CHAPTER S17

GUI Programming Basics

Objectives

- Understand the event-driven programming paradigm. In particular, understand what it means to fire an event, and understand an event handler.
- Use the `Stage`, `Scene`, and `FlowPane` classes to implement window functionality.
- Create and use `Label`, `TextField`, and `Button` components.
- Implement the `EventHandler` interface for the `TextField` and `Button` components.
- Understand what an inner class is and implement a handler as an inner class.
- Know the difference between an anonymous inner class and a standard inner class.
- Be able to distinguish multiple event sources.
- Be able to use a `Builder` to create and use a custom "constructor".
- Create and use dialog boxes.
- Acquire a general understanding of JavaFX GUI class organization
- Learn how to handle images and implement mouse event handlers.

Outline

17.1 Introduction

Chapters 17 and 18 presented *graphical user interface* (GUI) concepts using the Swing GUI toolkit, whereas this chapter and the next one present GUI concepts using the JavaFX GUI toolkit. Normally, teachers and readers will decide which toolkit they want to learn, and then they'll read either Chapters 17 and 18 or this chapter and the next one. Since this chapter and the next one provide concepts that are parallel to the concepts covered in Chapters 17 and 18, this chapter and the next one are considered supplements, and that's why they are named Chapters S17 and S18 (the *S*'s stand for "supplement").

You've probably heard the term GUI, and you probably know that it's pronounced "gooey." But do GUI's three words, Graphical User Interface, make sense? "Graphical" refers to pictures, "user" refers to a person, and "interface" refers to communication. Thus, GUI programming employs pictures—like windows, labels, text boxes, buttons, and so forth—to communicate with users. For example, Figure 17.1 shows a window with two labels, a text box and a button. We'll describe windows, labels, text boxes, and buttons in detail later on.

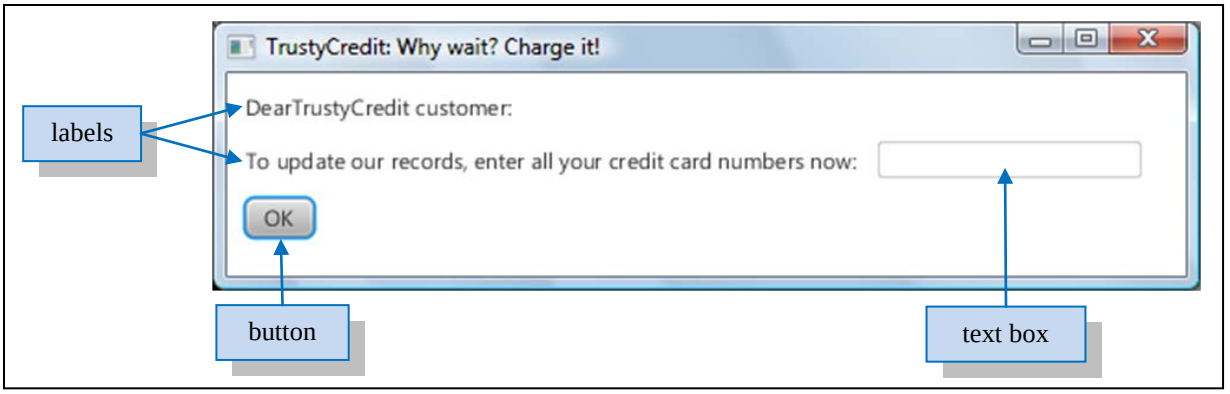


Figure 17.1 Example window that uses two labels, a text box, and a button

In the old days, program interfaces consisted of just text. Programs would prompt the user with a text question, and users would respond with a text answer. That's what we've done up to now. Text input/output (I/O) works well in many situations, but you can't get around the fact that some people consider text display to be boring. Many of today's users expect programs to be livelier. They expect windows, buttons, colors, and so on for input and output. They expect GUI.

Although companies still write many text-based programs for internal use, they normally write GUI-based programs for programs that are to be used externally. It's important that external programs be GUI based because external programs go to customers, and customers typically won't buy programs unless they are GUI based. So if you want to write programs that people will buy, you'd better learn GUI programming. We start this chapter with an overview of basic GUI concepts and terminology. We then move on to a bare-bones program where we introduce basic GUI syntax. We next describe various graphical components that can be placed inside a window. Then we describe handlers and the inner classes that contain them. After that we describe the `Color` class, for generating color. Finally, we describe images and mouse events. Section 17.18 shows how to use a mouse to drag an image around a window.

You may have noticed optional GUI-track sections at the end of about half of the prior chapters. The GUI material in this chapter and the next is different from the GUI material in the earlier chapters, and it does not depend on the earlier chapters' GUI material. So if you skipped the earlier GUI material, no worries.

To understand this chapter, you need to be familiar with object-oriented programming, arrays, inheritance, and exception handling. As such, you need to have read up through Chapter 15. This chapter does not depend on material covered in Chapter 16.

17.2 Event-Driven Programming Basics

GUI programs usually use *event-driven programming* techniques. The basic idea behind event-driven programming is that the program waits for events to occur and the program responds to events if and when they occur.

Terminology

So what is an event? An *event* is an object that tells your program that something has happened. For example, if the user clicks a button, then an event is generated, and it tells your program that a particular button was clicked. More formally, when the user clicks a button, we say that the button object *fires an event*. Note these additional event examples:

User Action	What Happens
Pressing the Enter key while the cursor is inside a text box.	The text box object fires an event, and it tells the program that the Enter key was pressed within the text box.
Clicking a menu item.	The menu item object fires an event, and it tells the program that the menu item was selected.
Closing a window (clicking on the window's top-right corner "X" button).	The window object fires an event, and it tells the program that the window's close button was clicked.

If an event is fired, and you want your program to handle the fired event, then you need to create a *handler* for the event. For example, if you want your program to do something when the user clicks a particular button, you need to create a handler for the button and *register* that handler with the button. For now, think of a handler as a dutiful attendant who listens carefully with a big ear. If an event is fired and there's nothing listening to it, then the fired event is never "heard" and there's no response to it. On the other hand, if there is a handler listening to a fired event, then the handler "hears" the event and responds by performing a specified set of operations. The way the program responds is by executing a chunk of code known as an *event handler*. See Figure 17.2. It depicts a button being pressed (see the mouse pointer), an event being fired (see the sound waves), a handler hearing the event (see the ear) and responding appropriately (see the arrow going down the event-handler code). This system of event handling is known as the *event-delegation model*—event handling is "delegated" to a particular handler.

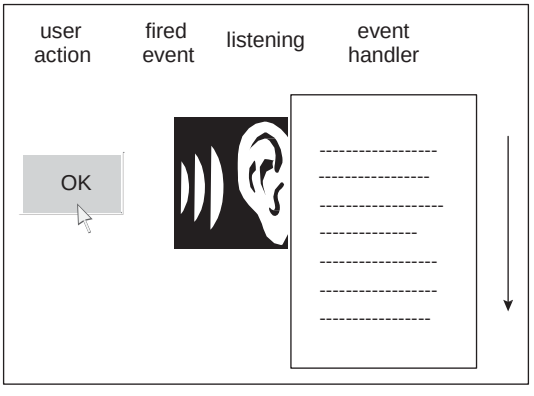


Figure 17.2 What happens when a button is pressed

The Event-Driven Programming Framework

Based on the above description, event-driven programming may feel like an altogether new type of programming. Particularly the part about firing an event and listening for a fired event. Many people are fine with the idea of event-driven programming being a new type of programming. But the truth of the matter is that it's really just

object-oriented programming with window dressing. Make that lots of window dressing. Oracle provides an extensive collection of GUI classes that, together, form a framework on which to build GUI applications. That framework is comprised of classes, methods, inheritance, and so on. In other words, it's comprised of OOP components. As a programmer, you don't have to understand all the details of how the framework works; you just have to understand it well enough to use it. For example, you have to know how to plug in your event handlers properly. Figure 17.3 provides a high-level, graphic illustration of what we're talking about.

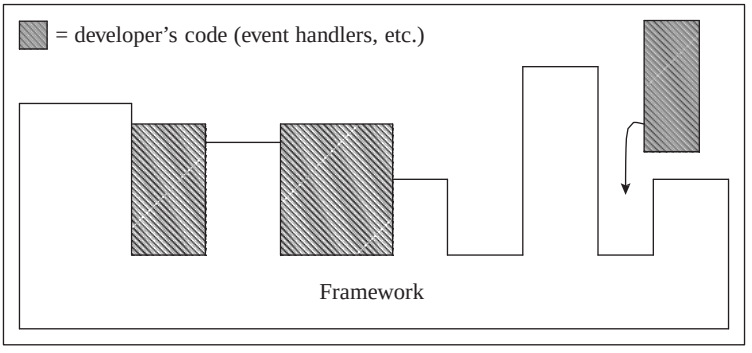


Figure 17.3 Event-driven programming framework

Why does Oracle bother to provide the event-driven programming framework? It satisfies the goal of getting maximum benefit from minimum code. With the help of the framework, Java programmers can get a GUI program up and running with a relatively small amount of effort. Initially, the effort might not seem so small, but when you consider all that the GUI program does (automatic event firing, listening for fired events, and so on), you'll find that your return on investment is quite good.

17.3 A Simple Window Program

OK. Enough talk about concepts. Time to roll up your sleeves and get your hands dirty with some code. To get a feel for the big picture, let's start with a simple GUI program that displays a single line of text inside a simple window, like this:

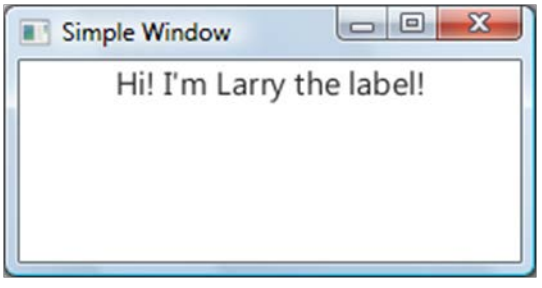


Figure 17.4 contains a SimpleWindow program that generates this window. Notice that even this very simple program requires a relatively large number of `import` statements. That's typical of GUI programs. We'll discuss these imports in more detail in the next section. At this point, let's focus on the principal features of a GUI program.

Every JavaFX GUI program is like a big event handler that responds to a *launch* event. The launch event occurs when a class that extends the JavaFX API `Application` class calls `Application`'s pre-written class

method, `launch`. This `launch` method creates an object which in turn calls a `start` method that the programmer's code must define because `start` is an `abstract` method in the extended `Application` class. This `start` method is our GUI program's front end.

```
/*********************************************************
* SimpleWindow.java
* Dean & Dean
*
* This program displays a label in a window.
*********************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.text.Font;
import javafx.geometry.Pos;

public class SimpleWindow extends Application
{
  private static final int WIDTH = 250;
  private static final int HEIGHT = 100;

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

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

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

  @Override
  public void start(Stage stage)
  {
    Label label = new Label("Hi! I'm Larry the label!");

    label.setFont(new Font(16));
    label.setAlignment(Pos.TOP_CENTER);
    stage.setTitle("Simple Window");
    stage.setScene(new Scene(label, WIDTH, HEIGHT));
    stage.show();
  } // end start
} // end class SimpleWindow
```

A subclass of the `Application` class.

This creates a label.

These modify label attributes.

This adds the label to the scene and the scene to the stage.

Figure 17.4 SimpleWindow program

Before examining the `start` method, look at the SimpleWindow program's `main` method. GUI programs typically create a window with GUI components, and then they just sit around waiting for the user to do something

like click a button, select a menu option, and so on. Thus, `main` is very short—it just calls the `launch` method, which creates an empty window and then calls `start`.

Look at the `start` method. For this initial example, our overriding `start` method instantiates a simple type of component called a `Label`. We'll describe `Label` in more detail later, but as Figure 17.4 shows you, it is basically just an object that displays a string of text. Within the `start` method, note how you can adjust the font in the label's text:

```
label.setFont(new Font(16));
```

The `setFont` method call specifies a label's font with the help of a `Font` object argument. A `Font` object is a wrapper for a font's name and size. In the code above, the `Font` object is created with a single argument of 16. That tells the JVM to use the default font type and a font size of 16 points. A point is a unit of measure that you should be familiar with if you use a word processor. A typical Microsoft Word document uses a point size of 12. Within the `start` method, the setAlignment method call specifies the location of the label in the window. Next, the `setTitle` method call puts a title in the banner at the top of the window frame:

```
stage.setTitle("Simple Window");
```

Notice how the `setTitle` method call is prefaced with a `stage` calling object. A JavaFX application's window frame is represented by a *stage*. As you can see in Figure 17.4, when the `start` method is called, it is automatically provided with a `stage` parameter.

Next in Figure 17.4's `start` method is the `setScene` method call. It passes in a `Scene` object, which stores and displays the picture that's in the stage's empty window frame. A scene must always contain exactly one item, but as you will see, there is great flexibility in what that item can be. In this simple case, the `setScene` method adds the previously created label to the scene, and then it puts that scene on the stage. Here's the relevant code:

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

In the `Scene` constructor call above, the first argument, `label`, is the one item the scene contains. The second and third arguments assign the width and height of the scene (the area inside the window's frame). See Figure 17.4 and note how the SimpleWindow program assigns 250 to the `WIDTH` named constant and assigns 100 to the `HEIGHT` named constant. The width and height values are specified in terms of pixels. A pixel is a computer monitor's smallest displayable unit, and it displays as a dot on the screen. If you create a scene with a width of 250 and a height of 100, then your window will consist of 100 rows with 250 pixels in each row. Each pixel displays with a certain color. The pixels form a picture by having different colors for the different pixels.

To provide perspective on how large a 250-by-100 pixel window is, you need to know the dimensions, in pixels, of an entire computer screen. The dimensions of a computer screen are referred to as the screen's *resolution*. A typical resolution for a desktop monitor would be 1280 x 1024. A typical resolution for a smartphone's display would be 960 x 640. A 960 x 640 resolution displays 640 rows with 960 pixels in each row.

If you call the `Scene` constructor without providing width and height arguments, the JVM will determine the window's dimensions based on the size of the window's contents. This is handy for quick development, but be aware that the JVM will often come up with window dimensions that are not quite what you have in mind. Thus, most of the time, we will provide explicit width and height arguments when calling the `Scene` constructor.

The final statement in the `start` method uses the `stage` variable to call the `show` method. Windows are invisible by default. To make a window and its contents visible, add the contents to the window and then call the `Stage` class's `show` method. Do it in that order—add contents first, then call `show`. Otherwise, you might get unexpected results.

17.4 Containers – Stage, Scene, and Pane

The previous section showed you a simple GUI example that involved `Stage`, `Scene`, and `Label` classes. In this section, we'll describe `Stage` and `Scene` in more detail and introduce you to another class that makes it possible for a scene to have more than one component – a `Pane` class. To make this introduction as gentle as possible, we'll illustrate it with a program that produces the same result as the previous SimpleWindow program.

Stage and Scene

In Figure 17.4, note the `stage` parameter in the `start` method's heading. The `stage` parameter is a reference to a `Stage` object generated in the launch process and passed to the `start` method. The name "Stage" should bring to mind the idea of a stage in a theatre. It's a platform upon which performers make presentations. Likewise, a `Stage` object is in charge of a GUI program's overall presentation. More specifically, the `Stage` class forms a JavaFX window's outer frame and provides the standard Windows features that you've come to know and love – a title bar, a border, a minimize button, a close-window button, the ability to resize the window, and so on. You could implement all those features from scratch in your own classes, but why "reinvent the wheel"? Use the `Stage` class!

If we did not use `Stage`'s `setScene` method to put something on the stage, the stage would be just a frame with nothing in it − a transparent hole. `Stage`'s `setScene` method fills that hole with a scene that contains exactly one item. In our simple window program, that one item was a `Label`. But usually, we want a scene to have more than one component, and we can do that by making the scene's one item be a container that is able to hold multiple components. In the most general case, the scene is a compositional tree, formally called the *scene graph*. The one item a scene always contains is the *root* of its scene graph.

The most general type of thing you can put into a `Scene` is a `Node`. A `Scene` can retrieve its root node by calling its `getRoot` method. Any `Node` instance can retrieve its underlying scene by calling its `getScene` method. If a node is not the root note, it has a parent container, and that node can retrieve its parent container by calling its `getParent` method. In the next subsection, we'll look at a particular type of parent container – a subclass of `Node` called `Pane`, and more specifically, a subclass of `Pane` called `FlowPane`.

FlowPane

Figure 17.5 shows a SimpleWindowWithPane program that generates exactly the same output as the SimpleWindow program in Figure 17.4. But this time, instead of adding the `Label` to the scene, we add a `FlowPane` to the scene. Then we add the `Label` to the `FlowPane`.

In the SimpleWindowWithPane program, again you can see the need to import many classes individually. But sometimes you'll find two or more classes in the same package. Recall that a package is a collection of pre-built classes. The last `import` statement imports a package. Recall that to import a package, you need to use an asterisk; that is, `import javafx.geometry.*`. The `*` is a wildcard, and it allows you to import all the classes within a particular package.

At first glance, you might think that the four `import` statements that start with `import javafx.scene` might also be combined into one `import` statement by using a wildcard − `import javafx.scene.*`. That doesn't work because the four imports actually import from four different packages − `javafx.scene`, `javafx.scene.layout`, `javafx.scene.control`, and `javafx.scene.text`. A common misconception among beginning Java programmers is that if two packages share the same words at the left of their package names, then they can be imported together by using a wildcard. Sorry − not true.

