to the following screenshot:


53.699

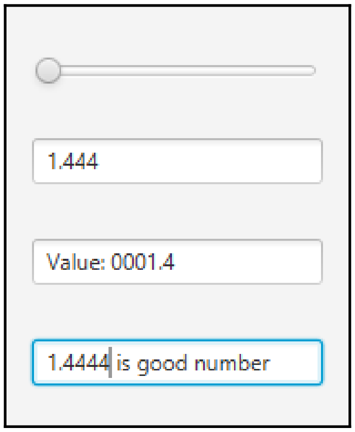
Value: 0053.7

Note that the converter will try to do its best to convert back and forth, but if you put the wrong value in the text field, nothing will be updated and `ParseException` will be thrown on the event thread (unfortunately, you can't catch or handle it there using the public API).

Also, you can provide your own converter by implementing the following very simple interface:

```
        // adding one more field to the previous example
        TextField tf3 = new TextField();
        tf3.textProperty().bindBidirectional(s1.valueProperty(), new
StringConverter<Number>() {
            @Override
            public String toString(Number number) {
                return number + " is good number";
            }

            @Override
            public Number fromString(String string) {
                return Double.valueOf(string.split(" ")[0]);
            }
        });
```

This code adds one more `TextField` to our previous example, which is governed by our new custom converter:



Creating custom bindings

If you have a complex logic, you can create your own binding by either extending the corresponding abstract base class—`DoubleBinding`, `StringBinding`, `ObjectBinding`—there are several options for different types; or by using a utility method from the `Bindings` class.

Implementing base binding classes

The concept of binding is simple:

- Add a listener to change events you want to track
- Compute the value you want to have

Correspondingly, in the following example, we call `bind()` for the desired property and override `computeValue()` method:

```
public void start(Stage primaryStage) {
    Button btn = new Button();
    btn.setText("Click me");

    StackPane root = new StackPane();
    root.setBackground(Background.EMPTY);
```

```
        root.getChildren().add(btn);
        Scene scene = new Scene(root, 300, 250);

        ObjectBinding<Paint> objectBinding = new ObjectBinding<Paint>() {
            {
                bind(btn.pressedProperty());
            }
            @Override
            protected Paint computeValue() {
                return btn.isPressed() ? Color.RED : Color.GREEN;
            }
        };
        scene.fillProperty().bind(objectBinding);

        primaryStage.setTitle("Hello World!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

In this example, the scene background color changes if the button is pressed.

Bindings helper function

The `Bindings` class has a helper function for custom bindings:

```
// chapter3.other.CreateBinding.java
Bindings.createObjectBinding(
    ()-> btn.isPressed() ? Color.RED : Color.GREEN, // computeValue logic
    btn.pressedProperty()                            // list of observed
values
);
```

And, as a small reminder, let's rewrite this binding once again using the Boolean binding operations we reviewed in the preceding code:

```
scene.fillProperty().bind(
        Bindings.when(btn.pressedProperty())
           .then(Color.RED)
           .otherwise(Color.GREEN));
```

Understanding binding collections

JavaFX 10 supports three main collection types—`List`, `Map`, and `Set`.

You can make any Java collection observable using `FXCollections` helper methods:

```
List observableList = FXCollections.observableArrayList(collection);
```

This makes a collection trigger a listener on every addition, removal, or change in the elements order.

The `FXCollections` class mimics `java.util.Collections` a lot, providing `observable` counterparts for `java.util.Collections` methods.

Note you can go even deeper, observing not only a collection but the collection elements' changes as well, using the following method:

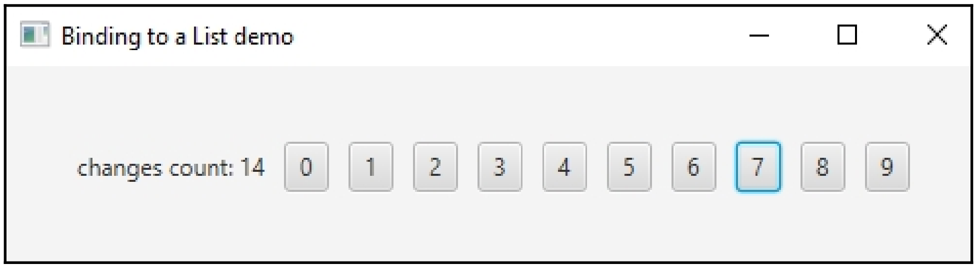
```
<E> ObservableList<E> observableList(List<E> list, Callback<E,
Observable[]> extractor)
```

Here, you need to additionally provide the `extractor` object, which will tells us which observable or observables in the element have to be tracked. Here is an example:

```
// chapter3/collections/Extractor.java
public void start(Stage stage) {
    // Lets have a list of Buttons
    List<Button> buttons = new ArrayList<>();
    for (int i = 0; i < 10; i++) {
        buttons.add(new Button(i + ""));
    }

    // Now lets have an observable collection which
    // will trigger listener when any button is pressed:
    ObservableList<Button> observableList =
FXCollections.observableList(buttons, (btn) -> {
        // in the extractor we need to return
        // a list of observables to be tracked
        return new Observable[] { btn.pressedProperty() };
    });

    // And add a listener
    IntegerProperty counter = new SimpleIntegerProperty(0);
    observableList.addListener((ListChangeListener.Change<? extends Button>
c) -> {
        counter.set(counter.intValue() + 1);
    });
```

```
        Label label = new Label();
        label.textProperty().bind(counter.asString("changes count: %1$s"));

