Chapter 11

More about Layout Panes for Precise Scene Design

In This Chapter

- Using four more layout panes to create spectacular layouts
- Introducing rectangle shapes
- Adding scroll bars to a layout
- Creating two complete programs

In Chapter 5, you can read about how to work with four basic layout pane classes that let you control the arrangement of controls in a scene: `HBox`, which arranges nodes horizontally; `VBox`, which arranges nodes vertically; `FlowPane`, which arranges nodes both horizontally and vertically; and `BorderPane`, which divides the scene into five regions: Top, Right, Bottom, Left, and Center.

In this chapter, you discover four additional layout panes that give you additional ways to arrange the elements in a scene. Specifically, you discover how to use the following five layout pane classes:

- **`StackPane`:** The `StackPane` class is a bit different than the other layout panes in that it doesn't visually separate nodes from one another. Instead, it displays nodes directly on top of each other. For example, if you add a rectangle shape and a text shape to a stack pane, the text will appear directly over the rectangle.

- **`AnchorPane`:** This layout lets you anchor nodes to the top, right, bottom, left, or center of the pane. As the pane resizes, the nodes are repositioned but remain tied to their anchor points. ***Note:*** A node can be anchored to more than one position. For example, you might anchor a node to the top and the right. Then, when you resize the pane, the node will remain near the top-right corner of the pane.
- **`GridPane`:** Arranges nodes in a grid of rows and columns. The grid does not have to be uniformly sized like a chess board. Instead, the width of each column and the height of each row can vary according to its content. In addition, content can span columns or rows. `GridPane` is an ideal layout type for forms that gather information from the user via user interface controls such as text boxes, list boxes, and so on.
- **`TilePane`:** If you want a layout that resembles a chess board, in which each cell in a grid is the same size, `TilePane` is the layout pane you're looking for. `TilePane` is ideal for organizing thumbnails of image files or other objects of the same size.
- **`ScrollPane`:** Technically, the `ScrollPane` class is not a layout pane at all; it inherits the `Control` class, not the `Pane` class. However, it's primary use is to create layouts that are too large to display all at once and so require a scroll bar to allow the user to pan left and right or up and down (or both) to see all its contents.

Keep in mind that layout panes are typically used in combinations to create the complete layout for your scene. For example, you might use a `GridPane` to organize user input controls and then place the `GridPane` in the center section of a `BorderPane` to place it in the middle of the scene. Or, you might use `VBox` panes to display labels beneath image thumbnails and then add the `VBox` panes to a tile pane to display the labeled images in a tiled arrangement.

Using the StackPane Layout

A *stack pane* layout is unusual in that it does not arrange its nodes by spreading them out so that you can see them all. Instead, it stacks its nodes one on top of the other so that they overlap. The first node you add to a stack pane at the bottom of the stack; the last node is on the top.

You will most often use a stack pane layout with shapes rather than controls. Because I haven't yet covered shapes, I limit the examples in this section to simple rectangles created with the `Rectangle` class. You can read more about this class in Chapter 13. For now, just realize that you can create a rectangle like this:

```
Rectangle r1 = new Rectangle(100,100);
```

To add a fill color, call the `setFill` method, like this:

```
r1.setFill(Color.RED);
```

The `Color` class defines a number of constants for commonly used colors. In this section, I use just three: `LIGHTGRAY`, `DARKGRAY`, and `DIMGRAY`.

The `Rectangle` class is in the `javafx.scene.shape` package, and the `Color` class is in `javafx.scene.paint`. Thus, you need to include the following import statements to use these classes:

```
import javafx.scene.shapes.*;
import javafx.scene.paint.*;
```

To create a stack pane, you use the `StackPane` class, whose constructors and methods are shown in Table 11-1.

Table 11-1 StackPane Constructors and Methods

Constructor	*Description*
`StackPane()`	Creates an empty stack pane.
`StackPane(Node... children)`	Creates a stack pane with the specified child nodes. This constructor lets you create a stack pane and add child nodes to it at the same time.
Method	***Description***
`ObservableList<Node> getChildren()`	Returns the collection of all child nodes that have been added to the stack pane. The collection is returned as an `ObservableList` type, which includes the methods `add` and `addAll`, which lets you add one or more nodes to the list.

(continued)

Table 11-1 ***(continued)***

Method	*Description*
`static void setAlignment (Pos alignment)`	Sets the alignment for child nodes within the stack pane. See Table 5-5 in Chapter 5 for an explanation of the `Pos` enumeration.
`static void setMargin(Node child, Insets value)`	Sets the margins for a given child node. See Table 5-2 in Chapter 5.
`void setPadding(Insets value)`	Sets the padding around the inside edges of the stack pane. See Table 5-2 in Chapter 5 for an explanation of the `Insets` class.

