CHAPTER S18

GUI Programming—Component Layout, Additional GUI Components

Objectives

- Know GUI design basics.
- Know the benefits of using a container that has a hidden layout manager.
- Understand `FlowPane` layout details.
- Understand `HBox` and `VBox` layout details.
- Understand `BorderPane` layout details.
- Understand `TilePane` and `GridPane` layout details.
- Learn how to embed layouts inside other layouts.
- Implement `TextArea` components for text that spans more than one line.
- Implement a `CheckBox` component for yes/no user input.
- Implement `RadioButton` and `ChoiceBox` or `ComboBox` components when the user needs to choose a value from among a list of predefined values.
- Become familiar with additional GUI components such as menus, scroll panes, and sliders.

Outline

18.1 Introduction
18.2 GUI Design and Layout Management
18.3 `FlowPane` and `GridPane` – Competing Layout Philosophies
18.4 Externally Driven `VBox` with Image
18.5 `BorderPane`
18.6 `TilePane`
18.7 Tic-Tac-Toe Example
18.8 Problem Solving: Winning at Tic-Tac-Toe (Optional)
18.9 Embedded Panes
18.10 MathCalculator Program
18.11 `TextArea` Component
18.12 `CheckBox` Component
18.13 `RadioButton` Component
18.14 `ComboBox` Component

18.15 Job Application Example
18.16 More GUI Components

18.1 Introduction

This is the second chapter in our two-chapter sequence on GUI programming. In the previous chapter, Chapter S17, you learned JavaFX GUI basics. You learned that a window's frame is a stage and the window's backing is a scene. You learned that a scene always contains exactly one node – the root node of the scene graph. In the simplest cases, the scene's one node is a simple component, like a `Label`, In more complex cases, that one node is a container, like a `FlowPane`, which can hold more than one component.

You learned that a `FlowPane` arranges its components like a word processor arranges Romance-language words on a page. It adds components left to right on the current row until it runs out of space on that row. Then it wraps around to the left side of the next row and continues from there. If the user changes the size of the window, the number of components one each row typically changes, and this can alter the layout in undesirable ways. To suppress or prevent undesirable layout alterations, we imposed explicit constraints by inserting methods like `setPrefColumnCount(10)`, or `setResizable(false)`.

In fact, the "undesirable layout alterations" we suppressed in Chapter S17 were manifestations of capabilities built into `Pane` containers that enable them to preserve layout arrangements. The automatic layout adjustments were undesirable only because (for simplicity) we kept using the same `FlowPane` container for every application. In this chapter you'll see that there are many other types of `Pane` containers, and if you select the type of container judiciously, the adjustments that occur automatically as users alter window size become a huge benefit.

In this chapter, instead of trying to suppress or prevent automatic layout adjustments, we will use them and celebrate them. We will contrast the `FlowPane` container with the `GridPane` container. And we will show you how to use several other types of containers – `HBox` and `VBox`, `BorderPane`, and `TilePane`.

In the previous chapter, we limited our discussion of components to `Labels`, `Buttons`, and `TextFields`. In this chapter, you will learn how to use several other types of components – `TextArea`, `CheckBox`, `RadioButton`, and `ComboBox`. Also, you will learn how to improve the visual appeal by applying various layout techniques to your windows' components. For example, you will learn how to insert layouts inside other layouts and put different layouts in different places in the scene.

For an preview of what you'll be learning, see Figure 18.1. Note the combo box, radio button, and check box components. Also, note how the radio buttons are grouped in the center, the check buttons are grouped on the right and the Next and Cancel buttons are grouped at the bottom center. In this chapter, you'll learn how to make such groupings, and you'll learn how to position them appropriately.

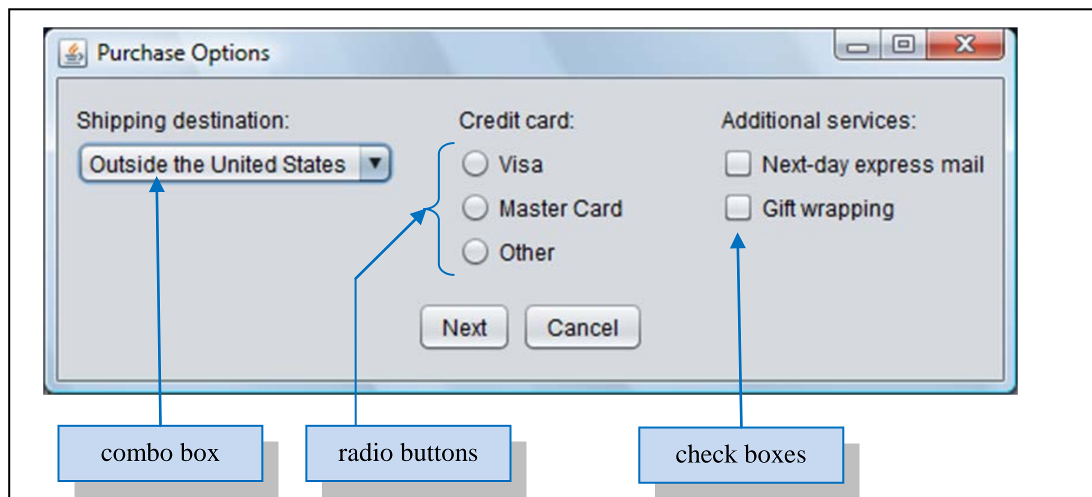


Figure 18.1 Example window that uses radio buttons, check boxes, and a combo box

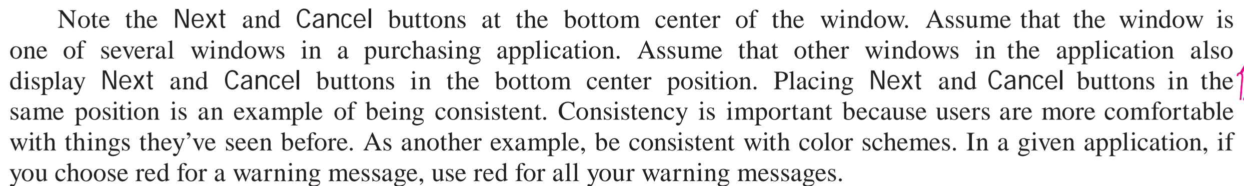
18.2 GUI Design and Layout Management

With text-based programs, it's relatively easy to tell users what to do. As a programmer, you just provide text instructions and the user enters input when prompted to do so. With GUI programs, it's more difficult to tell users what to do. As a programmer, you display a window with various components, set up handlers, and then wait for the user to do something. It's important that your display be easy to understand; otherwise your users won't know what to do. To make your display easy to understand, follow these guidelines:

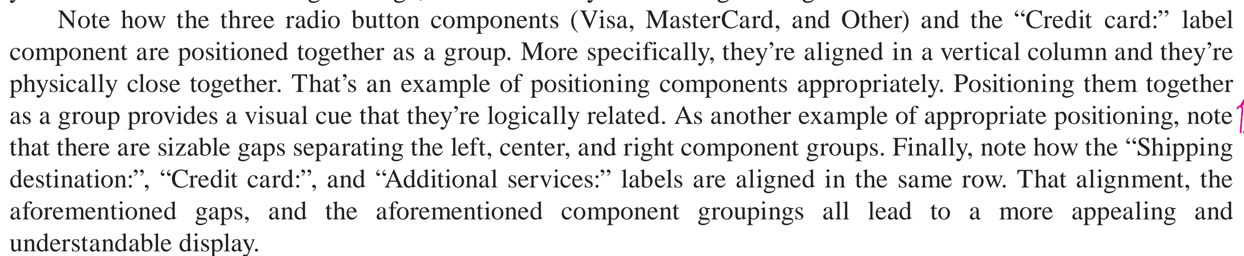
- Choose the right components.
- Be consistent.
- Position components appropriately.

GUI Design Basics

In Figure 18.1, note the small circles next to Visa, MasterCard, and Other. Those circles are radio button components (we describe radio buttons in Section 18.13). Using radio buttons for the credit card options is an example of choosing the right component. Radio buttons provide implicit instructions to the user about how to proceed. Most users recognize small circles as radio buttons, and when they see them, they know to click one of them with their mouse.



Note the `Next` and `Cancel` buttons at the bottom center of the window. Assume that the window is one of several windows in a purchasing application. Assume that other windows in the application also display `Next` and `Cancel` buttons in the bottom center position. Placing `Next` and `Cancel` buttons in the same position is an example of being consistent. Consistency is important because users are more comfortable with things they've seen before. As another example, be consistent with color schemes. In a given application, if you choose red for a warning message, use red for all your warning messages.



Note how the three radio button components (Visa, MasterCard, and Other) and the "Credit card:" label component are positioned together as a group. More specifically, they're aligned in a vertical column and they're physically close together. That's an example of positioning components appropriately. Positioning them together as a group provides a visual cue that they're logically related. As another example of appropriate positioning, note that there are sizable gaps separating the left, center, and right component groups. Finally, note how the "Shipping destination:", "Credit card:", and "Additional services:" labels are aligned in the same row. That alignment, the aforementioned gaps, and the aforementioned component groupings all lead to a more appealing and understandable display.

Pane Layout Management

As you now know, positioning components appropriately is an important part of GUI design. In the old days, positioning components was a tedious, manual process. Programmers would spend hours calculating and devising formulas for the space needed for each component and the pixel coordinate positions for each component, as functions of window size. Today, programmers are freed from that tedium by a hidden layout manager associated with each types of pane. This hidden layout manager does the appropriate calculations in the background, automatically.

In general, each pane's layout manager's goal is to arrange components neatly and adjust that arrangement appropriately if the user re-sizes the window. Usually, the neatness goal equates to making sure components are aligned and making sure components are appropriately spaced within the pane. For example, in Figure 18.1, hidden layout managers are responsible for aligning the left components, aligning the middle components, aligning the right components, and for spacing the three component groups across the width of the window.

If a user adjusts a window's size, the Java virtual machine (JVM) asks each pane's hidden layout manager to recalculate the pixel coordinate positions for all of that pane's components. All this takes place automatically without any intervention on the programmer's part. How convenient! Give thanks for those hidden layout managers!

As we have said, there are different types of panes, and their respective layout managers employ different

strategies for positioning components within their panes. See the table in Figure 18.2. It describes the layouts for several different panes in the JavaFX API library.

Layout Type	Description
BorderPane	Splits container into five regions—top, bottom, left, right, and center. Allows one component per region.
FlowPane	Allows components to be added left to right, flowing to next row as necessary.
GridPane	Gives the programmer direct control over each component's row and column location. Grid cells can vary in size, with column width and row height set by largest contained component.
HBox and VBox	Allows components to be arranged in either a single row or a single column.
TilePane	Splits container into a rectangular grid of equal-sized cells. Allows one component per cell.

Figure 18.2 Several of the more popular layouts

In the previous chapter, we used the simplest type of layout – a `FlowPane`. The `FlowPane` is useful for some situations, but we'll often need other layouts for other situations. In this chapter, we'll compare the `FlowPane` layout with a `GridPane` layout, and we'll describe `VBox`, `HBox`, `BorderPane`, and `TilePane` layouts. Those are popular layouts, so you'll want to know what they do. Layout classes are in the `javafx.scene.layout` package, so you must import that package.

We assign a particular layout to a window when we instantiate a `Scene`, like this:

```
stage.setScene(new Scene(pane, WIDTH, HEIGHT));
```

In this code template, `pane` could be an instance of any of the `Pane` layout classes, like `FlowPane`, `HBox`, `VBox`, `TilePane`, `BorderPane`, or `GridPane`.

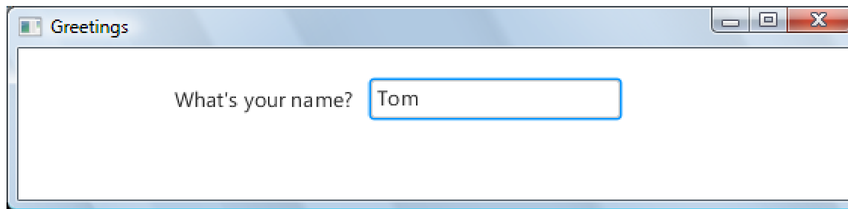
18.3 FlowPane and GridPane – Competing Layout Philosophies

In the previous chapter, we wanted to present GUI basics without getting bogged down in layout details. So we chose a simple `FlowPane` layout, which didn't require much explanation. We just used it and didn't dwell on particulars. Now it's time to explain the particulars, so you can take advantage of its functionality more fully. A `FlowPane` is very easy to use, but it's sometimes frustrating, because you don't have much control over what it does. If you find this to be a problem, you can switch to a `GridPane`, which gives you more direct control over where things go. This section's example will illustrate the trade-off.

FlowPane Layout Mechanism

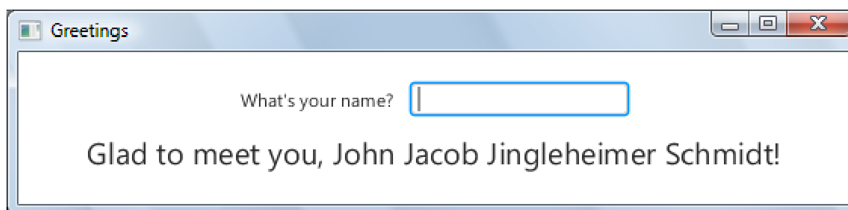
The `FlowPane` class implements a simple one-compartment layout scheme that allows multiple components to be inserted into that compartment. When a component is added to the compartment, it is placed to the right of any components that were previously added to the compartment. If there is not enough room to add a component to the right of the previously added components, the new component is placed on the next line (i.e., it "flows" to the next line). Now let's go back and re-visit the previous chapter's Greeting program, described and used in Sections 17.8 through 17.10.

That program prompts the user to enter his/her name and prints a personalized greeting after the user presses enter. We'll show you a sample session that starts with a wider (WIDTH = 550) window and a short name. Here's what the program displays after the user enters Tom:

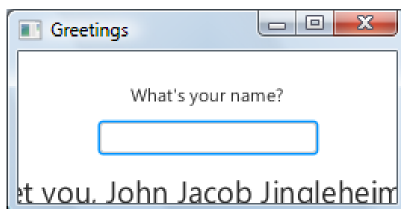


And here's what the program displays after the user presses enter:

If the user enters a longer name, like John Jacob Jingleheimer Schmidt, the greeting label can't fit on the first line, so it wraps to the next line:



If the programmer specifies a narrower window (WIDTH = 250) or the user manually resizes the window to that width, the text box can no longer fit on the first line, so it wraps to the next line, and the response wraps to a third line, like this:



Alignment

By default, the `FlowPane` manager positions its components using top-left alignment. But in this example, the computer tries to center everything horizontally and maintain a constant spacing down from the top. That's because the `createContents` method in the Greeting program contains this explicit alignment statement:

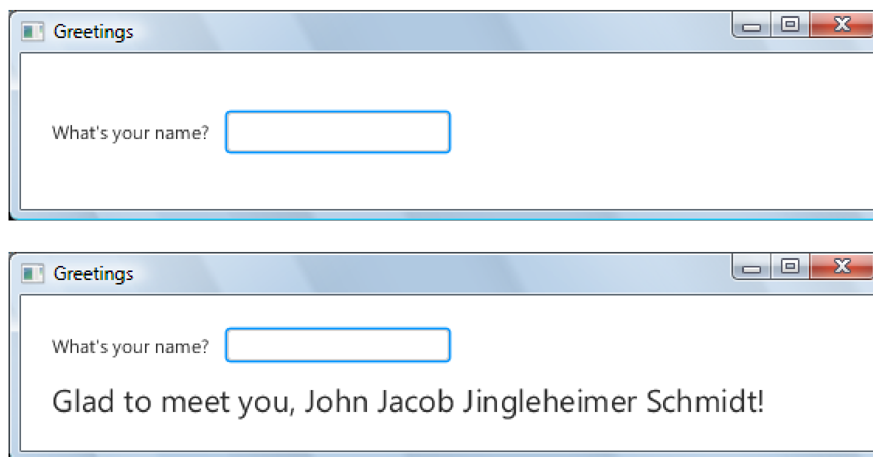
```
pane.setAlignment(Pos.TOP_CENTER);
```

It tries to produce this alignment, but sometimes the result is not satisfactory. In the last image above, notice how the third line jams up against the line above, and notice how it pushes as far as possible to the left but still doesn't all fit on the right.

However, if you click on a corner of the screen image and drag it away from the center, you can enlarge the image to where all the text in the last line does fit, and the spacing between lines becomes uniform. In fact, by using the mouse to adjust the window size, you can reproduce any of the images above, regardless of the program's width and height specifications. So the layout manager manages not only the initial layout. It continues to manage dynamically if a user chooses to alter the window's size and shape with a mouse. If you'd like to change the alignment, in the argument of the `setAlignment` method call, after the `Pos.`, change the alignment constant to another one of these:

```
TOP_LEFT, TOP_CENTER, TOP_RIGHT,
CENTER_LEFT, CENTER, CENTER_RIGHT,
BOTTOM_LEFT, BOTTOM_CENTER, BOTTOM_RIGHT
```

Here's what our Greeting program displays with `CENTER_LEFT` alignment:



`GridPane` Layout

Go back and look at the first four Greeting program displays above. You'll see that a `FlowPane` layout manager likes to move components around as the size of user-entered data and/or the window changes. This can be disconcerting. The last two displays above suggest that this disconcerting movement decreases with left alignment, and decreases further with (the default) top-left alignment. However, if you think about the second and third displays, you'll realize that even with the most stable top-left alignment, the point at which wrap-around occurs still changes with data length and window size.

If you know enough about your data sizes and you make the window big enough to show everything you want, you can stabilize your windows by switching to a different layout – the layout produced by a `GridPane`. A `GridPane` splits the container into a rectangular grid, and you explicitly specify the column and row for each component. This keeps components from changing column and row positions.

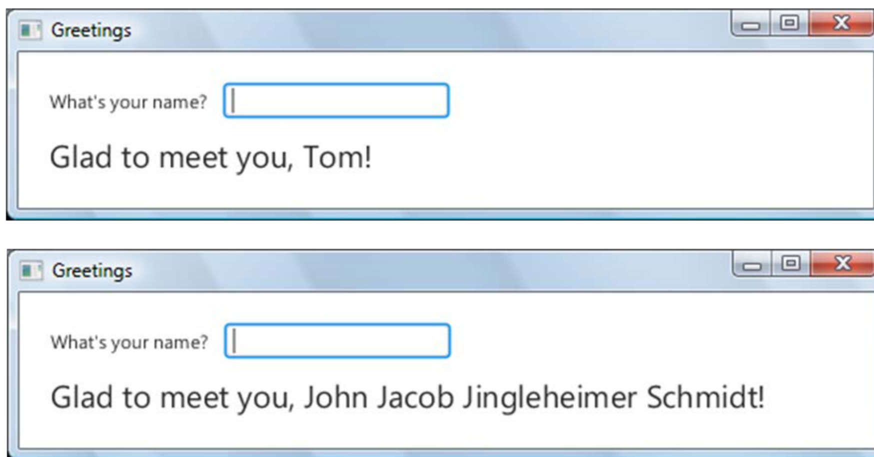
A `GridPane` makes each column's width and each row's height big enough to hold the largest component in that column or row. This may cause the locations of column and row boundaries to change with changes in component size, but another feature may help you solve this problem – you can allocate more than one cell to any component. Assuming the layout contains some fixed-sized components, you may be able to use these to establish column widths and row heights. Then you can allocate enough cells to each variable-size component so that variable-size components never get big enough to push their boundaries.