        HBox root = new HBox(10);
        root.setAlignment(Pos.CENTER);
        root.getChildren().add(label);
        root.getChildren().addAll(buttons);
        stage.setScene(new Scene(root, 500, 100));
        stage.setTitle("Binding to a List demo");
        stage.show();
    }
```

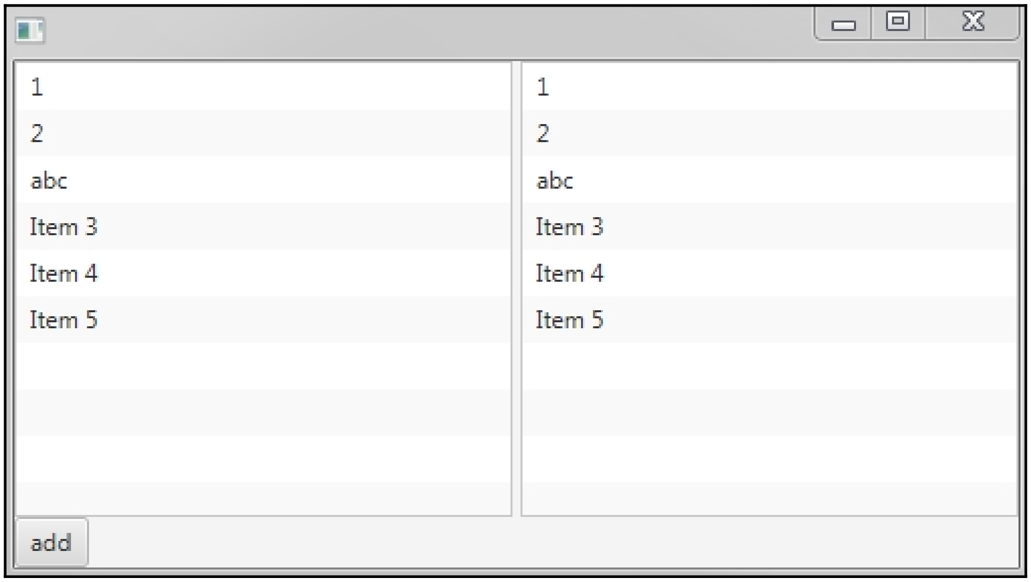
This code will produce an application where one binding track clicks for any number of buttons:



Once we have the observable list, we can use the binding as well. For example, in the following code, we connect two `ListView` objects to make them show similar data:

```
        public void start(Stage stage) {
            ListView<String> listView = new ListView<>();
            listView.getItems().addAll("1", "2", "3");
            ListView<String> listView2 = new ListView<>();
            listView2.itemsProperty().bind(listView.itemsProperty());

            Button button = new Button("add");
            button.setOnAction((e)->listView.getItems().add("Item " +
    listView.getItems().size()));

            BorderPane root = new BorderPane();
            root.setLeft(listView);
            root.setRight(listView2);
            root.setBottom(button);
            stage.setScene(new Scene(root, 500, 250));
            stage.show();
        }
```

Try adding elements to one `ListView` in the resulting application and the changes will be reflected in the other one. Refer to the following screenshot:



Summary

In this chapter, we looked at the binding functionality, which allows us to simplify the logic of the components' interactions. We learned that every property of JavaFX can be monitored for changes and quickly hooked to other properties through binding calls.

The Binding API is another step forward Java makes to the usability of the functional languages. The first was the Java8 Streams API. They both allow us to build a complex logic sequentially without using excessive syntax such as anonymous class creation.

In the following chapter, we will review another important tool introduced by JavaFX—the markup description language FXML.

4
FXML

FXML is a powerful tool used to build a complex JavaFX UI and separate business logic from UI design. This chapter will cover the following topics:

- Using FXML to design an FX application
- Working with SceneBuilder—the drag-and-drop designer provided by Oracle
- Viewing FXML as the **model-view-controller** (**MVC**) pattern

Introduction to FXML

When we were talking about Scenegraph, we saw how conveniently UI is represented by a tree graph structure. What else is good to describe tree structures? XML!

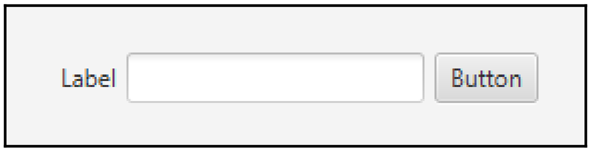
That is what FXML is all about. It allows us to describe the UI by the XML files. It's a common approach to describing static UI pages. For example, XAML for C#, or Android GUI, uses a similar approach.

Basics of FXML

In FXML, each XML node corresponds to a JavaFX entity:

- Adding a tag to the FXML file is similar to calling a constructor for a corresponding class.
- XML attributes correspond to the constructor parameters or setter method calls.
- For `Parent` classes, the inner nodes correspond to the children nodes, building hierarchy on the fly.
- For other classes, the inner nodes correspond to any entities these classes may contain. For example, you can populate `FXCollection` through FXML.

Let's compare an application and the FXML that describes it. Here is the application:



The FXML that describes it is as follows:

```
 <?xml version="1.0" encoding="UTF-8"?>

 <?import javafx.scene.control.*?>
 <?import java.lang.*?>
 <?import javafx.scene.layout.*?>