The simplest way to create a stack pane is to first create the nodes that you will place in the pane and then call the `StackPane` constructor, passing the child nodes as parameters. For example:

```
Rectangle r1 = new Rectangle(100,100);
r1.setFill(Color.DARKGRAY);
Rectangle r2 = new Rectangle(50,50);
r2.setFill(Color.LIGHTGRAY);
StackPane stack = new StackPane(r1, r2);
```

Here, I first create a pair of rectangles, one 100x100, the other 50x50. The larger rectangle is filled dark gray, the smaller one light gray. Then, I create a stack pane that holds the two rectangles. Figure 11-1 shows how this pane appears when displayed. As you can see, the smaller rectangle is displayed within the larger one.

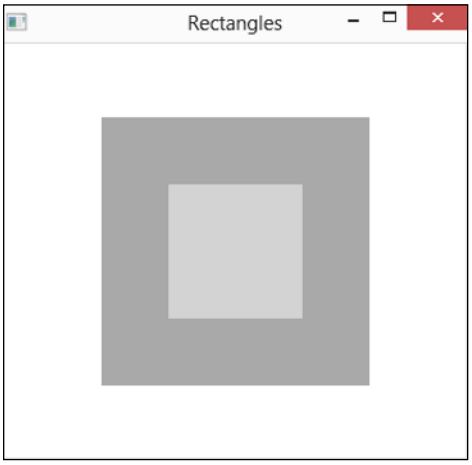

Figure 11-1: Two rectangles displayed in a StackPane.

If you prefer, you can call the `getChildren` method to add nodes to the stack pane, like this:

```
stack.getChildren().add(r1);
stack.getChildren().add(r2);
```

Or like this:

```
stack.getChildren().addAll(r1,r2);
```

Note: The order in which you add nodes to a stack pane has a major impact on how the child nodes are displayed. For example, suppose you reversed the order in which the two rectangles are added:

```
stack.getChildren().addAll(r2,r1);
```

Then, the larger rectangle will be displayed over the top of the smaller one. The result is that the user will see only the larger rectangle. (Unless, of course, the larger rectangle is transparent. I discuss how to create transparent shapes in Chapter 13.)

By default, the objects in a stack pane are centered on top of one another. You can change that by using the `setAlignment` method. The argument for this method is of type `Pos`, the same as for other layout panes that have a `setAlignment` method. If you need a refresher on the `Pos` enumeration, flip to Table 5-5 in Chapter 5. Here's an example that displays three rectangles of various sizes aligned at the top left of the stack pane:

```
Rectangle r1 = new Rectangle(400,150);
r1.setFill(Color.DARKGRAY);

Rectangle r2 = new Rectangle(200, 400);
r2.setFill(Color.LIGHTGRAY);

Rectangle r3 = new Rectangle(150,150);
r3.setFill(Color.DIMGRAY);

StackPane stack = new StackPane(r1, r2, r3);
stack.setAlignment(Pos.TOP_CENTER);
```

Figure 11-2 shows how this pane appears when displayed in a scene.

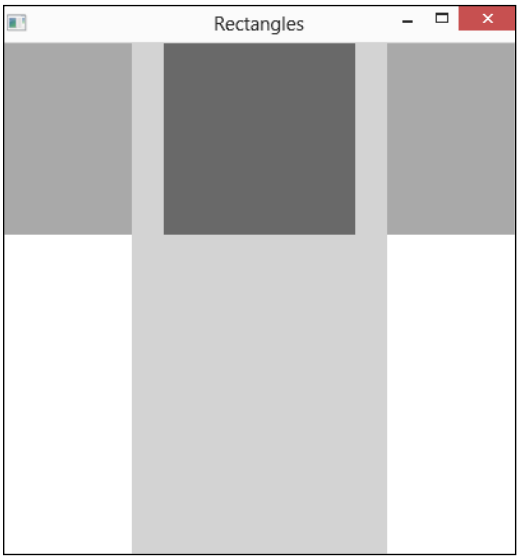

Figure 11-2: Three rectangles displayed with top-center alignment.

As with other layout panes, you can use the `setPadding` method to add padding around the perimeter of the pane. For example, the following line creates a 50-pixel buffer around the edge of the pane:

```
stack.setPadding(new Insets(50));
```

The `setPadding` method accepts an argument of type `Insets`. For more information about the `Insets` class, flip to Table 5-2 in Chapter 5.

You can also add margins to individual nodes within a stack pane. To do so, call the `setMargin` method, passing both the node and an `Insets` object that describes the margin:

```
stack.setMargin(r1, new Insets(25));
```

Using the TilePane layout

The *tile pane layout* is similar to the flow pane layout: It arranges nodes in neat rows and columns, either horizontally or vertically. The crucial difference is that in a tile pane layout, all the cells are the same size. The tile pane layout

calculates the size of the largest node in its child node collection and then uses that size as the size for each cell. This creates a nice grid-like appearance, as shown in Figure 11-3.

By default, a tile pane shows five nodes in each row, using as many rows as necessary to display all its nodes. Thus, the tile pane in Figure 11-3 displays its 12 rectangles in two rows of five and a third row of just two.

If you adjust the size of the tile pane, the number of nodes per row adjusts automatically. For example, Figure 11-4 shows the same tile pane resized so that the 12 rectangles are displayed in three rows of four.

Figure 11-3: A dozen rectangles displayed in a tile pane.

