Chapter 7

Introducing the JavaFX Node Hierarchy

In This Chapter

- Introducing the most important packages and classes that make up JavaFX
- Looking at the important methods that all controls inherit via the `Node`, `Parent`, and `Region` classes

The simplest definition of a JavaFX control is this: A *control* is an object created from a class that directly or indirectly inherits the JavaFX `Control` class. The `Control` class provides all the basic functions that are required for a JavaFX object to be considered a control. For example, any class that inherits the `Control` class has a visual representation in a scene, can be added to a layout pane, can automatically adjust its size within parameters you set by calling methods such as `setMaxWidth` or `setMinHeight`, and can have a tooltip that pops up when the user hovers the mouse over the control.

Although all controls have those features in common, not all those features are provided directly by the `Control` class. That's because the `Control` class itself inherits the `Region` class, which in turn, inherits the `Parent` class, which in turn inherits the `Node` class. Each of these classes along this inheritance chain contributes features to every JavaFX control.

In this chapter, you read about some of the more important features that are common to every JavaFX control by virtue of the fact that every control inherits the `Control` class, which in turn inherits the `Region`, `Parent`, and `Node` classes.

An Overview of JavaFX Packages

Before I look at the classes that make up the `Node` class hierarchy, I want to briefly discuss the various packages that make up JavaFX. JavaFX itself consists of a total of 665 classes that are spread out over 36 distinct packages, which all begin with the root name `javafx`.

So far in this book, you've seen JavaFX classes from the following seven packages:

- **`javafx.application`:** The most important class in this package is `Application`, which provides the basic lifecycle functions of a JavaFX application.

  As I discuss in Chapter 2, all JavaFX programs extend the `Application` class and implement the `start` method, which is called to initiate the application. The `Application` class also creates the application's primary stage and passes it to the `start` method via the `primaryStage` parameter. This allows the program to display a scene in the application's window.

- **`javafx.stage`:** The most important class in this package is `Stage`, which represents a window in which a user interface can be displayed. You read about the `Stage` class in Chapter 4. There are other classes in this class that may occasionally be useful, such as `FileChooser` and `DirectoryChooser`, which display dialog boxes that let you select files and directories.

- **`javafx.scene`:** This package contains several important classes that deal with creating user-interface scenes that can then be displayed in a stage. The two most important classes in this package are

  - `Scene`, which creates a scene object.

    You can read about the `Scene` class in Chapter 4.

  - `Node`, which is the base class for all objects contained in a scene, including controls and layout panes.

    For more information about the `Node` class, see the section "The Node Class" later in this chapter.

- **`javafx.scene.control`:** This package contains most of JavaFX's user-interface control classes, including `Button`, `Label`, `CheckBox`, and `RadioButton`. Also included in this package is `Control`, the base class from which all user-interface controls are derived. For more information about the `Control` class, see the section "The Control Class" later in

this chapter. (***Note:*** There are a few JavaFX controls that are defined in other packages, including `javafx.scene.control.cell` and `javafx.scene.web`.)

- **`javafx.scene.layout`:** This package contains the layout pane classes, such as `HBox`, `VBox`, and `BorderPane`. Two other important classes defined in this package are `Pane` and `Region`. All the layout pane classes are based on the `Pane` class, and both the `Pane` class in this package and the `Control` class in the `javafx.scene.control` class are based on the `Region` class. For more information about the `Region` class, see the section "The Region Class" later in this chapter.
- **`javafx.geometry`:** This is a relatively small package that defines several classes and enumerations that are related to the geometry of JavaFX nodes. In Chapter 5, you figure out how to use the `Insets` class to control spacing within a layout pane as well as the `Pos` enumeration to specify alignments.
- **`javafx.collections`:** This package defines the `ObservableList` class, which is used by the `getChildren` method of the `Pane` class. You also encounter several control classes in the next few chapters that require this package.

Because these seven packages contain most of the JavaFX classes you'll use in applications that work mostly with controls (as opposed to classes that work with other user-interface objects such as graphs, shapes, or animations), I recommend you import all the classes in these seven packages in all your programs:

```
import javafx.application.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.scene.geometry.*;
import javafx.collections.*;
```

Although you don't need all the classes in all these packages in every program, including these entire packages every time eliminates the need to keep track of which programs need which specific classes.

In the rest of this chapter, I take a closer look at the classes that are inherited by all JavaFX controls: `Node`, `Parent`, `Region`, and `Control`.