To see how this strategy works, let's apply it to the Greeting program. First, you will need to import `javafx.scene.layout.GridPane`. Substitute `GridPane` for all occurrences of `FlowPane`, and delete the `setAlignment` statement. Then replace the single `pane.getChildren().addAll` method call with these three method calls:

```
pane.add(namePrompt, 0, 0); // (component, colIndex, rowIndex)
pane.add(nameBox, 1, 0);    // (component, colIndex, rowIndex)
// (component, colIndex, rowIndex, numberOfColumns, numberOfRows)
pane.add(greeting, 0, 1, 3, 1);
```

Instead of just putting the three components one after another in a continuous sequence, these three method calls put each component in a particular place in the grid. (You could change the order of these three statements and it wouldn't matter.) The first parameter is the component. The second parameter is the index of the grid column where you want that component to go. The third parameter is the index of the grid row where you want that component to go. The optional fourth and fifth parameters are the number of columns and rows, respectively that you allocate to that component, if you need for that component to use more than one cell. Thus, the greeting component goes into column zero and row 1, and it has three columns and one row allocated to it. In other words, the greeting component may use all the space it needs in the first, second, and third columns in the second row.

Assuming the same window size as that in the first three displays in this section, here is what the `GridPane` version of the Greeting program produces for each of our two inputs:



The first two column widths are set by the default widths of the name prompt and the name box, respectively. You can see that greeting takes all the space in both these columns plus a lot more. Since we allocated three columns for the greeting, the third column becomes wide enough to handle the greeting overflow. If we had allocated just two columns to the greeting, the second column would have expanded to handle this very long name and then shrunk back to the original text box width if we had subsequently entered a short name like Tom.

If you don't have enough fixed-size components to establish all column widths and row heights, `GridPane` provides another alternative. Starting at column zero, you can specify successive column widths explicitly, with statements like these:

```
pane.getColumnConstraints().add(new ColumnConstraints(125));
pane.getColumnConstraints().add(new ColumnConstraints(150));
```

The `ColumnConstraints` class is in the `javafx.scene.layout` package. These two statements force the first column to be 125 pixels wide and the second column to be 150 pixels wide, regardless of the sizes of components in those columns. Using a zero-parameter `ColumnConstraints` constructor lets the `GridPane`

do what it wants for that column. You can do a similar thing with rows. Just substitute `Row` for `Column` in the above statements.

18.4 Externally Driven `VBox` with Image

A `FlowPane` is popular because it's easy to use. Just add components and the container accumulates them like words on a page of text. It starts at the top, filling each line from left to right, and then moves automatically to the next line. If everything's in one row, there's an even simpler alternative, an `HBox`, where H means "horizontal." If everything's in one column, there is another simple kind of pane, a `VBox`, where V means "vertical." A `VBox` arranges items vertically down one column. You'll see `HBox` later, in another context. The example in this section employs a `VBox`.

Example – a Dance Recital Poster

Suppose you want to implement a program that prints a poster with details about an upcoming dance recital. You want your program to be flexible, so it can be reused later on for other dance recital events. Thus, rather than hard-coding the event details in print statements, the program prompts the user for event details, and then prints them in a pleasing format as a poster. The poster should show the following five items, one above the other:

- The name of the dance recital performance.
- A promotional image, like a photograph of the featured performer.
- The dance recital's date.
- The dance recital's time.
- The dance recital's location.

Given this input:

Sample session:

```
Name of the performance: Swan Lake
Image file: dancer.jpg
Dance recital's date: February 14, 2014
Dance recital's time: 7:30 pm
Dance recital's venue: Park University, Alumni Hall
```

Figure 18.3 shows what we're talking about.

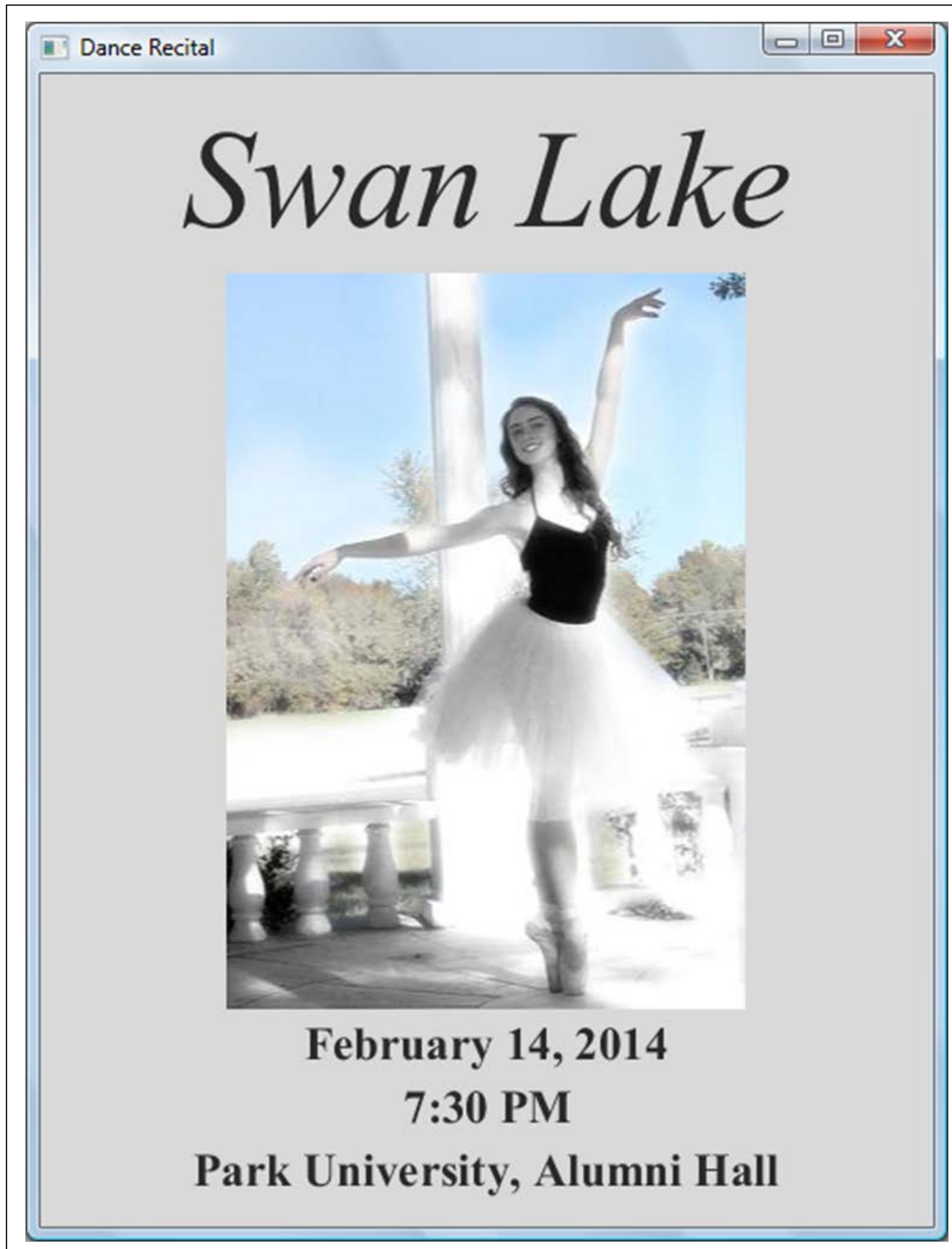


Figure 18.3 Representative output for the DanceRecital Program
The pictured ballerina, Jessica Sapenaro, is a dance teacher and a Park University computer science student.

Figure 18.4 contains the dance recital program's driver. The driver first instantiates an array of strings called `data`. For each string in the array, the driver prompts the user to enter the relevant information. It populates the array with user inputs for the name of the performance, an image, the date, the time, and the venue. Then the driver calls the two-parameter version of driven class's class method, `launch`, using the driven class's class as the first argument.

```
/***************************************************************
* DanceRecitalDriver.java
* Dean & Dean
*
* This driver collects & stores dance recital information.
***************************************************************/

import java.util.Scanner;

public class DanceRecitalDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String[] data = new String[5];

    System.out.print("Name of the performance: ");
    data[0] = stdIn.nextLine();
    System.out.print("Image file: ");
    data[1] = stdIn.nextLine();
    System.out.print("Dance recital's date: ");
    data[2] = stdIn.nextLine();
    System.out.print("Dance recital's time: ");
    data[3] = stdIn.nextLine();
    System.out.print("Dance recital's venue: ");
    data[4] = stdIn.nextLine();

    DanceRecital.launch(DanceRecital.class, data);
  } // end main
} // end class DanceRecitalDriver
```


Figure 18.4 Driver of `DanceRecital` class in Figures 18.5a and 18.5b

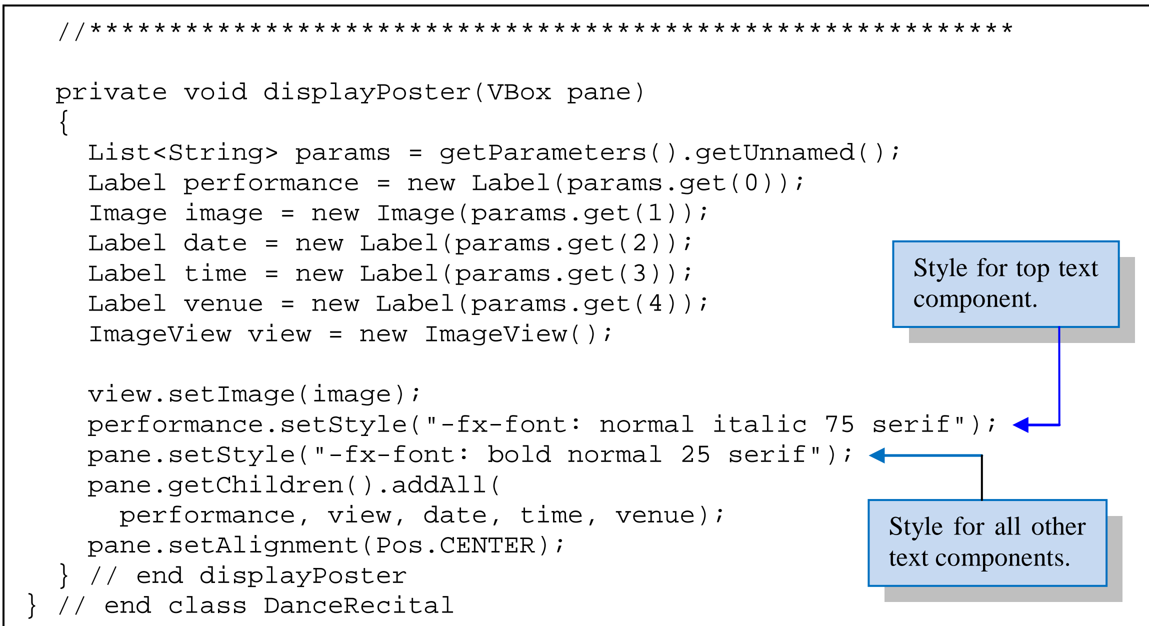
Now look at the first part of the driven class in Figure 18.5a. Notice that it `extends Application`. The `launch` method creates an instance of the driven class, but it stores the data in a `static` inner class called `Parameters`, so there can be only one data set at any one time. After initialization, the `Application` object calls the `start` method, which our class must override (because `Application`'s `start` method is `abstract`).

```
/*************************************************************
* DanceRecital.java
* Dean & Dean
*
* This prints previously stored dance recital information.
*************************************************************/

import java.util.List;
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.image.*; // Image, ImageView;
import javafx.scene.control.Label;
import javafx.scene.paint.Color;
import javafx.geometry.Pos;

public class DanceRecital extends Application
{
  @Override
  public void start(Stage stage)
  {
    VBox pane = new VBox(5);
    Scene scene = new Scene(pane, 480, 620);

    stage.setTitle("Dance Recital");
    stage.setScene(scene);
    scene.setFill(new Color(0.85, 0.85, 0.85, 1.0));
    displayPoster(pane);
    stage.show();
  } // end start
```


Figure 18.5a The `DanceRecital` class – part A

The `start` method creates a `VBox` pane, which defines a vertical layout strategy. The next three statements are standard stuff, which you've seen before. At this point, if we did nothing more except `stage.show()`, we would see just a labeled window with a white pane. But Figure 18.4 shows a light gray background. The get this, we use a color object in a `setFill` method call to paint the scene light gray. As indicated earlier, the first three numbers in the `Color` constructor are the fractional intensities of red, green, and blue, respectively, and the fourth number indicates that this new color is completely opaque. Then we call the `displayPoster` method, which acts like a `createContents` method.

Now look at Figure 18.5b. This is what makes the window look like a dance recital poster. The first statement employs a `getParameters().getUnnamed()` method call chain to retrieve the previously supplied array-of-strings data in the form of a `List`. The `getUnnamed` method filters out any map entries (see Chapter 10) that `Parameters` might contain. The next five statements get each string from the list and convert it into an appropriate GUI object. The only tricky part of these operations is converting the image. First we convert the image's url string to an `Image` object. Then we instantiate an `ImageView`. Then we use a `setImage` method call to load the `Image` into that `ImageView`. We could use `ImageView` methods to process the image in various ways, but here we just take the image as it is.



```
  //***********************************************************

  private void displayPoster(VBox pane)
  {
    List<String> params = getParameters().getUnnamed();
    Label performance = new Label(params.get(0));
    Image image = new Image(params.get(1));
    Label date = new Label(params.get(2));
    Label time = new Label(params.get(3));
    Label venue = new Label(params.get(4));
    ImageView view = new ImageView();

    view.setImage(image);
    performance.setStyle("-fx-font: normal italic 75 serif");
    pane.setStyle("-fx-font: bold normal 25 serif");
    pane.getChildren().addAll(
      performance, view, date, time, venue);
    pane.setAlignment(Pos.CENTER);
  } // end displayPoster
} // end class DanceRecital
```


Figure 18.5b The `DanceRecital` class – part B

The next two statements use `setStyle` method calls to establish the fonts we want for the poster's text. The particular CSS style format used here is an abbreviated format that simultaneously specifies four different properties in this order: font weight, font slope, font size, and font type. In the first of these two statements, the calling object is one of the components – the label at the top of the poster. It specifies normal weight, italicized, 75 point size, and serif characters. In the second of these two statements, the calling object is the `VBox` pane – the container that holds all the components. This specifies font for all contained components that do not have overriding specifications like the performance object does. It specifies bold weight, normal slope, 25 point size, and serif characters for all of the text below the image. The `addAll` method call puts all five components onto the pane, and the `setAlignment` method call centers everything in both directions.

For each GUI program in the previous chapter, we drove the code that generates the GUI directly from a `main` method within the class that generates the GUI. Similarly, we might have put our driver's data-entry code into a `main` method in the driven class. Then, we could have replaced the driver's two-parameter `launch` method call with the simpler one-parameter method call:

```
launch(data);
```

But to promote modularity, to show that a GUI class can be called from a different non-GUI class, and to show one way to actually do that, we extracted the keyboard entry from the GUI display and put that keyboard entry into a separate driver class. Such separation makes it possible to use the code in either class in other contexts. We could transmit the keyboard input to a file or remote site, or we could drive the GUI from a file or remote site.

You may have noticed that the `VBox` that's part of the title of this section didn't get as much attention as we have given to `FlowPane` or even to `GridPane`. That's because it doesn't need much attention. It just arranges components from top to bottom in order of entry by `getChildren().add` or `getChildren().addAll` method calls. Its simplicity is its greatest virtue.

18.5 BorderPane

`FlowPane` and `VBox` layouts are nice because they allow you to enter components one after another and let the computer handle the details. `GridPane` is nice because it lets you specify the details. But sometimes you want something that's more sophisticated than `FlowPane` or `VBox` and more automatic than `GridPane`. You may want a `BorderPane`.

BorderPane Regions

A `BorderPane` is particularly appropriate for windows that need components near their edges. It's common to put a title near the top edge of a window. It's common to put a menu near the left edge of a window. It's common to put buttons near the bottom edge of a window. The `BorderLayout` accommodates all those situations by splitting up its container into five *regions,* or compartments. Four of the regions are near the edges and one is in the center. You access the four edge regions with the names—top, bottom, left, and right. Note the regions' positions in Figure 18.6.

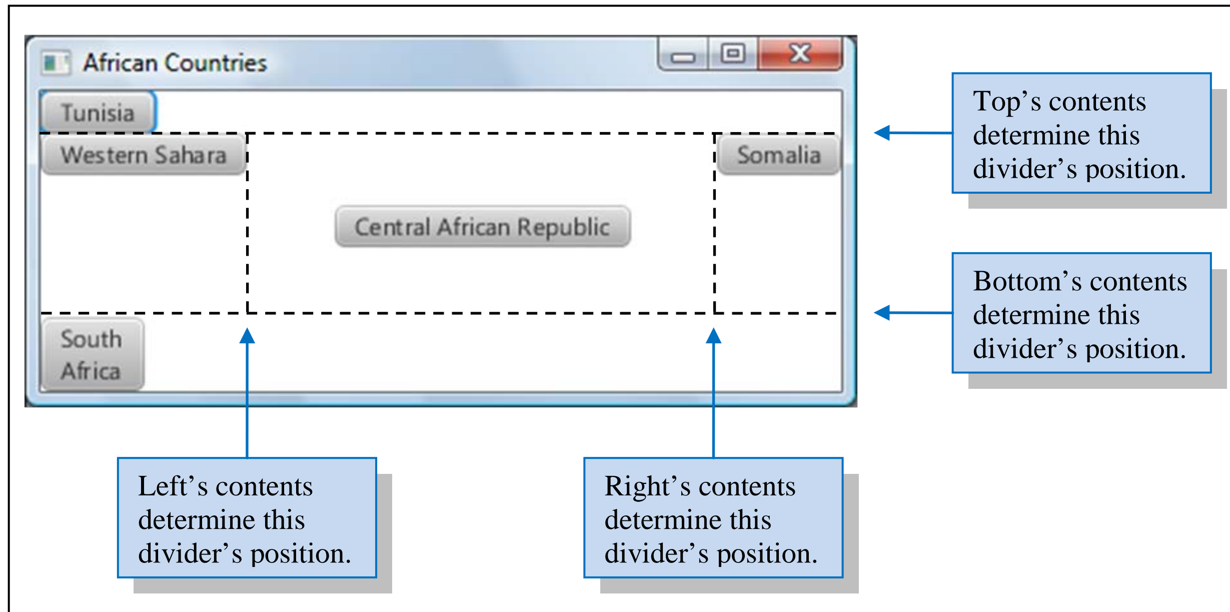


Figure 18.6 BorderPane regions and default alignment within those region
The Tunisia button in the top region is left aligned. The Western Sahara button in the left region is top aligned, the Central African Republic button in the center region is center aligned. The Somalia button in the right region is right aligned. The South Africa button in the bottom region is left aligned.