Figure 11-4: A tile pane automatically rearranges its tiles when the pane is resized.

To create a tile pane, you use the `TilePane` class, as I describe in Table 11-2.

Table 11-2 TilePane Constructors and Methods

Constructor	*Description*
`TilePane()`	Creates an empty tile pane.
`TilePane(Node. . . children)`	Creates a tile pane with the specified child nodes.
`TilePane(double hgap, double vgap)`	Creates an empty tile pane with the specified gaps between rows and columns.
`TilePane(Orientation orientation)`	Creates an empty tile pane with the specified orientation. You can specify `Orientation.HORIZONTAL` or `Orientation.VERTICAL`.
`TilePane(double hgap, double vgap, Node. . .children)`	
`TilePane(Orientation orientation, double hgap, double vgap)`	
`TilePane(Orientation orientation, Node. . . children)`	
`TilePane(Orientation orientation, double hgap, double vgap, Node. . . children)`	
Method	***Description***
`ObservableList<Node> getChildren()`	Returns the collection of all child nodes that have been added to the tile pane. The collection is returned as an `ObservableList` type, which includes the methods `add` and `addAll`, which lets you add one or more nodes to the list.

Method	*Description*
`void setHgap(double value)`	Sets the size of the gap that appears between columns.
`void setVgap(double value)`	Sets the size of the gap that appears between rows.
`void setOrientation(Orientation orientation)`	Sets the orientation. Allowable values are `Orientation.HORIZONTAL` and `Orientation.VERTICAL`.
`void setPrefColumns(int value)`	Sets the number of columns preferred for this tile pane.
`void setPrefRows(int value)`	Sets the number of rows preferred for this tile pane.
`void setPrefTileWidth(double value)`	Sets the preferred width for each cell.
`void setPrefTileHeight(double value)`	Sets the preferred height for each cell.
`static void setMargin(Node node, Insets value)`	Sets the margin for a particular node. See Table 5-2 in Chapter 5 for an explanation of the `Insets` class.
`void setMinHeight(double value)`	Sets the minimum height of the tile pane.
`void setMaxHeight(double value)`	Sets the maximum height of the tile pane.
`void setPrefHeight(double value)`	Sets the preferred height of the tile pane.
`void setMinWidth(double value)`	Sets the minimum width of the tile pane.
`void setMaxWidth(double value)`	Sets the maximum width of the tile pane.

(continued)

Table 11-2 *(continued)*

Method	*Description*
`void setPrefWidth(double value)`	Sets the preferred width of the tile pane.
`void setPadding(Insets value)`	Sets the padding around the inside edges of the tile pane. See Table 5-2 in Chapter 5 for an explanation of the `Insets` class.

Here's the code that creates the tile pane shown in Figures 11-3 and 11-4:

```
TilePane tile1 = new TilePane();
tile1.setHgap(10);
tile1.setVgap(10);
tile1.setPadding(new Insets(10,10,10,10));
for (int i=1; i<13; i++)
{
    Rectangle r = new Rectangle(100, 100);
    r.setFill(Color.LIGHTGRAY);
    Label l = new Label("A-" + i);
    StackPane s = new StackPane(r, l);
    tile1.getChildren().add(s);
}
```

As you can see, a `for` loop is used to create 12 labeled rectangles, which are added to the tile pane. In the `for` loop, I first create a 100x100-pixel rectangle and set its color to light gray. Then, I create a label and assign it a text value that consists of the string "`A-`" followed by an integer value. Finally, I create a stack pane and add the rectangle and the label to it. The result is that the label appears on top of the rectangle. I then add the stack pane to the tile pane.

Using the ScrollPane Layout

When a layout is too large to fit in a window, you want to provide horizontal or vertical scroll bars (or both) so the user can scroll to see the entire layout. The easiest way to do that is with a scroll pane. A *scroll pane* envelops a single node with an area that automatically displays horizontal or vertical

scroll bars whenever necessary. Thus, the scroll bars are not displayed if the entire layout fits within the scroll pane. If the layout is taller than the scroll pane, a vertical scroll bar appears. And if the layout is wider than the scroll pane, a horizontal scroll bar appears.

Technically, a scroll pane is not really a layout pane. The `ScrollPane` class is a descendant of the `Control` class, not the `Pane` class. Even so, scroll panes are typically used in conjunction with layout panes to accommodate layouts that are too large to fit onscreen.

Figure 11-5 shows a tile pane similar to the ones shown in the preceding section contained within a scroll pane. As you can see, the tile pane in this example displays two tiles per row, and a vertical scroll bar is visible, allowing the user to scroll to see all the tiles. To create margins around the scroll pane, I added the scroll pane to a stack pane and then set margins on the stack pane.