In the interest of modularization, instead of trying to do everything the `start` method, we delegate some of the work to a `createContents` helper method. This program's `createContents` method is fairly short, and

we could easily have put all of its statements in the `start` method. But normal GUI programs need many statements to create and add multiple components and to register event handlers. And they need other statements to adjust appearance. All that takes up quite a few lines. If you stick all those lines in the `start` method, it can get pretty long. Better to break things up and stick them in a helper method.

```
/*******************************************************
* SimpleWindowWithPane.java
* Dean & Dean
*
* This program displays a label in a window.
*******************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.FlowPane;
import javafx.scene.control.Label;
import javafx.scene.text.Font;
import javafx.geometry.*;     // Insets, Pos

public class SimpleWindowWithPane extends Application
{
  private static final int WIDTH = 250;
  private static final int HEIGHT = 100;

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

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

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

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

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

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

  private void createContents(FlowPane pane)
  {
    Label label = new Label("Hi! I'm Larry the label!");

    pane.getChildren().add(label);
    pane.setStyle("-fx-font-size: 16");
    pane.setAlignment(Pos.TOP_CENTER);
  } // end createContents
} // end class SimpleWindowWithPane
```

This creates a multiple-component container and layout manager.

This adds the pane to the scene and the scene to the stage.

This adds the label to the pane.

These modify display properties.

Figure 17.5 SimpleWindowWithPane program

The `createContents` method instantiates the `Label` component and adds it to the previously created `FlowPane`. Then the `FlowPane` object calls two methods, `setStyle` and `setAlignment`. These produce the same results as the two methods, `setFont` and `setAlignment`, that were called by the `Label` object in the SimpleWindow program. But the mechanisms are quite different. `FlowPane`'s `setStyle` method sets the font for all of the `FlowPane`'s contained components – unless one of these components overrides this blanket specification with its own individual specification. With only one component, it's just as easy to apply the font style directly to the component, but with more than one component, `FlowPane`'s blanket font specification avoids code duplication. `Label`'s `setAlignment` method aligns the text within the label, whereas `FlowPane`'s `setAlignment` aligns the label within the pane.

`Pane` classes have built-in algorithms that dynamically manage the layout of a scene's components to fit the available space. A `FlowPane` lays out components like a simple text editor lays out words. As the left image in Figure 17.6 suggests, it starts in the upper-left corner. It adds additional components on the right, until there's no longer enough room. Then it wraps around and puts the next component on the next line, and so forth, like words in a text document, except now it is complete components instead of individual words. The `FlowPane` class gets its name from the manner in which components are added to its scene – they "flow" across rows from left to right, progressing down to the next row when the right boundary is reached.[1]

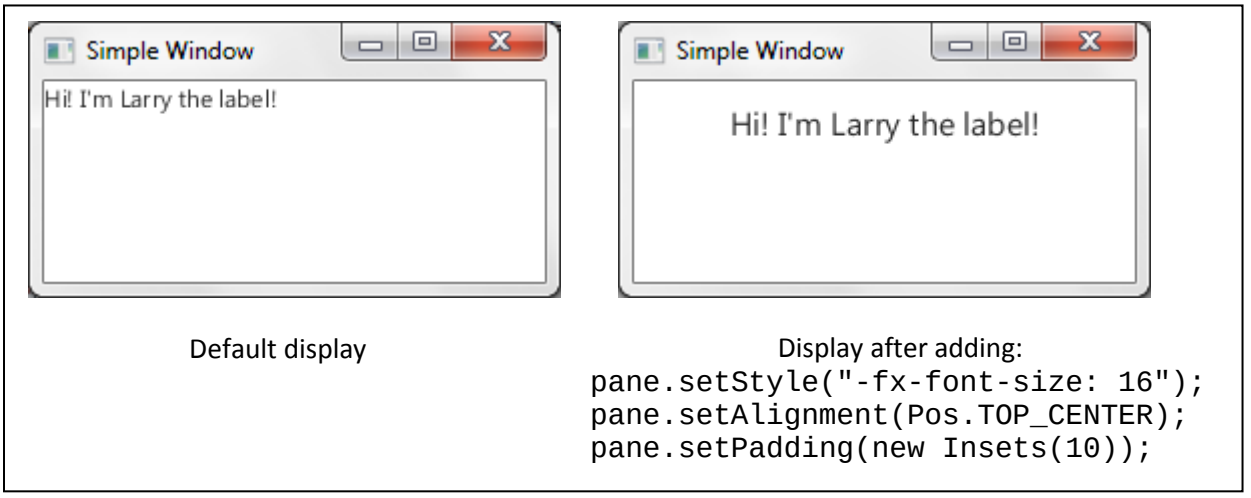


Figure 17.6 SimpleWindow program output – before and after cosmetic adjustments

The `FlowPane` class is one of many subclasses of Java's `Pane` class, and we call an instance of any of these subclasses a pane. In addition to enabling the incorporation of more than one component, a pane manages how those components are laid out relative to each other. Each type of pane has its own strategy for laying out components. You'll learn about other panes and their layout strategies in the next chapter. We're sticking with the `FlowPane` in this chapter because it's the easiest pane to use, and we're trying to keep things simple for now.

To add a pane to a scene, we pass it (instead of the previous label) to a `Scene` constructor,. Here's the relevant code from the SimpleWindowWithPane program, where pane is a reference to a `FlowPane` object:

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

To add a component to a pane, the `Pane` object calls its `getChildren` method. The `getChildren` method returns a list of the components already added to the pane. Then the returned list calls its `add` method, which adds a new component to the list. If the pane has been assigned to a scene and the scene has been assigned

[1] Instead of having new components flow horizontally across rows (the default), as an alternative, you can have new components flow vertically down columns. For vertical flow, use this alternate construction:

```
FlowPane pane = new FlowPane(Orientation.VERTICAL);
```

to a stage, the component gets added to what you see in the stage's window. The following code from the SimpleWindowWithPane program illustrates what we're talking about. It adds a label to the pane reference.

```
pane.getChildren().add(label);
```

If you'd like to add multiple components to a pane with just one statement, feel free to use the `addAll` method, like this:

```
pane.getChildren().addAll(<component1>, <component2>, <component3>, ...);
```

The `FlowPane` class and its super-classes provide many methods that allow us to adjust the pane's appearance – too many to mention here. In the SimpleWindowWithPane program, we give you a taste of what's available by calling `setStyle` and `setAlignment`. The right side of Figure 17.6 shows the syntax for calling these two methods.

A `FlowPane` object cannot call a `setFont` method like the `Label` object called in the SimpleWindow program in Figure 17.4, but – as indicated earlier -- it can specify font size for its components with the `setStyle` method call, copied here for your convenience:

```
pane.setStyle("-fx-font-size: 16");
```

This method's parameter is a specially formatted CSS (Cascading Style Sheet) string, which we'll describe later in Section 17.16. It may be hard to guess exactly how to write a legitimate CSS specification. And the Java compiler won't tell you if you get it wrong. It just ignores a bad one. But fortunately, it's usually not hard to understand a good CSS specification. This particular `setStyle` method argument is a CSS JavaFx specification of font size, and that size is 16 point.

To help with our explanation of `setAlignment`, we copy `setAlignment`'s method call here for your convenience:

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

`FlowPane`'s `setAlignment` method specifies the position and alignment of the group of components in the pane. In the code above, the `Pos.TOP_CENTER` argument means that the components will be laid out in the top center of the pane. `Pos` is an enumerated type that defines positioning and alignment values, and `TOP_CENTER` is one of those values.

The right side of Figure 17.6 also shows another `Pane` method, `setPadding`,

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

`FlowPane`'s `setPadding` method specifies the minimum blank border around the outside of the pane's contents (just a single label, in this case). We specify that padding with the help of an `Insets` object argument. An `Insets` object is a set of four offsets for the four sides of a rectangular area, top, right, bottom, and left, in that sequence. In the particular constructor used in Figure 17.6, the `Insets` object is created with a single argument of 10, which means that each of the window's four sides uses the same minimum offset value of 10 pixels. You can see the effect of the padding specification by comparing the image on the right side of Figure 17.6 with the image at the beginning of Section 17.3.

When a `FlowPane` contains more than one component (the usual case), you'll probably want to create horizontal gaps between adjacent components. You can do that with a method call like this:

```
pane.setHgap(25);
```

This inserts a 25-pixel horizontal gap between adjacent components. Similarly, you can insert vertical gaps with a method call like this:

```
pane.setVgap(5);
```

This inserts additional 5-pixel gaps between rows of components.

17.5 Components

Now let's elaborate on the individual objects that go into a scene—its components. Here are just a few of the many types of Java components:

- `Label`, `TextField`, `Button`,
- `TextArea`, `CheckBox`, `RadioButton`, `ComboBox`
- `MenuBar`, `ScrollPane`, `Slider`

These aren't all of the Java components, just some of the more commonly used ones. We'll describe the first three of these components in this chapter and the others in the next chapter.

All of these component classes are in the `javafx.scene.control` package, so you must import that package in order to use them. All of these component classes are subclasses of the `Control` class. The `Control` class is a subclass of the `Node` class. Containers like `FlowPane` are also subclasses of the `Node` class. Other classes like `Shape` and `ImageView` are also subclasses of the `Node` class. The `Node` class is the base class of all objects that go into a scene. Figure 17.7 shows an abbreviated view of the JavaFX class hierarchy. This picture shows classes discussed in this chapter. Those with bold outlines appear in example programs in this chapter.

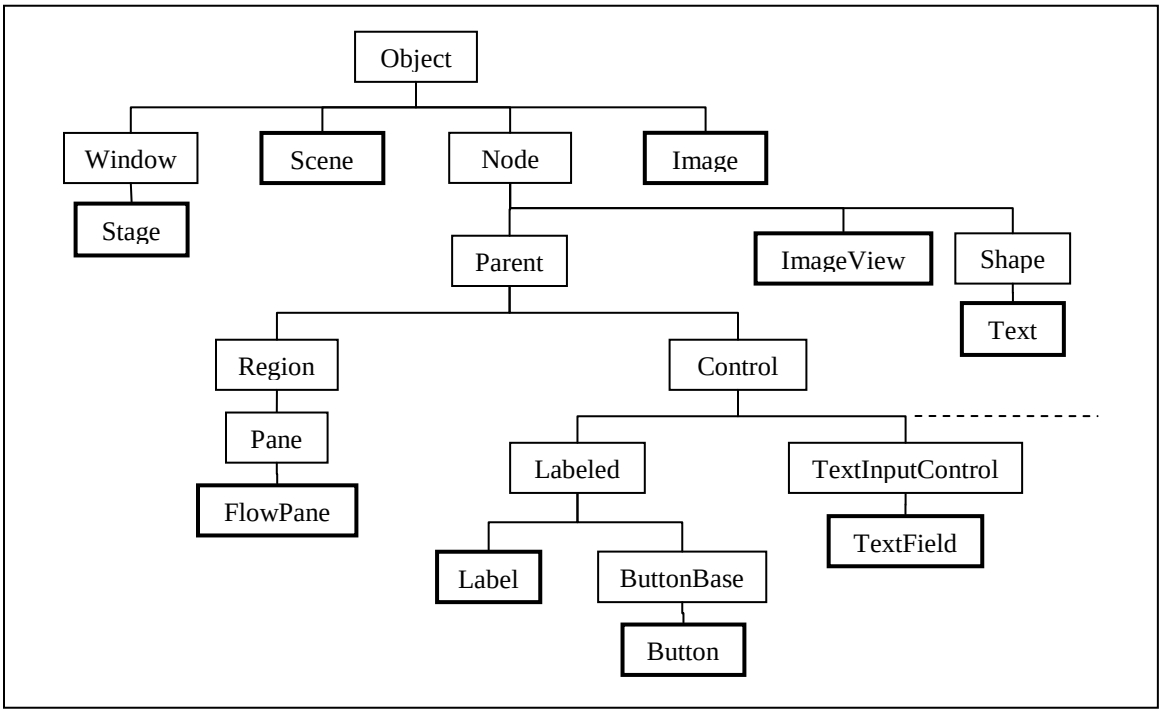


Figure 17.7 Inheritance relationships among JavaFX classes discussed in Chapter S17
Heavy bordered classes appear in example programs in this chapter.

For a more comprehensive picture of the JavaFX class hierarchy, skip forward and look at Figure 18.23 near the end of the next chapter. For complete information, look at Oracle's JavaFX documentation.

All instances of the `Node` class are unique, like individual people. Therefore, a node can be at only one place at a time. If you add an already-existing node to a different scene or pane, that add operation automatically removes it from its previous location and moves it to the new location. The `Node` class defines many features and more than two hundred methods. Some of these methods perform scaling, translation, and rotation. Others register distinctive handlers that respond to very specific event types. A `Parent` class can have children. The `Control` class adds features and methods that make components easy for users to manipulate. We'll describe and illustrate just a few of the many available component methods in subsequent discussion.

17.6 `Label` Component

User Interface

The `Label` component doesn't do much. It simply displays text. It's considered to be a read-only component because the user can read it but the user cannot directly change it. Normally, the `Label` component displays a single line of text, but that's not a requirement. One or more \n newline designators may be included in the text string to force the use of multiple lines.

Implementation

The `Label` class needs the `javafx.scene.control` package, so import that package if you use a label. To create a `Label` object, call the `Label` constructor like this:

```
Label <Label-reference> = new Label(<label-text>);
```

optional

The *<label-text>* is the text that appears in the `Label` component. If this argument is omitted, the `Label` component displays nothing. Why instantiate an empty label? So you can fill it in later on with text that's dependent on some condition. To add a `Label` object to a container called `pane`, use this syntax:

```
pane.getChildren().add(<Label-reference>);
```

<Label-reference> comes from above initialization statement.

Methods

The `Label` class inherits many methods from other classes.

From the `Node` class it inherits more than two hundred methods, like this one:

`public final void setStyle(String specifier)`

This sets the style using a CSS specifier. (This is the same method that the `FlowPane` object in the SimpleWindowWithPane program used to set the font size of all contained components.)

From the `Labeled` class, it inherits many additional methods, like these:

`public final String getText()`

Returns the labeled component's text.

`public final void setAlignment(Pos value)`

Determines how the labeled component's text should be aligned within the component when the labeled component has more space than it needs. (The `Label` in the SimpleWindow program used this method.)

`public final void setFont(Font value)`

Sets the font of the text in the labeled component. (The `Label` in the SimpleWindow program used this method.)

`public final void setText(String value)`

Assigns the label's text. Note that the programmer can update the label's text even though the user cannot.

`public final void setTextAlignment(TextAlignment value)`

Specifies how multiple lines of text are handled relative to the component's boundaries. The possible arguments are: `TextAlignment.LEFT`, `TextAlignment.CENTER`, `TextAlignment.RIGHT`,

and `TextAlignment.JUSTIFY`.

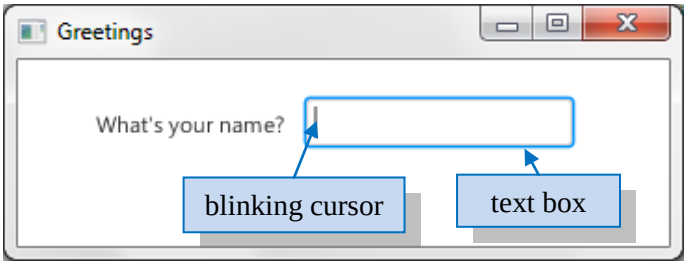
`public final void setTextFill(Paint value)`

Sets the color of this labeled component's text (`Color` is a sub-class of `Paint`).

17.7 `TextField` Component

User Interface

The `TextField` component displays a rectangle and allows the user to enter text into that rectangle. Here's an example:



Implementation

To create a `TextField` component (often called a *text box*), call the `TextField` constructor like this:

`TextField` <*TextField-reference*> `= new TextField(`<*default-text*>`);`

optional

The <*default-text*> is the text that appears in the text box by default and is read by the program if the user does not alter it. If the user enters more characters than the text box can display at one time, the leftmost characters scroll off the display. If the default-text argument is omitted, the default is an empty string.

To add a `TextField` object to a container called `pane`, use this syntax:

`pane.getChildren().add(`<*TextField-reference*>`);`

Methods

The `TextField` class inherits many methods from other classes.

From the `Node` class it inherits more than two hundred methods – the same methods that a `Label` inherits. Here's another example:

`public final void setVisible(boolean flag)`

Makes the text box visible or invisible.

From the `TextInputControl` class, it inherits many useful methods, like these:

`public void appendText(String text)`

Appends the specified `text` to the end of the text box's contents.

`public final int getLength()`

Retuns the number of characters in the text box.

`public final String getText()`

Returns the text box's contents.

`public final void setText(String text)`
Assigns the text box's contents.

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

The `TextField` class doesn't define many methods itself, but here are three useful ones:

`public void setOnAction(EventHandler<ActionEvent> handler)`
Registers an event handler with the text box.

`public final void setPrefColumnCount(int value)`
Gives the programmer some control over the width of the text box.

`public final void setPromptText(String text)`
Inserts text into the text box to instruct user on what to enter, but this prompt text is not read by the program.

Components are visible by default, but there are some instances where you might want to call `setVisible(false)` and make a component disappear. For example, after you calculate a result, you might want just the result to appear without the clutter of other components. When a component becomes invisible, the window automatically reclaims its space so other components can use that space.

Text boxes are editable by default, which means that users can type inside them. If you want to prevent users from editing a text box, call `setEditable` with an argument value of `false`. Calling `setEditable(false)` prevents users from updating a text box, but it does not prevent programmers from updating a text box. Programmers can call the `setText` method regardless of whether the text box is editable or non-editable.

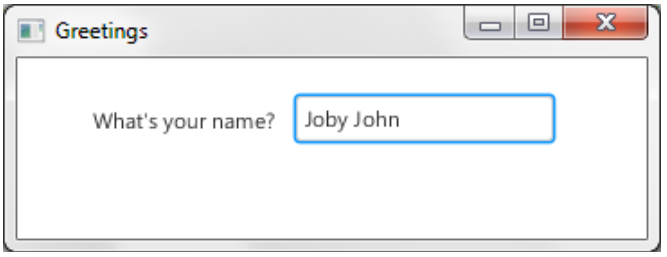
When a `TextField` component calls `setOnAction`, the JVM registers an event handler with the text box, and that enables the program to respond to the user pressing Enter within the text box. We'll describe handlers in more detail soon enough, but first we're going to step through an example program that puts into practice what you've learned so far. . . .