Be aware that the dashed lines don't appear on the actual display. We've drawn them in to show you the region boundaries. Also, note that to get this particular display, we did not use the one-parameter `Scene` constructor in the method call:

```
stage.setScene(new Scene(pane));
```

Instead, we used the three-parameter `Scene` constructor, like this:

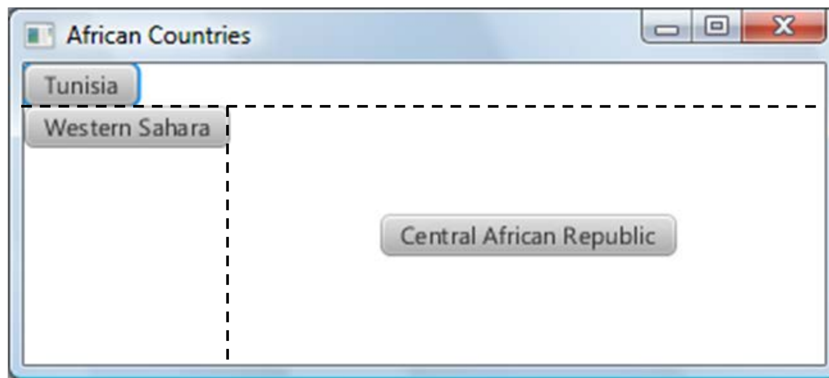
```
stage.setScene(new Scene(pane, 400, 150));
```

The second parameter, 400, sets the window's width at 400 pixels, and the third parameter, 150, sets the window's height at 150 pixels. If we had used the one-parameter `Scene` constructor, the window would have shrunk down to jam all the components together, and the default alignments would not be as obvious.

The sizes of the five regions are determined at runtime, and they're based on the contents of each region. Thus, if the left region contains a long label, the layout manager attempts to widen the left region. Likewise, if the left region contains a short label, the layout manager attempts to narrow the left region.

If an outer region is empty, it collapses so that it does not take up any space. But what exactly happens during the collapse? Each outer region controls only one dividing line, so only one dividing line moves for each collapsed region. Figure 18.6 shows you that the left region's dividing line is the boundary between left and center, the top region's dividing line is the boundary between top and below, and so on. So if the top region is empty, the top dividing line moves all the way up to the top border, and the left, center, and right regions all expand upward. What

happens if the right and bottom regions are both empty? The right region being empty causes the right dividing line to move all the way to the right border. The bottom region being empty causes the bottom dividing line to move all the way down to the bottom border. Here's the resulting layout:



Once again, the dashed lines don't appear on the actual window. And again, we constructed `Scene` with width and height specified as 400 and 150, respectively.

What happens if the center region is empty? The center region doesn't control any of the dividing lines, so nothing happens.

Adding Components

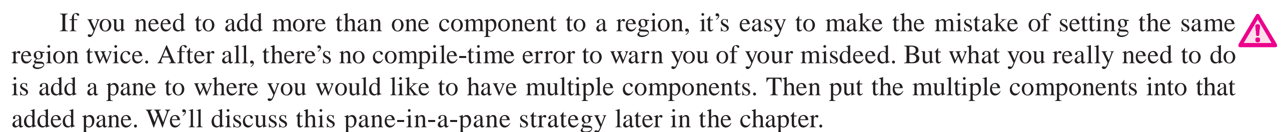
To add a Tunisia button to the top region of a `BorderPane`, do this:

```
BorderPane pane = new BorderPane();
...
pane.setTop(new Button("Tunisia"));
```

The `BorderPane` layout provides five regions/compartments—top, bottom, left, right, and center—in which to insert components.

With a `FlowPane`, you can add as many components as you like. With a `BorderPane`, you can add only five components total, one for each of the five regions. If you add a component to a region that already has a component, then the new component overlays the old component. Thus, in executing the following lines, the Somalia button replaces the Djibouti button:

```
pane.setRight(new Button("Djibouti"));
pane.setRight(new Button("Somalia"));
```



If you need to add more than one component to a region, it's easy to make the mistake of setting the same region twice. After all, there's no compile-time error to warn you of your misdeed. But what you really need to do is add a pane to where you would like to have multiple components. Then put the multiple components into that added pane. We'll discuss this pane-in-a-pane strategy later in the chapter.

AfricanCountries Program with Buttons

Let's put this `BorderPane` material into practice by using it within a complete program. In our AfricanCountries program, we add African-country buttons to the five regions of a `BorderPane` layout. Figure 18.7 shows the program's output. The five buttons you see are centered in the five regions. There is spacing between adjacent regions, and there is padding around the outside.

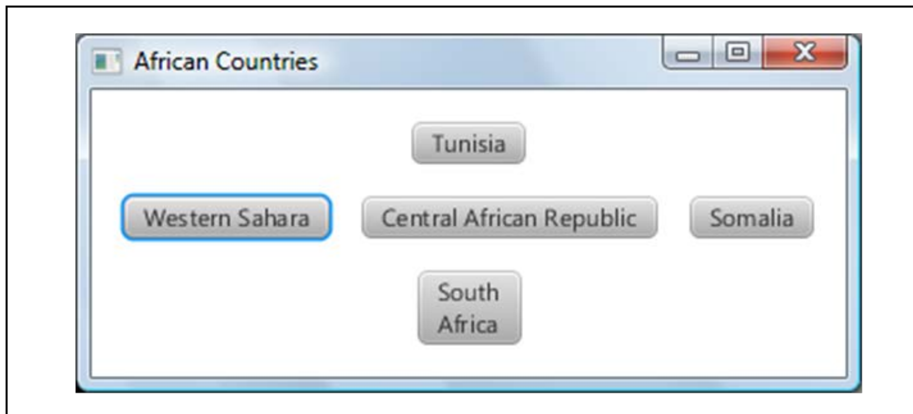


Figure 18.7 Display generated by the AfricanCountries program in Figure 18.8

Skim through the AfricanCountries program listing in Figure 18.8. Most of the code is straightforward. But notice that now we use the one-parameter `Scene` constructor to let the computer optimize the window size. Also notice the `for` loop near the bottom. This loop steps through the `BorderPane`'s five "children" (the five regions). The `setAlignment` method call centers the alignment for each child. The `setMargins` method call puts an 8-pixel blank border around whatever is in each region. After this loop, the `setPadding` method call puts an additional 8-pixel wide blank border around everything. In this case, the two `Insets` constructors have the same effect. The four-parameter constructor provides the alternative of specifying different border widths for top, right, bottom, and left, respectively.

Finally, it's worth noting that wherever we used `Button`, you could substitute any other component that is a subclass of the `Node` class. Familiar examples are `Label`, `TextField`, and `ImageView`.

```
/**************************************************************
* AfricanCountries.java
* Dean & Dean
*
* This program shows component layout for a BorderPane.
**************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.*;          // Scene, Node
import javafx.scene.layout.BorderPane;
import javafx.scene.control.*;  // Button, Label
import javafx.geometry.*;       // Pos, Insets

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

  //***********************************************************

  @Override
  public void start(Stage stage)
  {
    BorderPane pane = new BorderPane();

    stage.setTitle("African Countries");
    stage.setScene(new Scene(pane));
    pane.setTop(new Button("Tunisia"));
    pane.setLeft(new Button("Western Sahara"));
    pane.setCenter(new Button("Central African Republic"));
    pane.setRight(new Button("Somalia"));
    pane.setBottom(new Button("South\nAfrica"));
    for (Node child : pane.getChildren())
    {
      BorderPane.setAlignment(child, Pos.CENTER);
      BorderPane.setMargin(child, new Insets(8));
    }
    pane.setPadding(new Insets(8, 8, 8, 8));
    stage.show();
  } // end start
} // end class AfricanCountries
```

Figure 18.8 AfricanCountries program

18.6 `TilePane`

The `BorderPane` layout's partitioning scheme (top, bottom, left, right, center) works well for many situations,

but not for all situations. Often, you'll need to display information using a table format; that is, you'll need to display information that's organized by rows and columns. The `BorderPane` doesn't work well for table formats. If the table is large, and you use a `GridPane`, specifying every individual location becomes tiresome. For a regular pattern, a `TilePane` is easy going, and it may be just what you want. The `TilePane` lays out its components in a rectangular grid. The grid is divided into equal-sized cells, and each cell can hold one component.

Gaps and Direction of Layout Flow

When you instantiate a `TilePane`, you can use a zero-parameter constructor, or you can use constructors that specify one or both of two kinds of features. Back in Section 18.4, when we instantiated a `VBox`, we also specified a spacing between adjacent elements. You can do the same kind of thing when you instantiate a `TilePane`. Only this time, there are two spacings, a horizontal gap (hgap) and a vertical gap (vgap), both specified in pixels. The constructor that enables you to specify these gaps looks like this:

```
TilePane(<hgap>, <vgap>)
```

Adding components to a `TilePane` is like adding components to a `FlowPane`. You just start adding them, and they fill the pane sequentially. By default, the flow moves left-to-right across the top row. When space on that row runs out, the flow jumps back to the left side of the next row and moves left-to-right across that row, and so forth, as with a `FlowPane`.

With a `TilePane`, however, you have another option – vertical orientation. This makes flow move top-to-bottom down the left column. When space in that column runs out, the flow jumps up to the top of the next column and moves top-to-bottom down that column, and so forth. The constructor that enables you to specify vertical orientation looks like this:

```
TilePane(javafx.geometry.Orientation.VERTICAL)
```

A fourth constructor allows you to specify both gaps and vertical orientation:

```
TilePane(javafx.geometry.Orientation.VERTICAL,  <hgap>, <vgap>)
```

Specifying Number of Columns or Number of Rows and Adding Components

With a horizontal orientation, if you do not specify width, by default the computer will make the window just wide enough for five columns and add rows as needed. With vertical orientation, if you do not specify height, by default the computer will make the window just high enough for five rows and add columns as needed.

If you want positive control over the number of columns (with horizontal orientation) or the number of rows (with vertical orientation), use the one-parameter `Scene` constructor, and do not specify the window's width and height. For horizontal orientation, instead of specifying width, specify a preferred number of columns with a statement like this:

```
pane.setPrefColumns(<your-desired-number-of-columns>);
```

For vertical orientation, instead of specifying height, specify a preferred number of rows with a statement like this:

```
pane.setPrefRows(<your-desired-number-of-rows>);
```

If you try to specify both columns and rows, the computer will ignore the redundant specification. After all, your specification is only a "preferred" number. If the orientation is horizontal, it will ignore your preferred number of rows and supply just enough rows to accommodate the total number of components added. Conversely, if the orientation is vertical, it will ignore your preferred number of columns and supply just enough columns to accommodate the total number of components added.

Consider this example. Suppose your program creates a `TilePane` called `pane` that has 5-pixel horizontal and vertical gaps, and it uses the following statement to provide a 5-pixel outside blank border:

```
pane.setPadding(new Insets(5));
```

And suppose it includes a code fragment like this:

```
for (int i=0; i<5; i++)
{
  Button button = new Button(Integer.toString(i+1));
  button.setPrefSize(80, 40);    // pixel width and height
  pane.getChildren().add(button);
} // end for i
```

Now, suppose your `TilePane` has horizontal orientation and code before the above fragment includes this statement:

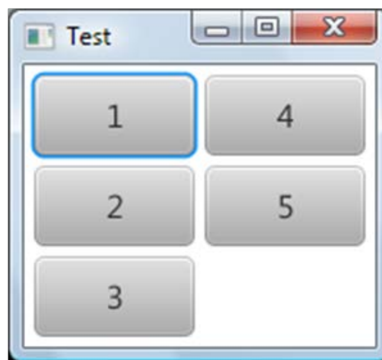
```
pane.setPrefColumns(3);
```

Here's what you would see:

On the other hand, suppose your `TilePane` has vertical orientation and code before the above fragment includes this statement:

```
pane.setPrefRows(3);
```

Here's what you would see this time:



The window size does not affect the component size, but if you allow the window (the `Scene`) to adapt, it will adjust to fit component size and quantity.

17.6 Tic-Tac-Toe Example

In this section, we present a simple tic-tac-toe program. We've chosen tic-tac-toe because we wanted to illustrate `TilePane` details. And tic-tac-toe, with its three-row by three-column board, provides the perfect opportunity for that.

User Interface

The program initially displays a three-row, three-column grid of blank buttons. Two users, player X and player O, take turns clicking blank buttons. Player X goes first. When player X clicks a button, the button's label changes from blank to X. When player O clicks a button, the button's label changes from blank to O. Player X wins by getting three X's in a row, 3 X's in a column, or 3 X's in a diagonal. Player O wins in the same manner except that O's are looked at instead of X's. To get a better handle on all this, see the sample session in Figure 18.9.

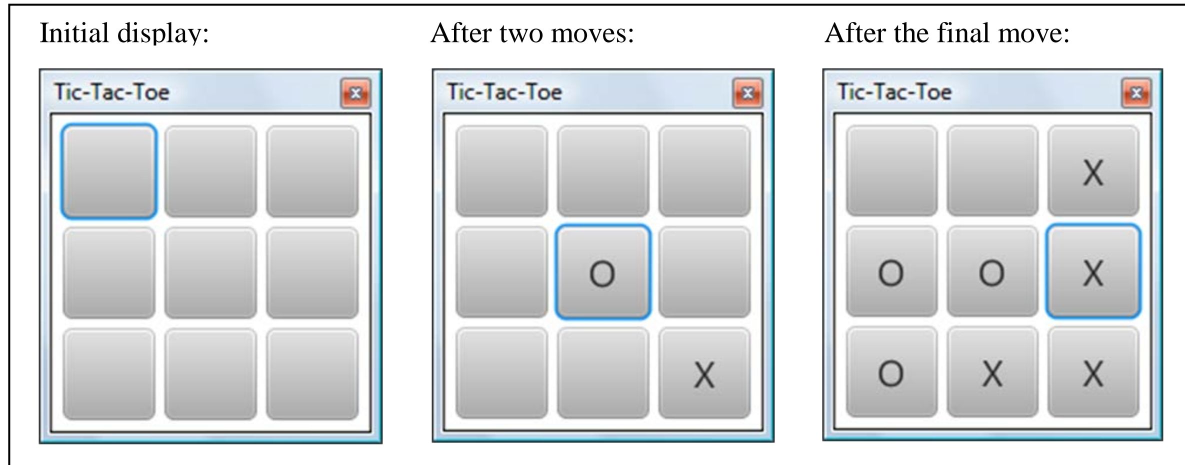


Figure 18.9 Sample session for the TicTacToe program

Program Details

See the TicTacToe program listing in Figures 18.10a and 18.10b. Most of the code in Figure 18.10a should make sense already since its structure parallels the structure in our previous GUI programs. The only unusual feature in this part of the program is the statement:

```
stage.initStyle(StageStyle.UTILITY);
```

We included this to shrink the buttons in the upper-right corner and provide more space for the "Tic-Tac-Toe" title.

Now look at Figure 18.10b. Note the `setPrefColumns` method call. Since we have not explicitly specified an orientation, by default the orientation is horizontal. Therefore, to control the geometry, we must specify the number of columns. Having set the number of columns to three, when we subsequently add nine buttons, the `TilePane` lays them out as a `FlowPane` would, ending up with three filled rows. The subsequent `setPadding`, `setHgap`, and `setVgap` method calls put white space around and between the buttons.

Next look at the `Handler` class in Figure 18.10b. In particular, note the statement where we get the clicked button and save it in a local variable:

```
Button button = (Button) e.getSource();
```

The `(Button)` cast operator is used because if there were no cast operator, the compiler would generate an error. Why? Because the compiler would see an `Object` at the right being assigned into a `Button` at the left. It sees an `Object` at the right because `getSource` is defined with an `Object` return type. In this case, since `getSource` really returns a `Button`, it's legal to cast its returned value to `Button`, and that satisfies the compiler and eliminates the error.

Let's examine the `Handler` class's `if` statement:

```
if (button.getText().isEmpty())
{
  button.setText(xTurn ? "X" : "O");
  xTurn = !xTurn;
}
```

```
/*************************************************************
* TicTacToe.java
* Dean & Dean
*
* This program implements the game of tic-tac-toe.
* When the first blank button is clicked, its label changes
* to an X. Subsequent clicked blank buttons change their labels
* to O and X in alternating sequence.
*************************************************************/

import javafx.application.Application;
import javafx.stage.*;          // Stage, StageStyle
import javafx.scene.Scene;
import javafx.scene.layout.TilePane;
import javafx.scene.control.Button;
import javafx.geometry.Insets;
import javafx.event.*;          // ActionEvent, EventHandler
import javafx.scene.text.Font;

public class TicTacToe extends Application
{
  private final double BUTTON_SIZE = 50;
  private boolean xTurn = true; // keeps track of who's up

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

  //***********************************************************

  @Override
  public void start(Stage stage)
  {
    TilePane pane = new TilePane();

    stage.initStyle(StageStyle.UTILITY); // makes room for title
    stage.setTitle("Tic-Tac-Toe");
    stage.setScene(new Scene(pane));
    createContents(pane);
    stage.show();
  } // end start
```

Figure 18.10a TicTacToe program – part A

We first check to ensure that the button is a blank button. We then reassign the button's label by using a conditional operator. If `xTurn` holds `true`, then X is assigned to the button label. Otherwise, O is assigned to the button label. We then change the value of `xTurn` by assigning its negated value into it. More specifically, if `xTurn` is `false`, we assign `true` into `xTurn`, and if `xTurn` is `true`, we assign `false` into `xTurn`.

```
  //************************************************************

  // Create components and add to pane

  private void createContents(TilePane pane)
  {
    Button button;              // for all buttons

    pane.setPrefColumns(3);
    for (int i=0; i<9; i++)
    {
      button = new Button();
      button.setPrefSize(BUTTON_SIZE, BUTTON_SIZE);
      button.setOnAction(new Handler());
      pane.getChildren().add(button);
    } // end for i
    pane.setPadding(new Insets(5));
    pane.setHgap(5);
    pane.setVgap(5);
  } // end createContents

  //************************************************************

  // If user clicks a button, change its label to "X" or "O".

  private class Handler implements EventHandler<ActionEvent>
  {
    public void handle(ActionEvent e)
    {
      Button button = (Button) e.getSource();
      button.setFont(new Font(20));

      if (button.getText().isEmpty())
      {
        button.setText(xTurn ? "X" : "O");
        xTurn = !xTurn;
      }
    } // end handle
  } // end class Handler
} // end class TicTacToe
```

Figure 18.10b TicTacToe program – part B

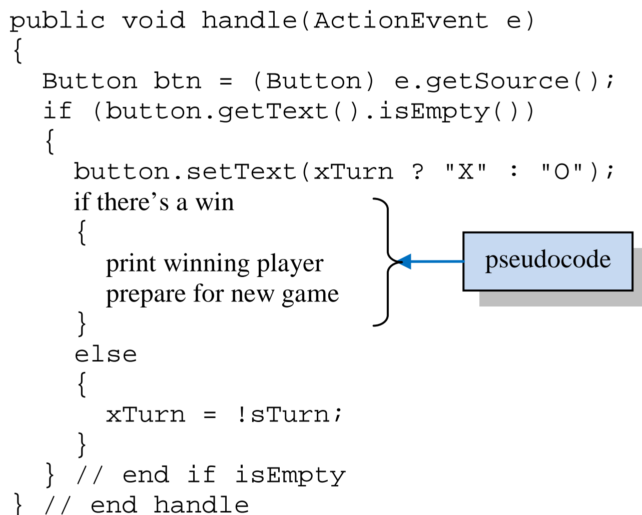
18.8 Problem Solving: Winning at Tic-Tac-Toe (Optional)

As you might have noticed, the previous section's TicTacToe program doesn't check for a winning move. As a problem-solving exercise, let's now discuss how to add that functionality. Rather than provide you with a Java solution, we'll provide you with the thought process for coming up with a solution. We'll codify the thought process using pseudocode. One of the chapter's projects asks you to finish the job by implementing a complete Java program solution.