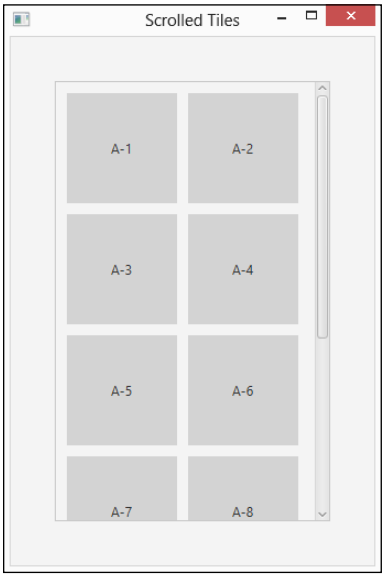

Figure 11-5: A tile pane contained within a scroll pane.

To create a scroll pane, use the `ScrollPane` class depicted in Table 11-3.

Table 11-3 ScrollPane Constructors and Methods

Constructor	***Description***
`ScrollPane()`	Creates an empty scroll pane.
`ScrollPane(Node node)`	Creates a scroll pane with the specified child node.
Method	***Description***
`void getContent(Node node)`	Sets the node contained within this scroll pane.
Note: The allowable values for the following two methods are shown at the end of this table.	
`void setHbarPolicy(ScrollPane.ScrollBarPolicy value)`	Sets the policy for the horizontal scroll bar.
`void setVbarPolicy(ScrollPane.ScrollBarPolicy value)`	Sets the policy for the vertical scroll bar.
`void setPannable(boolean value)`	If `true`, the user can pan the contents of the scroll pane using the mouse. The default is `false`.
`void setMinHeight(double value)`	Sets the minimum height of the tile pane.
`void setMaxHeight(double value)`	Sets the maximum height of the tile pane.
`void setPrefHeight(double value)`	Sets the preferred height of the tile pane.
`void setMinWidth(double value)`	Sets the minimum width of the tile pane.
`void setMaxWidth(double value)`	Sets the maximum width of the tile pane.
`void setPrefWidth(double value)`	Sets the preferred width of the tile pane.
`void setPadding(Insets value)`	Sets the padding around the inside edges of the tile pane. See Table 5-2 in Chapter 5 for an explanation of the `Insets` class.

ScrollBarPolicy Enumeration	*Description*
`ScrollPane.ScrollBarPolicy.ALWAYS`	Always show a scroll bar.
`ScrollPane.ScrollBarPolicy.NEVER`	Never show a scroll bar.
`ScrollPane.ScrollBarPolicy.AS_NEEDED`	Show a scroll bar only when it's needed.

The easiest way to create a scroll pane is to call the `ScrollPane` constructor and pass the node you want scrolled as a parameter, like this:

```
ScrollPane spane = new ScrollPane(tile1);
```

You will most likely also want to set the size constraints for the scroll pane. The following code fixes the width at 250 and allows the layout to determine the height, with a preferred height of 400:

```
spane.setMaxWidth(250);
spane.setMinWidth(250);
spane.setPrefWidth(250);
spane.setPrefHeight(400);
```

If you want, you can set a policy for the vertical and horizontal scroll bars. By default, the scroll bars appear only if necessary. If you want a scroll bar to always appear, even when it isn't necessary, set the policy to `ALWAYS` as in this example:

```
spane.setVBarPolicy(ScrollBarPolicy.ALWAYS);
```

The complete program used to create the scroll pane shown in Figure 11-5 is shown in Listing 11-1.

Listing 11-1: The ScrolledTile Program

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

(continued)

Listing 11-1 ***(continued)***

```
public class ScrolledTile extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }

    @Override public void start(Stage primaryStage)
    {

        TilePane tile1 = new TilePane();                                    →20
        tile1.setHgap(10);
        tile1.setVgap(10);
        tile1.setPrefColumns(2);
        tile1.setPadding(new Insets(10,10,10,10));
        for (int i=1; i<13; i++)                                            →25
        {
            Rectangle r = new Rectangle(100, 100);
            r.setFill(Color.LIGHTGRAY);
            Label l = new Label("A-" + i);
            StackPane s = new StackPane(r, l);
            tile1.getChildren().add(s);
        }

        ScrollPane spane = new ScrollPane(tile1);                           →34
        spane.setMinWidth(250);
        spane.setPrefWidth(250);
        spane.setMaxWidth(250);
        spane.setPrefHeight(400);
        spane.setVbarPolicy(ScrollPane.ScrollBarPolicy.ALWAYS);             →39

        StackPane stack = new StackPane(spane);                             →41
        stack.setMargin(spane, new Insets(40,40,40,40));

        Scene scene = new Scene(stack);                                     →44
        primaryStage.setScene(scene);
        primaryStage.setTitle("Scrolled Tiles");
        primaryStage.show();
    }
}
```

The following paragraphs describe the highlights of this program:

- → **20:** This line and the four lines that follow it create the tile pane, set the horizontal and vertical gaps to 10 pixels, set the preferred width to two columns, and set the padding.
- → **25:** A `for` loop is used to create 12 labeled rectangles and add them to the tile pane.
- → **34:** These lines create the scroll pane and set its size. The width is fixed at 250 pixels. The preferred height is 400 pixels, but the scroll pane's height can grow or shrink as needed to fill the scene.
- → **39:** The vertical scroll bar will always be displayed, even if it isn't necessary.
- → **41:** A stack pane is used here for the sole purpose of providing 40 pixels of margin around the scroll pane.
- → **44:** The stack pane is added to the scene, and the scene is finalized and displayed.