The Node Class

The JavaFX `Control` class hierarchy begins at the `Node` class, which represents an object that can be added to the JavaFX scene. Well, actually, the topmost class in the `Control` class hierarchy is the `Object` class, but that's hardly worth mentioning here because *all* Java class hierarchies begin with the `Object` class — `Object` is the mother of all Java classes.

All objects that are a part of a scene belong to a *scene graph,* which is a tree structure that contains all the nodes that make up a user interface. To be a part of a scene graph, an object must inherit the `Node` class. Thus, the `Node` class is the base class for all classes that can be added to a JavaFX scene.

Like any other tree structure, a scene graph begins with a single node — the *root node* — which can have one or more child nodes, each of which in turn can have one or more child nodes. A node that has at least one child node is a *branch node;* a node that has no children is a *leaf node.* A scene can have only one root node, but it can have many branch and leaf nodes.

The `Node` class is an abstract class, which means that you can't directly create an instance of it. In other words, the following code results in a compiler error:

```
Node myNode = new Node();
```

However, you can use the `Node` class to hold nodes whose type you are uncertain of or don't care about. For example:

```
Node myNode = new Button();
```

In this case, the `myNode` variable is of type `Node`, but it's used to hold a reference to a `Button` control.

Many methods of classes up and down the `Node` hierarchy accept or return objects of type `Node`. For example, the `getChildren` method used with layout panes, such as `HBox` and `FlowPane`, returns list of `Node` objects. And the `add` method used to add an object to a layout pane's node list accepts a `Node` object as a parameter. In other words, any `Node` object can be added to a layout pane.

Table 7-1 lists just a few of the methods you're likely to use in most JavaFX programs. Note that this table drastically simplifies the complexity of the `Node` class. There are actually more than 300 methods defined by this class. More than one third of them are related to event handling: The `Node` class is

the class that's responsible for most event handling for all nodes, including events that handle mouse, keyboard, and touchscreen interaction with the node.

Table 7-1 Commonly Used Methods of the Node Class

Method	*Explanation*
`Parent getParent()`	Returns this node's `Parent` node.
`String getId()`	Returns the ID of this node.
`void setId(String id)`	Sets the ID of this node. The ID should be unique within the scene graph.
`Node lookup(String id)`	Searches the node's children for a node whose ID matches the parameter.
`String getStyle()`	Returns the CSS style string for the node.
`Void setStyle(String style)`	Sets the CSS style string for the node.

The `getParent` method returns a node's parent node. This method returns an object of type `Parent`, which is a node that can have children (as you discover in the following section). Every node in a scene graph except the root node must have a parent node, and that parent node will always be of type `Parent`. If you call this method on the root node, null will be returned.

Notice that every node can have a unique string identifier, which makes it easy to distinguish nodes from one another in complicated scene graphs and can also be helpful when you use CSS to format your scenes. You can set the string identifier by calling the `setID` method, which accepts a string argument like this:

```
myNode.setId("LBL3");
```

Here, the string `LBL3` is associated with the node.

You can later find this node by calling the `lookup` method. This method is a little quirky in that you must preface the ID you're looking for with a hash symbol (#). For example, here's how you might search an entire scene graph for a control whose ID is `LBL3`:

```
Node myNode = scene.getRoot().lookup("#LBL3");
```

Here, the `getRoot` method of the `scene` variable (which I assume to be of type `Scene`) is called to get the root node of the scene. Then, the `lookup` method is called to return the node whose ID is `LBL3`.

There are many other methods of the `Node` class that let you apply common formatting or other features for all types of nodes. For example, the `setStyle` method lets you apply CSS-style formatting to any type of node. And the `setRotate` method lets you rotate any node. You can read about these and other `Node` methods in later chapters of this book.

The Parent Class

Although ten different classes directly inherit the `Node` class, the only one you need to be concerned with when working with JavaFX controls is the `Parent` class. For an explanation of all ten of the subclasses of Node, see the sidebar "Ten Different Kinds of Nodes" in this chapter.

The `Parent` class has all the capabilities of the `Node` class, plus the added ability to have child nodes. Its main job is to manage a collection of child nodes, which is represented as a standard Java list. You can access this list by calling the `getChildren` method.

You've seen this method in action in layout panes, such as `HBox` and `VBox`. For example, the following code creates an `HBox` pane and then adds two controls to it:

```
Label lblAddress = new Label("Address:");
TextField txtAddress = new TextField();
HBox hbox = new HBox();
hbox.getChildren().addAll(lblAddress, txtAddress);
```

The `getChildren` method returns an object of type `ObservableList`, which in turn extends the `List` interface. Between them, these two interfaces define a few dozen methods that you can use to manipulate the parent's child nodes. Table 7-2 lists a few of the more commonly used of these methods.