To check for a win (i.e., to check for three in a row, three in a column, or three in a diagonal), the handler needs to access multiple buttons. As it stands now, the `TicTacToe` handler can access only one button—the button that was clicked. It gets that button by calling `getSource`. So how should you change the program so the handler can access multiple buttons?

To access multiple buttons, you need to declare multiple buttons. You could declare nine separate buttons, but the more elegant solution is to declare a three-row, three-column, two-dimensional array of buttons. The next question is, where should you declare the array? Do you declare it as a local variable inside the handler or as an instance variable at the top of the program? In general, local variables are preferred, but in this case, a local variable won't work. Local variables don't persist. You need to be able to update a button from within the handler and have that update be remembered the next time the handler is called. Thus, you need to declare the buttons array as an instance variable.

You need to check for a win only when the user clicks a button. So add check-for-a-win code to the `handle` method inside the button's handler. In adding the code, use top-down design. In other words, don't worry about the low-level details; just assume they work. Here's the updated `handle` method. The added code is in pseudocode:



```
public void handle(ActionEvent e)
{
  Button btn = (Button) e.getSource();
  if (button.getText().isEmpty())
  {
    button.setText(xTurn ? "X" : "O");
    if there's a win
    {
      print winning player
      prepare for new game
    }
    else
    {
      xTurn = !sTurn;
    }
  } // end if isEmpty
} // end handle
```


The pseudocode contains three tasks—checking for a win, printing the winner, and preparing for a new game. Checking for a win requires the most thought, so we'll postpone that task for now. Let's discuss the other two tasks first.

There are several possible ways to print the winner. You could print a congratulatory message in a special dialog box, as described in the previous chapter. If you save the `Start` method's `Stage` parameter as an instance variable, you could use it to add the winner to the title bar with a statement like this:

```
stage.setTitle(stage.getTitle() + " -> XO Wins");
```

The "XO" represents the winner's name, X or O, which can be obtained with the conditional operator expression,

```
xTurn ? "X" : "O".
```

Another alternative is to insert a `Label` or `Button` just below the `TilePane`, using the embedded-pane

technique described in the next section.

Preparing for a new game should be straightforward. Just assign the empty string to the board's button labels and assign `true` to the `xTurn` variable (X always goes first). Feel free to implement the print-winning-player and prepare-for-new-game tasks as embedded code inside the `if` statement or as separate helper methods. Either way is fine. But the checking-for-a-win task should definitely be implemented as a separate helper method. Why? Note how cleanly `win` is called in the above pseudocode. You can retain that clean look in the final Java code only if you implement the checking-for-a-win task as a `boolean` method, not as embedded code.

In implementing the `win` method, you need to check the two-dimensional buttons array for three in a row, three in a column, or three in a diagonal. Normally, when you access a group of elements in an array, you should use a `for` loop. So you might want to use a `for` loop to access the elements in the first row, use an- other `for` loop to access the elements in the second row, and so on. But that would require eight `for` loops:

```
for loop for first row
for loop for second row
. . .
for loop for second diagonal
```

Yikes! That's a lot of `for` loops! Is there a better way? How about taking the opposite approach and using no `for` loops? Let "XO" represent the current player, and use one big `if` statement like this:

```
if (buttons[0][0] equals XO and buttons[0][1] equals XO and buttons[0][2] equals XO) or
   (buttons[1][0] equals XO and buttons[1][1] equals XO and buttons[1][2] equals XO) or
   . . .
   (buttons[0][2] equals XO and buttons[1][1] equals XO and buttons[2][0] equals XO)
{
   return true
}
else
{
   return false
}
```

That works fine, but if you're bothered by the length of the `if` condition (eight lines long), you might want to try the following. Use one `for` loop for all the rows, one `for` loop for all the columns, and one `if` statement for the two diagonals:

```
for (i←0; i<3; i++)
{
   if (buttons[i][0] equals XO and buttons[i][1] equals XO and buttons[i][2] equals XO)
   {
      return true
   }
}
for (j←0; j<3; j++)
{
   if (buttons[0][j] equals XO and buttons[1][j] equals XO and buttons[2][j] equals XO)
   {
      return true
   }
}
if (buttons[0][0] equals XO and buttons[1][1] equals XO and buttons[2][2] equals XO) or
   (buttons[0][2] equals XO and buttons[1][1] equals XO and buttons[2][0] equals XO)
{
   return true
}
```

Of the three solutions, we prefer the last one because we feel its code is the most understandable.

To make the tic-tac-toe program more "real world," you'd probably want to provide additional functionality. In particular, you'd want to check for a "cat's game," which is when the board is filled and no one has won. You're asked to implement that functionality in one of the chapter's projects.

18.9 Embedded Panes

Suppose you'd like to implement the math-calculator window shown in the upper-left of Figure 18.11.

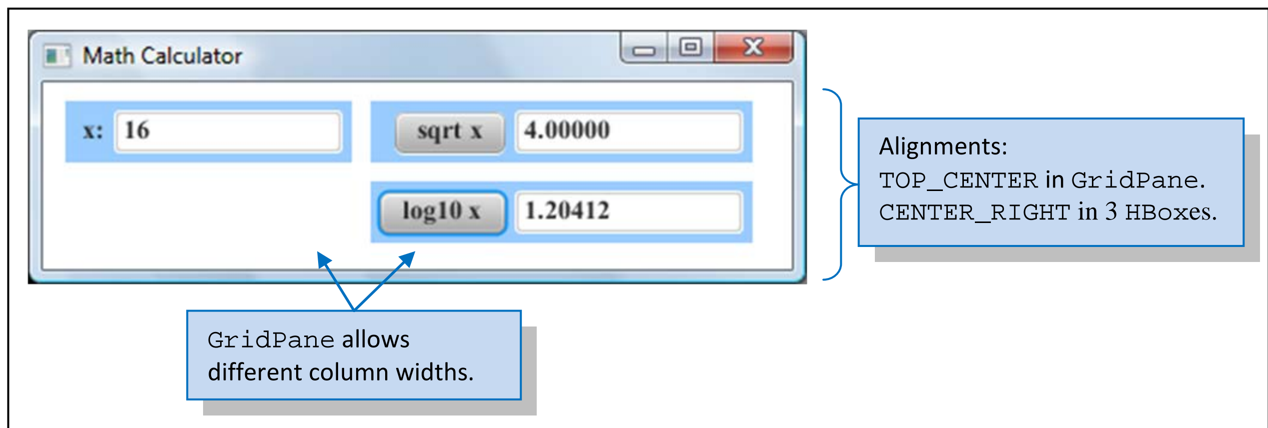


Figure 18.11 GridPane with three embedded HBoxes, each containing two components

What type of layout scheme should you use? The callouts in Figure 18.11 show our solution, but coming up with a good layout scheme often requires some thought. So let's walk through the creative process for this math-calculator example.

Trying Out Different Layouts

The math-calculator window appears to have two rows and four columns. So is a two-row by four-column TilePane scheme appropriate? The TilePane might be adequate for positioning components in an organized tabular fashion, but it's limited by one factor—each of its cells must be the same size. If we use a two-row by four-column TilePane for the math-calculator window, then we'll have eight same-sized cells. That's fine for most of the cells, but not for the top-left cell. The top left cell would hold the x: label. With such a small label, we would want a relatively small cell for it. But with a TilePane, a "relatively small cell" is not an option.

Since the TilePane is less than ideal, you might want to consider a FlowPane. That could work if you use right-aligned components and sized the window just right. But then you'd be at the mercy of the user to not resize the window. If the user widens the window, then the log10 x button would flow up to the top line, and you don't want that. So the FlowPane is also less than ideal. A simple BorderPane isn't even close. So what's the solution?

Using an Embedded Layout Scheme

In coming up with layouts for more complex windows, the key is often to embed panes inside other panes. Delegate.

Let's first tackle the outer pane. For the math-calculator window, we want the input at the left and the output at the right. To the extent that the input and output columns are approximately the same width, it makes sense to consider using a two-column TilePane for them. The left column would contain the input components—the x label and the input text box. The right column would contain the output components—the square root's button and output text box and the logarithm's button and output text box. However, the label in the left column is significantly smaller than the buttons in the right column, so it would be better if the columns could be different widths. So how about the GridPane, which we described in Section 18.3? That allows each column width to conform the widest component in that column, and this allows the column on the left to be narrower.

We'd like to organize the output components so that the square root's items are above the logarithm's items. If we used a TilePane for the overall container, we could insert an empty subordinate pane to align the log sub-pane under the square-root sub-pane. But since a GridPane provides better column-width conformity, we selected a GridPane for our overall container. With a GridPane, we individually specify where we want each component to go, and it doesn't matter if some intermediate cells are empty.

A TilePane or a GridPane allows only one component per cell. But Figure 18.11 shows two components in three of the outer pane's cells. To implement that organization scheme, we need to group each of the two-component pairs into their own separate containers. And for proper layout, we want containers that have appropriate layouts. We could use a FlowPane for each of these three cells, but again, that leaves us vulnerable to wrap-around. A better choice is an HBox. An HBox is just a horizontal version of the VBox you saw in Section 18.4. Components are still added sequentially, as in a FlowPane or VBox, but with an HBox everything stays on the same row. So what we end up with is three HBoxes in a GridPane. Voila, panes of one type inside a pane of another type. Pretty cool, eh?

When you have a non-trivial window, it's very common to have embedded panes. When that happens, it might take some tweaking to get your windows to look right. Despite the tweaking, using embedded panes is a lot easier than having to manually position many individual components with pixel values.[1]

18.10 MathCalculator Program

Now look at the MathCalculator program listing in Figures 18.12a, 18.12b, 18.12c, and 18.12d. You should peruse the entire program on your own, but we'll focus primarily on the embedded-pane aspects.

[1] If you want to set component positions manually, perhaps because you don't want them to jump around, instead of embedding a pane, you can embed a Group. A Group (found in the javafx.scene package) is a Node that can contain other Nodes like Label and Button, but a Group does not have an associated layout manager. So it does not adjust its components automatically if something alters window size. However, as its name implies, a Group can contain multiple components, so it provides another way to insert more than one component into a single cell in a BorderPane, TilePane, or GridPane. We describe Groups later in Section 18.16.

```
/*************************************************************
* MathCalculator.java
* Dean & Dean
*
* This program uses embedded layout managers to display
* the square root and logarithm of a user-entered number.
*************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.text.*;   // Font, FontWeight
import javafx.scene.layout.*; // GridPane, HBox
// Button, Control, Label, Labeled, TextField:
import javafx.scene.control.*;
import javafx.geometry.*;     // Insets, Pos
import javafx.event.*;        // Event Handler, ActionEvent

public class MathCalculator extends Application
{
  private static final int WIDTH = 400;
  private static final int HEIGHT = 100;

  private TextField xBox;     // user's input value
  private Button xSqrtButton;
  private TextField xSqrtBox; // generated square root
  private Button xLogButton;
  private TextField xLogBox;  // generated logarithm

  //**********************************************************

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

  //***********************************************************

  @Override
  public void start(Stage stage)
  {
    GridPane pane = new GridPane();

    stage.setTitle("Math Calculator");
    stage.setScene(new Scene(pane, WIDTH, HEIGHT));
    createContents(pane);
    stage.show();
  } // end start
```

Figure 18.12a MathCalculator program – part A

In Figure 18.12b the `createContents` method starts by declaring variables for the three subordinate panes. Then, for each of these subordinate panes it creates its two components and calls a common helper method to add those components to that subordinate pane and adjust the layout in that subordinate pane. Then it modifies the layout of the parent pane, establishes a font style that is inherited by all components, and adds handlers to the

two button components.

```
//**********************************************************

private void createContents(GridPane pane)
{
  HBox xPane;         // holds x label and its text box
  HBox xSqrtPane;     // holds "sqrt x" label and its text box
  HBox xLogPane;      // holds "log x" label and its text box
  Label xLabel;

  // Create the x pane:
  xLabel = new Label(" x:");
  xBox = new TextField();
  xPane = createSubPane(xLabel, xBox, true);

  // Create the square-root pane:
  xSqrtButton = new Button("sqrt x");
  xSqrtBox = new TextField();
  xSqrtPane = createSubPane(xSqrtButton, xSqrtBox, false);

  // Create the logarithm pane:
  xLogButton = new Button("log10 x");
  xLogBox = new TextField();
  xLogPane = createSubPane(xLogButton, xLogBox, false);

  // Add panels to the parent pane:
  pane.add(xPane, 0, 0);        // left upper
  pane.add(xSqrtPane, 1, 0);    // right upper
  pane.add(xLogPane, 1, 1);     // right lower

  // Modify layout in parent pane
  pane.setPadding(new Insets(10));
  pane.setHgap(10);
  pane.setVgap(10);
  pane.setAlignment(Pos.TOP_CENTER);

  // Establish font for all components
  pane.setStyle("-fx-font: bold normal 14 serif");

  // Add handlers
  xSqrtButton.setOnAction(new Handler());
  xLogButton.setOnAction(new Handler());
} // end createContents
```

Figure 18.12b MathCalculator program – part B

Figure 18.12c contains the helper method which takes care of the details of adding the components to a subordinate pane and adjusting the layout within that subordinate pane. Notice the `setMinWidth` method call. It keeps the buttons from collapsing whenever a user tries to reduce window width with a mouse. In addition to preserving button visibility, it also establishes a minimum overall window width.

```
//***********************************************************

// This adds components to subordinate panes and styles them.

private HBox createSubPane(
  Labeled control, TextField field, boolean editable)
{
  HBox subPane = new HBox(5);  // 5 pixels between components

  // populate and configure this subordinate pane
  subPane.getChildren().addAll(control, field);
  subPane.setPadding(new Insets(5));
  subPane.setAlignment(Pos.CENTER_RIGHT);
  subPane.setStyle("-fx-background-color: #99ccff");

  // modify features of the individual components
  control.setMinWidth(Control.USE_PREF_SIZE);
  field.setPrefColumnCount(8);
  field.setMaxHeight(25);
  field.setEditable(editable);

  return subPane;
} // end createSubPane

//***********************************************************
```

Figure 18.12c MathCalculator program – part C

Figure 18.12d contains the handler code. If you study this, you can see that it considers and deals with the most common types of user errors.

```
  // Inner class for math calculations

  private class Handler implements EventHandler<ActionEvent>
  {
    public void handle(ActionEvent e)
    {
      double x;             // numeric value for user entered x
      double result;        // calculated value

      try
      {
        x = Double.parseDouble(xBox.getText());
      }
      catch (NumberFormatException nfe)
      {
        x = -1;             // indicates an invalid x
      }

      if (e.getTarget().equals(xSqrtButton))
      {
        if (x < 0)
        {
          xSqrtBox.setText("undefined");
        }
        else
        {
          result = Math.sqrt(x);
          xSqrtBox.setText(String.format("%7.5f", result));
        }
      } // end if

      else // calculate logarithm
      {
        if (x < 0)
        {
          xLogBox.setText("undefined");
        }
        else
        {
          result = Math.log10(x);
          xLogBox.setText(String.format("%7.5f", result));
        }
      } // end else
    } // end handle
  } // end class Handler
} // end class MathCalculator
```

Figure 18.12d MathCalculator program – part D

18.11 `TextArea` Component

In the previous chapter, we introduced you to a few GUI components—`Label`, `TextField`, and `Button`—that provide basic input/output functionality. Now we'll introduce you to a few more GUI components—`TextArea`,

CheckBox, RadioButton, and ComboBox—that provide more advanced input/output functionality. We'll start with the TextArea component.

User Interface

The Label component works great for displaying a single line of text. As described in Chapter S17, you can use a Label component to display multiple lines of text, but the preferred technique for displaying multiple lines of text is to use a TextArea component. The larger lower area in Figure 18.13 is a TextArea component. The smaller upper area is a CheckBox component. We'll describe CheckBox components in the next section.

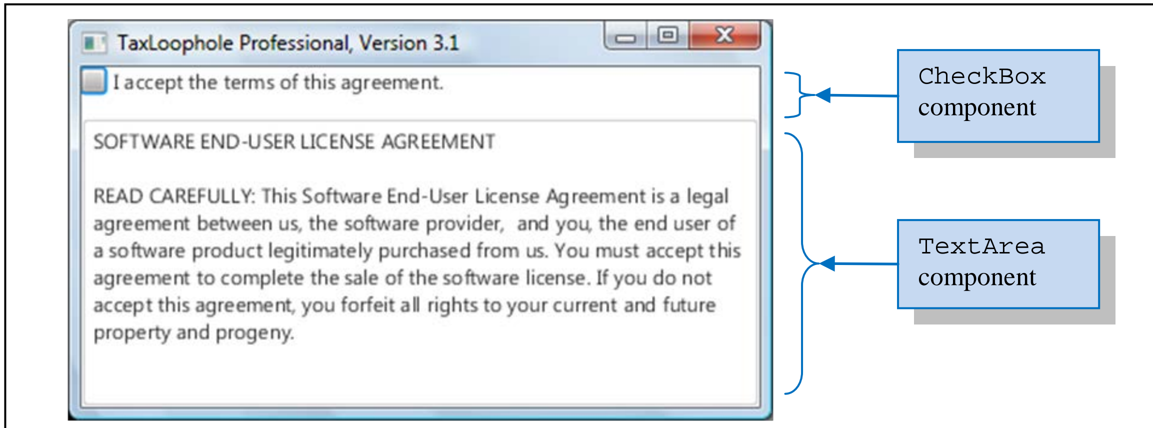


Figure 18.13 A window with a TextArea component and a CheckBox component

Implementation

To create a TextArea component, call the TextArea constructor like this:

TextArea <*TextArea-reference*> = new TextArea(<*display-text*>);

The *display-text* is the text that appears in the TextArea component. If the display-text argument is omitted, then the TextArea component displays no initial text.