Using the GridPane Layout

The grid pane layout manager lets you arrange GUI elements in a grid of rows and columns. Unlike a tile pane, the rows and columns of a grid pane do not have to be the same size. Instead, the grid pane layout automatically adjusts the width of each column and the height of each row based on the components you add to the panel.

Here are some important features of the grid pane layout manager:

- You can specify which cell you want each component to go in, and you can control each component's position in the panel.
- You can create components that span multiple rows or columns, such as a button two columns wide or a list box four rows high.
- You can tell `GridPane` to stretch a component to fill the entire space allotted to it if the component isn't already big enough to fill the entire area. You can specify that this stretching be done horizontally, vertically, or both.
- If a component doesn't fill its allotted area, you can tell the grid pane layout manager how you want the component to be positioned within the area — for example, left- or right-aligned.

The following sections describe the ins and outs of working with grid pane layouts.

Sketching out a plan

Before you create a grid pane layout, draw a sketch showing how you want the components to appear in the panel. Then slice the panel into rows and columns, and number the rows and columns starting with zero in the top-left corner. Figure 11-6 shows such a sketch for an application that lets a user order a pizza.

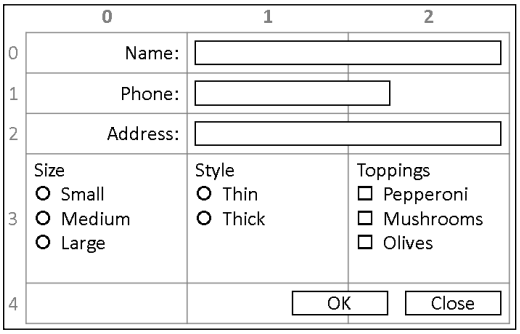

Figure 11-6: Sketching out a panel.

After you have the panel sketched out, list the components, their *x* and *y* coordinates on the grid, their alignment, and whether each component spans more than one row or column. Here's an example:

Component	***x***	***y***	***Alignment***	***Spans***
Label `"Name"`	0	0	Right	
Label `"Phone"`	0	1	Right	
Label `"Address"`	0	2	Right	
Name text field	1	0	Left	2
Phone text field	1	1	Left	2
Address text field	1	2	Left	2
Size radio buttons	0	3	Left	
Style radio buttons	1	3	Left	
Toppings check boxes	2	3	Left	
OK and Close buttons	2	4	Right	

After you lay out the grid, you can write the code to put each component in its proper place.

Creating a grid pane

Table 11-4 shows the most frequently used constructors and methods of the `GridPane` class, which you use to create a grid pane.

Table 11-4 GridPane Constructors and Methods

Constructor	*Description*
`GridPane()`	Creates an empty grid pane.
Method	***Description***
`void add(Node node, int col, int row)`	Adds a node at the specified column and row index.
`void add(Node node, int col, int row, int colspan, int rowspan)`	Adds a node at the specified column and row index with the specified column and row spans.
`void addColumn(int col, Node... nodes)`	Adds an entire column of nodes.
`void addRow(int row, Node... nodes)`	Adds an entire row of nodes.
`<ObservableList> getColumnConstraints()`	Returns the column constraints. For more information, see to Table 11-5.
`<ObservableList> getRowConstraints()`	Returns the row constraints. For more information, see Table 11-6.
`void setColumnSpan(Node node, int colspan)`	Sets the column span for the specified node.
`void setRowSpan(Node node, int colspan)`	Sets the row span for the specified node.
`void setHalignment(Node node, HPos value)`	Sets the horizontal alignment for the node. Allowable values are `HPos.LEFT`, `HPos.CENTER`, and `HPos.RIGHT`.
`void setValignment(Node node, VPos value)`	Sets the vertical alignment for the node. Allowable values are `HPos.BOTTOM`, `HPos.CENTER`, and `HPos.TOP`.

(continued)

Table 11-4 *(continued)*

Method	*Description*
`void setHgap(double value)`	Sets the size of the gap that appears between columns.
`void setVgap(double value)`	Sets the size of the gap that appears between rows.
`static void setMargin(Node node, Insets value)`	Sets the margin for a particular node. See Table 5-2 in Chapter 5 for an explanation of the `Insets` class.
`void setPadding(Insets value)`	Sets the padding around the inside edges of the grid pane. See Table 5-2 in Chapter 5 for an explanation of the `Insets` class.
`void setMinHeight(double value)`	Sets the minimum height of the grid pane.
`void setMaxHeight(double value)`	Sets the maximum height of the grid pane.
`void setPrefHeight(double value)`	Sets the preferred height of the grid pane.
`void setMinWidth(double value)`	Sets the minimum width of the grid pane.
`void setMaxWidth(double value)`	Sets the maximum width of the grid pane.
`void setPrefWidth(double value)`	Sets the preferred width of the grid pane.