 <HBox alignment="CENTER" spacing="5.0" onAction="#buttonHandler"
xmlns:fx="http://javafx.com/fxml" xmlns="http://javafx.com/javafx/8"
fx:controller="chapter4.first.FirstController">
   <children>
      <Label text="Label" />
      <TextField fx:id="textField" />
      <Button fx:id="button" text="Button" />
   </children>
</HBox>
```

Here, we have four nodes for JavaFX entities—`HBox` and its children, `Label`, `TextField`, and `Button`—and a service node, `children`, which corresponds to the children collection of the `HBox`.

To use this FXML in a program, you need to use `FXMLLoader` to get a top-level node as a result, which can be used just like a regular FXML node:

```
// chapter4/first/DemoFXML.java
HBox root = (HBox) FXMLLoader.load(
                   getClass().getResource("FirstDocument.fxml"));

Scene scene = new Scene(root, 300, 80);
stage.setScene(scene);
stage.show();
```

The next question is where to put the logic for FXML programs.

Note the `fx:controller` attribute of the FXML. It's the name of the special controller class that handles the business logic. The controller and FXML are connected through the tags, such as `fx:id`, which JavaFX uses to assign a JavaFX object to all FXML tags:

```
// chapter4/first/FirstController.java
public class FirstController implements Initializable {
    @FXML
    private Button button;

    @FXML
    private TextField textField;
    @Override
    public void initialize(URL url, ResourceBundle rb) {
        button.setText("hello");
    }

    @FXML
    private void buttonHandler(ActionEvent event) {
        textField.setText("hello");
    }
}
```

Any field annotated by `@FXML` will automatically be initialized with a corresponding tag with the same `fx:id` as the field's name. At the time the `initialize` method is called, all these fields will be ready.

Note, at the moment of `initialize()` not all UI functionality is ready yet. For example, you can't set focus to a UI control. To address it, you can use the following trick—wrap such code with `Platform.runLater()` and it will be called on the next good opportunity. Note, though, it may not work for complex applications.

Also, methods with the same `@FXML` tags will be associated with the `#methodName` attribute's values in the FXML file. See the `buttonHandler` methods in the preceding examples.

Benefits of FXML

This division of the UI layout and the business logic pattern gives us several benefits.

First of all, separation of View and Controller logic gives more flexibility and supportability to your code. Once you have established an interface between View and Controller (which is defined by `fx-id` objects and handler names), you can modify them separately or share this work across the team.

Secondly, having an intermediate media to store app layout allows using separate tools to work with it. We will talk about SceneBuilder in detail later in this chapter, which allows designing JavaFX apps in WYSIWYG mode.

And finally, even without extra tools, FXML is much closer to actual application layout than Java code. You can clearly see which parent has which nodes, and all nodes' attributes are gathered in one place. In Java code, all this logic is often spread all over the method bodies or even between different classes.

Limitations of FXML

FXML can describe only static user interfaces; for any UI changes, you'll need to use Java code or prepare a separate FXML.

Also, debugging FXML loading is tricky as it works through reflection and you need to be very careful with the names of `@FXML` variables.

For the same reason, refactoring of the code that uses FXML may miss text constants inside FXML. Some IDEs (for example, the latest versions of NetBeans) may help with that.

Working with FXML loaders

There is only one class responsible for loading FXML—`FXMLLoader`—but working with it requires care, as shown in the following sections.

Working with resources

Let's look again at loading FXML:

```
FXMLLoader.load(getClass().getResource("FirstDocument.fxml")); // URL
```

The `load()` method's parameter here is a URL to the FXML file during **runtime**. It usually looks like the following:

```
jar:file:/path/to/jar/file/FxmlDemo.jar!/demo/FirstDocument.fxml
```

So, you will almost never set it directly as a `String` but through the `getResource()` method. The `getResource` parameter can be relative to the current class, as in the preceding examples, or absolute to your JAR directory structure, for example:

```
FXMLLoader.load(getClass().getResource("/demo/FirstDocument.fxml"));
```

Using absolute paths allows you to store FXML files in a separate folder rather than among the Java code, which is more convenient for complex projects.

Often, the project structure will look like the following:

```
./src/demo/FirstController.java
./src/demo/FxmlDemo.java
./resources/fxmls/FirstController.fxml
```

Here, the JAR routine will combine both built classes and FXML files into one folder, so the JAR content will look as follows:

```
FxmlDemo.jar!/demo/FirstController.java
FxmlDemo.jar!/demo/FxmlDemo.java
FxmlDemo.jar!/fxmls/FirstController.fxml
```

In this case, your FXML loader will have to address the FXML file by an absolute address:

```
FXMLLoader.load(getClass().getResource("/fxmls/FirstDocument.fxml"));
```

Using the FXMLLoader API

Aside from only loading and FXML, you may need to use other `FXMLLoader` functionality. In this case, you need to construct an object:

```
FXMLLoader loader = new
FXMLLoader(getClass().getResource("FXMLDocument.fxml"));
loader.load();
FirstController controller = loader.getController();
Parent root = loader.getRoot();
```

Due to an unfortunate name choice, there are several overridden `load()` methods in FXML and most of them are static. Make sure you are using non-static methods for calls from the instantiated `FXMLLoader`. See the following example:

```
FXMLLoader loader = new FXMLLoader();
loader.load(getClass().getResource("FXMLDocument.fxml"));
// STATIC method!
FirstController controller = loader.getController(); //
returns null
Parent root = loader.getRoot(); // returns null
```

Working with the fx:root attribute and custom components

You can revert the workflow of the FXML node creation and inject FXML content into your own object. This is especially useful if you want to create custom controls with content managed by FXML.

Firstly, the `fx:root` attribute should be used instead of the direct class name for the root node:

```
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<fx:root type="HBox" xmlns:fx="http://javafx.com/fxml">
   <children>
      <Label fx:id="label" />
      <TextField fx:id="textField" />
   </children>
</fx:root>
```

Also, note that you may not set `fx:controller` here and select the desired type of the root.

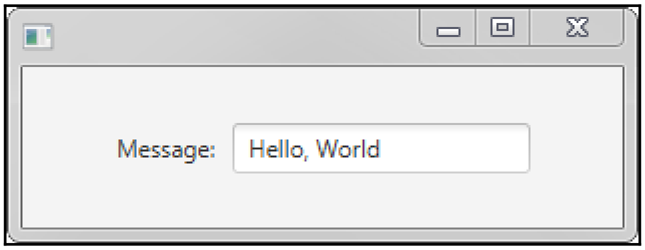
Now, we can create the class that will be used as a root for such FXML:

```
// chapter4/fxroot/MyControl.java
public class MyControl extends HBox {
    @FXML
    private TextField textField;

    @FXML
    private Label label;
    public MyControl(String text) throws IOException {
        // here we initialize our HBox
        setAlignment(Pos.CENTER);
        setSpacing(5);
        // loading FXML and using current object as it's root and
controller
        FXMLLoader fxmlLoader = new
FXMLLoader(getClass().getResource("MyControl.fxml"));
        fxmlLoader.setRoot(this);
        fxmlLoader.setController(this);
        fxmlLoader.load();
        // now we already can use @FXML initialized controls
        textField.setText(text);
        label.setText("Message: ");
    }
}
```

Here, we are using the same class for the Controller and Root, which allows us to contain all logic in one place. So, we can create our new control like a regular JavaFX class:

```
// chapter4/fxroot/FxRootDemo.java
public class FxRootDemo extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        StackPane stackPane = new StackPane();
        stackPane.getChildren().add(
                new MyControl("Hello, World"));
        Scene scene = new Scene(stackPane, 300, 80);
        stage.setScene(scene);
        stage.show();
    }
}
```

And, we will get the following application:


Message: Hello, World

Note that your own controls can be used in FXML as well. You just need to provide the correct includes.

Working with Controllers

A question that is often asked is how to transfer data between an FXML Controller and other parts of the application. Let's look into the available options.

Enhancing Controllers

FXML Controllers are not set in stone; you can add methods to them and use them to transfer information.

Consider the following example for the `FirstDocument.fxml` we used earlier in this chapter:

```
public class SmartController implements Initializable {
    public void setText(String newText) {
        textField.setText(newText);
    }

    @FXML
    private Button button;

    @FXML
    private TextField textField;
    @Override
    public void initialize(URL url, ResourceBundle rb) {
        button.setText("hello");
    }
}
```

Now, we can work with the FXML variables from other classes, as in the following example, from `Application`:

```
FXMLLoader loader = new
FXMLLoader(getClass().getResource("FirstDocument.fxml"));
HBox root = loader.load();
loader.<SmartController>getController().setText("Text from App");
```

Also, you can always declare variables in a `Controller` public and use them directly. While it's not advised for production code, it can simplify prototyping.

Using a preconstructed Controller

The approach from the previous section will not work if you need to have certain data accessible at the time of the `initialize()` call. If the controller is set by the `fx:controller` attribute in FXML, it will be constructed automatically by JavaFX.

For such cases, there is an option to use a preconstructed controller:

```
public class PreconstructedController implements Initializable {
    private final String newText;
    public PreconstructedController(String newText) {
        this.newText = newText;
    }

    @FXML
    private Button button;

    @FXML
    private TextField textField;
    @Override
    public void initialize(URL url, ResourceBundle rb) {
        button.setText(newText);
    }
}
```

Also, you need to remove the `fx:controller` attribute from the FXML file.

After that, you can create and initialize your controller in advance and then use it for FXML:

```
FXMLLoader loader = new
FXMLLoader(getClass().getResource("FirstDocument.fxml"));
PreconstructedController pc = new PreconstructedController("new text");
loader.setController(pc);
HBox root = loader.load();
```

Working with data

An FXML node resembles the MVC pattern very closely. FXML as a View and the Controller present by default, the only thing left is Model—a separate entity to store data and communicate with the outside part of an application.

It can be either a global context in a form of a `Singleton` class or a combination with one of the previous methods to set your model class to a controller.

Syntax details of FXML

All the FXML syntax we have already used in this chapter was very self-explanatory (another benefit of FXML!) but, there are additional useful options which we will review in the following sections.

Reviewing the basics of FXML

Let's briefly review the basics we learned at the beginning of this chapter:

- Tags correspond to JavaFX classes and attributes correspond to these classes' properties
- By setting the `fx:id` attribute, you can link a field from the Controller to these tags
- You can set the Controller by the `fx:controller` attribute of the root node
- Action handlers can be linked using the `#` symbol

Take a look at the following code snippet:

```
<HBox alignment="CENTER" spacing="5.0" onAction="#buttonHandler"
            xmlns:fx="http://javafx.com/fxml"
xmlns="http://javafx.com/javafx/8"
            fx:controller="chapter4.first.FirstController">
   <children>
      <Label text="Label" />
      <TextField fx:id="textField" />
      <Button fx:id="button" text="Button" />
   </children>
</HBox>
```

Also, note the root tag should have XML namespace attributes set: `xmlns:fx` and `xmlns`.

Importing packages

The next thing you need to pay attention to is the imports section. All used classes should have corresponding packages mentioned in the import header:

```
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
```

Including other FXML files

You can include one FXML in another, as shown in the following code:

```
<?import javafx.scene.layout.*?>
<HBox xmlns:fx="http://javafx.com/fxml">
    <children>
       <fx:include source="my_fields.fxml"/>
       <TextField fx:id="textField" />
       <Button fx:id="button" text="Button" />
    </children>
</HBox>
```

Here, `my_fields.xml` contains all the listed data:

```
<?import javafx.scene.control.*?>
<Label text="Label" />
```

Note that in this document, `Label` is a root element for simplicity. But, if you want to have several controls, you'll have to introduce a layout manager to handle them—there can be only one root element in the FXML.

But, what to do if the included FXML has its own `Controller`? They are called nested controllers and can be handled through the `fx:id` attribute, similar to regular JavaFX classes.

First of all, you need to set a name for your `include`:

```
<?import javafx.scene.layout.*?>
<?import javafx.scene.control.*?>
<HBox xmlns:fx="http://javafx.com/fxml"
fx:controller="chapter4.includes.FirstController">
    <children>
       <fx:include fx:id="myLabel" source="MyLabel.fxml"/>
       <TextField fx:id="textField" />
       <Button text="Button" onAction="#btnAction"/>
    </children>
</HBox>
```

Now, let's create a slightly more sophisticated `MyLabel` FXML with a `Controller`:

```
<?import javafx.scene.layout.*?>
<?import javafx.scene.control.*?>
<VBox xmlns:fx="http://javafx.com/fxml"
fx:controller="chapter4.includes.NestedController">
    <Label text="MyLabel" />
    <Button fx:id="myBtn" text="I'm nested" onAction="#myBtnAction" />
</VBox>
```

And, `NestedController` provides a small API to alter its button:

```
// chapter4/includes/NestedController.java
public class NestedController implements Initializable {
    @FXML
    private Button myBtn;
    @FXML
    void myBtnAction(ActionEvent event) {
        System.out.println("Hello from " + myBtn.getText());
    }
    public void setButtonText(String text) {
        myBtn.setText(text);
    }

    @Override
    public void initialize(URL url, ResourceBundle rb) {
    }
}
```

Now, in the controller for the first FXML, we can use that API:

```
// chapter4/includes/FirstController.java
public class FirstController implements Initializable {
    @FXML
    private NestedController myLabelController; // name is tricky! See
below.
    @FXML
    void btnAction(ActionEvent event) {
        myLabelController.setButtonText(textField.getText());
    }
    @FXML
    private TextField textField;
    @Override
    public void initialize(URL url, ResourceBundle rb) {
    }
}
```

Note that the name of the corresponding variable should be the corresponding `fx:id` value concatenated with the *Controller* word.

Using FXML defines

To structure your FXML better, you can use `fx:define` to separate usage and declaration of various elements. For example, the preceding examples with includes can be rewritten by defining the include first and using it later through the `$` prefix. Refer to the following code snippet:

```
<?import javafx.scene.layout.*?>
<?import javafx.scene.control.*?>
<HBox xmlns:fx="http://javafx.com/fxml"
fx:controller="chapter4.includes.FirstController">
    <fx:define>
        <fx:include fx:id="myLabel" source="MyLabel.fxml"/>
    </fx:define>
    <children>
       <StackPane children="$myLabel">
    </children>
</HBox>
```

Another common use case for `define` is setting a toggle group for radio buttons:

```
<?import javafx.scene.layout.*?>
<?import javafx.scene.control.*?>
<HBox xmlns:fx="http://javafx.com/fxml" >
    <fx:define>
        <ToggleGroup fx:id="toggleGroup"/>
    </fx:define>
    <children>
        <RadioButton text="radio1" toggleGroup="$toggleGroup"/>
        <RadioButton text="radio2" toggleGroup="$toggleGroup"/>
        <RadioButton text="radio3" toggleGroup="$toggleGroup"/>
    </children>
</HBox>
```

Default properties

In some cases, you can skip declaring properties, such as `children`:

```
<?import javafx.scene.layout.*?>
<?import javafx.scene.control.*?>
<HBox xmlns:fx="http://javafx.com/fxml"
fx:controller="chapter4.includes.FirstController">
    <TextField fx:id="textField" />
    <Button text="Button" onAction="#btnAction"/>
</HBox>
```

The most common properties are marked as *default* by the JavaFX API, and FXML allows us to skip them to remove obvious parts of the code.

You can detect such properties by looking into JavaDoc or the source code. They are set through the `@javafx.beans.DefaultProperty` annotation:

```
@DefaultProperty("children")
public class Pane
extends Region
```

TIP

Using this annotation, you can set the default property for your own classes as well.

Referring to resources from FXML

Certain JavaFX controls work with external resources. For example, the `Image` class requires a URL of the represented image. To simplify work with such controls in FXML, you can refer to a path relative to the FXML location by using the `@` prefix. Refer to the following code snippet:

```
<ImageView>
    <Image url="@image.png"/>
</ImageView>
```

Here, JavaFX assumes that `image.png` is located in the same folder (either on disk or in a JAR) as the FXML.

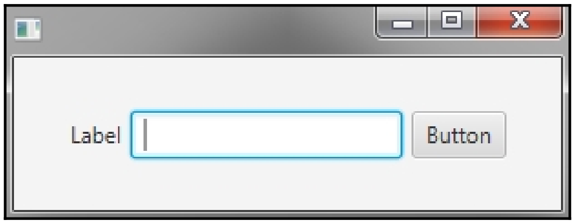
Adding business logic to FXML

Although I strongly advise against doing it, you can put bits of the business logic directly into FXML by using the JavaScript engine and the `fx:script` tag.

In the next example, we declare a handler directly in the FXML and assign it to the button:

```
<?language javascript?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<HBox alignment="CENTER" spacing="5.0" xmlns:fx="http://javafx.com/fxml/1">
    <fx:script>
        function handleButtonAction(event) {
           textField.setText("clicked");
        }
    </fx:script>
   <children>
      <Label text="Label" />
      <TextField fx:id="textField" />
      <Button fx:id="button" text="Button"
            onAction="handleButtonAction(event);"/>
   </children>
</HBox>
```

And, on button click, the `TextField` value will be updated:



Also, you can use binding by using expression bindings:

```
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<HBox alignment="CENTER" spacing="5.0" xmlns:fx="http://javafx.com/fxml/1">
   <children>
      <Label text="${textField.text}" />
      <TextField fx:id="textField" />
   </children>
</HBox>
```

Here, we bind the `Label` text property to the TextField content.

There are also ways to build a more complex business logic inside FXML, but I really advise not using these options for the following reasons:

- You can't debug this logic
- It's not tracked by the IDE's Find Usage or Refactor mechanisms
- It's easy to miss them in a text file
- It can be very confusing to maintain

Using static methods in FXML

Some JavaFX APIs require the use of static methods, like those we used to adjust BorderPane in `Chapter 1`, *Stages, Scenes, and Layout*:

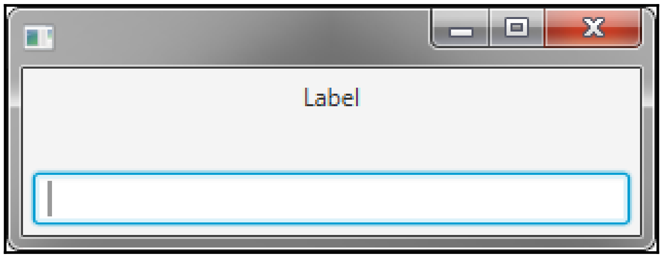
```
BorderPane.setAlignment(label, Pos.CENTER);
```

To use them in FXML, you need to call them backward—inside a parameter to be adjusted, you need to assign a value to the class name and property name combination—`BorderPane.alignment="CENTER"`.

Here is a full example of such FXML (note another example of using `define`, adding it to make an image look better):