17.8 Greeting Program

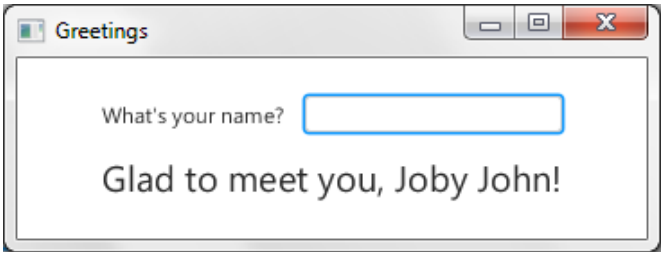
Now let's look at a label and text box example that invites the user to enter a name and then responds with a personal greeting, like this sample session:

Initial display:

After the user makes an entry in the text box:



After the user hits Enter:



Figures 17.8a and 17.8b show the program which produces these displays, the Greeting program. Most of the code should look familiar since it closely parallels the code in the SimpleWindowWithPane program. The `main` methods are similar. The only difference is that we have dispensed with the class-name-dot prefix in the `launch` method call. The proper class name is inferred when the `launch` method call is the same class as the `start` method. But keep in mind that if the `launch` method call is in a different class, it needs to have the `start` method's class as an explicit prefix. The SimpleWindow and SimpleWindowWithPane `start` methods contain similar calls to `setTitle`, `setScene`, `createContents`, and `show`. The `createContents` helper method creates the components and adds them to the scene.

This time, instead of just one component, we have three − two labels called `namePrompt` and `message`, and a text box called `nameBox`. We use the pane's `getChildren().addAll()` method call to add all three of these components to the pane simultaneously. Next, we call the `setFont`, `setPadding`, and `setAlignment` methods. They should look familiar since we used them in the previous SimpleWindow programs. We then call the `setHgap` method to set the horizontal gap between components in the same row and the `setVgap` method to set the vertical gap between the rows. In the three displays above, the horizontal gap is the gap between the right side of the "What's your name?" label and the left side of the subsequent text box. In the final display above, the vertical gap is the gap between the two rows.

In Figure 17.8a, note the location of `nameBox`'s declaration. It's declared as an instance variable at the top of the class. Why an instance variable instead of a `createContents` local variable? Aren't local variables preferred? Yes, but in this case, we need to access `nameBox` not only in `createContents`, but also in the event handler (which we'll get to next). It's possible to use a local variable within `createContents` and still access it from the event handler, but that's a bit of a pain.[2] For now, we'll keep things simple and declare `nameBox` as an instance variable. The same rationale applies to the `greeting` label. We need to access it in `createContents` and also in the event handler, so we make it an instance variable.

[2] If you declare a variable locally within `createContents`, you can retrieve it from an event handler by calling `getSource`. We describe the `getSource` method in Section 17.15.

```
/*************************************************************
* Greeting.java
* Dean & Dean
*
* This program demonstrates text boxes and labels.
* When the user presses Enter after typing something into the
* text box, the text box value displays it in the label below.
*************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.FlowPane;
import javafx.scene.control.*;  // TextField, Label
import javafx.scene.text.Font;
import javafx.geometry.*;       // Insets,Pos
import javafx.event.*;          // EventHandler, ActionEvent

public class Greeting extends Application
{
  private static final int WIDTH = 350;
  private static final int HEIGHT = 100;
  private TextField nameBox;    // holds user's name
  private Label greeting;       // personalized greeting

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

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

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

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

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

1. Import this package for event handling.

Figure 17.8a Greeting program – part A

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

  private void createContents(FlowPane pane)
  {
    Label namePrompt = new Label("What's your name?");

    nameBox = new TextField();
    greeting = new Label();
    pane.getChildren().addAll(namePrompt, nameBox, greeting);
    greeting.setFont(new Font(20));
    pane.setPadding(new Insets(20));
    pane.setHgap(10);                  // horizontal spacing
    pane.setVgap(10);                  // vertical spacing
    pane.setAlignment(Pos.TOP_CENTER);
    nameBox.setOnAction(new Handler());
  } // end createContents

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

  // Inner class for event handling.

  private class Handler implements EventHandler<ActionEvent>
  {
    public void handle(ActionEvent e)
    {
      String message =
        "Glad to meet you, " + nameBox.getText()+ "!";

      nameBox.setText("");
      greeting.setText(message);
    } // end handle
  } // end class Handler
} // end class Greeting
```

2. Register a handler.

3. handler class heading

4. event handler

Figure 17.8b Greeting program – part B

The four callouts highlight features of the event handler, which specifies what happens when the user presses Enter while the cursor is in the text box. The next section describes event handling.

17.9 Event Handlers

When the user interacts with a GUI component (like when the user clicks a button or presses Enter while the cursor is in a text box), that component fires an event. If the component has a handler attached to it, that handler "hears" the fired event. Consequently, the handler handles the event by executing its `handle` method. In this

section, you'll learn how to make all that work by creating a handler with its requisite `handle` method.

Below, in top-down sequence, we describe the steps needed to implement a handler for a GUI component like a text box. These steps correspond to the numbered callouts in Figures 17.8a and 17.8b:

1. Import the `javafx.event` package. Event handling requires the use of the `EventHandler` interface and the `ActionEvent` class. Those entities are in the `javafx.event` package, so that package must be imported for event handling to work. To see the `import` statements within a complete program, look at callout 1 in Figure 17.8a.

2. Register your handler class. More specifically, that means adding your handler class to a text box component by calling the `setOnAction` method. Here's the syntax:

   <*text-box-component*>`.setOnAction(new` <*handler-class*>`());`

   For an example, look at callout 2 in Figure 17.8b. The registration enables your text box can find a handler when an event fires. For example, an event fires whenever the user presses Enter while the cursor is within the text box. Registering a handler is like registering your car. When you register your car, nothing much happens at that point. But later, when some event occurs, your car registration comes into play. What event would cause your car registration to be used? If you get caught speeding, the police can use your registration number as part of a traffic citation. If you get into a wreck, your insurance company can use your registration number to raise your insurance rates.

3. Define a class with an `implements EventHandler<ActionEvent>` clause appended to the class's heading. To see an example, look at callout 3 in Figure 17.8b. This `implements` clause tells the Java complier that you promise to define the `handle` method that the event-handling framework will call whenever an event fires. In a top-down development process, the `handle` method can initially be just a dummy method that does nothing.

4. Fully define the promised `handle` method in your handler's class. Here's a skeleton of the `handle` method that be in every handler class:

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

   Even if your `handle` method doesn't use the `ActionEvent` parameter (`e`, above), you still must include that parameter in the method heading to make your method's signature match the signature of the method the event-handling framework will call when an event fires. To see an example of a complete `handle` method, look at callout 4 in Figure 17.8b. It refers to a handler class that's named `Handler`. Note that `Handler` is not a reserved word—it's just a good descriptive name we picked for the handler class in our Greeting program.

17.10 Inner Classes

Here's a reprint of the Greeting program, in skeleton form:

```
public class Greeting extends Application
{
  ...
    private class Handler implements EventHandler<ActionEvent>
    {
```

```
        public void handle(ActionEvent e)
        {
          String message =
            "Glad to meet you, " + nameBox.getText()+ "!";

          nameBox.setText("");
          greeting.setText(message);
        } // end handle
      } // end class Handler
    ...
  } // end class Greeting
```

Do you notice anything odd about the position of the `Handler` class in the `Greeting` program? See how the `Handler` class is indented and how its closing brace is before the `Greeting` class's closing brace? The `Handler` class is inside of the `Greeting` class!

If a class is limited in its scope such that it is needed by only one other class, you should define the class as an *inner class* (a class inside of another class). Since a handler is usually limited to handling just one class, handlers are usually implemented as inner classes.

It's not required by the compiler, but inner classes should normally be `private`. Why? Because the main point of using an inner class is to further the goal of encapsulation and using `private` means the outside world won't be able to access the inner class. Note the `private` modifier in the above `Handler` class heading.

Besides furthering the goal of encapsulation, there's another reason to use an inner class as opposed to a *top-level* class (top-level class is the formal term for a regular class—a class not defined inside of another class). An inner class can directly access its enclosing class's instance variables. Since handlers normally need to access their enclosing class's instance variables, this is an important benefit.

17.11 Anonymous Inner Classes

Take a look at the GreetingAnonymous program in Figures 17.9a and 17.9b. It's virtually identical to the previous Greeting program. Can you identify the difference between the GreetingAnonymous program and the Greeting program?

```
/*************************************************************
* GreetingAnonymous.java
* Dean & Dean
*
* This program demonstrates text boxes and labels.
* When the user presses Enter after typing something into the
* text box, the text box value displays it in the label below.
*************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.FlowPane;
import javafx.scene.control.*;  // TextField, Label
import javafx.geometry.*;       // Insets,Pos
import javafx.event.*;          // EventHandler, ActionEvent

public class GreetingAnonymous extends Application
{
  private static final int WIDTH = 350;
  private static final int HEIGHT = 100;
  private TextField nameBox;    // holds user's name
  private Label greeting;       // personalized greeting

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

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

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

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

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

Figure 17.9a GreetingAnonymous program – part A

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

  private void createContents(FlowPane pane)
  {
    Label namePrompt = new Label("What's your name?");

    nameBox = new TextField();
    greeting = new Label();
    pane.getChildren().addAll(namePrompt, nameBox, greeting);
    greeting.setFont(new Font(20));
    pane.setPadding(new Insets(20));
    pane.setHgap(10);                // horizontal spacing
    pane.setVgap(10);                // vertical spacing
    pane.setAlignment(Pos.TOP_CENTER);
    nameBox.setOnAction(

      // anonymous inner class for event handling
      new EventHandler<ActionEvent>()
      {
        public void handle(ActionEvent e)
        {
          String message =
            "Glad to meet you, " + nameBox.getText()+ "!";

          nameBox.setText("");
          greeting.setText(message);
        } // end handle
      } // end EventHandler anonymous inner class
    ); // end setOnAction method call
  } // end createContents
} // end class GreetingAnonymous
```

Figure 17.9 b GreetingAnonymous program – part B

In the Greeting program, we implemented a handler class named `Handler`, starting with this code:

```
private class Handler implements EventHandler<ActionEvent>
{
```

This code does not appear in the GreetingAnonymous program—there is no class named `Handler`. But we still need a handler object so that the text box's enter event is detected and acted upon. This time, instead of declaring a handler class with a name (like `Handler`), we implement a handler class anonymously (without a name).

We've discussed anonymous objects previously. That's where you instantiate an object without storing its reference in a variable. In our previous Greeting program, we instantiated an anonymous `Handler` object with this line:

```
nameBox.setOnAction(new Handler());
```

The point of using an anonymous object is to avoid cluttering the code with a variable name when an object needs to be used only one time. The same idea can be applied to classes. The point of using an *anonymous inner class* is to avoid cluttering the code with a class name when a class needs to be used only one time. For example, if a particular handler class handles just one object, then the handler class needs to be used only one time as part of an `setOnAction` method call. Therefore, to un-clutter your code, you may want to use an anonymous inner

class for the handler.

Using an anonymous inner class is not a compiler requirement. It's an elegance issue. In industry, you'll find some people who say anonymous inner classes are elegant and you'll find other people who say anonymous inner classes are confusing. Do as you see fit. Better yet, do as your teacher sees fit.

Below, we show the syntax for an anonymous inner class. Naturally, there's no class name. But there is an interface name. So anonymous inner classes aren't built from scratch; they're built with the help of an interface. Note the `new` operator. Formally speaking, the `new` operator isn't part of the anonymous inner class. But practically speaking, since there's no point in having an anonymous inner class without instantiating it, you can think of the `new` operator as being part of the anonymous inner class syntax.

```
new <interface-name>()
{
 <class-body>
}
```

Here's an example of an anonymous inner class, taken from the GreetingAnonymous program:

```
nameBox.setOnAction(
  new EventHandler<ActionEvent>()
  {
    public void handle(ActionEvent e)
    {
      ...
    } // end handle
  } // end inner-class constructor
);
```

`EventHandler<ActionEvent>` is an interface.

For comparison purposes, here's an example of a named (non-anonymous) inner class. It's taken from the Greeting program:

```
private void createContents()
{
  ...
  nameBox.setOnAction(new Handler());
} // end createContents

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

There are only two syntactic differences between the two code fragments—the `setOnAction` call and the handler class heading. There are no semantic differences between the two code fragments, so the Greeting program and the GreetingAnonymous program behave the same.

17.12 `Button` Component

It's now time to learn another GUI component—a button component.

User Interface

If you press a button on an electronic device, something usually happens. For example, if you press the power button on a television, the television turns on or off. Likewise, if you press/click a GUI *button* component, something usually happens. For example, in Figure 17.1's TrustyCredit window, if you click the OK button, then the entered credit card numbers get processed by the TrustyCredit company.

Implementation

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

```
Button helloButton = new Button("Press me");
```

When this button is displayed, it says "Press me" in the center of the button. The label argument is optional. If it's omitted, the label gets the empty string by default and the button displays with a blank face (no writing on it).

After you have created the `helloButton`, add it to your scene or to a pane, like this:

```
pane.getChildren().add(helloButton);
```

To make the button useful, you'll need to implement a handler. As with the text box handlers, button handlers must implement the `EventHandler<ActionEvent>` interface. As indicated earlier, this interface dictates that your handler must define a `handle` method. The code skeleton looks like this:

```
private class Handler implements EventHandler<ActionEvent>
{
  public void handle(ActionEvent e)
  {
    <do-something>
  } // end handle method
} // end Handler class
```

We're using `private` instead of `public` for the handler class because a handler normally is implemented as an inner class, and inner classes are normally `private`. We're using a named inner class instead of an anonymous inner class because named inner classes are slightly more flexible. They allow you to create a handler that's used with more than one component. We'll provide an example in an upcoming program.

To register the above handler with our `helloButton` component, do this:

```
helloButton.setOnAction(new Handler());
```

The `Button` class needs the `javafx.scene.control` package, but that may be available already, since it's needed for the `Label` and `TextArea` classes. The `EventHandler` interface and the `ActionEvent` class need the `javfx.event` package, so import that package.

Methods

Since the `Button` method is a subclass as the `Labeled` class, it inherits all the methods from the `Labeled` and

Node classes. Methods inherited from the Labeled class include the methods identified in Section 17.6 – getText, setAlignment, setFont, setText, setTextAlignment, and setTextFill. Methods inherited from the Node class include the setStyle method identified in Section 17.6 and the setVisible method identified in Section 17.7.

As indicated in Figure 17.7, the Button class is also a subclass of the ButtonBase class, which defines the obviously important method:

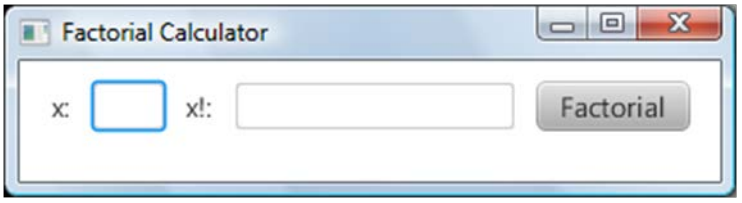
```
public final void setOnAction(EventHandler<ActionEvent> value)
```

Adds a handler to the button.

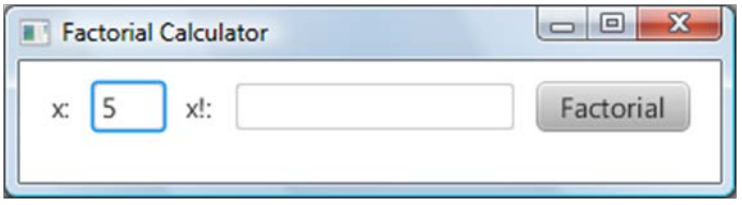
FactorialButton Program

It's time to put all this Button syntax into practice by showing you how it's used within a complete program. We've written a FactorialButton program that uses a Button component to calculate the factorial for a user-entered number.[3] The program generates the following sequence of displays:

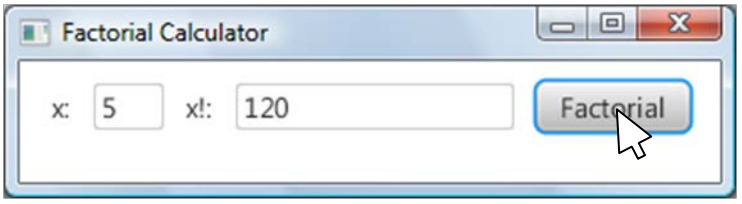
Initial Display:



Display after typing the number 5 in the text box:



Display after clicking the Factorial button:



Figures 17.10a, 17.10b, and 17.10c contain the FactorialButton program listing. Most of the code should already make sense since the program's structure parallels the structure in our previous GUI programs. We'll skip the more familiar code and focus on the more difficult code.

[3] The factorial of a number is the product of all positive integers less than or equal to the number. The factorial of n is written as n! Example: The factorial of 4 is written as 4!, and 4! is equal to 24 because 1 times 2 times 3 times 4 equals 24.

```
/*************************************************************
* FactorialButton.java
* Dean & Dean
*
* When user clicks button or presses Enter in input text box,
* entered number's factorial displays in the output text box.
*************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.FlowPane;
import javafx.scene.control.*;    // TextField, Label
import javafx.geometry.*;         // Insets, Pos
import javafx.event.*;            // ActionEvent, EventHandler

public class FactorialButton extends Application
{
  private static final int WIDTH = 350;
  private static final int HEIGHT = 60;
  private TextField xBox = new TextField();  //  user entry
  private TextField xfBox = new TextField(); //  computed factorial

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

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

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

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

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

Figure 17.10a FactorialButton program – part A

We declare the two text box components as instance variables at the top of the program, but we declare most of our GUI variables locally within `createContents`. Why the difference? As discussed earlier, normally you should declare components as local variables to help with encapsulation. But if a component is needed in `createContents` and also in an event handler, it's fine to declare it as an instance variable where it can be shared more easily. In the FactorialButton program, we declare the two text boxes as instance variables because we need to use them in `createContents` and also in the event handler's `handle` method.