Methods

The TextArea class, like all the GUI component classes, has quite a few methods. Here are the API headings and descriptions for the more popular TextArea methods:

```
public String getText()
```
Returns the text area's text.

```
public String getSelectedText()
```
Return selected portion of text area's text.

```
public void setText(String text)
```
Assigns the text area's text.

```
public void setEditable(boolean flag)
```
Makes the text box editable or non-editable.

```
public void setWrapText(boolean flag)
```
Turns line wrap on or off. The wrapping occurs only at word boundaries.

TextArea components are editable by default, which means users are allowed to type inside them. They are also editable if code includes the method call, setEditable(true). To prevent users from editing a TextArea component, call setEditable(false). Doing so prevents users from updating the text area, but

it does not prevent programmers from updating the text area. Programmers can call the `setText` method to modify the text, regardless of whether the text area is editable or non-editable. Other methods allow programmers to select a portion of the text. Then they can use the `replaceSelection` method to edit that selection.

`TextArea` components have line wrap turned off by default. Normally, you'll want to turn line wrap on by calling `setWrapText(true)`. This wraps at word boundaries. It makes long lines automatically wrap to the next row, instead of disappearing when they reach the text area's right boundary. If text runs past the area's bottom boundary, a vertical scroll bar automatically appears. This allows users to bring overflowing text into view.

Regardless of whether the code includes `setEditable(true)`, a user can select any portion of the text displayed, and (given an appropriate stimulus) the program can process that selected text. For example, suppose some component that is a subclass of the `ButtonBase` class (like a button or a check box) has been given a handler by a `setOnAction` method call. Then, after a click on that `ButtonBase` component, the handler can retrieve the selected text by having the `TextArea` object call its `getSelectedText` method.

License-Agreement Example

Figure 18.14 contains the code that creates the display in Figure 18.13.

```
private void createContents(BorderPane pane)
{
  CheckBox confirmBox = new CheckBox(
    "I accept the terms of this agreement.");
  TextArea license = new TextArea(
    "SOFTWARE END-USER LICENSE AGREEMENT\n\n" +
    "READ CAREFULLY: This Software End-User License Agreement" +
    " is a legal agreement between us, the software provider, " +
    " and you, the end user of a software product legitimately" +
    " purchased from us. You must accept this agreement to" +
    " complete the sale of the software license. If you do not" +
    " accept this agreement, you forfeit all rights to your" +
    " current and future property and progeny.");

  pane.setTop(confirmBox);
  pane.setBottom(license);
  license.setWrapText(true);
  license.setEditable(false);
} // end createContents
```

Figure 18.14 Populating a pane with a `TextArea` component and a `CheckBox` component

This example uses a `BorderPane` layout scheme. The check box is in the top region, and the text area is in the bottom region. Note the `setWrapText(true)` and `setEditable(false)` calls. These calls are common for `TextArea` components. Unfortunately, the appearance of what you see in Figure 18.13 is not the greatest. By default, the background is white, and the alignment in each region is top-left. The text seems to be jammed up against the top and left boundaries.

To improve the appearance, let's add some shading to the text area. Here's a statement that does that:

```
license.setStyle("-fx-background-color: #cdcdcd");    // light gray
```

Next, let's put a 20-pixel border around everything. Here's a statement that does that:

```
pane.setPadding(new Insets(20));
```

Finally, let's insert some additional space between the confirm box and the text area. The sequence of parameters

in the four-parameter `Inset` constructor is top, right, bottom, left. So to put a 20-pixel space above the text area, we can use this:

```
BorderPane.setMargin(license, new Insets(20, 0, 0, 0));
```

Figure 18.15 shows the result of these cosmetic improvements. The additional padding and spacing reduces the area available for the text to where it cannot fit into the available space. This condition automatically inserts a scroll bar on the right side of the `TextArea` to allow users to see the text that runs past the bottom of the text area. Using a mouse to expand the entire window causes the text area to increase. If this action enables all the text to fit into the text area once again, the scroll bar goes away.

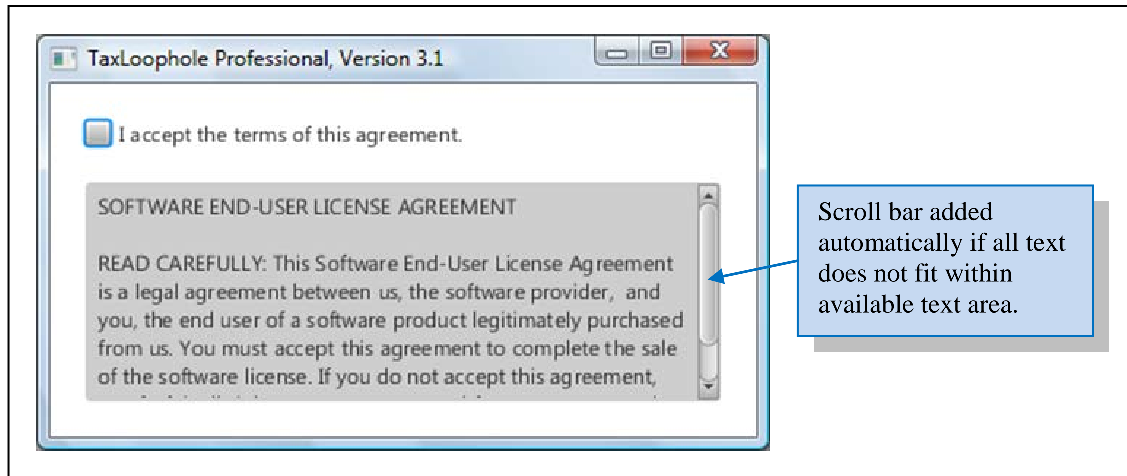


Figure 18.15 License agreement with modified background color, padding, and margin

18.12 `CheckBox` Component

User Interface

Now look at the *check box* component at the top of Figure 18.15. Use a check box component if you want to present an option. A check box component displays a small square with a label to its right. When the square is blank, the check box is unselected. When the square contains a check mark, the check box is selected. By default, a check box is initially unselected, and when users click on a check box, it toggles between unselected and selected.

Implementation

To create a check box component, call the `CheckBox` constructor like this:

```
CheckBox <CheckBox-reference> = new CheckBox(<label>);
```

The *label* argument specifies the text that appears at the right of the check box's square. If the label argument is omitted, then no text appears at the right of the check box's square. The check box is initially unselected. Here's how the check box was created in the license-agreement window:

```
confirmBox = new JCheckBox("I accept the terms of this agreement.");
```

Methods

Here are the API headings and descriptions for the more popular `CheckBox` methods:

`public boolean isSelected()`
Returns `true` if the check box is selected and `false` otherwise.

`public void setVisible(boolean flag)`
Makes the check box visible or invisible.

`public void setSelected(boolean flag)`
Makes the check box selected if the parameter is true.

`public void setDisable(boolean flag)`
Disables the check box if the parameter is true.

`public void setOnAction(EventHandler<ActionEvent> handler)`
Adds a handler to the check box.

The `isSelected` and `setVisible` methods are straightforward, but the other three methods need further explanation. Let's start with `setSelected`. Why might you want to call `setSelected` and adjust the selection status of a check box? Because you might want one user input to impact another user input. For example, in Figure 18.16, the user's selection of standard versus custom[4] should impact the check box selections. More specifically, if the user selects the Standard option, the check box selections should go to their "standard" settings. As you can see in Figure 18.16's left window, the standard settings for the check boxes are the top two selected and the bottom two unselected. To have your program select the top two check boxes, those two check boxes should call `setSelected(true)`. To have your program unselect the bottom two check boxes, those two check boxes should call `setSelected(false)`.

To have your program disable a check box, the checkbox should call `setDisable(true)`. Why might you want to call `setDisable(true)` and disable a check box? Because you might want to keep the user from modifying that box's value. For example, if the user selects the Standard option as shown in Figure 18.16's left window, the check box selections should be set to their standard settings (as explained above), and then each check box should call `setDisable(true)`. That way, the user cannot make changes to the standard-configuration check box values. In Figure 18.16's left window, note that the check boxes are gray. We say that they're *grayed out*. That's the standard GUI way of telling the user that something is disabled.

[4] The Standard and Custom circles at the top of Figure 18.16 are called radio buttons. We'll describe `RadioButton` components in the next section.

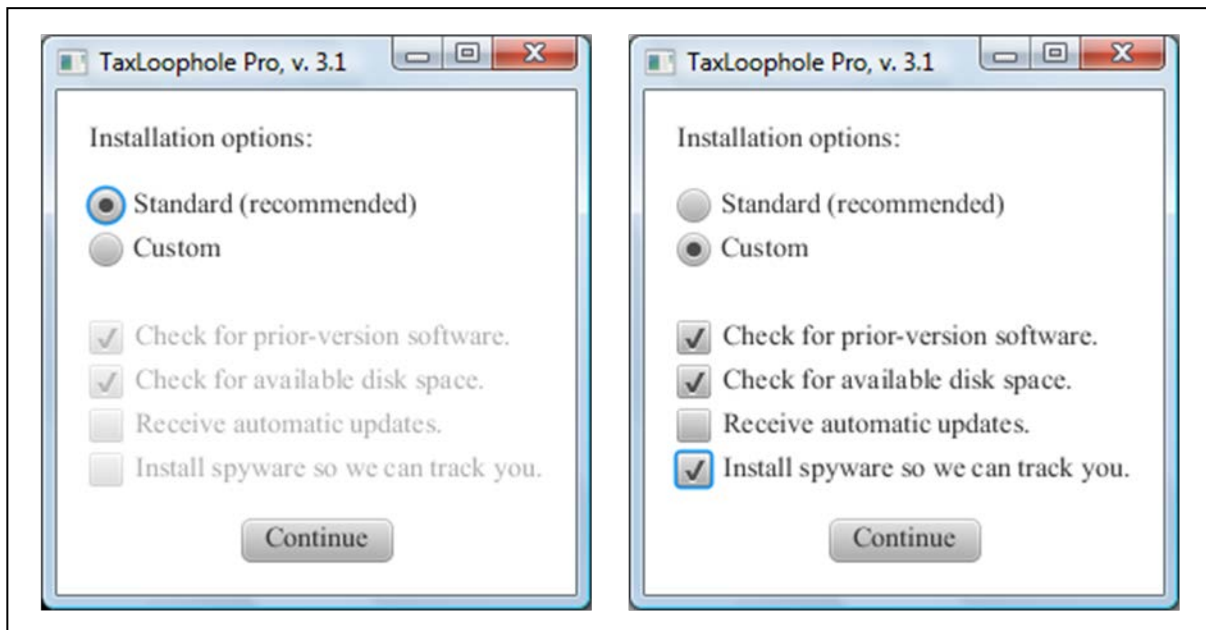


Figure 18.16 Example that illustrates `CheckBox`'s `setSelected` and `setDisable` methods

CheckBox Handlers

With a `Button` component you'll almost always want an associated handler. But with a `CheckBox` component, you may or may not want an associated handler. If you have a check box with no handler, then the check box simply serves as an input entity. If that's the case, then the check box's value (checked or unchecked) would typically get read and processed when the user clicks a button. On the other hand, if you want something to happen immediately (right when the user selects a check box), then add a handler to the check box component. Suppose you have a Green Background check box. If you want the window's background color to change to green right when the user clicks the check box, add a handler to the check box. The syntax for adding a handler to a `CheckBox` component is the same as the syntax for adding a handler to a `Button` component. Provide a handler that implements the `EventHandler` interface and then add the handler to the `CheckBox` component by calling `setOnAction`.

Be aware that the Java API provides many alternative event handlers. For example, there is a `ListChangedListener<E>` interface which specifies an `onChanged` method that gets called whenever a change occurs to a `ObservableList`, which is what's returned by the familiar `getChildren()` method call. To add a `ListChangedListener<E>`, you call the `addListener` method specified by the `ObservableList` interface. Since the `EventHandler` interface is the preferred interface for most situations, we'll stick with it when implementing `CheckBox` handlers. When we get to the `RadioButton` and `ComboBox` components in the next sections, we'll continue to use the `EventHandler` interface.

Installation-Options Example

It's now time to put these check box concepts into practice by showing you some code. Look back at the installation-options windows in Figure 18.16. In Figure 18.17, we provide the handler code associated with those windows. Let's walk through the code. In the `if` statement's condition, we check to see whether the standard option was selected. If that's the case, we disable the check boxes by calling `setDisable(true)` for each check box. We then assign the check boxes to their standard settings by calling `setSelected(true)` or `setSelected(false)` for each check box. In the `else` block, we handle the custom option being selected. We enable the check boxes by having each box call `setDisable(false)`. This enables the user to control whether the box is selected or not.

```
private class Handler implements EventHandler<ActionEvent>
{
  public void handle(ActionEvent e)
  {
    if (e.getSource() == standard)  // standard option chosen
    {
      prior.setDisable(true);
      diskSpace.setDisable(true);
      updates.setDisable(true);
      spyware.setDisable(true);
      prior.setSelected(true);
      diskSpace.setSelected(true);
      updates.setSelected(false);
      spyware.setSelected(false);
    }
    else                            // custom option chosen
    {
      prior.setDisable(false);
      diskSpace.setDisable(false);
      updates.setDisable(false);
      spyware.setDisable(false);
    }
  } // end handle
} // end Handler
```

Figure 18.17 Handler code for Figure 18.16's installation-options windows

18.13 `RadioButton` Component

User Interface

Look at the circles in the windows in Figure 18.16. They're called *radio buttons*. A `RadioButton` component displays a small circle with a label to its right. When the circle is blank, the radio button is unselected. When the circle contains a large dot, the radio button is selected.

According to the description so far, radio buttons sound a lot like check boxes. They display a shape and a label, and they keep track of whether something is on or off. The key difference between radio buttons and check boxes is that radio buttons almost always come in groups called *toggle groups*. And within a radio button toggle group, only one radio button can be selected at a time. If a user clicks an unselected radio button, the clicked button becomes selected, and the previously selected button in the group becomes unselected. If a user clicks a selected radio button, no change occurs (i.e., the clicked button remains selected). In contrast, if a user clicks a selected check box, the check box changes its state from selected to unselected.

Implementation

To create a `RadioButton` component, call the `RadioButton` constructor like this:

```
RadioButton <RadioButton-reference> = new RadioButton(<label>);
```

The *label* argument specifies the text that appears at the right of the radio button's circle. If the label argument is omitted, no text appears at the right of the radio button's circle. The radio button is initially unselected.

The following statements show how we created the standard and custom radio buttons in the installation-options program:

```
standard = new RadioButton("Standard (recommended)");
custom = new RadioButton("Custom");
```

To enable the only-one-button-selected-at-a-time radio-button-group functionality, create a ToggleGroup object and add individual radio button components to it. Here's how:

```
<first-button-in-group>.setToggleGroup(<ToggleGroup-reference>);
...
<last-button-in-group>.setToggleGroup(<ToggleGroup-reference>);
```

The following statements show how we created the radio button group for the standard and custom radio buttons in the installation-options program:

```
ToggleGroup radioGroup = new ToggleGroup();
standard.setToggleGroup(radioGroup);
custom.setToggleGroup(radioGroup);
```

In addition to adding radio buttons to a toggle group, you also must add them to a container. Radio buttons work the same as other components in terms of adding them to a container. You can add them individually, or you can add them along with other components, like this:

```
VBox vPane = new VBox(5);
vPane.getChildren().addAll(standard, custom, new Label(""),
  prior, diskSpace, updates, spyware);
```

You need to add each radio button twice—once to a radio button group and once to a container. If you like shortcuts, you might be thinking: Why does Java make you add the individual radio buttons to the container? Why are they not added automatically when the radio button group is added? Adding the buttons separately from the button group gives you freedom in positioning the buttons. If you wanted to, you could even put them in different panes.

Since the RadioButton class has Button in its name, you can correctly assume that it's in the javafx.scene.control package. But what about the ToggleGroup class? It doesn't describe an explicit component, but it's also in the javafx.scene.control package.

Methods

Here are the API headings and descriptions for the more popular RadioButton methods:

`public boolean isSelected()`

Returns true if radio button is selected and false otherwise.

`public void setSelected(boolean flag)`

Makes radio button selected if argument is true. Does nothing if argument is false.

`public void setDisable(boolean flag)`

Makes radio button disabled or enabled. If not disabled, it responds to mouse clicks.

`public void setOnAction(EventHandler<ActionEvent> handler)`

Adds a handler to the radio button.

We described these same methods in the `CheckBox` section. Only one of them needs further attention— the `setSelected` method. To understand how `setSelected` works, you first need to understand fully how a user interacts with a radio button group. To select a radio button, a user clicks it. That causes the radio button to become selected and all other radio buttons in the group to become unselected. To programmatically select a radio button, you have the radio button call `setSelected(true)`. That causes the radio button to become selected and all other radio buttons in the group to become unselected. As mentioned above, there is no way for a user to unselect a button. Likewise, there is no way for a program to unselect a button. That's why calling `setSelected(false)` doesn't do anything. It compiles and runs, but it doesn't cause any buttons to change their selected status.

18.14 `ComboBox` Component

User Interface

A *combo box* allows a user to select an item from a list of items. Combo boxes are sometimes called *drop-down lists* because if a user clicks the box's down arrow, a list of selection items drops down from the original display. Then, if a user clicks a selection from the drop-down list, the list disappears and only the selected item remains displayed. Figure 18.18a shows a label followed by a combo box in an `HBox`, which is like a `VBox` except it is horizontal rather than vertical. You already know about labels and an `HBox` is very similar to a `VBox`, so we'll ignore them and focus on the combo box.

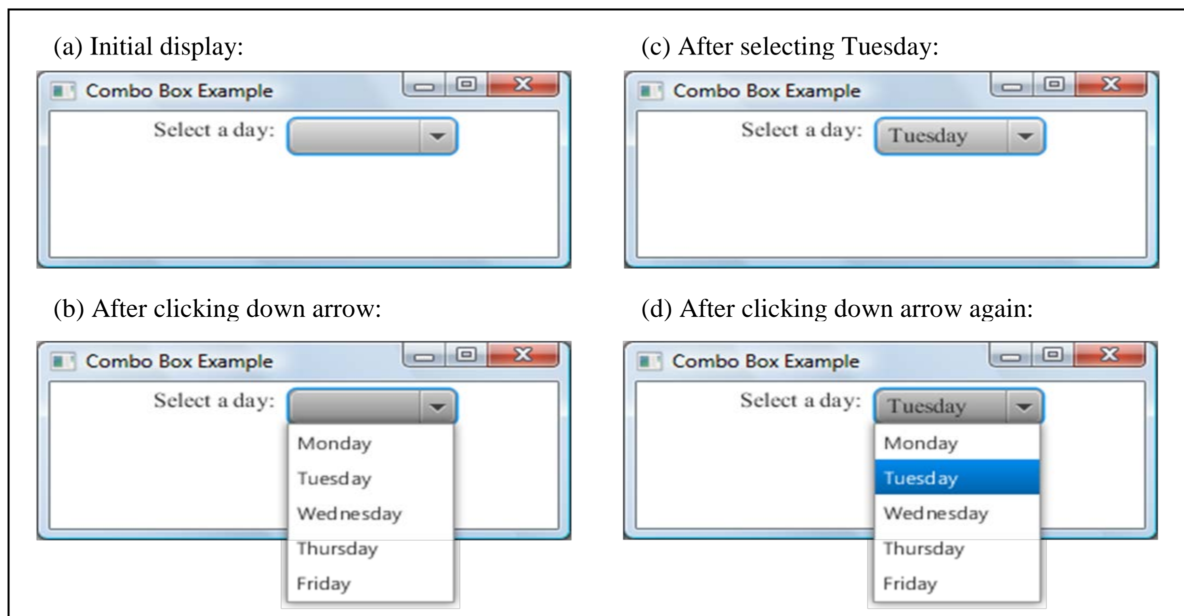


Figure 18.18a Select-a-day `ComboBox` example – base case

By default, nothing is selected initially, so when a combo box first displays, it's empty. But after a selection, it shows the most recent selection. Combo boxes are similar to radio button groups in that they both allow the user to select one item from a list of items. But a combo box takes up less space in the window. Therefore, if you have a long list of items to choose from, and you want to save space, you should use a combo box rather than a group of radio buttons. By default, the drop-down list displays no more than ten items at a time. If the list contains more than ten items, a vertical scroll bar appears, like the scroll bar on the right side of the text area in Figure 18.15. This allows users to find any item in a long list.