To create a basic grid pane, you first call the `GridPane` constructor. Then, you use the `add` method to add nodes to the grid pane's cells. The parameters of the `add` method specify the node to be added, the node's column index, and the node's row index. For example, the following code snippet creates a label, and then creates a grid pane and adds the label to the cell at column 0, row 0:

```
Label lblName = new Label("Name");
GridPane grid = new GridPane();
grid.add(lblName, 0, 0);
```

The typical way to fill a grid pane with nodes is to call the `add` method for each node. However, if you prefer, you can add an entire column or row of nodes with a single call to either `addColumn` or `addRow`. For example, this example creates a label and a text field, and then creates a grid pane and adds the label and the text field to the first row:

```
Label lblName = new Label("Name");
TextField txtName = new TextField();
GridPane grid = new GridPane();
grid.addRow(0, lblName, txtName);
```

If a node should span more than one column, you can call the `setColumnSpan` method to specify the number of columns the node should span. For example:

```
grid.setColumnSpan(txtName, 2);
```

Here, the `txtName` node will span two columns. You use the `setRowSpan` in a similar way if you need to configure a node to span multiple rows.

To control the horizontal alignment of a node, use the `setHalignment` method as in this example:

```
grid.setHalignment(lblName, HPos.RIGHT);
```

Here, the `lblName` node is right-aligned within its column. The `setValignment` method works in a similar way.

Like other layout panes, the `GridPane` class has a host of methods for setting spacing and alignment details. You can use the `setHgap` and `setVgap` methods to set the spacing between rows and columns so that your layouts won't look so cluttered. You can use the `setPadding` and `setMargins` methods to set padding and margins, which work just as they do with other layout panes. And you can set the minimum, maximum, and preferred width and height for the grid pane.

Working with grid pane constraints

You can control most aspects of a grid pane's layouts using methods of the `GridPane` class, but unfortunately, you can't control the size of individual columns or rows. To do that, you must use the `ColumnConstraints` or `RowConstraints` class, as described in Tables 11-5 and 11-6.

Table 11-5 The ColumnConstraints Class

Constructor	*Description*
`ColumnConstraints()`	Creates an empty column constraints object.
`ColumnConstraints(double width)`	Creates a column constraint with a fixed width.
`ColumnConstraints(double min, double pref, double max)`	Creates a column constraint with the specified minimum, preferred, and maximum widths.
Method	***Description***
`void setMinWidth(double value)`	Sets the minimum width of the column.
`void setMaxWidth(double value)`	Sets the maximum width of the column.
`void setPrefWidth(double value)`	Sets the preferred width of the column.
`void setPercentWidth(double value)`	Sets the width as a percentage of the total width of the grid pane.
`void setHgrow(Priority value)`	Determines whether the width of the column should grow if the grid pane's overall width increases. Allowable values are `Priority.ALWAYS`, `Priority.NEVER`, and `Priority.SOMETIMES`.
`void setFillWidth(boolean value)`	If true, the grid pane will expand the nodes within this column to fill empty space.
`void setHalignment(HPos value)`	Sets the horizontal alignment for the entire column. Allowable values are `HPos.LEFT`, `HPos.CENTER`, and `HPos.RIGHT`.

Table 11-6 The RowConstraints Class

Constructor	*Description*
`RowConstraints()`	Creates an empty row constraints object.
`RowConstraints(double height)`	Creates a row constraint with a fixed height.
`RowConstraints(double min, double pref, double max)`	Creates a row constraint with the specified minimum, preferred, and maximum heights.
Method	***Description***
`void setMinHeight(double value)`	Sets the minimum height of the row.
`void setMaxHeight(double value)`	Sets the maximum height of the row.
`void setPrefHeight(double value)`	Sets the preferred height of the row.
`void setPercentHeight (double value)`	Sets the height as a percentage of the total height of the grid pane.
`void setVgrow(Priority value)`	Determines whether the height of the row should grow if the grid pane's overall height increases. Allowable values are `Priority.ALWAYS`, `Priority.NEVER`, and `Priority.SOMETIMES`.
`void setFillHeight(boolean value)`	If `true`, the grid pane will expand the nodes within this row to fill empty space.
`void setValignment(VPos value)`	Sets the vertical alignment for the entire row. Allowable values are `VPos.TOP`, `VPos.CENTER`, and `VPos.BOTTOM`.

To use column constraints to set a fixed width for each column in a grid pane, first create a constraint for each column. Then, add the constraints to the grid pane's constraints collection. Here's an example:

```
ColumnConstriants col1 = new ColumnConstraints(200);
ColumnConstriants col2 = new ColumnConstraints(200);
ColumnConstriants col3 = new ColumnConstraints(200);
GridPane grid = new GridPane();
grid.getColumnConstraints().addAll(col1, col2, col3);
```

One of the most useful features of column constraints is their ability to distribute the width of a grid pane's columns as a percentage of the overall width of the grid pane. For example, suppose the grid pane will consist of three columns and you want them to all be of the same width regardless of the width of the grid pane. The following code accomplishes this:

```
ColumnConstriants col1 = new ColumnConstraints();
col1.setPercentWidth(33);
ColumnConstriants col2 = new ColumnConstraints();
col2.setPercentWidth(33);
ColumnConstriants col3 = new ColumnConstraints();
col3.setPercentWidth(33);
GridPane grid = new GridPane();
grid.getColumnConstraints().addAll(col1, col2, col3);
```

In this example, each column will fill 33 percent of the grid.