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

private void createContents(FlowPane pane)
{
  Label xLabel = new Label("x:");
  Label xfLabel = new Label("x!:");
  Button btn = new Button("Factorial");
  Handler handler = new Handler();

  pane.getChildren().addAll(
    xLabel, xBox, xfLabel, xfBox, btn);
  pane.setStyle("-fx-font-size: 14");
  pane.setPadding(new Insets(10));
  pane.setHgap(10);
  pane.setAlignment(Pos.TOP_CENTER);
  xBox.setPrefColumnCount(2);   // makes a narrow box
  xfBox.setPrefColumnCount(10); // makes a wider box
  xfBox.setEditable(false);
  xBox.setOnAction(handler);
  btn.setOnAction(handler);
} // end createContents

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

Here we register the same handler with two different components.

Figure 17.10b FactorialButton program – part B

Note this line near the end of the `createContents` method:

```
xfBox.setEditable(false);
```

This causes the factorial text box, `xfBox`, to be non-editable (i.e., the user won't be able to update the text box). That should make sense since `xfBox` holds the factorial, and it's up to the program (not the user) to generate the factorial. In all three sample-session displays above the factorial text box has a dim border. You get that visual cue free of charge whenever you call `setEditable(false)` from a text box component. Cool!

Again from the `createContents` method:

```
Handler handler = new Handler();
...
xBox.setOnAction(handler);
btn.setOnAction(handler);
```

Note that we're registering the same handler with two different components. This gives the user two ways to trigger a response. The user can press Enter when the cursor is in the input text box (`xBox`) or the user can click on the button (`btn`). Either way causes the handler to react. Whenever you register the same handler with two different components, you need to have a name for the handler. That's why we use a named inner class for this program. You could use two anonymous handler instantiations, but an anonymous inner class wouldn't work.

Figure 17.10c's `handle` method is chock full of interesting code. Of greatest importance is the `Integer.parseInt` method call. If you ever need to read numbers or display numbers in a GUI program, you have to use string versions of the numbers. Thus, to read a number from the input text box, we first read it in as a string, and then we convert the string to a number. To accomplish this, we read the string using `xBox.getText()`, and then we convert it to a number using `Integer.parseInt`.

```
  // Inner class for event handling.

  private class Handler implements EventHandler<ActionEvent>
  {
    public void handle(ActionEvent e)
    {
      int x;          // numeric value for user-entered x
      int xf;         // x factorial

      try
      {
        x = Integer.parseInt(xBox.getText());
      }
      catch (NumberFormatException nfe)
      {
        x = -1;       // indicates an invalid x
      }
      if (x < 0)
      {
        xfBox.setText("undefined");
      }
      else
      {
        if (x == 0 || x == 1)
        {
          xf = 1;
        }
        else
        {
          xf = 1;
          for (int i=2; i<=x; i++)
          {
            xf *= i;
          }
        } // end else

        xfBox.setText(Integer.toString(xf));
      } // end else
    } // end handle
  } // end class Handler
} // end class FactorialButton
```

Convert user entry from a string to a number.

This can be improved.

Figure 17.10c FactorialButton program – part C

Ideally, you should always check user input to make sure it's valid. In the `handle` method, we check for two types of invalid input—a non-integer input and a negative number input. Those inputs are invalid because the factorial is mathematically undefined for those cases. The negative number case is easier, so we'll start with it. Note this code in the middle of the `handle` method:

```
if (x < 0)
{
  xfBox.setText("undefined");
}
```

The x is the user's entry after it's been converted to an integer. If x is negative, the program displays undefined in the xfBox component.

Now for the non-integer input case. Note this code near the top of the handle method:

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

The Integer.parseInt method attempts to convert xBox's user-entered value to an integer. If xBox's user-entered value is a non-integer, then parseInt throws a NumberFormatException. To handle that possibility, we put the Integer.parseInt method call inside a try block, and we include an associated catch block. If parseInt throws an exception, we want to display undefined in the xfBox component. To do that, we could call xfBox.setText("undefined") in the catch block, but then we'd have redundant code – xfBox.setText("undefined") in the catch block and also in the subsequent if statement. To avoid code redundancy and its inherent maintenance problems, we assign –1 to x in the catch block. That makes the subsequent if statement true, and this causes xfBox.setText("undefined") to be called.

After validating the input, the handle method calculates the factorial. It first takes care of the special case when x equals 0 or 1. It then takes care of the x > 2 case by using a for loop. Study the code. It works fine, but do you see a way to make it more compact? You can omit the block of code that starts with

```
if(x == 0 || x == 1)
```

because that case is handled by the else block. More specifically, you can delete the six lines above the second xf = 1; line, initialize xf with 1 when it's declared, and remove the outer else's closing parentheses. Here's what the handle method looks like after you clean it up:

Write compact code.

```
private class Handler implements EventHandler<ActionEvent>
{
    public void handle(ActionEvent e)
    {
      int x;          // numeric value for user-entered x
      int xf = 1;     // initialized x factorial

      try
      {
        x = Integer.parseInt(xBox.getText());
      }
      catch (NumberFormatException nfe)
      {
        x = -1;       // indicates an invalid x
      }
      if (x < 0)
      {
        xfBox.setText("undefined");
      }
      else
      {
        for (int i=2; i<=x; i++)
        {
          xf *= i;
        }
        xfBox.setText(Integer.toString(xf));
      } // end else
    } // end handle
  } //end class Handler
```

improved code

Figure 17.11 Refinement of FactorialButton program's `Handler` class

17.13 Using an Event's `getSource` Method

Before now, the event handler's `ActionEvent` parameter was just a minor annoyance. Now, you'll see examples in which this object comes in handy.

The getSource Method

Suppose you register a handler with two components. To respond appropriately to the event, you'll often need to determine which component fired the event. That way, you can customize your event handling: Do one thing if component X fired the event, and do another thing if component Y fired the event.

From within a handler, how can you determine the source of an event? In other words, how can you identify the component that fired an event? Call `getSource`, of course! More specifically, within the `handle` method, use the `handle` method's `ActionEvent` parameter to call `getSource`. The `getSource` method returns a reference to the component whose event was fired. To see which component that was, you can use `==` to compare the returned reference with a saved component reference. For example, in the below code fragment, we compare the returned value to a button component named `okButton`.

```
public void handle(ActionEvent e)
{
  if(e.getSource() == okButton)
  {
    ...
```

For More Source Information

Suppose you don't want to save references to all sources that might fire events. Can you figure out what fired the event from what `getSource` returns? What it returns, of course, is a reference to the source object, and if you try to print it, you'll get what that object's `toString` method returns. For example, suppose you're in a handler that is registered with several different components. In an attempt to learn more about the particular source that fired the event, after retrieving that source, you might print it, like this:

```
Node source = (Node) e.getSource();
System.out.println(source);
```

Assuming the object that fired the event was a button, the printout might look like this:

```
Button[id=null, style Class=button]
```

This tells you the source was a button. If that's enough information, fine.

However, suppose there are several buttons, and any one of them might have been the source. How could you determine which button fired the event? Since the `Button` class is a subclass of the `Labeled` class, assuming you gave your buttons distinctive text labels, you could determine which button fired the event with a statement like this:

```
System.out.println(((Labeled) source).getText());
```

As you can see in Figure 17.7, not all GUI items are subclasses of the `Labeled` class, but don't despair. The above printout generated by `System.out.println(source)` suggests another alternative. Notice that the first item in that printout's square brackets is `id=null`. You can give an instance of any subclass of the `Node` class a string identification by having that instance call it's `setId(String id)` method. Then, later, you can retrieve that identification and display it with a code fragment like this:

```
Node source = (Node) e.getSource();
System.out.println(source.getId());
```

Moreover, you can associate any object with any `Node` object, using `Node`'s `setUserData(Object obj)` method. Then, later, you can retrieve that other object using `Node`'s `getUserData` method. This other object can be of any type. Thus, it could contain any amount of other data in any form, and it could call methods that can do anything. This is an extremely powerful option. It gives you the ability to associate arbitrary amounts of information and/or arbitrary methods with any GUI node.

But what about GUI objects that are not `Node`s? Figure 17.7 shows that the `Scene` class is not a subclass of the `Node` class. If the source is a `Scene` object, you can use `Scene`'s `getRoot` method call to get a reference to the one `Node` object that every scene must contain. Since every scene always has exactly one root node, you can use `Node`'s `setId` and/or `setUserData` method to put a string identifier and/or any other descriptive material about a scene in its root node. Then later you can retrieve that scene identifier and/or other descriptive material with something like this:

<scene-reference>`.getRoot().getId();`

The `Stage` class is another class that is not a subclass of the `Node` class. If the source object is an instance of the `Stage` class, you can use `Stage`'s `getScene` method to get the window's current scene. Then you can use `Scene`'s `getRoot().getId()` to obtain the identifier of the scene currently in that window. You'll see an example of this later in Section 17.15. The `Node` class also has a `getScene` method, so any node in a complex scene graph can use this to obtain a reference to the underlying scene. Then, if the scene's root node has an identifier, you can use the scene reference and `getRoot().getId()` to obtain the identifier of the node's

scene.

If the source object is an instance of the `Stage` class, you might have still another option. If you used the `setTile` method call to give the window a title, you can use `Stage`'s `getTile` method call to retrieve a useful window identifier – its string title. The `Scene` class includes a `getWindow` method, which returns a reference to the underlying window. Thus, if the window has a title, any node could obtain a descriptor of its underlying window by using a chained method call like this:

<node-reference>`.getScene().getWindow().getTitle();`

With `getSource` you can learn a great deal about the event's environment.

17.14 Builders

The standard way to create a GUI container or component is to use a constructor with particular parameter values for the most commonly specified properties. Then call a sequence of set methods to establish whatever other properties you need. It's often a motley process, involving a combination of diverse or clashing elements. When you have a basic pattern with just a few variations, a *builder* provides an attractive alternative. Once you get the idea and express it in a well-structured format, builder code is easy to read, easy to modify, and robust. This section explains what a builder is and several ways to use it.

To illustrate, let's modify the FactorialButton program. This modification requires an additional import:

```
import javafx.scene.layout.FlowPaneBuilder;
```

It also requires that we combine some of the operations that were split between the `start` method and the `createContents` method. This motivates us to dispense with the `createContents` method and bring everything it would do back into the `start` method. Figure 17.12 shows a modification of the FactorialButton program that does everything in the `start` method. The inherent compactness of the builder technology reduces the need for a separate `createContents` helper method.

Builder Details

In Figure 17.12, we'll focus on the individual method calls in the method-call chain that instantiates the `FlowPane` object called `pane`. First consider just the "book ends", `FlowPaneBuilder.create()` and `.build()`. The `create` method call returns a reference to a new `FlowPaneBuilder` object. The final `build` method call returns a reference to the desired `FlowPane` object. If there were nothing between this pair of book ends, the result would be the same as a simple `FlowPane` instantiation, that is, `new FlowPane();` What you'd see in the displayed window frame would be just a plain white fill.

Each method call between the `create` and `build` book ends returns `this`, that is, a reference to the `FlowPaneBuilder` object that called it. In Figure 17.12, each of the indented `FlowPaneBuilder` methods between the book ends corresponds to one of the parameters in a `FlowPane` constructor parameter or one of `FlowPane`'s many set methods:

- `FlowPaneBuilder`'s `children` method call is equivalent to `FlowPane`'s more lengthy `getChildren().addAll` method call.
- `FlowPaneBuilder`'s `style` method call is equivalent to `FlowPane`'s `setStyle` method call.
- `FlowPaneBuilder`'s `padding` method call is equivalent to `FlowPane`'s `setPadding` method call.
- `FlowPaneBuilder`'s `hgap` method call is equivalent to `FlowPane`'s `setHgap` method call.
- `FlowPaneBuilder`'s `alignment` method call is equivalent to `FlowPane`'s `setAlignment` method call.

Because they all avoid using the `set` prefix, the meanings of `FlowPaneBuilder`'s methods are more immediately apparent than the meanings of the corresponding `FlowPane` methods. An important aspect of this

builder methodology is that each method sets just one attribute, and there are no multi-parameter constructors.

```
// <Figure 17.10a imports plus FlowPaneBuilder import>

public class FactorialButton2 extends Application
{
  private static final int WIDTH = 350;
  private static final int HEIGHT = 60;
  private TextField xBox = new TextField();  //  user entry
  private TextField xfBox = new TextField(); //  generated factorial

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

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

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

  @Override
  public void start(Stage stage)
  {
    Label xLabel = new Label("x:");
    Label xfLabel = new Label("x!:");
    Button btn = new Button("Factorial");
    FlowPane pane = FlowPaneBuilder.create()
      .children(xLabel, xBox, xfLabel, xfBox, btn)
      .style("-fx-font-size: 14")
      .padding(new Insets(10))
      .hgap(10)
      .alignment(Pos.TOP_CENTER)
      .build();
    Handler handler = new Handler();

    stage.setTitle("Factorial Calculator");
    stage.setScene(new Scene(pane, WIDTH, HEIGHT));
    xBox.setPrefColumnCount(2);   // makes a narrow box
    xfBox.setPrefColumnCount(10); // makes a wider box
    xfBox.setEditable(false);
    xBox.setOnAction(handler);
    btn.setOnAction(handler);
    stage.show();
  } // end start

  // <Handler code from Figure 17.11>

} // end class FactorialButton2
```

This creates a builder.

This educates the builder.

This produces the `FlowPane` object.

Figure 17.12 FactorialButton program modifications that illustrate use of a builder

Another way to visualize a builder is to think of it as a customized overloaded constructor, which you create yourself with parameters for all the attributes you want and no others. This customized "constructor" is better than

any ordinary multi-parameter constructor, because its "parameters" may be in any order, so you cannot accidently introduce a logical error by accidently switching the order of two parameters of the same type. Moreover, the function of each "parameter" is automatically self-documented by its `FlowPaneBuilder` method name.

Now, suppose you need to construct several similar objects that have just a few slightly different attributes. You can see how clean and easy it is to modify code to construct and initialize a slightly different object. Just copy the original builder method-call chain. Then delete old builder methods that correspond to attributes you don't want and insert new builder methods that correspond to attributes you do want. And suit yourself on sequence.

There's also another way to do it. In the above example, we created an anonymous builder object (a `FlowPaneBuilder`), which contained all the information needed to build a desired target object (a `FlowPane`). Then we used that anonymous builder object immediately to instantiate the desired target object. But it's not necessary to use a builder object immediately. Instead, you can declare a builder variable and assign it the result produced by chaining all but the final `build` method call. That saves the builder object for later use.

Then later, you can have that builder object call its `build` method to generate a target object with the builder's already-stored attributes. After that, if you want another target object with just one differing attribute, you can mutate the builder object by having it call the one method which sets that one attribute, followed by the `build` method call that creates the new target object. If you need a large number of target objects in which only one attribute differs, we can obtain them easily and efficiently by using a simple `for` loop.

There's still another way to do it. You can stop the building process before it is complete and save the partially completed builder. For example, you can create and save a builder that contains only the common information for some desired set of similar target objects. Then you can use that partially completed builder as the starting point for different chaining operations that append methods for differentiating attributes only and finish with `build` method calls that create the desired target objects. In the next section, you'll see an example of this procedure.

Our discussion and the programs we use to illustrate it use one particular type of GUI object, a `FlowPane`. But if you look at JavaFX documentation, you'll see that essentially all JavaFX containers and components have builders in addition to ordinary constructors. Therefore, if you want, you can also employ this builder technique with many other GUI containers and components.

17.15 Dialog Boxes

A *dialog box*—often referred to simply as a *dialog*—is a specialized type of window. The primary difference between a dialog box and a standard window is that a dialog box is more constrained in terms of what it can do. While a standard window usually remains on the user's screen for quite a while (often for the duration of the program) and performs many tasks, a dialog box remains on the screen only long enough to perform one specific task. While standard windows may be configured in almost any form, dialog boxes have a basic pattern with just a few variations. Since dialog boxes are basically similar with a few variations, they are good candidates for JavaFX builder technology.

User Interface

There are basically three types of dialogs—a *message dialog*, an *input dialog*, and a *confirmation* dialog. Each type performs one specific task. The message dialog displays output. The input dialog displays a question and an input field. The confirmation dialog displays a yes/no question and yes/no/cancel button options. See what the different types look like in Figure 17.13.

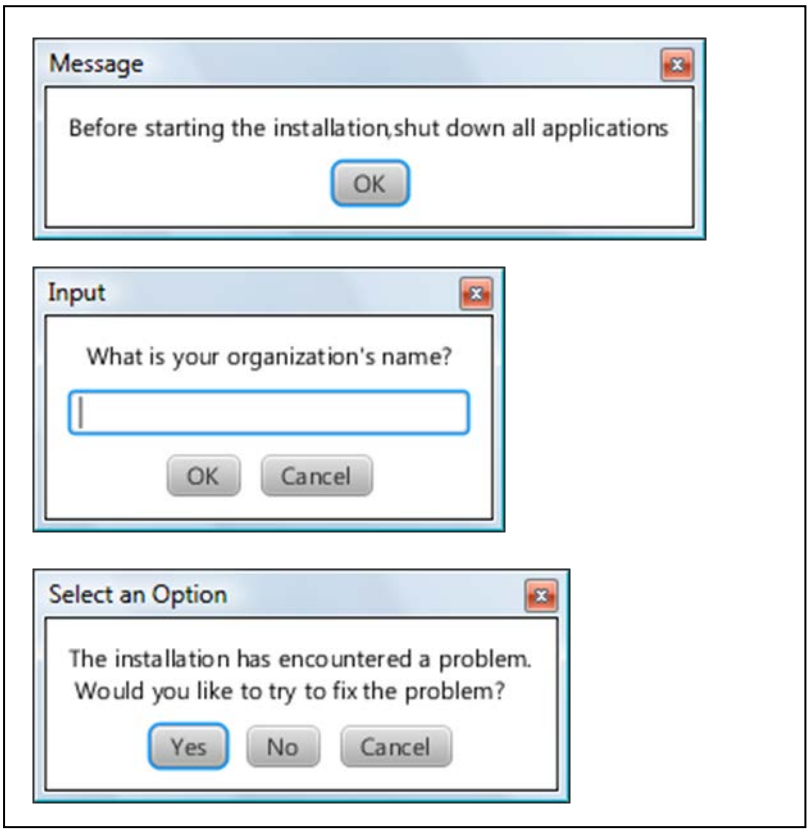


Figure 17.13 Three different types of dialog box – message, input, and confirm

Definition of Dialog Boxes

Now let's look at the code that defines these dialog boxes and describes what happens when a user interacts with one of their components. Figure 17.14a shows the first part of the `Dialog` class. This class does not extend the `Application` class. There is no `main` method and no `start` method. This class just provides class methods that respond to calls from another class that does extend the `Application` class and does have `main` and `start` methods. We'll describe that other class later.