Implementation

The ComboBox class is in the javafx.scene.control package. Creating a combo box component is a two-step process. First, instantiate the box, using this syntax:

```
ComboBox<<item-type>> <box-reference> = new ComboBox<>();
```

Then populate the box with a list of items, using this syntax:

```
<box-reference>.getItems().addAll(<varargs-list-of-items>);
```

For example, here's how we created the combo box in Figure 18.18a:

```
ComboBox<String> daysBox = new ComboBox<>();

daysBox.getItems().addAll(
  "Monday", "Tuesday", "Wednesday", "Thursday", "Friday");
```

A combo box's items must be references, and they are usually strings. ComboBox's getItems method call returns an ObservableList. This like what's returned by a pane's getChildren method call. Thus, the addAll method is the same addAll method used to add components to a pane. But beware: Although a pane typically contains component types like Label and Button, which are subclasses of the Node class, the item list in a combo box should never contain a Node or any of its subtypes. That's because a Node can be in only one place at a time, and as you can see in the second image in Figure 18.18a, the "Saturday" item appears in two places – in the pull-down list and in the text field.

Methods

Here are the API headings and descriptions for the more popular ComboBox methods:

```
public void setValue(T value)
```
Selects a particular item in the item list from within the program.

```
public void setEditable(boolean flag)
```
Makes the combo box's top portion editable or non-editable.

```
public void setOnAction(EventHandler<ActionEvent> handler)
```
Adds a handler to the combo box.

```
public T getValue()
```
Use this in a handler. If the ComboBox is not editable, this method returns the selected item. If it is editable, it returns the most recent user action − either the value input by the user, or the last selected item.

The setValue method allows the program to select an item. The type of that item (T) must match the item type declared in the ComboBox instantiation. If the argument passed to the setValue method is not already in the list, it's automatically added to the list. The getValue method enables the program to retrieve the most recently selected item.

By default, a user cannot modify the item list. However, a setEditable(true) method call enables the user to make entries in the field that displays the currently selected item. In this case that field acts like the TextField in the previous chapter's Greeting program. A user entry in an editable ComboBox text field becomes the ComboBox's selected item. A subsequent getValue method call

retrieves this entered value.

The effect of such an entry depends on the program. If the ComboBox does not have its own handler, a user entry into its text field doesn't have any effect until some other component's handler makes a call to the combo box's getValue method. In the meantime, if the user substitutes a different entry or makes a selection of one of the items in the drop-down list, the original entry will be lost. If you want a user entry to have immediate effect, when you set things up (perhaps in a createContents method) have the ComboBox object call its setOnAction method to add its own event handler.

ComboBox with Editing and Event Handling

To enable editing and add an event handler, include these two statements during the set-up phase of the program:

```
daysBox.setEditable(true);
daysBox.setOnAction(new Handler());
```

You also need to provide the event handler, and here's an example of what it might look like:

```
// This adds a distinct user entry to the list of items.

private class Handler implements EventHandler<ActionEvent>
{
  public void handle(ActionEvent e)
  {
    String selection = daysBox.getValue();
    System.out.println(selection);
    if(!daysBox.getItems().contains(selection))  // avoids duplication
    {
      daysBox.getItems().add(selection);
    }
  } // end handle
} // end inner class Handler
```

This adds new items entered by the user into the item list. Since the item list is a List, it may contain duplicate elements, which is probably undesirable. When we initialize the combo box, we can avoid duplication by being careful. The if statement in the above handler prevents the user from duplicating existing list items.

Figure 18.18b shows a typical sample session. In image (a) on the left, notice that the text field is now white, and a cursor invites the user to make an entry. In this example, the user types in "Saturday" and then hits the Enter key. This Enter event causes the above handler to append the entered string to the item list. A subsequent user click displays the new item list in image (b) on the right.

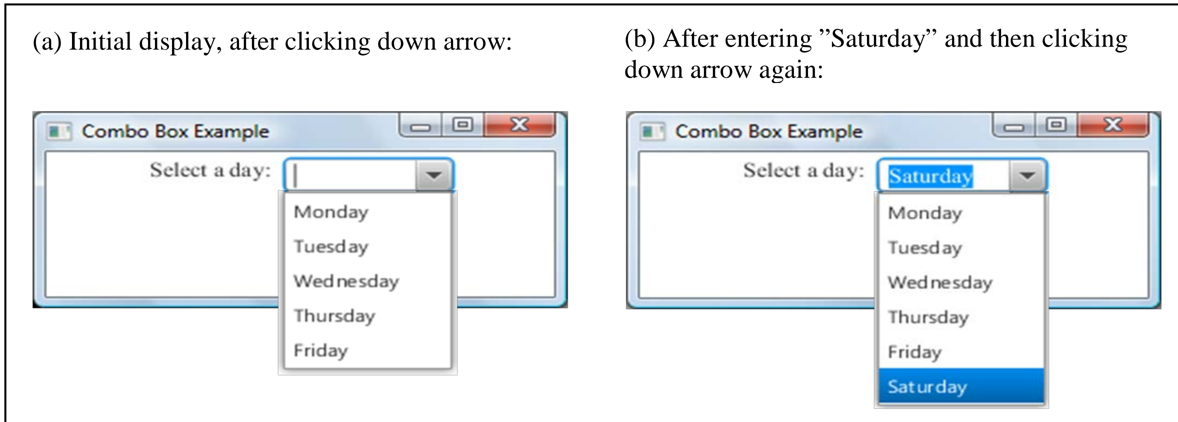


Figure 18.18b Select-a-day `ComboBox` example with editing that adds to the item list

If you'd like to force a combo box's text field to display a particular value, call `setValue` with that particular value as the argument. You can use `setValue` to select an existing value or to add a new value. In either case, the `setValue` argument becomes the selected value.

If you'd like to remove one item, like Tuesday, from the item list, use a statement like this:

```
daysBox.getItems().remove("Tuesday");
```

If you'd like to remove all items from the list and start over, use a statement like this:

```
daysBox.getItems().clear();
```

Since a combo box's items are a `List`, you can also access them by index value. In our example, because of the order of the varargs in our original `addAll` method call, Monday is 0, Tuesday is 1, Wednesday is 2, and so forth. Here are examples of statements you might want to use:

```
int index = daysBox.getItems().indexOf("Tuesday");
String item = daysBox.getItems().get(index);
```

For example, you might want to find the item that is immediately after the currently selected item. To do that you could use a code fragment like this:

```
String selection = daysBox.getValue();
int index = daysBox.getItems().indexOf(selection);
String item = daysBox.getItems().get(index + 1);
```

Now for a brain teaser. Suppose you tried to use this code fragment in the handler above with a subsequent `daysBox.setValue(item)` method call to reset the prompt to the next day. What would happen? The change in the selection value would trigger another call to the handler and produce a loop that crashes the program with an `IndexOutOfBoundsException`.

18.15 Job Application Example

In this section, we'll put into practice what you've learned in the previous three sections. We present a complete program that uses check boxes, radio buttons, and a combo box. The program implements a job application form. If the user enters values that are indicative of a good employee, the program displays an encouraging message ("Thank you for your application submission. We'll contact you after we process your information."). Figure 18.19

shows what we're talking about. The image at the top is the initial display. Making good selections like those on the left generates the pop-up box in the lower left corner. Making bad selections like those on the right generates the pop-up box in the lower right corner.

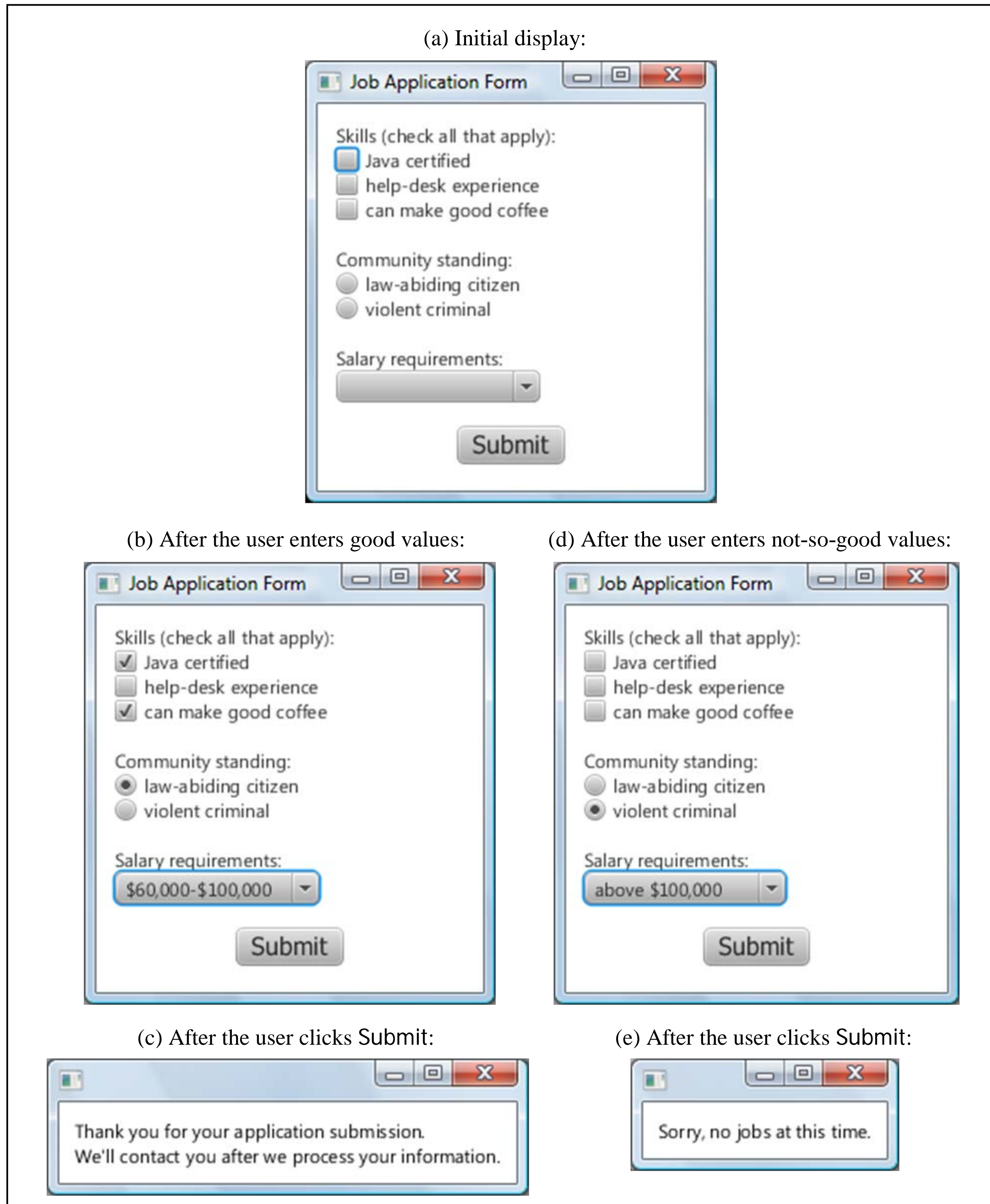


Figure 18.19 Sample session for the JobApplication program

Figures 18.20a, 18.20b, and 18.20c show the program. In Figure 18.20a notice how the program provides complete descriptions of all seven of the active components right at the beginning. These are all instance variables because the program needs to access each of them during the set-up phase and again later in the event handler. Defining these components completely at the beginning also provides nice self documentation and makes the program easier to understand.

```
/*****************************************************************
* JobApplication.java
* Dean & Dean
*
* This program implements job application questions
* with check boxes, radio buttons, and a combo box.
*****************************************************************/

import javafx.application.Application;
import javafx.stage.*;          // Stage, Modality
import javafx.scene.Scene;
import javafx.scene.layout.*;   // BorderPane, VBox, VBoxBuilder
import javafx.geometry.*;       // Insets, Pos
import javafx.scene.text.*;     // Text, Font
import javafx.scene.control.*; // CheckBox, RadioButton, ToggleGroup
import javafx.event.*;          // EventHandler, ActionEvent

public class JobApplication extends Application
{
  private static final int WIDTH = 250;
  private static final int HEIGHT = 250;

  private CheckBox java = new CheckBox("Java certified");
  private CheckBox helpDesk = new CheckBox("help-desk experience");
  private CheckBox coffee = new CheckBox("can make good coffee");
  private RadioButton good = new RadioButton("law-abiding citizen");
  private RadioButton bad = new RadioButton("violent criminal");
  private ComboBox<String> salary = new ComboBox<>();
  private Button submit = new Button("Submit");

  //**************************************************************

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

Figure 18.20a JobApplication program – part A

Figure 18.20b creates the scene. For the overall layout we use a `BorderPane`. This helps us display the Submit button differently from everything else by centering it in the `BorderPane`'s bottom region. Then we put everything else in a `VBox` embedded in the `BorderPane`'s center region. After thus populating the `BorderPane`'s center and bottom regions with a `VBox` and the Submit button, we add all the other components to the `VBox`, using two blank `Label` components to create blank lines between the three different types of information. Next we associate the two radio buttons in a toggle group and add the three salary ranges to the `ComboBox`. Then we add a handler to the Submit button.

The last five statements in the `createContents` method tweak the display cosmetics. Embedding a `VBox` in a `BorderPane` helps us use different font, spacing, and alignment for the Submit button than for the other components.

```
//***************************************************************

@Override
public void start(Stage stage)
{
  BorderPane pane = new BorderPane();

  stage.setTitle("Job Application Form");
  stage.setScene(new Scene(pane, WIDTH, HEIGHT));
  createContents(pane);
  stage.show();
} // end start

//***************************************************************

// Create components and add to window.

private void createContents(BorderPane pane)
{
  VBox centerPanel = new VBox();   // for all but submit button
  ToggleGroup radioGroup = new ToggleGroup();

  pane.setCenter(centerPanel);
  pane.setBottom(submit);
  centerPanel.getChildren().addAll(
    new Label("Skills (check all that apply):"),
    java, helpDesk, coffee, new Label(),
    new Label("Community standing:"), good, bad,
    new Label(), new Label("Salary requirements:"), salary);
  good.setToggleGroup(radioGroup);
  bad.setToggleGroup(radioGroup);
  salary.getItems().addAll(
    "$20,000-$59,000", "$60,000-$100,000", "above $100,000");
  submit.setOnAction(new Handler());

  submit.setFont(Font.font("Tahoma", 16));
  pane.setPadding(new Insets(12));
  BorderPane.setMargin(submit, new Insets(5));
  BorderPane.setAlignment(pane.getCenter(), Pos.CENTER_LEFT);
  BorderPane.setAlignment(pane.getBottom(), Pos.CENTER);
} // end createContents
```

Figure 18.20b JobApplication program – part B

In Figure 18.20c, the `showMessageDialog` method defines another window that presents the computer's response to the user's input. This method creates one of those special dialog windows described in the previous chapter. The `initModality` method call gives this new dialog window (the `dialogStage`) priority over all other windows by suppressing event firing from all other windows until the user closes the dialog window by clicking the X in its upper right corner. The `setScene` method instantiates a `new Scene` with a `VBox`. Then it adds a label with the appropriate text message. The last two statements center the message and surround it with a 10-pixel border.

```
  //***************************************************************

  private void showMessageDialog(String text)
  {
    Stage dialogStage = new Stage();
    VBox box = new VBox();

    dialogStage.initModality(Modality.APPLICATION_MODAL);
    dialogStage.setScene(new Scene(box));
    box.getChildren().add(new Label(text));
    box.setAlignment(Pos.CENTER);
    box.setPadding(new Insets(10));
    dialogStage.show();
  } // end showMessageDialog

  //***************************************************************

  // Read entered values and display an appropriate message.

  private class Handler implements EventHandler<ActionEvent>
  {
    public void handle(ActionEvent e)
    {
      if (
        (java.isSelected() || helpDesk.isSelected()
                           || coffee.isSelected()) &&
        (good.isSelected()) && (!salary.getSelectionModel().
          getSelectedItem().equals("above $100,000")))
      {
        showMessageDialog(
          "Thank you for your application submission.\n" +
          "We'll contact you after we process your information.");
      }
      else
      {
        showMessageDialog(
          "Sorry, no jobs at this time.");
      }
    } // end handle
  } // end class Handler
} // end class JobApplication
```

Figure 18.20c JobApplication program – part C

The message that's printed in the dialog box is determined by an `if` condition in the handler. For a favorable response, the user must have checked at least one of the check boxes, have selected the law-abiding citizen radio button, and not have asked for more than $100,000 in salary.

18.16 More GUI Components

In this chapter and the previous chapter, you've learned quite a bit about JavaFX. Enough to get up and running for most basic GUI needs. This final section presents some additional GUI components.

Menus and Scroll Panes

Figure 18.21 shows examples of the `MenuBar`, `Menu`, and `MenuItem` classes and the `ScrollPane` class. The `MenuBar`, `Menu`, and `MenuItem` classes allow you to add a *menu bar* and *menus* to the top of a window. The `ScrollPane` class allows you to create a scrollable container. It's like a `TextArea`, but more versatile.

After clicking Brightness > Brighter; Font Size > Larger; Font Size:

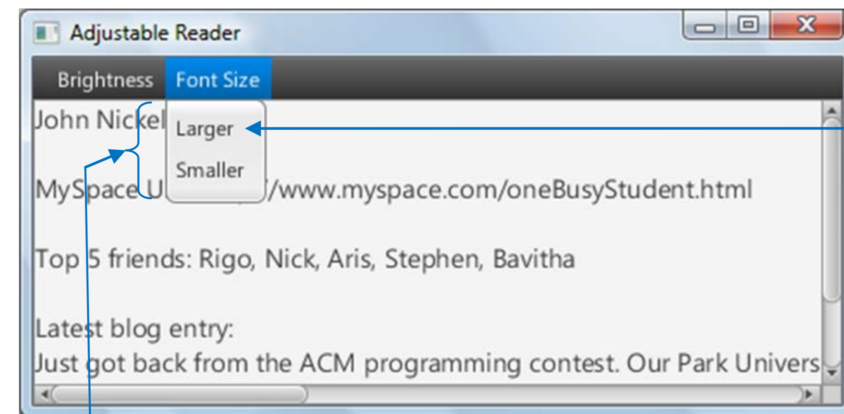


Figure 18.21 A reader program that uses a menu bar and scroll bars to adjust the view
The user can also pan the text area by dragging it with the mouse.

In this example, the menu bar contains two menus—one allows the user to adjust the brightness of the text, and the other allows the user to adjust the size of the text. The two scroll bars are part of the scroll pane. They allow the user to scroll up and down and back and forth and view the contents of a large pane. The user can also move the pane's contents around by panning (dragging it) with a mouse.

By default a scroll pane provides as many scroll bars as the content requires, or the program can specify how

scroll bars are used. The program can enable panning with a mouse or not, and it can configure several other scroll pane features. A scroll pane can contain multiple components and other panes, which themselves can contain multiple components. Therefore, a single scroll pane can contain any combination of GUI components in any arrangement.

Figures 18.22a, 18.22b, and 18.22c show the program that generates and manages what you see in Figure 18.21. By now, most of the code should be familiar and straightforward. However, the brightness and font-size algorithms have some subtleties that merit explanation.