Several of the attributes that can be set with column or row constraints mirror attributes you can set for individual nodes via the `GridPane` class. For example, you can set the horizontal alignment of an individual node by calling the `setHalignment` method on the grid pane. Or, you can set the horizontal alignment of an entire column by creating a column constraint, setting its horizontal alignment, and then applying the column constraint to a column in the grid pane.

Examining a grid pane example

Listing 11-2 shows the code for a program that displays the scene I drew for Figure 11-6, and Figure 11-7 shows how this scene appears when the program is run. Figure 11-7 shows that the final appearance of this scene is pretty close to the way I sketched it.

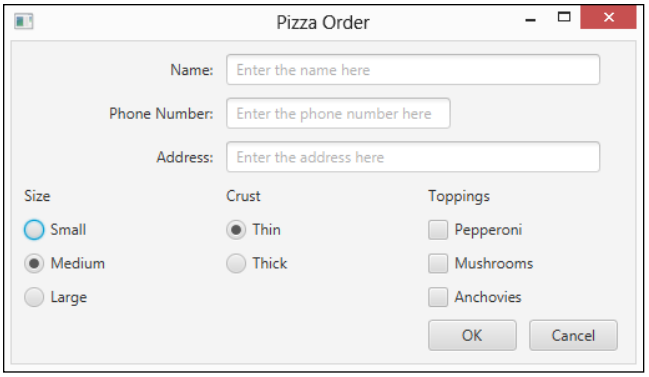

Figure 11-7: The Pizza Order application in action.

Listing 11-2: The Pizza Order Application

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

public class PizzaOrder extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }

    Stage stage;
    TextField txtName;
    TextField txtPhone;
    TextField txtAddress;
    RadioButton rdoSmall;
    RadioButton rdoMedium;
    RadioButton rdoLarge;
    RadioButton rdoThin;
    RadioButton rdoThick;
    CheckBox chkPepperoni;
    CheckBox chkMushrooms;
    CheckBox chkAnchovies;
```

(continued)

Listing 11-2 ***(continued)***

```
@Override public void start(Stage primaryStage)
{
    stage = primaryStage;

    // Create the name label and text field                          →32
    Label lblName = new Label("Name:");
    txtName = new TextField();
    txtName.setMinWidth(100);
    txtName.setPrefWidth(200);
    txtName.setMaxWidth(300);
    txtName.setPromptText("Enter the name here");

    // Create the phone number label and text field                  →40
    Label lblPhone = new Label("Phone Number:");
    txtPhone = new TextField();
    txtPhone.setMinWidth(60);
    txtPhone.setPrefWidth(120);
    txtPhone.setMaxWidth(180);
    txtPhone.setPromptText("Enter the phone number here");

    // Create the address label and text field                       →48
    Label lblAddress = new Label("Address:");
    txtAddress = new TextField();
    txtAddress.setMinWidth(100);
    txtAddress.setPrefWidth(200);
    txtAddress.setMaxWidth(300);
    txtAddress.setPromptText("Enter the address here");

    // Create the size pane                                          →56
    Label lblSize = new Label("Size");
    rdoSmall = new RadioButton("Small");
    rdoMedium = new RadioButton("Medium");
    rdoLarge = new RadioButton("Large");
    rdoMedium.setSelected(true);
    ToggleGroup groupSize = new ToggleGroup();
    rdoSmall.setToggleGroup(groupSize);
    rdoMedium.setToggleGroup(groupSize);
    rdoLarge.setToggleGroup(groupSize);

    VBox paneSize = new VBox(lblSize, rdoSmall, rdoMedium, rdoLarge);
    paneSize.setSpacing(10);
```

```
// Create the crust pane                                          →70
Label lblCrust = new Label("Crust");
rdoThin = new RadioButton("Thin");
rdoThick = new RadioButton("Thick");
rdoThin.setSelected(true);
ToggleGroup groupCrust = new ToggleGroup();
rdoThin.setToggleGroup(groupCrust);
rdoThick.setToggleGroup(groupCrust);

VBox paneCrust = new VBox(lblCrust, rdoThin, rdoThick);
paneCrust.setSpacing(10);

// Create the toppings pane                                       →82
Label lblToppings = new Label("Toppings");
chkPepperoni = new CheckBox("Pepperoni");
chkMushrooms = new CheckBox("Mushrooms");
chkAnchovies = new CheckBox("Anchovies");

VBox paneToppings = new VBox(lblToppings, chkPepperoni,
    chkMushrooms, chkAnchovies);
paneToppings.setSpacing(10);

// Create the buttons                                             →92
Button btnOK = new Button("OK");
btnOK.setPrefWidth(80);
btnOK.setOnAction(e -> btnOK_Click() );