Interestingly, the `getParent` method is defined in the `Parent` class with `protected` access. That means that although the `getParent` method is available to any class that inherits the `Parent` class, the `getParent` method is *not* accessible to the outside world. For the `getParent` method to become public, a class that inherits the `Parent` class must override the `getParent` method with `public` access.

Table 7-2 Commonly Used ObservableList Methods

Method	*Explanation*
`void add(Node node)`	Adds a single child node to the existing list of children.
`void addAll(Node nodes...)`	Adds multiple child nodes.
`void remove(Node node)`	Removes the specified node from the list of children.
`void clear()`	Removes all child nodes.
`int size()`	Returns the number of child nodes.

That's precisely what the `Pane` class does. The `Pane` class is the base class of all layout panes; it inherits the `Parent` class and then overrides the `getParent` method. Here's a snippet of the actual code from the `Pane` class:

```
@Override public ObservableList<Node> getChildren()
{
    return super.getChildren();
}
```

As you can see, the `getChildren` method in the `Pane` class simply calls the `getChildren` method of its superclass (`Parent`) and returns the result.

One class derived from the `Parent` class which you may use on occasion is the `Group` class. The `Group` class is a bit like a layout pane such as `HBox` or `FlowPane`, except that it doesn't provide any actual layout for the child nodes it contains. When you create a `Group`, you can pass the child nodes to the constructor, like this:

```
Group group = new Group(Node1, Node2, Node3);
```

Or, you can use the default constructor and add child nodes via the `getChildren` method, like this:

```
Group group = new Group();
group.getChildren().addAll(Node1, Node2, Node3);
```

You'll see examples of `Group` nodes occasionally throughout this book.

Ten Different Kinds of Nodes

In all, ten different classes directly inherit the `Node` class, creating ten distinct inheritance branches beneath `Node`. The only one of these ten I discuss in the chapters that make up this part of the book is the `Parent` class because all JavaFX controls and layout panes are derived from the `Parent` class. However, to give you a general idea of what other types of objects besides controls and layout panes can be added to a scene graph, the following paragraphs give a brief summary of what each of the ten subclasses of `Node` provide:

- **`Camera`:** An object that's used to graphically render a three-dimensional screen on a flat display.

  A camera is a node because in scene graphs that represent 3D layout, the camera can be positioned at a specific location within that layout, thus rendering the flat image of the 3D layout from a specific perspective.

- **`Canvas`:** A node that you can draw on using drawing commands, much like an artist can draw on a canvas to create a painting.

  A canvas is a two-dimensional object that has a height and a width.

- **`ImageView`:** Represents an image viewer, used to display a two-dimensional image.

- **`LightBase`:** An abstract class that serves as the base class for lighting sources that illuminate a scene rendered by a camera.

  Like a camera, a light source is a node so that you can position it at a specific location within a scene to create realistic lighting effects.

- **`MediaView`:** Represents a media viewer that can play media, such as sound or video.

- **`Parent`:** A node that can contain child nodes. All controls and layout panes inherit the `Parent` class.

- **`Shape`:** A two-dimensional shape such as a rectangle or a circle.

  The `Text` class also inherits the `Shape` class, providing an easy way to display text within a scene.

- **`Shape3D`:** A three-dimensional shape such as a box, cylinder, or sphere.

- **`Subscene`:** Marks a branch of a scene that can be rendered with its own camera.

- **`SwingNode`:** Allows you to incorporate Swing objects into a JavaFX scene graph.

The Region Class

Next in line in the `Node` class hierarchy is the `Region` class. `Region` is the last common ancestor class shared by both the `Control` class and the `Pane` class. Thus, the `Region` class is the last class from which controls and layout panes share common features.

As its name implies, the `Region` class defines a visible area of the scene that has a physical size — that is, a height and a width. The size of a region depends on a number of factors, but by default will be determined by the size of the content it contains. You can set minimum, maximum, and preferred size constraints that the region will honor, and you can specify a fixed amount of padding that provides a margin between the region's content and its outer edges. In addition, the visual style of a region can be set by a Cascading Style Sheet.

Table 7-3 shows the most commonly used methods provided by the `Region` class, which are all related to setting the size of the region.