```
/********************************************************************
* AdjustableReader.java
* Dean & Dean
*
* Display a text area with menu-based viewing options.
********************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.text.Text;
import javafx.scene.layout.BorderPane;
import javafx.scene.control.*; // MenuBar,Menu, MenuItem, ScrollPane
import javafx.event.*;         // EventHandler, ActionEvent

public class AdjustableReader extends Application
{
  private static final int WIDTH = 465;
  private static final int HEIGHT = 200;
  private static final double BRIGHTNESS_RATIO = 3.0;
  private static final double FONT_RATIO = 1.25;

  private Text text = new Text(
    "John Nickell\n\n" +
    "MySpace URL: http://www.myspace.com/oneBusyStudent.html\n\n" +
    "Top 5 friends: Rigo, Nick, Aris, Stephen, Bavitha\n\n" +
    "Latest blog entry:\n" +
    "Just got back from the ACM programming contest." +
    " Our Park University team took second place. Whoohoo!\n\n" +
    "About me:\n" +
    "I enjoy school, work, cars, and mountain bikes, though I" +
    " can’t wait to finish the first one! I like to experience" +
    " different countries and cultures. I've spent time in Armenia,"
    + " Germany, Kazakhstan, and Brazil.");
  private double brightness = 1.0; //  dim = 0; bright = +infinity
  private double fontSize = 12.0;  //  cast to int before using

  //*****************************************************************

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

Figure 18.22a AdjustableReader program – part A

```
//***************************************************************

@Override
public void start(Stage stage)
{
  BorderPane pane = new BorderPane();
  Scene scene = new Scene(pane, WIDTH, HEIGHT);

  stage.setTitle("Adjustable Reader");
  stage.setScene(scene);
  createContents(pane);
  stage.show();
} // end start

//***************************************************************

private void createContents(BorderPane pane)
{
  MenuBar mBar = new MenuBar();
  Menu menu1 = new Menu("Brightness");
  Menu menu2 = new Menu("Font Size");
  MenuItem mi1 = new MenuItem("Brighter");
  MenuItem mi2 = new MenuItem("Dimmer");
  MenuItem mi3 = new MenuItem("Larger");
  MenuItem mi4 = new MenuItem("Smaller");
  ScrollPane scroll = new ScrollPane();

  pane.setTop(mBar);
  pane.setCenter(scroll);
  mBar.getMenus().addAll(menu1, menu2);
  menu1.getItems().addAll(mi1, mi2);
  menu2.getItems().addAll(mi3, mi4);
  scroll.setContent(text);

  mi1.setOnAction(new BrightnessHandler());
  mi2.setOnAction(new BrightnessHandler());
  mi3.setOnAction(new SizeHandler());
  mi4.setOnAction(new SizeHandler());

  scroll.setPannable(true);
  // brighten text by making it more opaque
  text.setOpacity(brightness / (1.0 + brightness));
  text.setStyle("-fx-font-size: " + (int) fontSize + "pt;");
} // end createContents
```

Figure 18.22b AdjustableReader program – part B

Text brightness is altered by the `Text.setOpacity` method calls near the end of the `createContents` method in Figure 18.22b and at the end of the `handle` method in the `BrightnessHandler` in Figure 18.22c. On a white background, the black text is dim when the text is nearly transparent, and it is bright when the text is opaque. With opacity given by the expression, brightness / (1.0 + brightness), as brightness varies between 0.0 and +infinity, text opacity varies between 0.0 and 1.0.

Text size is altered by multiplying or dividing by a constant ratio. The obvious type of variable for font size is an `int`. However, once an integer size got scaled down to 1, it would get stuck there, and the user could not scale back up again. That's because if we multiply the integer 1 by 1.25, the integer product is still 1. To keep from getting stuck at 1, we declare `fontSize` to be a `double`. Then, before we use the `fontSize` variable to create the font for the display, we cast it to an `int`.

```
  //***************************************************************

  // Adjust opacity of the text area

  private class BrightnessHandler implements EventHandler<ActionEvent>
  {
    public void handle(ActionEvent e)
    {
      if (((MenuItem) e.getSource()).getText().equals("Brighter"))
      {
        brightness *= BRIGHTNESS_RATIO;
      }
      else                      // dimmer selected
      {
        brightness /= BRIGHTNESS_RATIO;
      }
      text.setOpacity(brightness / (1.0 + brightness));
    } // end handle
  } // end class BrightnessHandler

  //***************************************************************

  // Adjust font size of the text area's text

  private class SizeHandler implements EventHandler<ActionEvent>
  {
    public void handle(ActionEvent e)
    {
      if (((MenuItem) e.getSource()).getText().equals("Larger"))
      {
        fontSize *= FONT_RATIO;
      }
      else                      // smaller selected
      {
        fontSize /= FONT_RATIO;
      }
        text.setStyle("-fx-font-size: " + (int) fontSize + "pt;");
    } // end handle
  } // end class SizeHandler
} // end class AdjustableReader
```

Figure 18.22c AdjustableReader program – part C

Groups

You have now seen many examples of components that are subclasses of the `Control` class, like `Label`, `Button`, `RadioButton`, `ComboBox`, and `ScollPane`. You also know about containers that are subclasses of the `Pane` class, like `FlowPane`, `BorderPane`, `GridPane`, `TilePane`, `VBox`, and `HBox`. Typically we put the `Control` components into `Pane` containers because `Panes` automatically adjust their component locations as more components are added and as users resize windows. You have seen several examples where panes contain other panes. In our brief discussion of the AdjustableReader program, we mentioned components that a component like a `ScrollPane` can contain other components, or panes that themselves can contain other panes and/or components. All this nesting is possible because all of these classes are subclasses of the `Parent` class, which first defines that ubiquitous `getChildren` method. Figure 18.23 extends the previous chapter's Figure 17.7 to include the new classes introduced in this chapter.

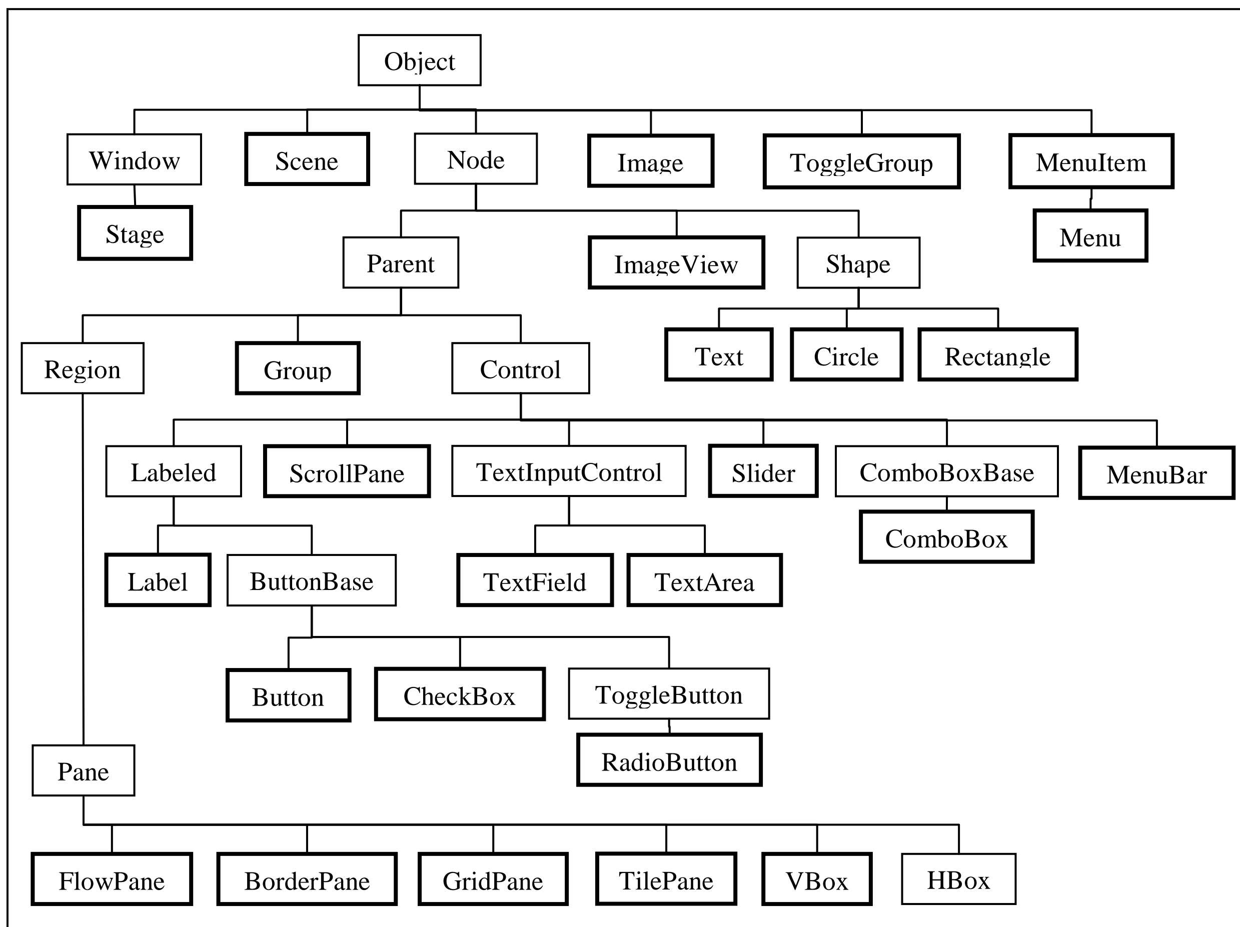


Figure 18.23 Inheritance relationships among JavaFX classes discussed in Chapters S17 and S18
The heavy bordered classes appear in examples in either Chapter S17 or Chapter S18 or both.

Figure 18.23 shows that the `Parent` class has another subclass called `Group`. Conceptually, a `Group` is halfway between a `Pane` and a `Control`. It doesn't perform full-fledged layout management like a `Pane` does, but it does some layout management. For example, as a program adds children to a `Group`, each new child gets piled on top of all previous children. By default, a `Group` makes itself wide enough and high enough to accommodate its widest and tallest children. Typically, we use methods like `setPrefWidth`,

setPrefHeight, setTranslateX, and setTranslateY to re-size and reposition each component within the group's area.

Like a Control and a Pane, a Group can contain any kind of Node object. The Node class is the parent of the Parent class, and one of Parent's siblings under Node is the Shape class. Since Shape is not a subclass of Parent, a Shape cannot have children, but Shapes like Circle and Rectangle are quite useful, nevertheless. Some Circle and Rectangle constructors include offset as well as size parameters, so when you add one of these shapes to a group, you can set its dimensions and location at the same time.

By default, most components are opaque. So when you add another child to a group, the new child covers all previously added children that happen to be under it. You use translation methods or offset parameters to put different components at different places, so that no two components share the same area. Then, the order of entry into the group doesn't matter, and all components can be seen in their entirety. But frequently you'll want to put some components on top of others to create more sophisticated structures. Then you can specify the amount of opaqueness in a continuous range from 0.0 (completely transparent) to 1.0 (completely opaque.) You can also specify color gradients. By carefully controlling relative locations, opaqueness, and color gradients, you can create some very interesting effects.

Figure 18.24 shows a sequence of outputs from a program that simulates an eclipse of the moon by the earth.

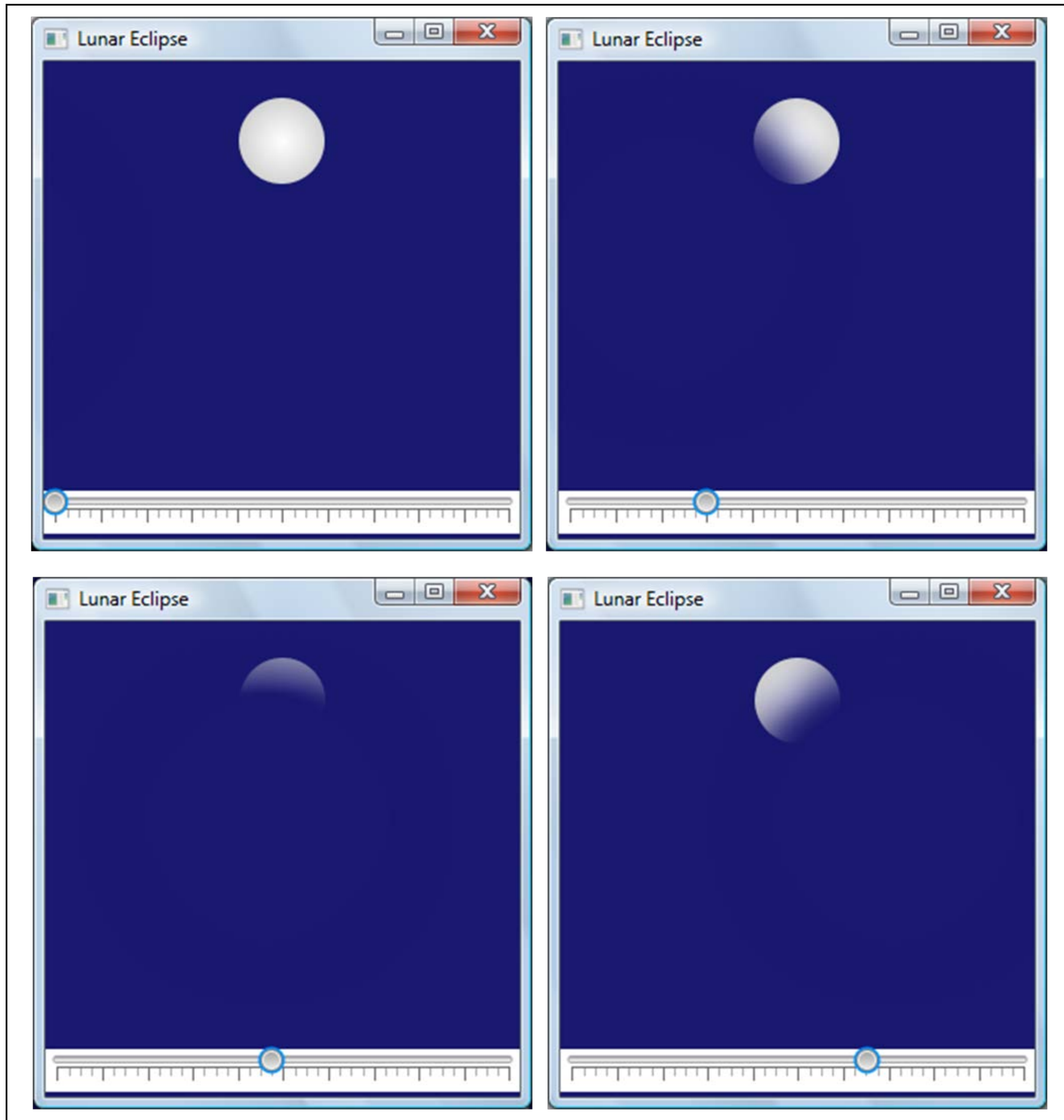


Figure 18.24 Four displays of a program that uses a slider to mimic a lunar eclipse

This program adds three distinct objects to a `Group`, in the following order:

(1) A dark blue `Rectangle` object that represents the night sky (we don't bother with stars). This object establishes the size of the `Group`'s area.

(2) A light `Circle` object that represents the moon. This covers part the rectangular sky behind. It's white in the center, and starting at half its radius, its color grades to light gray at the perimeter. This makes it look spherical.

(3) Another `Circle` object that represents the earth's shadow. Its diameter is almost four times the diameter of the moon. It covers part of the sky and depending on its position it can also cover part or all of the moon. To represent the fact that the sun is not a point source, we use a special effect to blur the edges of

the earth's shadow.

The moon revolves around the earth in the direction of the earth's rotation. If we track the moon, it remains at a fixed point in our field of view, and the earth's shadow appears to cross the moon from left to right. The program allows the user to simulate this effect by repositioning the earth's shadow relative to the moon. As the earth's shadow passes over the moon, it hides the part of the moon it covers. To add interest, we model a lunar eclipse that is not quite total by placing the vertical position of the center of the earth's shadow somewhat below the vertical position of the center of the moon. The horizontal position of the center of the earth's shadow lines up with the little round button in the *slider* at the bottom of the four images. The user controls the position of the earth's shadow by using the mouse to drag this slider button horizontally. We'll discuss the slider in more detail in the following subsection.

Figure 18.25a shows the first part of the program that generates this output. The five class constants are the width and height of the night sky rectangle, the radius of the earth, and the x- and y-coordinates of the center of the moon. The program has two instance variables, a `Circle` called `earth` to represent the earth's shadow, and the `Slider`. The program accesses these two variables from two different methods, the `createContents` method and the `handler` method. For simplicity, we initially align the earth's shadow with the moon in the x-direction. But in the y-direction, we put the center of the earth's shadow 75 pixels below the center of the moon.

```
/*****************************************************************
* LunarEclipse.java
* Dean & Dean
*
* This program mimics an eclipse of the moon by the earth.
* The slider allows the user to drag a shadow across a circle.
*****************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.*;        // Scene, Group
import javafx.scene.paint.*;  // Color, RadialGradient, CycleMethod, Stop
import javafx.scene.control.Slider;
import javafx.scene.input.MouseEvent;
import javafx.scene.shape.*;  // Circle, Rectangle,
import javafx.scene.effect.*; // BoxBlur, BlendMode
import javafx.event.*;        // EventHandler, Event

public class LunarEclipse extends Application
{
  private static final int WIDTH = 300;
  private static final int HEIGHT = 300;
  private static final int RADIUS = 100;     // radius of earth
  private static final int CENTER_X = 150;   // center of moon
  private static final int CENTER_Y = 50;    // center of moon
  private Circle earth =
    new Circle(CENTER_X, CENTER_Y + 75, RADIUS);
  private Slider slider = new Slider(0, 100, 0); // min, max, init

  //*************************************************************

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

  //*************************************************************

  public void start(Stage stage)
  {
    Group root = new Group();
    Scene scene = new Scene(root, WIDTH, HEIGHT);

    stage.setTitle("Lunar Eclipse");
    stage.setScene(scene);
    createContents(root);
    stage.show();
  } // end start
```

Figure 18.25a LunarEclipse program – part A

The `start` method creates a `Group` called `root`, which represents the root of the structural tree that contains all the components in the scene. This `root` is analogous to the object we called `pane` in previous programs.