The first class variable is a partially educated builder. This one builder will build the `FlowPane` object for the scene in any of the three dialog boxes in Figure 17.13. At this point we teach the builder all shared attributes – all attributes that apply to any of the three dialog boxes. The second class variable stores user-entered information for the input and confirm boxes.

The bottom half of Figure 17.14a shows the method that defines a message dialog box. This method defines and creates the dialog box, but it does not show the box. It returns a reference to the box, so the method caller can manage its use. Inside the `createMessageDialog` method, the first statement creates the dialog box. Its `StageStyle.UTILITY` argument minimizes the banner decoration. In Figure 17.13, notice that the only clickable item in the banners is the little "X" button on the right. Alternatively, we could get this relatively clean banner by using the zero-parameter `Stage` constructor and then later making a method call like this:

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

Now, skip over the event handler to the `dialogStage` method calls below. The `initModality` method gives the dialog window priority. It prevents a user from interacting with anything else on the screen until the dialog box closes. The `setTitle` method puts the word "Message" in the dialog box's title bar. The next statement sets the scene. In this case, the scene is a `FlowPane` formed by using the previously constructed and partially educated `FlowPaneBuilder` object as a starting point. It finishes the builder's education by adding the

pane’s children. Then it converts the builder to a pane. Then it uses the pane to create the scene. And finally it sets the scene in the dialog window.

```
/*********************************************************
* Dialog.java
* Dean & Dean
*
* This implements three dialog boxes with JavaFX builders.
*********************************************************/

import javafx.stage.*;         // Stage, StageStyle, Modality
import javafx.scene.Scene;
import javafx.scene.layout.FlowPaneBuilder;
import javafx.scene.text.Text;
import javafx.scene.control.*; // Button, TextField, Labeled
import javafx.geometry.*;      // Pos, Insets
import javafx.event.*;         // EventHandler, ActionEvent

public class Dialog
{
  // create partially educated builder
  private static FlowPaneBuilder builder = FlowPaneBuilder.create()
    .prefWrapLength(150)
    .alignment(Pos.CENTER)
    .vgap(10)
    .hgap(10)
    .padding(new Insets(10));
  private static String entry; // to store user input or entry

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

  public static Stage createMessageDialog(String text)
  {
    Stage dialog = new Stage(StageStyle.UTILITY);
    Button okButton = new Button("OK");
    EventHandler<ActionEvent> handler = new EventHandler<ActionEvent>()
    {
      public void handle(ActionEvent e)
      {
        Button source = (Button) e.getSource();
        entry = "";
        source.getScene().getWindow().hide();    // close dialog box
      } // end handle
    }; // end anonymous inner class EventHandler

    dialog.initModality(Modality.APPLICATION_MODAL);
    dialog.setTitle("Message");
    dialog.setScene(new Scene(builder
      .children(new Text(text), okButton)
      .build()));
    dialog.setResizable(false);
    okButton.setOnAction(handler);
    return dialog;
  } // end createMessage
```

Figure 17.14a `Dialog` class part A – up through `createMessage` method

The `setResizable(false)` method call preserves the scene's layout by preventing the user from resizing the window. The `setOnAction` method call registers an event handler called `handler` with the `okButton`. Its definition was the code we skipped over earlier. When an event occurs, the first statement in the `handle` method retrieves the event's source and casts it into type `Button`. The next statement erases any previous value that might be in the class variable, `entry`, since this dialog box is just a message. The last statement employs a method-call chain to make the message dialog box disappear.

Notice that we could have defined this handler anonymously in the `setOnAction` argument. If this class had no other methods, what's what we would have done, because it would be closer to where it's used. But that strategy would not work as well for the other two methods in this class. So, in the interest of parallelism, we defined it in a way that also works well for the other two methods.

Figure 17.14b shows the `createInputDialog` method. It has essentially the same structure as the previous `createMessageDialog` method. Except down near the bottom, notice that this time we register the handler with two different buttons, `okButton` and `cancelButton`.

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

public static Stage createInputDialog(String text)
{
  Stage dialog = new Stage(StageStyle.UTILITY);
  final TextField field = new TextField();
  Button okButton = new Button("OK");
  Button cancelButton = new Button("Cancel");
  EventHandler<ActionEvent> handler = new EventHandler<ActionEvent>()
  {
    public void handle(ActionEvent e)
    {
      Button source = (Button) e.getSource();
      if (source.getText().equals("OK"))
      {
        entry = field.getText();
      }
      else // source is "Cancel"
      {
        field.setText("");
      }
      source.getScene().getWindow().hide();    // close dialog box
    } // end Handle method
  };

  dialog.initModality(Modality.APPLICATION_MODAL);
  dialog.setTitle("Input");
  dialog.setScene(new Scene(builder
    .children(new Text(text), field, okButton, cancelButton)
    .build()));
  dialog.setResizable(false);
  okButton.setOnAction(handler);
  cancelButton.setOnAction(handler);
  return dialog;
} // end createInputDialog
```

Figure 17.14b `Dialog` class part B – the `createInputDialog` method

Since the same handler responds to an event from either button, it needs to know which button fired the current event. If the event source is the "OK" button, the handler copies the field's value into the class's `entry` variable. Otherwise, it erases the field's value. (Aside: Before Java 1.8, the compiler required that the `field` variable be `final` because it's accessed from within an anonymous method.)

Figure 17.14c contains the `createConfirmDialog` method. It has essentially the same structure as the previous `createMessageDialog` and `createInputDialog` methods. Except this time we register the handler with three buttons, `yesButton`, `noButton`, and `cancelButton`. And this time, the handler responds the same for all three sources. Figure 17.14c also contains a `getEntry` method, which allows an outsider to retrieve the value saved when the user clicks any dialog button.

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

  public static Stage createConfirmDialog(String text)
  {
    Stage dialog = new Stage(StageStyle.UTILITY);
    Button yesButton = new Button("Yes");
    Button noButton = new Button("No");
    Button cancelButton = new Button("Cancel");
    EventHandler<ActionEvent> handler = new EventHandler<ActionEvent>()
    {
      public void handle(ActionEvent e)
      {
        Button source = (Button) e.getSource();
        entry = source.getText();
        source.getScene().getWindow().hide();    // close dialog box
      } // end Handle method
    };

    dialog.initModality(Modality.APPLICATION_MODAL);
    dialog.setTitle("Select an Option");
    dialog.setScene(new Scene(builder
      .children(new Text(text), yesButton, noButton, cancelButton)
      .build()
    ));
    dialog.setResizable(false);
    yesButton.setOnAction(handler);
    noButton.setOnAction(handler);
    cancelButton.setOnAction(handler);
    return dialog;
  } // end createConfirmDialog

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

  public static String getEntry()
  {
    return entry;
  } // end getEntry
} // end class Dialog
```

Figure 17.14c `Dialog` class part C – the `createConfirmDialog` method and the `getEntry` method

Now you can probably see why we chose to put our handler definitions at the beginnings of these methods. Parallelism makes code easier to understand. Because handlers are important, we want them in a prominent place. Code should be local. It should be short. It should not be redundant. There is conflict among these goals. For

example, to make handlers as local as possible, we define them as anonymous arguments of the `setOnAction` method calls. As indicated earlier, that strategy would work well in the `createMessageDialog` method. But it would result in three identical definitions in the `createConfirmDialog` method. Thus, for the `Dialog` class as a whole, a uniform strategy that emphasized localization would increase code length and introduce redundancy. That's why we chose the next-best form of localization − anonymous definitions in a local variable at the beginning of each method. In the next subsection, for the sake of clarity, we back one step farther away from the goal of strict localization and define our handlers in inner classes.

Use of Dialog Boxes

Figure 17.15 shows the initial display produced by a driver class that illustrates use of dialog boxes. The scene contains a `FlowPane` with five components: (1) a prompt label, (2) a message button, (3) an input button, (4) a confirm button, and (5) another label. The flow pane's line wrapping puts the fifth component on the left side of a second row. This fifth component is not initially visible because it is initially empty.

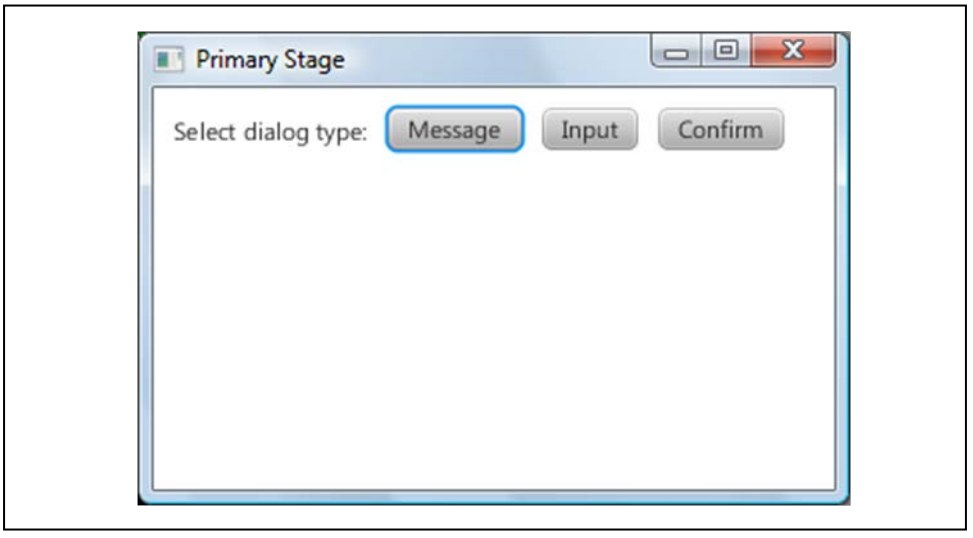


Figure 17.15 Primary stage generated by the `DialogDriver` class

Figure 17.16a contains the first part of the `DialogDriver` class – the part of the program that generates the display in Figure 17.15 and registers event handlers with each of the buttons in that display. The window you see in Figure 17.15 is the `Stage` object that's passed into our `start` method by Java API `Application` code. In Java lingo, it's called the *primary stage*. Our dialog stages are secondary stages.

By now, you should recognize the imports. The only novel feature is that now we import the entire `javafx.stage` package, so that we can access the `WindowEvent` class in addition to the familiar `Stage` class. This `DialogDriver` class `extends Application` and includes a `main` method and the obligatory `start` method, so execution of our program starts here.

The first four instance variables are the four components you see in Figure 17.15. The `result` label is the currently empty (and therefore invisible) component on the left side of the second row. The `Stage` variable, `dialogStage` will get a reference to the particular dialog box in Figure 17.13 that corresponds to the button the user clicks in Figure 17.15.

Now look at the `start` method in Figure 17.16a. The first statement sets the primary stage's title. The long second statement sets the primary stage's scene by creating an anonymous `Scene` whose argument is an anonymous `FlowPane`. As you can see, this `FlowPane` is created by a `FlowPaneBuilder`. This time, the builder is anonymous, and the method-call chain proceeds all the way through to the `build` method call, which returns the desired `FlowPane` object. Since the builder method-call chain before the final `build` method call includes prescription of all desired primary stage features (including the addition of its five components), we do

not need a `createContents` method to organize and fill the primary stage.

```
/*********************************************************
* DialogDriver.java
* Dean & Dean
*
* Creates primary stage and dialog boxes and gathers entries.
*********************************************************/

import javafx.application.Application;
import javafx.stage.*;          // Stage, WindowEvent
import javafx.scene.Scene;
import javafx.scene.layout.FlowPaneBuilder;
import javafx.scene.control.*;  // Button, Label
import javafx.geometry.Insets;
import javafx.event.*;          // EventHandler, ActionEvent

public class DialogDriver extends Application
{
  private Label prompt = new Label("Select dialog type:");
  private Button message = new Button("Message");
  private Button input = new Button("Input");
  private Button confirm = new Button("Confirm");
  private Label result = new Label();

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

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

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

  @Override
  public void start(Stage primaryStage)
  {
    primaryStage.setTitle("Primary Stage");
    primaryStage.setScene(new Scene(
      FlowPaneBuilder.create()
        .children(prompt, message, input, confirm, result)
        .prefWidth(340)
        .prefHeight(200)
        .hgap(10)
        .padding(new Insets(10))
      .build()
    ));
    primaryStage.setResizable(false);
    message.setOnAction(new ShowHandler());
    input.setOnAction(new ShowHandler());
    confirm.setOnAction(new ShowHandler());
    primaryStage.show();
  } // end start
```

Figure 17.16a `DialogDriver` class – part A

As in the previous `Dialog` class, the `setResizable(false)` method call preserves the scene's layout by preventing the user from resizing the window. The three `setOnAction` method calls just before the `show` method call register instances of a common event handler called `ShowHandler`. The `show` method call displays what you see in Figure 17.15.

The `DialogDriver` class continues in Figure 17.16b. This contains an inner class that defines the `ShowHandler` event handler and another inner class that defines a `HideHandler` event handler.

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

  private class ShowHandler implements EventHandler<ActionEvent>
  {
    public void handle(ActionEvent e)
    {
      Stage dialogStage = null;
      Button source = (Button) e.getSource();

      switch (source.getText())
      {
        case "Message":
          dialogStage = Dialog.createMessageDialog("Before starting "
            + "the installation,shut down all applications");
          break;
        case "Input":
          dialogStage = Dialog.createInputDialog(
            "What is your organization's name?,");
          break;
        case "Confirm":
          dialogStage = Dialog.createConfirmDialog(
            "The installation has encountered a problem\n" +
            "Would you like to try to fix the problem?");
      }
      dialogStage.getScene().getRoot().setId(source.getText());
      dialogStage.setOnHiding(new HideHandler());
      dialogStage.show();
    } // end handle method
  } // end ShowHandler

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

  private class HideHandler implements EventHandler<WindowEvent>
  {
    public void handle(WindowEvent e)
    {
      Stage dialogStage = (Stage) e.getSource();
      String dialogId = dialogStage.getScene().getRoot().getId();