Table 7-3 Common Methods of the Region Class

Method	*Explanation*
`void setMaxHeight(double height)`	Sets the maximum height for the region.
`void setMinHeight(double height)`	Sets the minimum height for the region.
`void setPrefHeight(double height)`	Sets the preferred height for the region.
`void setMaxWidth(double width)`	Sets the maximum width for the region.
`void setMinWidth(double width)`	Sets the minimum width for the region.
`void setPrefWidth(double width)`	Sets the preferred width for the region.
`double getHeight()`	Gets the actual height of the region.
`double getWidth()`	Gets the actual width of the region.
`void setPadding(Insets value)`	Sets the padding around the inside edges of the `Hbox`.

The `Region` class provides three distinct parameters that let you control the height and width of the region. For both the height and the width, you can set a minimum value, a preferred value, and a maximum value. As their names imply, JavaFX will not make the control smaller than the minimum size or larger than the maximum size, and, if possible, will shoot for the preferred size.

Within these parameters, JavaFX will determine the ideal height and width for the region based on the content it contains. For a control, such as a label or a button, the content is the text that's displayed by the label or the button. For more complex controls, the content is more complex. And for layout panes, the content consists of the aggregate of all the nodes that are added to the pane.

If you don't specify any height or width constraints, all three — minimum, preferred, and maximum — default to the actual computed size of the control's contents.

Note: In many cases, the contents of a region will resize automatically to fill whatever space is available to it. Thus, if the user dynamically resizes the window that contains the scene, the size of all the regions contained within the scene may expand or contract to fill the available space.

If you want to set an exact value for a region's width or height, set all three parameters (minimum, preferred, and maximum) to the same value. For example:

```
lbl.setMinWidth(150);
lbl.setPrefWidth(150);
lbl.setMaxWidth(150);
```

You might, in this case, prefer to create a constant:

```
final static int LABEL_WIDTH = 150;
lbl.setMinWidth(LABEL_WIDTH);
lbl.setPrefWidth(LABEL_WIDTH);
lbl.setMaxWidth(LABEL_WIDTH);
```

That way, if you change your mind about the width of the label, you have to change the value in only one place.

One other setting affects the height or width of a region: the amount of padding you specify. *Padding* provides margins around the edges of the region to prevent crowded-looking scenes. ***Note:*** You're more likely to use padding with layout panes than with controls.

To specify padding, use the `Insets` class, which is defined in the `javafx.geometry` package. `Insets` provides two constructors. The first lets you set even margins around all four sides of a region:

```
pane.setPadding(new Insets(10));
```

Or, you can set different values for the top, right, bottom, and left edges:

```
pane.setPadding(new Insets(10,0,10,0);
```

In this example, the top and bottom margins are set to 10 pixels, but the right and left margins are set to 0.

For more information about padding, flip to Chapter 5.

The Control Class

The ultimate purpose of this chapter is to give you an overview of the features that are common to all JavaFX controls by virtue of the fact that all controls inherit the `Control` class. Now that you've finally made it to a discussion of the `Control` class itself, brace yourself for a little disappointment: The `Control` class itself isn't all that interesting. As Table 7-4 reveals, there are really only three methods of interest. It turns out that most of the features that are common to all controls are actually provided by the `Region`, `Parent`, and `Node` controls.

Table 7-4 Methods of the Control Class

Method	*Explanation*
`void setTooltip(Tooltip value)`	Sets a tooltip for the control.
`void setContextmenu(Contextmenu value)`	Sets a context menu for the control. For more information, see Chapter 10.
`void setSkin(Skin value)`	Sets a skin for the control. For more information, see Chapter 12.

The `Control` adds three main features to the `Region` class: the ability to add tooltips, context menus, and CSS skins. You can read about context menus in Chapter 10, and you see how CSS skins work in Chapter 12. So for now, I just look at tooltips.

A *tooltip* is a pop-up balloon that provides an explanation of a control's function. Creating a tooltip couldn't be easier: You call the `Tooltip` constructor, passing the text of the tooltip as an argument, and then assign the tooltip to a control by calling the control's `setTooltip` method. Here's an example:

```
btnSave.setTooltip(new Tooltip("Saves the file"));
```

Then, when the user hovers the mouse over the button, the tooltip appears.

Congratulations! You now know about the most important methods that are available to all JavaFX controls by virtue of the fact that they all inherit the `Control` class, which in turn inherits `Region`, which inherits `Parent`, which inherits `Node`.

Now, in the remaining chapters of this Part, you discover how to use some of the most commonly used and useful JavaFX controls, including radio buttons, check boxes, choice boxes, lists, tree views, tables, and menus.