Button btnCancel = new Button("Cancel");
btnCancel.setPrefWidth(80);
btnCancel.setOnAction(e -> btnCancel_Click() );

HBox paneButtons = new HBox(10, btnOK, btnCancel);

// Create the GridPane layout                                     →103
GridPane grid = new GridPane();
grid.setPadding(new Insets(10));
grid.setHgap(10);
grid.setVgap(10);
grid.setMinWidth(500);
grid.setPrefWidth(500);
grid.setMaxWidth(800);

// Add the nodes to the pane                                      →112
grid.addRow(0, lblName, txtName);
grid.addRow(1, lblPhone, txtPhone);
grid.addRow(2, lblAddress, txtAddress);
grid.addRow(3, paneSize, paneCrust, paneToppings);
grid.add(paneButtons,2,4);
```

(continued)

Listing 11-2 ***(continued)***

```
        // Set alignments and spanning                                    →119
        grid.setHalignment(lblName, HPos.RIGHT);
        grid.setHalignment(lblPhone, HPos.RIGHT);
        grid.setHalignment(lblAddress, HPos.RIGHT);
        grid.setColumnSpan(txtName,2);
        grid.setColumnSpan(txtPhone,2);
        grid.setColumnSpan(txtAddress,2);

        // Set column widths                                              →127
        ColumnConstraints col1 = new ColumnConstraints();
        col1.setPercentWidth(33);
        ColumnConstraints col2 = new ColumnConstraints();
        col2.setPercentWidth(33);
        ColumnConstraints col3 = new ColumnConstraints();
        col3.setPercentWidth(33);
        grid.getColumnConstraints().addAll(col1, col2, col3);

        // Create the scene and the stage                                 →136
        Scene scene = new Scene(grid);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Pizza Order");
        primaryStage.setMinWidth(500);
        primaryStage.setMaxWidth(900);
        primaryStage.show();

    }

    public void btnOK_Click()                                             →146
    {

        // Create a message string with the customer information
        String msg = "Customer:\n\n";
        msg += "\t" + txtName.getText() + "\n";
        msg += "\t" + txtPhone.getText() + "\n\n";
        msg += "\t" + txtAddress.getText() + "\n";
        msg += "You have ordered a ";

        // Add the pizza size
        if (rdoSmall.isSelected())
            msg += "small ";
        if (rdoMedium.isSelected())
            msg += "medium ";
        if (rdoLarge.isSelected())
            msg += "large ";
```

```
        // Add the crust style
        if (rdoThin.isSelected())
            msg += "thin crust pizza with ";
        if (rdoThick.isSelected())
            msg += "thick crust pizza with ";

        // Add the toppings
        String toppings = "";
        toppings = buildToppings(chkPepperoni, toppings);
        toppings = buildToppings(chkMushrooms, toppings);
        toppings = buildToppings(chkAnchovies, toppings);
        if (toppings.equals(""))
            msg += "no toppings.";
        else
            msg += "the following toppings:\n"
                + toppings;

        // Display the message
        MessageBox.show(msg, "Order Details");
    }

    public String buildToppings(CheckBox chk, String msg)                →185
    {
        // Helper method for displaying the list of toppings
        if (chk.isSelected())
        {
            if (!msg.equals(""))
            {
                msg += ", ";
            }
            msg += chk.getText();
        }
        return msg;
    }

    public void btnCancel_Click()                                        →199
    {
        stage.close();
    }

}
```

The following paragraphs point out the highlights of this program:

→ **32:** A label and text field are created for the customer's name.

→ **40:** A label and text field are created for the customer's phone number.

→ **48:** A label and text field are created for the customer's address.

→ **56:** A label and three radio buttons are created for the pizza's size. The label and radio buttons are added to a `VBox` named `paneSize`.

→ **70:** A label and two radio buttons are created for the pizza's crust style. The label and radio buttons are added to a `VBox` named `paneStyle`.

→ **82:** A label and three check boxes are created for the pizza's toppings. The label and check boxes are added to a `VBox` named `paneToppings`.

→ **92:** The OK and Cancel buttons are created and added to an `HBox` named `paneButton`.

→ **103:** The grid pane layout is created. The padding and horizontal and vertical gaps are set to 10, and the width is set to range from 500 to 800.

→ **112:** The nodes are added to the pane. The name, phone number, and address labels and text fields are added to rows 0, 1, and 2. Then, the size, crust, and toppings `VBox` panes are added to row 3. Finally, the `HBox` that contains the buttons is added to column 2 of row 4. (***Remember:*** Row and column indexes are numbered from 0, not from 1.)

→ **119:** The column alignment and spanning options are set.

→ **127:** Column constraints are created to distribute the column widths evenly.

→ **136:** The scene is created, and the stage is displayed.

→ **146:** The `btnOK_Click` method is called when the user clicks OK. This method creates a summary of the customer's order and displays it using the `MessageBox` class.

→ **185:** `buildToppings` is simply a helper method that assists in the construction of the message string.

→ **199:** The stage is closed when the user clicks the Close button.