Figure 18.25b shows the rest of the LunarEclipse program. In the `createContents` method, the first three statements create the other two `Shape` components, `sky` and `moon`.

```
  //***************************************************************

  private void createContents(Group root)
  {
    Rectangle sky =
      new Rectangle(WIDTH, HEIGHT, Color.MIDNIGHTBLUE);
    RadialGradient gradient = new RadialGradient(
      0, 0, 0.5, 0.5, 0.5, true, CycleMethod.NO_CYCLE,
      new Stop(0.0, Color.WHITE), new Stop(1.0, Color.LIGHTGRAY));
    Circle moon =
      new Circle(CENTER_X, CENTER_Y, 0.27 * RADIUS, gradient);
    Group skyGroup = new Group(sky, moon, earth);

    earth.setFill(Color.MIDNIGHTBLUE);       // same color as sky
    earth.setEffect(new BoxBlur(40, 40, 1)); // due to sun's width
    root.getChildren().addAll(skyGroup, slider);
    slider.setTranslateY(HEIGHT - 30);
    slider.setPrefWidth(WIDTH);
    slider.setMajorTickUnit(10);
    slider.setShowTickMarks(true);
    slider.setStyle("-fx-background-color: white");
    slider.setOnMouseDragged(new SliderHandler());
    slider.fireEvent(new Event(MouseEvent.MOUSE_DRAGGED));
  } // end createContents

  //***************************************************************

  private class SliderHandler implements EventHandler<Event>
  {
    public void handle(Event e)
    {
      double earthCenterX = 2.0 * CENTER_X - 3.5 * RADIUS +
        4 * RADIUS * slider.getValue() / 100;

      earth.setCenterX(earthCenterX);
    } // end handle
  } // end class SliderHandler
} // end class LunarEclipse
```

We want this messy formula to be in just one place.

Figure 18.25b LunarEclipse program – part B

The second statement creates a *radial gradient*. This is a powerful tool, but the `RadialGradient` constructor parameters are tricky. The third and fourth parameters are the x- and y- coordinates of the center of an ellipse within which the radial gradient occurs, relative to a circle's diameter (or a rectangle's width or height). The fifth parameter is the distance over which the gradient occurs, relative to the diameter. The sixth parameter should be `true`. The seventh parameter tells whether you want the gradient to stop (`NO_CYCLE`), repeat or reflect when the distance relative to the diameter reaches the fifth parameter's value. The varargs starting at the eighth parameter specify gradient-bounding radii (relative to the radius) where the color equals the associated color. The first and second parameters allow you to offset the start of the gradient from the center specified in the third and fourth parameters. This simulates light coming from a point different from the viewpoint. The first parameter is the angle in clockwise degrees from the +x axis. The second parameter is the offset in this direction, relative to the radius. For example, if you want to simulate light coming from an infinitely distant source in the plane of the picture 45 degrees above and to the right of the viewpoint, make the first two parameters -45 and 1.0,

respectively.

The fourth statement creates another Group called skyGroup and populates it with sky, moon, and earth, in that order. The next statement uses a setFill method call to make the color of the earth's shadow the same as that of the sky. We could have done that earlier, when we instantiated earth, but we prefer to do it here, shortly after we instantiate the sky with the same color. The next statement uses a BoxBlur object to simulate the fact that the sun is not a point source.

Sliders

In the next statement in Figure 18.25b's createContents method, the Group called root adds to itself the other Group called skyGroup and a slider. A slider allows the user to select a value from a range of values. To select a value, the user drags the slider's button along a bar that portrays that range of values. As the user drags the slider's button to the right, the earth's shadow, which is centered over the button, moves to the right. Several slider method calls in the createContents method establish the slider's location, its width, and various rendering details. To make the slider responsive to the mouse, the next-to-last statement in the createContents method adds an event handler.

The last statement in createContents invokes the slider handler from within the program. The handler's purpose is responding to user mouse-drag events. So why do we also use a fireEvent method call to call the handler from the code? Because we want to complete the createContents initialization activity by doing some additional work that the handler does every time it responds to a mouse event, and we don't want to have to duplicate the messy formula which does this work. This messy formula calculates of the position of the earth's shadow in the x-direction, which – as we've said – depends on the current position of the slider's button.

Summary

- Hidden layout managers automate the positioning of components in Pane containers.
- The FlowPane class implements a simple one-compartment layout scheme and allows more than one component to be inserted into the compartment.
- The GridPane class implements a rectangular grid, and the programmer assigns each component to a particular column and row. Column widths and row heights are set by the widest and highest component in each column and row. The programmer may allocate more than one grid cell to any component.
- The HBox and VBox layouts align entries sequentially in one row or one column, respectively.
- The BorderPane layout provides five regions/compartments—top, bottom, left, right, and center—in which to insert components.
- The GridPane lays out its components in a rectangular grid. The programmer specifies each component's row and column. Column and row widths and heights are set by widths and heights of largest contained cells. Large components can use more than one adjacent cell. Each cell can hold no more than one component.
- The TilePane lays out its components in a rectangular grid of equal-sized cells. Each cell can hold only one component.
- If you have a complicated window with many components, you might want to compartmentalize them by using a pane instead of a component is a cell and thereby making it possible to put more than one component into the original cell.
- To display multiple lines of text, use a TextArea component or its more versatile cousin, a ScrollPane.
- A CheckBox component displays a small square with an identifying label. Users click the check box in order to toggle it between selected and unselected.
- A RadioButton component displays a small circle with a label to its right. If an unselected button is clicked, the clicked button becomes selected, and the previously selected button in the group becomes

unselected.

- A ComboBox component allows the user to select an item from a list of items. ComboBox components are called “combo boxes” because they are a combination of a text box (normally, they look just like a text box) and a list (when the down arrow is clicked, they look like a list).
- A MenuBar contains Menus, which in turn contain MenuItems.
- A Group is a stack of Nodes whose relative positions can be set by the program but do not change with window resizing.
- A Slider enables a user to enter a variable value by dragging a button along a tic-marked bar.

Review Questions

§18.2 GUI Design and Layout Management

1. Pane layouts adapt automatically to changes in the size of a container or one of its components. (T / F)
2. Which package has classes with hidden layout managers?

§18.3 FlowPane and GridPane – Competing Layout Philosophies

3. How does the FlowPane arrange its components?
4. Write a single statement that tells a Pane called pane to use center-right alignment.
5. For the GridPane version of the Greeting program, write a statement that makes the greeting start under the name box, and allocate just enough cells to avoid text-box resizing if the name is long.

§18.4 Externally Driven VBox with Image

6. Write the method call chain that returns strings previously passed to the hidden Application code by Application’s class method, launch.

§18.5 BorderPane

7. What are the five regions in a BorderPane layout?
8. The sizes of the five regions in a BorderPane are determined at runtime based on the contents of the four outer regions. (T / F)
9. By default, how many components can you put in any one region of a BorderPane?
10. Given the reference, `BorderPane pane = new BorderPane();` write a single statements that adds a new Label with the text “Stop” to the center region of a BorderPane. The label should be centered within the center region.

§18.6 TilePane

11. When you instantiate a TilePane, you should always specify both the number of rows and the number of columns. (T / F)
12. In a TilePane, all cells are the same size. (T / F)

§18.7 Tic-Tac-Toe Example

13. What happens to the xTurn variable in the Tic-Tac-Toe program if you click the same cell twice?

§18.9 Embedded Panes

14. Why are embedded panes particularly useful in BorderPane, TilePane, and GridPane outer containers (as opposed to FlowPane, VBox, and HBox outer containers)?

§18.11 TextArea Component

15. TextArea components are editable by default. (T / F).
16. TextArea components employ line wrap by default. (T / F).

§18.12 CheckBox Component

17. What happens if you click a check box that’s already selected?
18. Write a code fragment that creates a check box named attendance, with an “I will attend” label, and put it into its selected state.

§18.13 RadioButton Component

19. What happens if you click a radio button that is already selected?

20. What happens if you click an initially unselected radio button that is a member of a `ToggleGroup`?

§18.14 ComboBox Component

21. How are combo boxes and radio button groups similar?

22. What method call returns the current selection in a combo box?

§18.15 Job Application Example

23. In the JobApplication program, what happens if the following code is omitted?

```
ToggleGroup radioGroup = new ToggleGroup();
...
good.setToggleGroup(radioGroup);
bad.setToggleGroup(radioGroup);
```

§18.16 More GUI Components

24. The `Group` class implements the `getChildren` method. (T / F)

25. Provide a `Slider` constructor call where the minimum value is 0, the maximum value is 50, and the initial value is 10. Hint: Look up the answer on Oracle's Java API Web site.

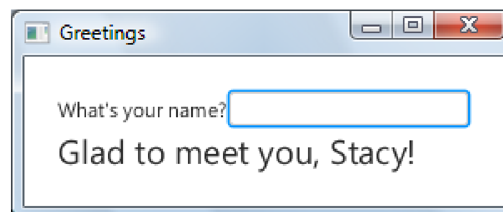
Exercises

1. [after §18.3] In a `FlowPane`, a button component expands so that it completely fills the size of the region in which it is placed. (T / F)

2. [after §18.3] Assuming you are using a `GridPane`, write a code fragment that fixes the width of the second column at 200 pixels but lets the `GridPane` manager establish the widths of all other columns.

3. [after §18.4] Rewrite the DanceRecital program to eliminate the separate driver by moving the `main` method to the driven class, and then use the simpler one-parameter `launch` method.

4. [after §18.5] Provide a complete program that is a modification of Chapter 17's Greeting program. Your new program should use a `BorderPane` (instead of a `FlowPane`), and it should generate the following display after a name has been entered. Use defaults to make your code as simple as possible.



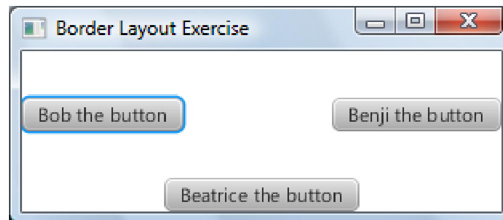
5. [after §18.5] In a `BorderPane`, what happens if the right region is empty? Said another way, which region(s), if any, expand(s) if the right region is empty?

6. [after §18.5] Assume you have this program:

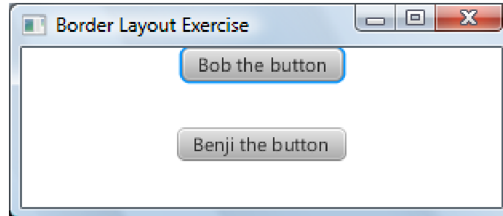
```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.scene.control.Label;

public class BorderPaneExercise extends Application
{
  @Override
  public void start(Stage stage)
  {
```

```
    BorderPane pane = new BorderPane();

    stage.setTitle("Border Layout Exercise");
    stage.setScene(new Scene(pane, 300, 100));
    pane.setTop(new Label("Lisa the label"));
    pane.setCenter(new Label("LaToya the label"));
    pane.setBottom(new Label("Lemmy the label"));
    stage.show();
  } // end start

  //*********************************************************

  public static void main(String[] args)
  {
    launch();
  } // end main
} // end class BorderPaneExercise
```

(a) Specify the changes you would make to the above code to produce this output:



(b) Specify the changes you would make to the above code to produce this output:



7. [after §18.6] If a `Button` component is directly added to a `TilePane` cell, it expands so that it completely fills the size of its cell. (T / F)

8. [after §18.6] Assume a `TilePane` is constructed with no specification of orientation. Given the following code fragment, draw a picture that illustrates the buttons' positions within the program's window.

```
public void start(Stage stage)
{
  TilePane pane = new TilePane();

  stage.setScene(new Scene(pane));
  pane.setPrefColumns(3);
  // This makes buttons big enough so that window can conform
  pane.setStyle("-fx-font-size: 16");

  for (int i=0; i<7; i++)
  {
    pane.getChildren().add(new Button(Integer.toString(i+1)));
  } // end for i
  stage.show();
} // end start
```

9. [after §18.9] What type of container might you put into an individual `BorderPane` region or `TilePane` or `GridPane` cell to allow that region or cell to contain more than one component? [Hint: What class name sounds like it "can have children"? Look it up in Oracle API documentation. The answer may be far more general than you might expect.]

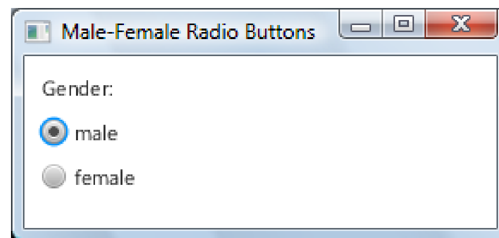
10. [after §18.11] Suppose you're given a window with two `TextArea` components, named `msg1` and `msg2`, and a `Button` component. When clicked, the button swaps the contents of the two text areas. Provide the code that performs the swap operation. More specifically, provide the code that goes inside the below `handle` method:

```
private class Handler implements EventHandler<ActionEvent>
{
  public void handle(ActionEvent e)
  {
    ...
  } // end handle
} // end class Handler
```

11. [after §18.12] Write a statement that creates a check box named `bold`. The check box should be unselected, and it should have a "boldface type" label.

12. [after §18.12] How can your code determine whether a check box is selected or not?

13. [after §18.13] Provide a `createContents` method for a program that displays this window:



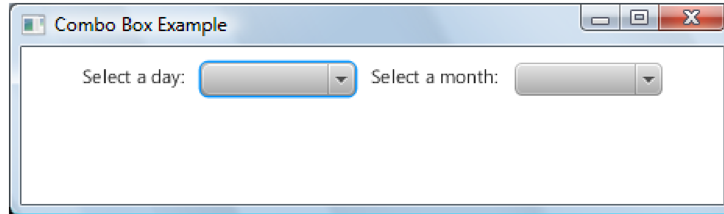
The male and female radio buttons should behave in the normal fashion—when one is selected, the other is unselected. Note that the male button is selected when the window initially displays. Your `createContents` method must work in conjunction with this program skeleton:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.geometry.Insets;
import javafx.scene.control.*; // RadioButton, ToggleGroup

public class MaleFemaleRadioButtons extends Application
{
  private RadioButton male = new RadioButton("male");
  private RadioButton female = new RadioButton("female");

  @Override
  public void start(Stage stage)
  {
    VBox pane = new VBox(10);

    stage.setScene(new Scene(pane, 275, 100));
    stage.setTitle("Male-Female Radio Buttons");
    createContents(pane);
    stage.show();
  } // end start

  //  <The createContents method goes here.>
```

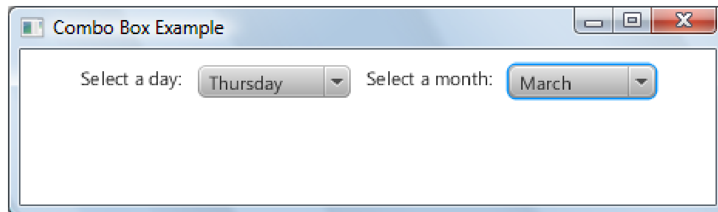
```
    public static void main(String[] args)
    {
      launch();
    }
  } // end class MaleFemaleRadioButtons
```

14. [after §18.14] The `CheckBox`, `RadioButton`, and `ComboBox` components are defined in what package(s)?

15. [after §18.14] Provide a `createContents` method for a program that initially displays this window:



Then, after the user clicks the day down arrow and selects Thursday and clicks the month down arrow and selects March, the program's display looks like this:



Your `createContents` method must work in conjunction with this program skeleton:

```
import javafx.scene.layout.HBox;
import javafx.scene.control.*;    // Label, ComboBox
import javafx.geometry.*;         // Insets, Pos

public class ComboBoxExample extends Application
{
  private ComboBox<String> daysBox = new ComboBox<>();
  private ComboBox<String> monthsBox = new ComboBox<>();
  private String[] days =
    {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"};
  private String[] months = {"January", "February", "March",
    "April", "May", "June", "July", "August", "September",
    "October", "November", "December"};

  @Override
  public void start(Stage stage)
  {
    HBox pane = new HBox(10);

    stage.setTitle("Combo Box Example");
    stage.setScene(new Scene(pane, 450, 100));
    createContents(pane);
    stage.show();
  } // end start

  // <The createContents method goes here.>
```

```
    public static void main(String[] args)
    {
      launch();
    } // end main
  } // end class ComboBoxExample
```

Include appropriate padding and alignment.

Review Question Solutions

1. True.
2. Classes with hidden layout managers are in the `javafx.scene.layout` package.
3. The `FlowPane` manager places components left-to-right in a row until it runs out of space, and then it goes to the next row and does the same thing, and so on.
4. `pane.setAlignment(Pos.CENTER_RIGHT);`
5. `pane.add(greeting, 1, 1, 2, 1);`
6. To retrieve passed-in string data, use:
`List<String> params = getParameters().getUnnamed();`
7. The five regions of a `BorderPane` layout are Top at the top, Bottom at the bottom, and Left, Center, and Right in a row between them.
8. True.
9. `BorderPane`, `TilePane`, and `GridPane` containers can have no more than one component per region.
10. `pane.setCenter(new Label("Stop")); // center is the default alignment`
11. False. If orientation is unspecified or horizontal, specify columns. If orientation is vertical, specify rows.
12. True.
13. Nothing. It does not change value.
14. Embedded panes are particularly useful in `BorderPane`, `TilePane`, and `GridPane` outer containers (as opposed to `FlowPane`, `VBox`, and `HBox` outer containers) because each `BorderPane` region or each `TilePane` or `GridPane` cell can store only one component.
15. True. `TextArea` components are editable by default.
16. False. `TextArea` components do not employ line wrap by default.
17. If you click a check box that's already selected, the check box becomes unselected.
18. The following code creates a check box named `attendance` with an "I will attend" label and then puts it into its selected state:

```
CheckBox attendance = new CheckBox("I will attend");
attendance.setSelected(true);
```

19. Nothing. It stays selected.
20. The clicked button becomes selected and all other buttons in the group become unselected.
21. Combo boxes and radio button groups are similar in that they both allow the user to select one item from a list of items.
22. To determine the current selection for a combo box, call `getValue`.
23. If the `radioGroup` code is omitted from the JobApplication program, the program still compiles and runs, but the radio buttons operate independently. In other words, clicking one radio button will not cause the other one to be unselected.
24. True. The `Group` class is a subclass of the `Parent` class and therefore implements the `getChildren` method.

25. `Slider` constructor call:

```
new Slider(0, 50, 10);
```