```
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.geometry.*?>

<BorderPane padding="$insets" xmlns:fx="http://javafx.com/fxml/1">
    <fx:define>
        <Insets bottom="5.0" left="5.0" right="5.0" top="5.0"
fx:id="insets"/>
    </fx:define>
   <top>
      <Label text="Label" BorderPane.alignment="CENTER"/>
   </top>
   <bottom>
      <TextField fx:id="textField" />
   </bottom>
</BorderPane>
```

On running the preceding code, you'll see the following output:

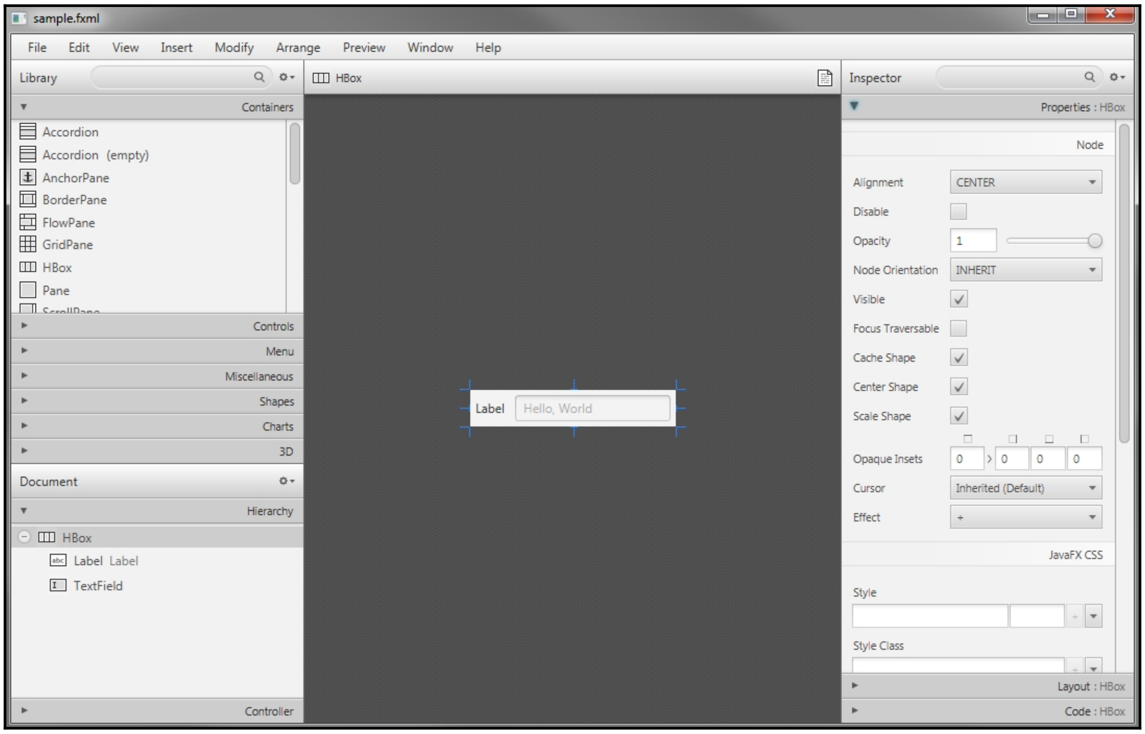


SceneBuilder

Finally, having gone through most of the features of FXML, we will look into a great tool to handle them—SceneBuilder. It's an open source tool, primarily developed by Oracle. It can be downloaded from the Oracle site or built from sources—see the instructions at `https://wiki.openjdk.java.net/display/OpenJFX/Building+OpenJFX`.

Working with a WYSIWYG editor

The **What You See Is What You Get** (**WYSIWYG**) conception is extremely useful for building UIs. You can design your application by dragging and dropping components and adjusting their properties:



Alignment CENTER
Disable ☐
Opacity 1
Node Orientation INHERIT
Visible ✓
Focus Traversable ☐
Cache Shape ✓
Center Shape ✓
Scale Shape ✓
Opaque Insets 0 0 0 0
Cursor Inherited (Default)
Effect +

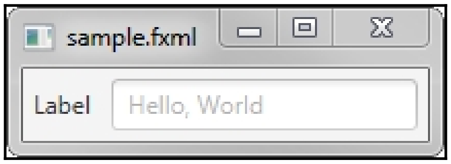
This is a base SceneBuilder UI. It contains the following components:

- Drawing board in the middle
- List of available components in the top-left corner
- List of properties for the currently selected component in the right-hand side panel
- Scenegraph representation, called Hierarchy (it works in a similar way to the ScenicView tool we used in previous chapters), in the bottom-left corner

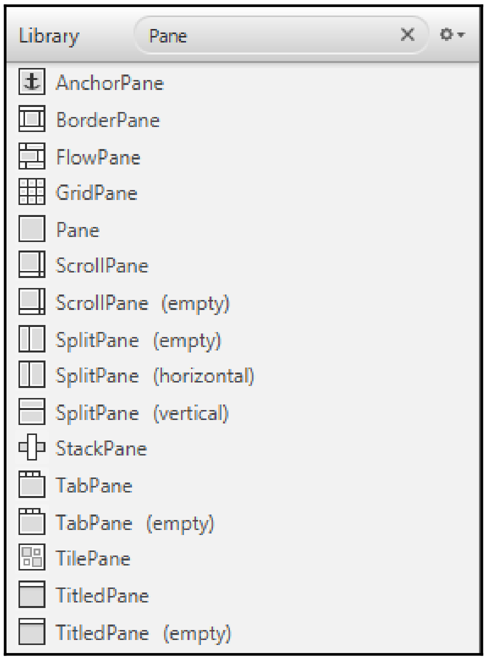
The main functionality is pretty straightforward, so I'll review only the less-noticeable features.

Features

You can preview the current state of the FXML app by using the **Preview** menu item:

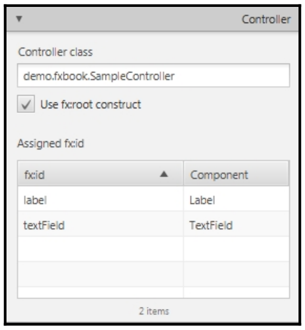


The **Components** window has a convenient search box at the top to avoid looking through the categories:



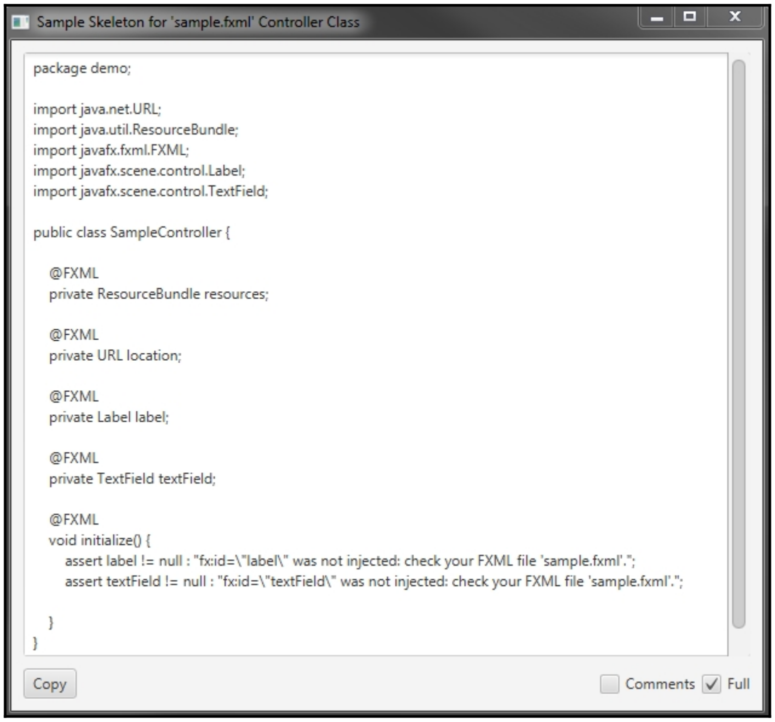
Below **Hierarchy,** there is a **Controller** view.

It allows you to set the `Controller` name—be careful of typos; it's just a text field for `SceneBuilder`. Also, all fields that you intend to use in the `Controller` and marked with `fx:id` will be listed here:



fx:id	Component
label	Label
textField	TextField

Based on this information, SceneBuilder can prepare for you a skeleton of a real `Controller` based on your FXML. Use the menu item **View|Show Sample Controller Skeleton**:



```
package demo;

import java.net.URL;
import java.util.ResourceBundle;
import javafx.fxml.FXML;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;

public class SampleController {

    @FXML
    private ResourceBundle resources;

    @FXML
    private URL location;

    @FXML
    private Label label;

    @FXML
    private TextField textField;

    @FXML
    void initialize() {
        assert label != null : "fx:id=\"label\" was not injected: check your FXML file 'sample.fxml'.";
        assert textField != null : "fx:id=\"textField\" was not injected: check your FXML file 'sample.fxml'.";

    }
}
```

Make sure you are satisfied with your FXML state first—if you used this generated `Controller` class, added code to it, and changed the FXML after, there is no automated way to merge your changes with the updated generated `Controller`.

Specifying CSS files through the Preview menu

You can specify CSS files and localization files directly in SceneBuilder through the **Preview** menu. Note that these changes will not be reflected in the FXML you are editing as they have to be set in the Java code—they are added only for convenience.

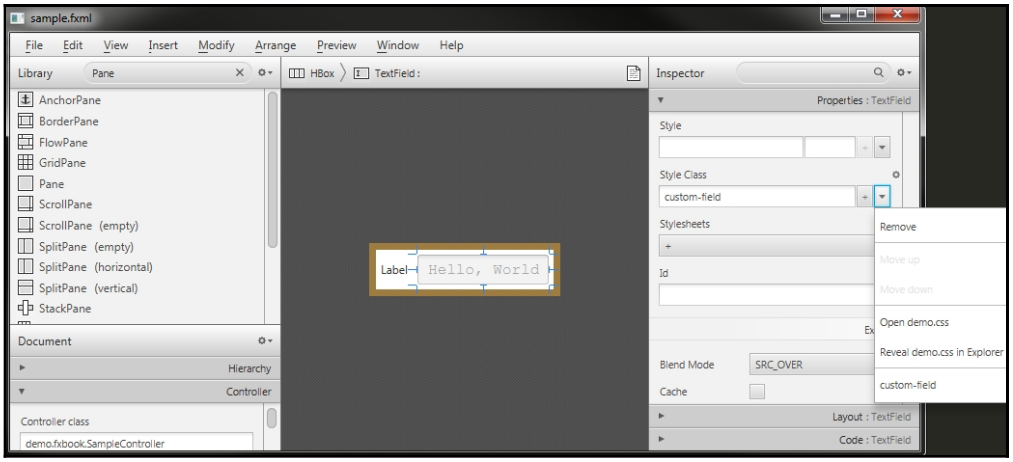
We will talk in detail about CSS in `Chapter 6`, *Styling Applications with CSS*, for this section, you need to know just a few facts:

- JavaFX supports CSS similar to HTML
- You can assign various properties to CSS classes and assign these classes to JavaFX objects

Here is a sample CSS used for the following screenshot:

```
.custom-field {
    -fx-font: 16px "Courier New";
}
```

Once you've selected a CSS file, it will instantly be applied to your editing interface. Also, all CSS classes will be available on the **Property** page, so you can select them from the list instead of typing—no typos! Check the bottom of the popup in the following screenshot:



Style Class: custom-field

Blend Mode: SRC_OVER

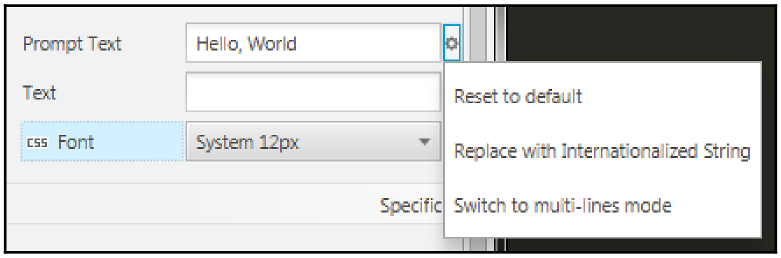
Cache

Controller class: demo.fxbook.SampleController

In this screenshot, `custom-field` is a class from the CSS file that alters the `TextField` font.

Localization in Preview

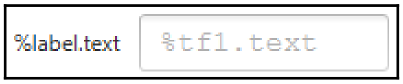
For localization, you can change any text fields to i18n labels by using the small gear icon:



For example, let's look at the following two i18n files:

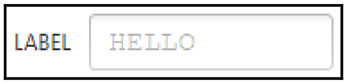
```
#demo_en.properties
label.text=LABEL
tf1.text=HELLO

#demo_fr.properties
label.text=étiqueter
tf1.text=bonjour
```

Let's apply them to our small UI. First, we change all text values to i18n variables:

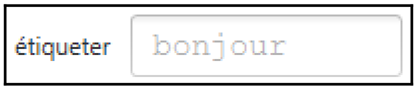


And now, we can select English and French versions and preview the application without even compiling any code!

Here is the English version:



Here is the French version:



This way, you can be sure that your UI looks good in all supported localizations.

In Java code, you can set the resource bundle used for a specific FXML localization in the Loader API:

```
// for current system locale
loader.setResources(ResourceBundle.getBundle("chapter4.demo"));
// for specific locale
loader.setResources(ResourceBundle.getBundle("chapter4.demo", new
Locale("fr", "FR"));
```

Here, `chapter4` is a package name and `demo` is the first part of the filename (`demo_fr.properties`).

Summary

In this chapter, we studied FXML syntax and capabilities, studied common questions, and learned about the SceneBuilder tool. With these tools, building a UI with FXML became easier due to using a WYSIWYG editor and decoupling the UI from the code.

In the next chapter, we'll look into the Animation API, which helps with dynamic content.