      result.setText(dialogId + ": " + Dialog.getEntry());
    } // end handle method
  } // end HideHandler
} // end DialogDriver
```

Figure 17.16b `DialogDriver` class – part B

The `ShowHandler` event handler responds to a click on any of the three buttons on the primary stage (in Figure 17.15) by calling the `Dialog` class method which corresponds to that clicked primary-stage button. Since instances of the same `ShowHandler` are registered with all three primary-stage buttons, the `ShowHandler` must be able to differentiate among those three buttons. It does this by using a `getSource` method call to retrieve the source of the fired event. It casts the returned value to a `Button` and then uses it as the argument in a `switch` statement. The `switch` statement's three `case` constants are the text labels on the three primary-stage buttons.

In the appropriate case block, the `Dialog` class method call returns a reference to the dialog box that corresponds to the clicked primary-stage button, and this returned reference goes into the `dialogStage`

variable. After the `switch` statement, `dialogStage` gives itself an identification that matches the text on the primary-stage button that invoked it. The next statement registers with the dialog stage an instance of another event handler called `HideHandler`. Notice that this registration is with a container (a secondary stage) rather than a component. No problem. Handlers are quite versatile. The final `show` method call displays the selected dialog stage on the computer screen.

The code near the bottom of Figure 17.16b defines the `HideHandler`. This responds to a `WindowEvent`, which fires when one of the `source.getScene().getWindow().hide()` methods calls in the `Dialog` class methods in Figure 17.14a, 17.14b, and 17.14c closes its dialog box. Notice that the registration method call is `setOnHide` (rather than `setOnAction`) and notice that the event is a `WindowEvent` (rather than an `ActionEvent`). No problem. There are many different kinds of events and many different ways to register them.

The `HideHandler` response is relatively simple. It uses its `getSource` method to retrieve a reference to the dialog box that just closed and casts it into a `Stage`. It uses that to retrieve the just-closed dialog box's identifier, which is in its seene's root node. This illustrates one of the identification techniques described earlier in Section 17.13. Then, in the `result` label on the second row of the primary stage, the `HideHandler` displays this identifier followed by the `Dialog` class entry that was saved when the user clicked a dialog button. One of the end-of-chapter exercises asks you to implement this hide handler differently – anonymously and inside the `ShowHandler` class.

Responding to Dialog Box Action – a Waiting Game

Let's see how this works in a particular example. Figure 17.17 shows what's on the screen after a user clicks the "Confirm" button on the primary stage shown in Figure 17.15.

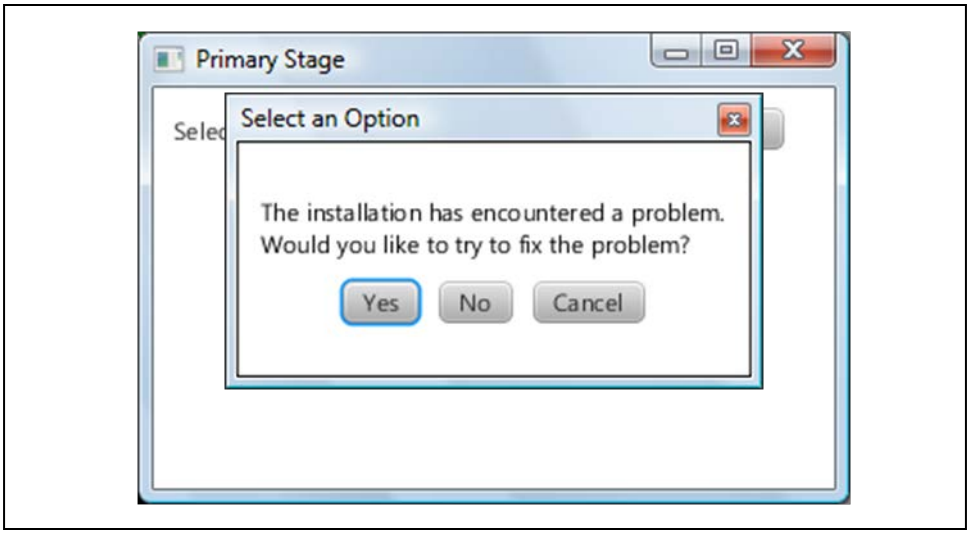


Figure 17.17 Display after user clicks the "Confirm" button on the primary stage shown in Figure 17.15

After the user clicks one of the buttons in the dialog box, the event handler in `Dialog`'s `createConfirmDialog` method stores the clicked button's text label in `Dialog`'s `entry` variable and hides (closes) the dialog window. Then `DialogDriver`'s `HideHandler` retrieves the value stored in `Dialog`'s `entry` variable and displays it in the `result` label on the second row on the primary stage. For example, in Figure 17.17, suppose the user clicks the dialog box's "Yes" button. The dialog box disappears, and the primary stage image changes to what you see in Figure 17.18. Notice that the string on the second row starts with a word (the dialog box's identifier) that matches the word on the primary-stage button the user clicked to display the dialog box. That's because `ShowHandler` copied dialog box's identifier from that button's text.

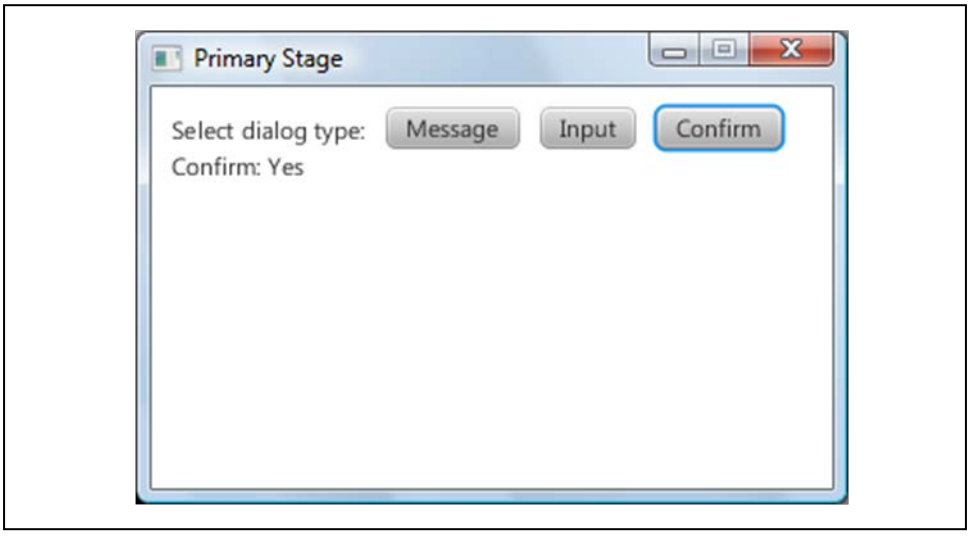


Figure 17.18 Display after user clicks the "Yes" button in the dialog box in Figure 17.17

You might be wondering, why not just have the `createInputDialog` or `createConfirmDialog` methods in Figures 17.14b and 17.14c return the user-entry information? Why do we need that `HideHandler` in Figure 17.16b?

Here's the reason: For simple operations, a computer is faster than a user. If the method that displays the dialog box tries to return the user's entry right after it presents the display, the user doesn't have time to think, type, or move and click the mouse. If we wanted the method that presents the display to also return the user's entry, before returning it would need to wait until the user has made the entry.[4]

Instead of waiting at the end of the method that presents the dialog, another alternative is to wait at the beginning of the method that reads the user's entry. That's how `Scanner`'s `nextLine` method works. It waits until the user hits the Enter key before trying to read the user's entry. And that's the strategy we use here. The principal difference is that `Scanner`'s code is implicit API code, whereas this program's code is explicit. It shows you how event-driven programs play the waiting game.

Our `DialogDriver` is rather lengthy because we wanted to illustrate many aspects of dialog-box use. Typically, though, you'll want to display just one dialog box and read the user's input into just that box. If you just want to display a message, you can write just one statement, like this:

```
Dialog.createMessage("<your-message>");
```

In this case you can ignore the problem of waiting for the user's response, because there is no user input. Another end-of-chapter exercise illustrates this case.

17.16 JavaFX CSS - Cascading Styles and Stylesheets

In Figures 17.5 and 17.10b, we had the flow pane call its `setStyle` method to establish a common font size for all of that pane's components. These methods calls employ a style-specification format like the format used in HTML's *Cascading Style Sheets* (CSS). A cascade is a sequence of falls, like a closely-coupled sequence of waterfalls. A cascading style specification falls down through a scene graph's compositional hierarchy. In Figure 17.5 or Figure 17.10b, the top of this compositional hierarchy – the root node – was a flow pane. In Figure 17.5, the font size specification cascaded from the flow pane down to a single label. In Figure 17.10b, it cascaded down to two labels, two text fields, and a button. CSS style specifications come in two forms – inline style specifications

[4] The `JOptionPane` class in `javax.swing` package does this kind of thing, but the industry is migrating away from swing implementations because they are relatively slow.

and class style specifications. A CSS inline style specification is like an anonymous Java class. A CSS class style specification is like a named Java class.

What you saw in Figures 17.5 and 17.10b were examples of inline style specifications. For your convenience, here's the relevant statement from Figure 17.10b:

```
pane.setStyle("-fx-font-size: 14");
```

Each node in a scene has a hidden variable called `style`, which contains a string that describes all the inline style properties established in a previous `setStyle` method call, and a `getStyle` method call retrieves that inline style description.

Each node also contains a `styleClass` variable, a `List` of `String`s that are *selectors* of style classes available to that node. Each style class contains one or more property/value pairs, like the property/value pair in quotation marks in the `setStyle` method call above. To make the property/value pairs in a style class identified by a selector called `selector-a` available to a particular node called `nodeA`, use a method call like this;

```
nodeA.getStyleClass().add("selector-a");
```

But this is only half of what it takes to make style class information available to a JavaFX program. You also need to tell your program where that information is. Each style class is in a specially formatted text file, which has a name like <*filename*>`.css`. Assuming that <*filename*>`.css` is in the same directory as the java program that uses it, to access style class information from any node in a JavaFX `Scene` called `scene`, use a method call like this:

```
scene.getStylesheets().add("<filename>.css");
```

To access style class information from or below a particular `Parent` node called `parent`, use a method call like this:

```
parent.getStylesheets().add("<filename>.css");
```

Inline Style Specification Syntax

To specify a single property, like font size, put everything on the same line, as we did above and in previous examples. To specify multiple properties, insert semicolons between individual property specifications. You may put more than one specification on the same line, like this:

```
pane.setStyle("-fx-font-size: 16; -fx-background-color: lightgreen");
```

If you want to specify many properties, extend the string onto additional lines, like this:

```
pane.setStyle(
  "-fx-font-size: 16;" +
  "-fx-alignment: top-center;" +
  "-fx-background-color: lightgreen;" +
  "-fx-background-insets: 5;" +
  "-fx-border-insets: 10;" +
  "-fx-border-width: 5;" +
  "-fx-border-color: blue;" +
  "-fx-border-radius: 15;" +
  "-fx-border-style: solid;" +
  "-fx-padding: 10");
```

You can probably guess what many of these specifications do, but if not, don't worry. We'll discuss that later.

At this point, however, we ask you to notice that CSS property names (before the colons) and values (after the colons) have a format that's different from typical Java format. For example, consider the second property/value pair in the argument above:

```
-fx-alignment: top-center
```

The property is `-fx-alignment`, and the value is `top-center`. If these were Java variable names, they

would probably be written in camel-case and all-caps, like this: `fxAlignment` and `TOP_CENTER`. But CSS properties and values are all lower case, with hyphens substituted for spaces between words, except there are no spaces or hyphens in multiple-word names of colors. Also note that all JavaFX CSS property names include the distinctive prefix, `-fx`.

As indicated earlier, CSS font properties automatically cascade down to lower scene-graph nodes. In addition to font properties, two other CSS properties automatically cascade down to lower nodes:

```
-fx-text-alignment
```

and:

```
-fx-cursor
```

For other properties to cascade down to a lower node, that node must make an explicit request by using `inherit` instead of a particular value. For example, suppose the previous `pane.setStyle` method call applies to the container of a component called `label`. If you want the label's text to be spaced away from the label's boundary by the same amount that the label is spaced away from the pane's boundary, you could use a method call like this:

```
label.setStyle("-fx-padding: inherit");
```

If you want to come back later and add another inline property specification to a node that already has one or more inline property specifications, precede your new specification with the old specification plus a semicolon, like this:

```
<node>.getStyle() + "; <new-specification>"
```

If the new specification happens to include a property that is the same as a property in the old specification, the new property's specification replaces the old property's specification. If you forget to insert the semicolon before your new specification, your new specification will replace everything in the old specification, and you might erase some specifications you want to keep. So if you come back later, be careful!

Class Style Specification Syntax

Now let's look at the other form of CSS style specification – the class style specification. To specify multiple style properties and to make those specifications usable in several different places, we put them into a separate text file named with the extension, `.css`. Figure 17.19 contains the CSS file that defines properties just like those in the previous `pane.setStyle` method call.

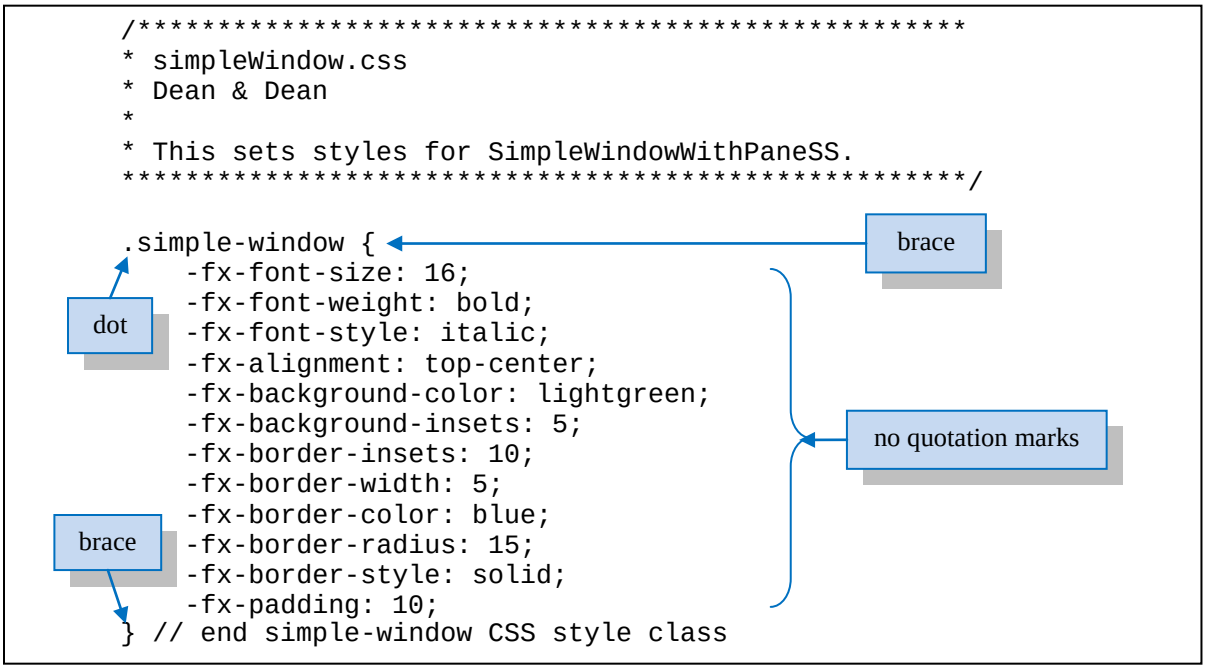


Figure 17.19 CSS file for SimpleWindowWithCSS program in Figure 17.20

The `simple-window` name is a style-class selector. In the CSS file, the selector name is preceded by a dot. The opening curly brace could be moved to the beginning of the next line to make the format look like a Java block. But this is not Java code. For convenience, we save this `test.css` text file in the same directory as the JavaFX class which uses it.

Figure 17.20 shows a JavaFX class that uses the CSS file in Figure 17.19. Notice that the number of imports needed is somewhat less than for many of our previous JavaFX programs. That's because now most of the GUI aspects are embedded in the CSS file, and they are handled behind the scenes at runtime. Because most of the GUI work is now in the CSS file, the remaining code fits easily into the obligatory `start` method.

The first five statements in `start` should be familiar to you. The sixth statement tells the program where the style sheet is, and the seventh statement makes that stylesheet's information available to the `FlowPane` container called `pane`. This container is the root node in the scene graph, and in this example it is the node that uses most of the stylesheet information.

```
/*************************************************************
* SimpleWindowWithCSS.java
* Dean & Dean
*
* This program displays a label in a window.
*************************************************************/

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

public class SimpleWindowWithCSS extends Application
{
  private static final int WIDTH = 300;
  private static final int HEIGHT = 150;

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

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

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

  @Override
  public void start(Stage stage)
  {
    FlowPane pane = new FlowPane();
    Scene scene = new Scene(pane, WIDTH, HEIGHT);
    Label label = new Label("Hi! I'm Larry the label!");

    stage.setTitle("Simple Window with CSS");
    stage.setScene(scene);
    scene.getStylesheets().add("simpleWindow.css");
    pane.getStyleClass().add("simple-window");
    pane.getChildren().add(label);
    label.setStyle("-fx-background-color: skyblue;"
      + "-fx-text-fill: red;"
      + "-fx-padding: inherit;"
    );
    stage.show();
  } // end start
} // end class SimpleWindowWithCSS
```

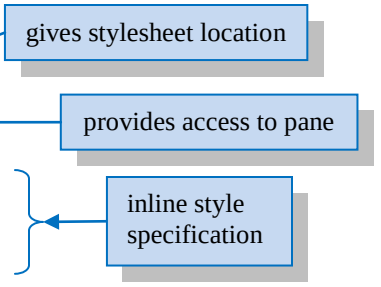


Figure 17.20 `SimpleWindowWithCSS` class that used the CSS file in Figure 17.19

Nevertheless, `pane` contains one child node called `label`, and the next statement uses an inline `setStyle` method call to specify those `label` properties that do not cascade down automatically from `pane`. Which properties cascade down automatically? The font properties. The `label` component automatically receives the CSS file's font-size specification, and because `label` does not override it with an explicit inline font-size specification, `label` uses the automatically inherited font size.

The label's inline `setStyle` method call specifies a background color that differs from the pane's

background color, and this makes it possible to see the label's extent. It also changes the text's color from the default black to red. It also specifies the label's padding – the space between the label's text and its boundary. It does this by using `inherit` to cascade the padding value down from pane. This example illustrates two kinds of cascading, the automatic cascading of a font-size property and an inherited cascading of a padding property.

Figure 17.21 shows the window generated by the `SimpleWindowWithCSS` class in Figure 17.20, using the CSS style sheet defined in Figure 17.19. The callouts indicate the dimensional properties specified by the style sheet in Figure 17.19 and explicitly inherited label padding. Notice how the label's light blue background specified by the `setStyle` method call overrides the pane's light green background specified in the style sheet.

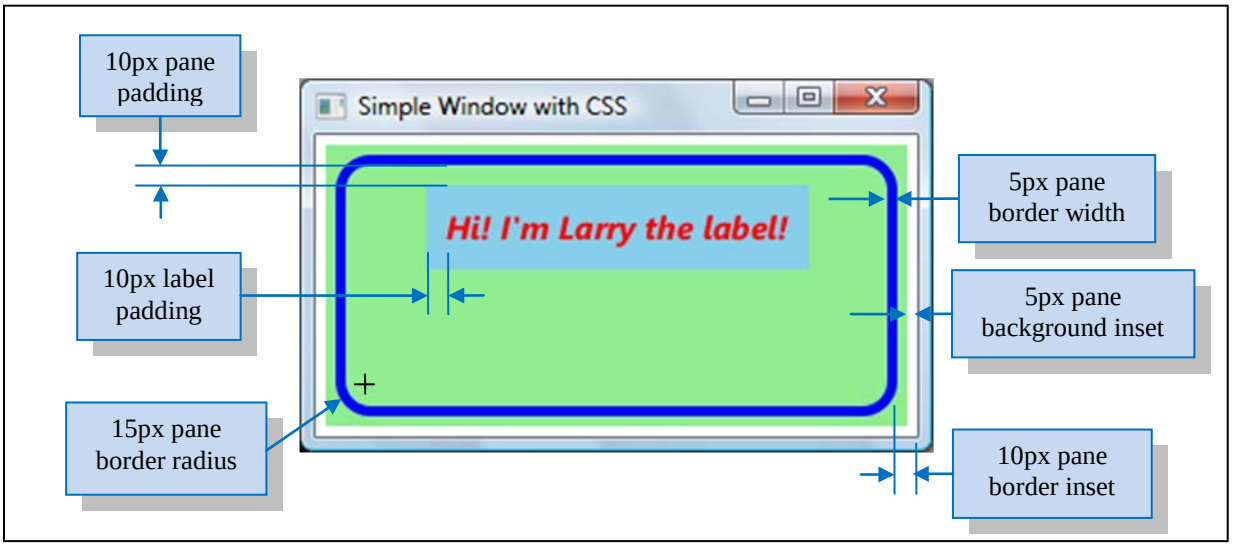


Figure 17.21 Display generated by Figure 17.20's SimpleWindowWithCSS program with Figure 17.19's stylesheet

Sometimes it's hard to remember the relative position of padding versus insets (or margins). Think of it this way: We put padding inside a fragile packing box. So padding is inside a border. In Figure 17.21, the pane padding is inside the pane border, and the label padding is inside the label's boundary.

Now suppose we add to the code in Figure 17.20 just this one statement:

```
label.getStyleClass().add("simple-window");
```

This applies the entire style sheet to the label (as well as to the pane). Figure 17.22 shows how the additional label border inset and border width pushes the text 15 pixels farther down in the overall window. And even after subtracting 5 pixels all around for the label background inset, the label background is still 20 pixel higher and wider than it is in Figure 17.21.

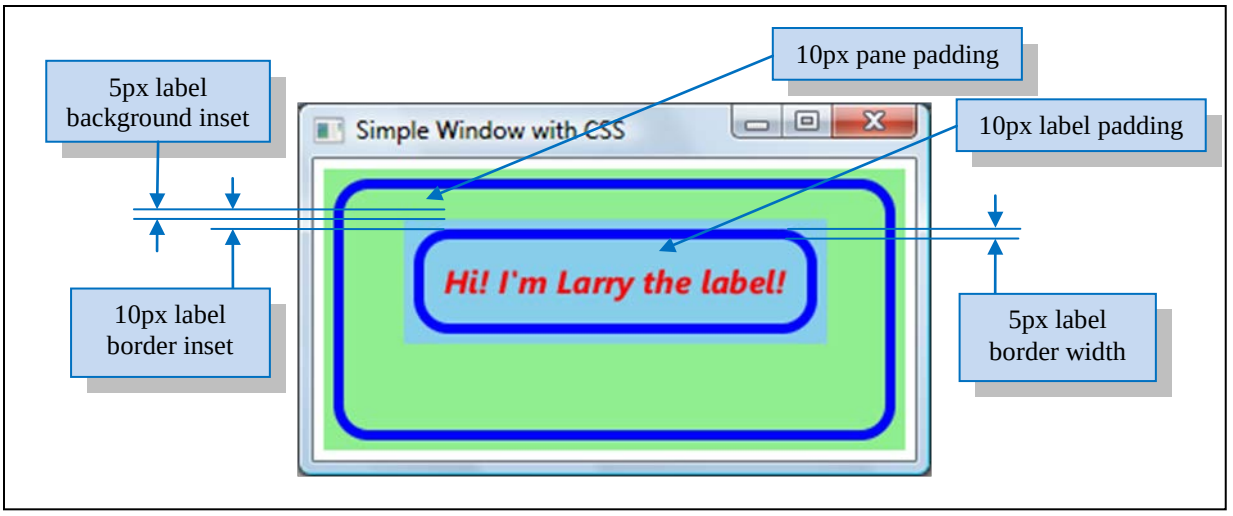


Figure 17.22 Display generated by the SimpleWindowWithCSS program in with the additional statement:
`label.getStyleClass().add("simple-window");`

CSS Properties

Notice that the font-size style specification applies only to the label, not to the pane. JavaFX Pane objects do not themselves use font. Therefore, they do not support font style specifications. They just ignore Figure 17.19's CSS font style specification, `-fx-font-size: 16;` However, Figure 17.22 indicates that the pane and the label both support the other style specifications in Figure 17.19.

The following table provides an abbreviated summary of important CSS properties, corresponding allowed values, and JavaFX classes that support those properties. The properties are listed in alphabetical order. Use this table as a reference for your convenience, after you have seen illustrative examples, or as a starting point for experimentation. For example, see if you can find all the style specifications in Figures 17.19 and 17.20 and verify their supporting classes. (Figure 17.7 shows that `Region` is a superclass of `Pane`.)

CSS Property	CSS Value	Classes
-fx-alignment	[top-left, top-center, top-right, center-left, center, center-right, bottom-left, bottom-center, bottom-right]	Labeled, FlowPane, GridPane, Hbox, VBox, TilePane
-fx-background-color	<paint>	Labeled, Region
-fx-background-insets	<size>	Labeled, Region
-fx-border-color	<color>	Labeled, Region
-fx-border-insets	<size>	Labeled, Region
-fx-border-radius	<size>	Labeled, Region
-fx-border-style	[solid, dotted, dashed]	Labeled, Region
-fx-border-width	<size>	Labeled, Region
-fx-column-halignment	[left, center, right]	FlowPane
-fx-effect	<effect>	Node
-fx-fill	<paint>	Shape
-fx-font	<family, size, style, and weight>	Text, Labeled
-fx-font-family	<family name>	Text, Labeled

-fx-font-size	<size>	Text, Labeled
-fx-font-style	[normal, italic, oblique]	Text, Labeled
-fx-font-weight	[normal, bold, bolder, lighter]	Text, Labeled
-fx-graphic	<location>	Labeled
-fx-graphic-text-gap	<size>	Labeled
-fx-grid-lines-visible	<boolean>	GridPane
-fx-hgap	<size>	FlowPane, GridPane, TilePane
-fx-opacity	[0.0-1.0]	Node
-fx-orientation	[horizontal, vertical]	FlowPane, TilePane
-fx-padding	<size>	Labeled, Region
-fx-pref-columns	<integer>	TilePane
-fx-pref-rows	<integer>	TilePane
-fx-pref-tile-height	<size>	TilePane
-fx-pref-tile-width	<size>	TilePane
-fx-rotate	<cw deg right>	Node
-fx-row-valignment	[top, center, bottom]	FlowPane
-fx-scale-x	<multiplier>	Node
-fx-scale-y	<multiplier>	Node
-fx-spacing	<size>	Hbox, VBox
-fx-strikethrough	<boolean>	Text
-fx-stroke	<paint>	Shape
-fx-stroke-line-cap	[square, butt, round]	Shape
-fx-stroke-type	[inside, outside, centered]	Shape
-fx-stroke-width	<size>	Shape
-fx-text-alignment	[left, center, right, justify]	Text, Labeled
-fx-text-fill	<paint>	Labeled
-fx-translate-x	<offset>	Node
-fx-translate-y	<offset>	Node
-fx-underline	<boolean>	Text, Labeled
-fx-vgap	<size>	FlowPane, GridPane, TilePane
-fx-wrap-text	<boolean>	Labeled
visibility	[visible, hidden]	Node

Where you see <paint>, it's usually just <color>, which we'll describe shortly. But <paint> can also be a <radial-gradient> or a <linear-gradient>. We'll show you a radial-gradient example in the next chapter.

CSS Value Types

Here is an abbreviated summary of the possible types of property values, and related notes. This list will help you figure out the more cryptic notations in the second column of the three-column table above..

- boolean: `"true"` or `"false"`
- strings: use `\"` for embedded quote and \A for embedded newline
- numbers & integers: decimal real or decimal integer values respectively
- size: a number or integer with appended length or percentage symbol, or pixels by default
- length: `px` (screen pixels), `in` (inches), `cm` (centimeters), `mm` (millimeters), `pt` (1 point = 1/72 inch)

- percentage: %
- angle: `deg`, `rad`, `grad`, or `turn` (complete rotation) in clockwise direction
- point: length length
- color-stop: color percentage, ... (like this: `red 10%, white 50%, blue 90%`)
- url: absolute or relative, with forward slashes, and with or without quotes
- effect: drop shadow or inner shadow (see CSS documentation)
- font-family: `'serif'`, `'sans-serif'`, `'cursive'`, `'fantasy'`, or `'monospace'`
- font-size: size (using syntax above)
- font-style: `normal`, `italic`, or `oblique`
- font-weight: `normal`, `bold`, `bolder`, or `lighter`
- font: style(optional) weight(optional) size family
- paint: linear gradients or radial gradients (see CSS documentation and later examples in this book)
- color: a large number of pre-defined color names, and numerical specs (see following subsection)

Any length, percentage, or angle symbol must be immediately appended to the preceding number, with no whitespace between. See CSS documentation for full details.

CSS Colors

There are several different ways to specify color in a `setStyle` method call or a CSS stylesheet. One way is to write the color's name, using all lowercase letters with no spaces between multiple-word names. Another way is to use the pound sign (#) followed by three two-hexit numbers. The first number is the value of red in the range from `00` to `ff`. The second number is the value of green in the same range. The third number is the value of blue in the same range. Thus, `#4488cc` means a little bit of red, a medium amount of green, and quite a bit of blue.

Figure 14.23 shows samples of 148 different colors, followed by the name and the hexadecimal representation of those colors. We got the information for this table from Oracle's CSS description, but the display you see here was generated by a Java program that uses three labels in a flow pane for each color. Then each color's flow pane is embedded in a large tile pane, which implements the whole table. We describe tile panes and pane-in-pane embedding in the next chapter. The color patch to the left of each color is actually two patches, side by side. The left side of the combined patch is generated by the name representation and the right side is generated by the hexadecimal representation. The fact that both sides of each combined color patch look the same verifies that the name and hexadecimal representations agree.

The very last entry, "transparent", uses an alternate numerical representation that we could have used for every other entry, also. In the parentheses after "rgba" the first item is the value of red, in the range 0 to 255. The second item is the value of green in the range 0 to 255. The third entry is the value of blue in the range 0 to 255. The fourth entry is the *alpha* or opaqueness in the range 0.0 to 1.0. In this particular case, since the alpha is 0.0, it's completely transparent, and the values of the other entries don't matter. But when the alpha is greater than 0.0, the values of the other entries do matter.

To convert any hexadecimal representation to the equivalent rgba representation, split the six hexits into three pairs. Then convert each pair to decimal. For example, here's how you could convert `yellowgreen`'s hexadecimal representation into the equivalent rgba representation:

#9acd32 → 9a cd 32 → 9*16 + 10, 12*16 + 13 , 3*16 + 2, 1.0 → rgba(154, 205, 50, 1.0)

Using an alpha of less than unity for a component's background color blends the component's background color with the background color of that component's container.

If you want to use the 0-255 representation and you want it to be completely opaque, you can omit the final item in the parentheses and use use rgb instead of rgba, like this:

#9acd32 → rgb(154, 205, 50)

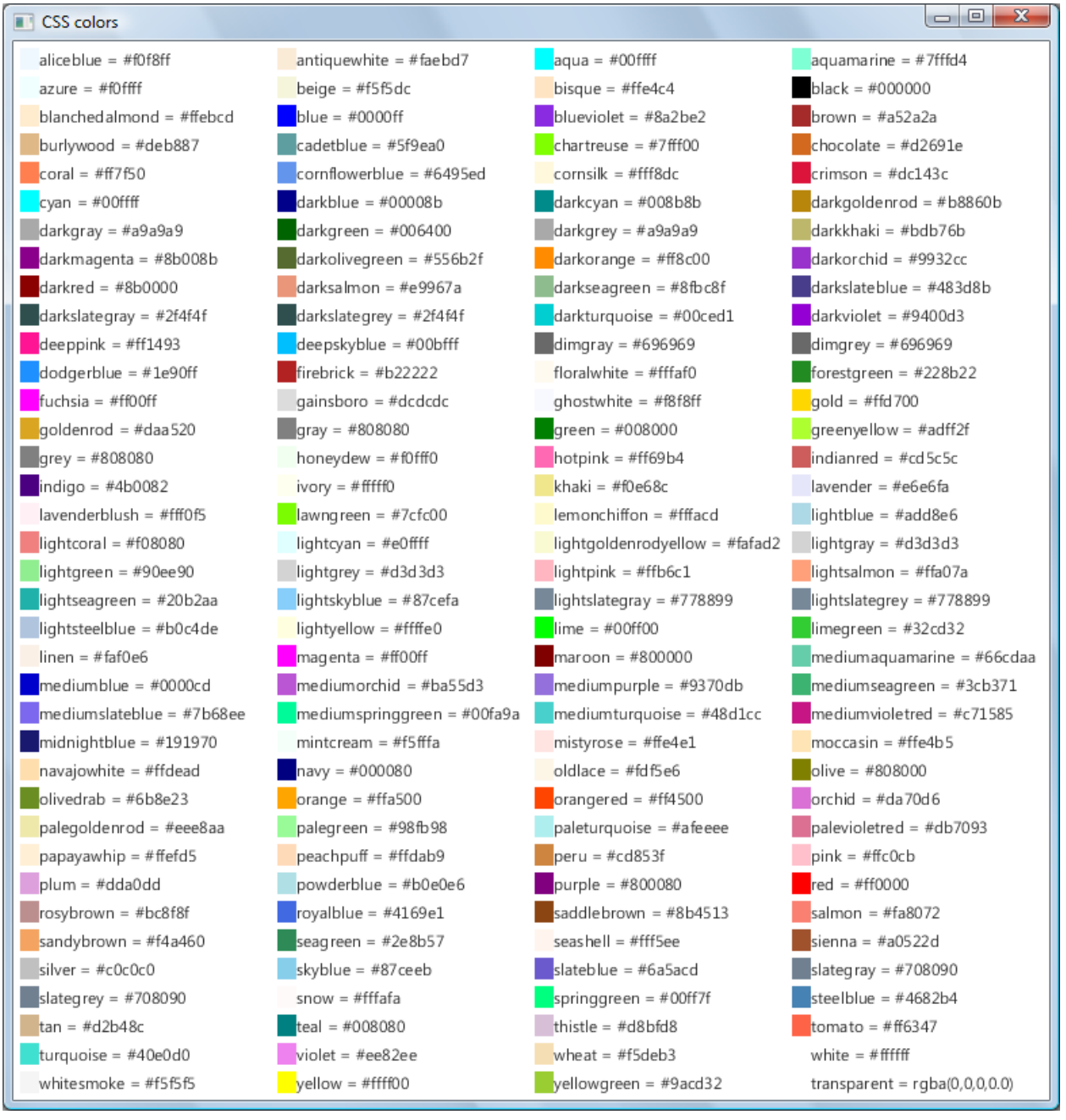


Figure 17.23 CSS colors with name and hexadecimal specification for each color

17.17 Overview of Java's GUI Libraries

Throughout this chapter, you've used Java's pre-built GUI classes from Oracle's JavaFX API library. For example, you used the `Application` class for creating a window and the `Button` class for creating a button. In this section, we describe how Oracles's pre-built GUI classes are grouped and organized, starting with legacy classes developed before the advent of JavaFX.

The AWT and Swing Libraries

In the first Java compiler, all GUI classes were bundled into one library known as the Abstract Windowing Toolkit (AWT). The AWT's GUI commands generate GUI components that look different on different platforms. In other words, if your program instantiates an AWT button component, the button will have a Macintosh look and feel if the program runs on a Macintosh computer, but it will have a Windows look and feel if the program runs on a Windows computer. That leads to portability issues. Your programs are still portable in the sense that they'll run on different platforms. But they'll run differently on different platforms. If you have a persnickety customer who demands the same appearance on all platforms, AWT components probably won't be satisfactory.

One of Java's strongest selling points was (and is) its portability, so soon after Java's initial release, Java language designers proceeded to develop a set of more portable GUI components. They put their new, more-portable components in a brand new library named Swing. To make the relationship clear between the new Swing components and the AWT components, they used the same component names except that they prefaced the new Swing components with a "J." For example, AWT has a `Button` component, so Swing has a `JButton` component.

The AWT GUI components are known as *heavyweight components*, while the Swing GUI components are known as *lightweight components*. The AWT components are heavyweight because they are built with graphics commands that are part of the computer's platform. Being part of the computer's platform, they're too "heavy" to move to other platforms. Swing components are lightweight because they're built with Java code. Being built with Java code means that they're "light" enough to "swing" from one platform to another.

The Swing library includes quite a bit more than just GUI component classes. It added lots of functionality to the AWT, but it did not replace the AWT entirely. In fact, for a long time, Java applet programmers typically used AWT only, even for GUI components, and they did not use the Swing library at all. Why? Because applets relay on browsers and, sadly, many browsers used old versions of Java, versions that did not include Swing.

The primary AWT packages are `java.awt` and `java.awt.event`. The primary Swing package is `javax.swing` (`javax.swing` is one of several `javax` packages). The "x" in `javax` stands for "extension" because the `javax` packages were a major extension to the core Java platform.

The JavaFX Library

On Oracle's JavaFX web site, technical writer Cindy Castillo describes the history of JavaFX as follows:

> At the JavaOne 2007 conference, Sun Microsystems introduced the JavaFX platform to help content developers and application developers to create content-rich applications for mobile devices, desktops, televisions, and other consumer devices. The initial offering consisted of the JavaFX Mobile platform and the JavaFX Script language. Multiple public releases were delivered after the initial announcement; the 1.3 version was released on April 22, 2010.
>
> After Oracle's acquisition of Sun Microsystems, Oracle announced during the JavaOne 2010 conference that support for the JavaFX Script language would be discontinued. However, it was also announced that the JavaFX Script APIs will be ported to Java and would be released as part of the JavaFX 2 product. This announcement meant that the JavaFX capabilities will become available to all Java developers, without the need for them to learn a new scripting language. With this announcement, Oracle has committed to making JavaFX the premier environment for rich client applications.

JavaFX 2 has a new graphics engine, which is designed to take maximum advantage of the latest hardware graphics processing units. This engine uses a new graphics pipeline called *Prism* and a new windowing tookkit called *Glass*. This new combination provides the speed of AWT with the portability of Swing.

17.18 Images and Mouse Events

If you look back at Figure 17.7, you'll see that the `Image` class is not a subclass of the `Node` class. Therefore, an `Image` instance cannot be a node in a scene graph. However, Figure 17.7 also has another class, called `ImageView`, which is a subclass of the `Node` class. Thus, an `ImageView` instance can be a node in a scene graph. To display an image, we create an `ImageView` object and have that object call its `setImage` method with the `Image` object as its

argument.

The `DragSmiley` program presented in this section will illustrate this mechanism with two different images, a smiley-face image, and a scared-face image. Initially, the image is a smiley face. If the user moves the cursor to within the boundary of this image and then presses the left mouse button, the image switches to a scared face – "Oh dear! What's happening?" Then, if the user moves the cursor with the left mouse button held down, the scared face moves along with it. In other words, the mouse drags the (now scared) image. When the user releases the button, the image changes back to a smiley face. And if the user moves the cursor away, the (now smiley) image remains where it was when the user released the mouse button.

In Figure 17.24, the window on the left shows the smiley face in its initial position. The window on the right shows the scared face while it's being dragged.

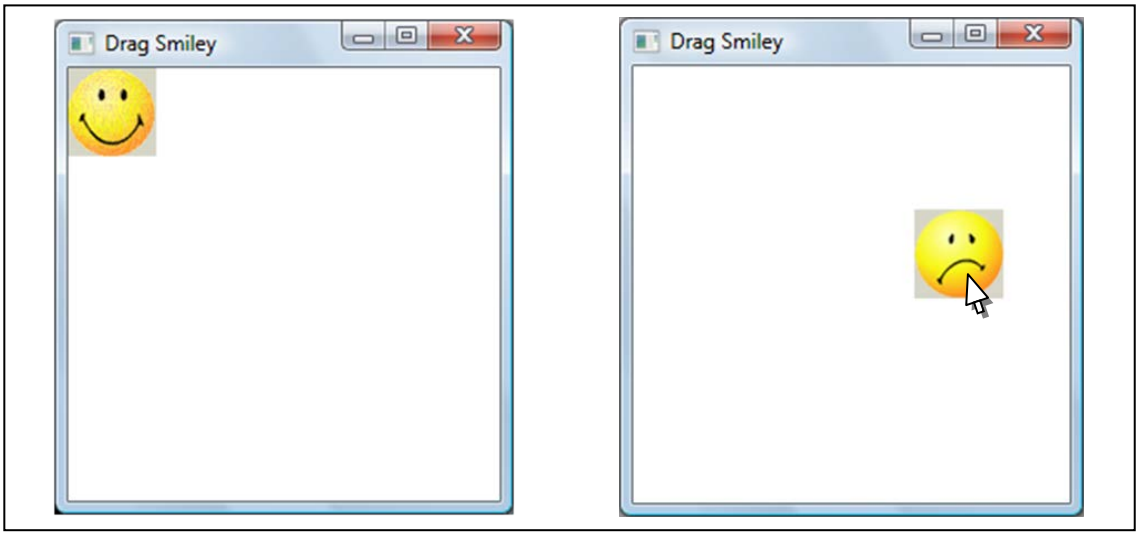


Figure 17.24 The initial smiley face on the left and the dragged scared face on the right

Most of our previous mouse clicks fell within the scope of an `ActionEvent`, which can be sourced by a text field or a button-like component. As you have seen, we can use an `ActionEvent` object to retrieve useful information about the event source. However, as you saw when we used a `WindowEvent` object, an `ActionEvent` is not the only kind of event. And it's not the best kind of event for determining the type of action that fired the event, because it is not very discriminating. For specific event-type information, it is better to use one of the more than twenty special `setOn` methods defined in the `Node` class. For our DragSmiley program, we'll use three of these methods to register event handlers for specific types of mouse events:

```
setOnMousePressed(EventHandler<MouseEvent> handler)
setOnMouseReleased(EventHandler<MouseEvent> handler)
setOnMouseDragged(EventHandler<MouseEvent> handler)
```

Instead of generating `ActionEvent`s, these three method calls generate `MouseEvent`s.

The `ActionEvent`, `WindowEvent` and `MouseEvent` classes are all subclasses of the `Event` class. Any `Event` instance can call its `getEventType` method to retrieve an object whose string representation is like the name of the `setOn` method that registers an event handler for that event. For example, if the handler is registered using a `setOnMousePressed` method call, in the handler you can distinguish a mouse-pressed event from other types of events with the condition:

```
e.getEventType().toString().equals("MOUSE_PRESSED")
```

During program development, you can learn the exact string representation of any event type by temporarily inserting into the handler a statement like this: `System.out.println(e.getEventType());`

Now let's consider the dragging process. When the mouse is dragging an image, we need a way to keep track

of the drag path. We'll do this by saving and updating the current mouse position in the scene. The first time we save this position will be when the mouse is pressed. Then whenever a mouse-drag operation fires a mouse event, we'll do the following: First we'll compute the change in mouse position from the previous mouse position. Then we'll move the image and correct the saved mouse position by the amount of the change.

Figure 17.25a contains the first part of the DragSmiley program. The instance constants include the two raw `Image` objects, `smiley.gif` and `scared.gif`. The `ImageView` object, `face`, is an instance variable. The `oldMouseX` and `oldMouseY` instance variables are the saved horizontal and vertical positions of the mouse.

```
/*********************************************************
* DragSmiley.java
* Dean & Dean
*
* This program displays a smiley face image.
* When the user presses the mouse on the image, it changes
* to a scared image. Then the user can drag the image.
*********************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.image.*;      // Image, ImageView
import javafx.event.EventHandler;
import javafx.scene.input.MouseEvent;

public class DragSmiley extends Application
{
  private static final int WIDTH = 250;
  private static final int HEIGHT = 250;
  private final Image SMILEY = new Image("smiley.gif");
  private final Image SCARED = new Image("scared.gif");
  private ImageView face = new ImageView();
  private double oldMouseX;
  private double oldMouseY;

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

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

Figure 17.25a DragSmiley program – part A

Figure 17.25b contains the rest of the DragSmiley program. Look at the `start` method. If `ImageView` were a subclass of `Parent`, we could set the scene with it, like we set the scene with a `Label` object in the original SimpleWindow program in Figure 17.4. But as you can see in Figure 17.7, `ImageView` is not a subclass of `Parent`, so we must put the `ViewImage` object in some kind of container. For that container, we chose to use a generic `Pane` object, because we do not need or want any background layout management. We want the user to manage the layout explicitly – with the mouse! After adding the face to the pane, we set the face to the smiley face. Then we call three `setOn` methods to register an event handler with `face` for three different types of events.

The event handler is an inner class called `MouseHandler`. Its `handle` method is one big `switch` statement. The `switch` condition is the `String` representation of the event's type. The three `case` constants are the `String`

representations of the types corresponding to the three `setOn` method calls in the `start` method. The rest of the code is straightforward.

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

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

    stage.setTitle("Drag Smiley");
    stage.setScene(new Scene(pane, WIDTH, HEIGHT));
    pane.getChildren().add(face);
    face.setImage(SMILEY);
    face.setOnMousePressed(new MouseHandler());
    face.setOnMouseReleased(new MouseHandler());
    face.setOnMouseDragged(new MouseHandler());
    stage.show();
  } // end start

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

  class MouseHandler implements EventHandler<MouseEvent>
  {
    public void handle(MouseEvent e)
    {
      switch (e.getEventType().toString())
      {
        case "MOUSE_PRESSED":
          face.setImage(SCARED);
          oldMouseX = e.getX();
          oldMouseY = e.getY();
          break;
        case "MOUSE_RELEASED":
          face.setImage(SMILEY);
         break;
        case "MOUSE_DRAGGED":
          double deltaX = e.getX() - oldMouseX;
          double deltaY = e.getY() - oldMouseY;

          face.setX(face.getX() + deltaX);
          face.setY(face.getY() + deltaY);
          oldMouseX += deltaX;
          oldMouseY += deltaY;
      } // end switch
    } // end handle method
  } // end inner class MouseHandler
} // end class DragSmiley
```

Figure 17.25b DragSmiley program – part B

Summary

- You define a JavaFX GUI class by extending `javafx.application.Application`. Your class's `main`

method calls `Application`'s pre-defined `launch` method. This calls a method your class must define, the `public void start(Stage stage)` method.

- The `Stage` object passed to your `start` method implements all the standard window features like a border, a title bar, a minimize button, a close-window button (the "X"), the ability to resize the window, and so on.
- In your `start` method, the `Stage` object calls its `setScene` method with a `Parent` node argument, optionally followed by scene width and height in pixels.
- After the `start` method or a helper method configures the scene, the `Stage` object calls its `show` method.
- `Label` is a read-only component node; it simply displays some text.
- The `TextField` component node allows the user to enter text into a text box.
- You can register an implementation of the `EventHandler` interface with `TextField` or `ButtonBase` node by having that node call its `setOnAction` method. Then when the user interacts with that node (perhaps by clicking a button or pressing Enter while the cursor is in a text box), the node fires an event.
- Your event handler "hears" the fired event and executes your event handler's obligatory `handle` method.
- If a class is limited in its scope such that it is only needed by one other class, then you should define the class as an inner class (a class inside of another class).
- If an inner class has only one instance, you can make it anonymous. An anonymous inner class has no name and is defined where it is instantiated. You can assign that single instance to a named variable and use that variable in multiple method calls. Or if you will use the instance only once, you can skip the assignment to a named variable and define the class right where you use it – typically in the argument of a method call.
- To identify the one of several possible nodes that fired an event, have the event call its `getSource` method. Then compare the returned source with a known reference to that source, or have the returned source call its `getText` or `getId` method to get an identifying string.
- For almost any JavaFX class, a builder accumulates just those attributes you specify in a `create(),...` method-call chain. A final `build()` call returns the built object.
- To display a simple window with a message, call the `Stage` constructor with a `StageStyle.UTILITY` argument. To prevent interactions with other windows until that simple window closes, have that new stage call `initModality(Modality.APPLICATION_MODAL)`.
- You can specify GUI node properties and values using an inline *<node>*`.setStyle` method call with semicolon-separated CSS property/string pairs in a string argument. Alternatively, you can put the CSS property/string pairs in a *<filename>*`.css` file in a block after a `.`*<selector>*. Then in Java code have the scene or a parent node call `getStylesheets().add(`*<filename>*`.css)`to locate the stylesheet, and have the node call `getStyleClass().add(`*<selector>*`)` to make the stylesheet's contents available to that node.
- To adjust the text color in a GUI component, use CSS's `"-fx-text-fill:` *<color>*`"`. To adjust the background color in a GUI container or component, use CSS's 's `"-fx-background-color:` *<color>*`"`. JavaFX also supports many other kinds of property specifications.
- To handle mouse events, register with a `setOn` method call that corresponds to the particular event of interest.

Review Questions

§17.2 Event-Driven Programming Basics

1. What is an event?
2. What is an event handler?

§17.3 A Simple Window Program

3. What kind of method is `launch`, and what does it do?

§17.4 Containers – `Stage`, `Scene`, and `Pane`

4. How many items might a scene contain?

§17.5 Components

5. Based on Figure 17.7 and the facts that a scene can contain any `Node` and a `Parent` can have children, indicate whether an instance of each of the following classes could be a scene-graph component and/or a scene-graph container by writing "yes" or "no" in the columns to the right:

class	component?	container?
`Image`		
`ImageView`		
`Text`		
`FlowPane`		
`Label`		
`Button`		
`TextField`		

§17.6 `Label` Component

6. Declare a `Label` reference variable named `hello` and initialize it with the string, "Hello World".

§17.7 `TextField` Component

7. Provide a statement that reduces a text field's current preferred column count by two. Assume the text field is called `nameBox`. Hint: As you might expect, there is also a `getPrefColumnCount` method.

§17.9 Event Handlers

8. Write a statement that registers an anonymous instance of an event handler defined in a class called `Responder` with a component named `component`.

9. When you define a class to handle an event, what do you add to the class's heading after the class name?

10. What is the heading of the one method the handler of an `ActionEvent` must implement?

§17.10 Inner Classes

11. An inner class is _____.

§17.11 Anonymous Inner Classes

12. If you want to implement an event handler with an anonymous inner class, what argument do you give to a `setOnAction` method to register the handler?

§17.12 `Button` Component

13. In the FactorialButton program in Figures 17.10a, 17.10b, and 17.10c, what component fires the event that the handler handles?

§17.13 Using an Event's `getSource` Method

14. Suppose the same handler is registered with several components, and the components and handler are defined within the same class. Within the handler, what `ActionEvent` method should you call to determine which component fires an event?

§17.14 Builders

15. In Figure 17.12, the `Scene` constructor argument contains seven chained method calls. What is returned by each of these calls?

§17.15 Dialog Boxes

16. What package contains the `WindowEvent` class?

17. Write a single statement that creates a secondary stage called `notice` with a banner that includes the 'X" (close) button only.

§17.16 JavaFX CSS – Cascading Styles and Stylesheets

18. Write a CSS statement that sets the style of a `Button` named `button1` so that its text is 20 points and white.

19. If you specify style properties for a container, all of that container's properties automatically apply to all of that container's components. (T / F)

§17.18 Images and Mouse Events

20. Write a statement that registers an instance of an inner class called `DragHandler` with a `Node` called `node` for mouse dragging events. Then write the header for that `DragHandler` class.

Exercises

1. [after §17.2] Give three examples of how a user might cause an event to be fired.

2. [after §17.3] For each of the following, what Java API package must you import?
a) `Application`
b) `Stage`
c) `Scene`
d) `Label`

3. [after §17.4] How many items might a scene contain?

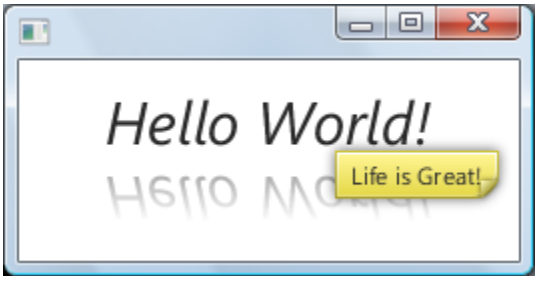
4. [after §17.5] Indicate how the output of Figure 17.5 changes if you replace:

```
pane.setStyle("-fx-font-size: 16");
```

with:

```
label.setStyle("-fx-font-size: 16");
```

5. [after §17.6] Provide a complete program that displays this "Hello World" message:



Note these label characteristics: (1) italics, (2) large font size (30 points), (3) tool tip that says "Life is Great!", (4) Reflection of text below. Use this program skeleton as a starting point:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.FlowPane;
import javafx.scene.effect.*;  // Effect, Reflection
import javafx.scene.text.*;    // Font, FontPosture
import javafx.scene.control.*; // Label, Tooltip
import javafx.geometry.*;      // Insets, Pos

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

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

  @Override
  public void start(Stage stage)
  {
    FlowPane pane = new FlowPane();
    Label label = new Label("Hello World!");

    <4-statement code fragment>

    <3-statement code fragment>
```

```
      stage.show();
    } // end start
  } // end BigHello class
```

In the 4-statement code fragment, set the scene with 250 pixels width and 100 pixels height, use top-center pane position, and an all-around pane padding of 10 pixels. For the 3-statement code fragment, in JavaFX's API, look up the `setFont`, `setToolTip`, and `setEffect` methods that `Label` inherits from `Labeled`, `Control`, and `Node` classes, respectively. For the `setFont` argument, use the class `font` method with "Serif" type, italic font posture, and 30.0 point size. For the `setTooltip` argument, create a `Tooltip` object with the indicated text value. For the `setEffect` method, create a `Reflection` object.

6. [after §17.7] What can you do to prevent users from updating a `TextField` component called `textBox`?

7. [after §17.9] The `EventHandler` interface and the `ActionEvent` class are in what JavaFX API package?

8. [after §17.10] An inner class can directly access its enclosing class's instance variables. (T / F)

9. [after §17.12] It's appropriate to use an anonymous inner class if you are going to use the class only once. In the FactorialButton program in Figures 17.10a, 17.10b, and 17.11, we used the `handler` object twice, so that `handler` object needed to have a name. Since we used that object's class only once (to instantiate the one object, the object's class did not need to have a name, and we could have used an anonymous class to create the `handler` object. For this exercise, modify the program to initialize `handler` with an anonymous `EventHandler` class instead of the named `Handler` class. [Hint: The program is already set up to facilitate this change—it's mostly cut-and-paste.]

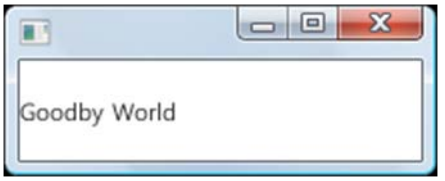
10. [after §17.13] By calling `setDisable(true)`, you can disable a button and give it a muted appearance and make its handler unresponsive to clicks on it. Modify the FactorialButton program in Figures 17.10a, 17.10b, and 17.11so that the factorial button is initially disabled. Enable it only after the user enters a character in the `xBox` text box. To enable it, add a `setOnKeyTyped` method call to also register a *key handler* with the `xBox` text box, and have the key handler's `handle` method call `setDisable(false)`. Use the following code skeleton:

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

You will also need to import `javafx.scene.input.KeyEvent` and change `btn` from a local variable to an instance variable.

11. [after §17.14] What `FlowPaneBuilder` method corresponds to `FlowPane`'s `getChildren` method?

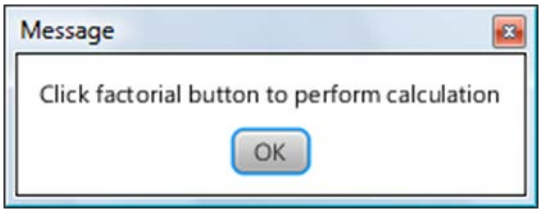
12. [after §17.15] Write a program that displays a primary window and a secondary window. The primary window should have no title and contain only a label that is initially empty (no text). The secondary window should have no title and its banner should include only the "X" (close) button. The secondary window's scene should contain only a label saying "Hello World", with default font, alignment and coloring and a frame size just big enough to display the label. While the secondary window is open it should not be possible to interact with any other windows. When the user clicks the "X" in the secondary window's banner, the secondary window should disappear and a handler should then insert the text, "Goodby World", into the primary window's label. Make the primary window's size be 200 pixels wide by 50 pixels high, but accept defaults for all other primary window features. When you execute the program you should see what's on the left below. When you click the secondary window's "X", you should see what's on the right below.



Put everything into one class with just two methods, a `main` method and a `start` method. This means you'll need to use an anonymous inner class for the event handler.

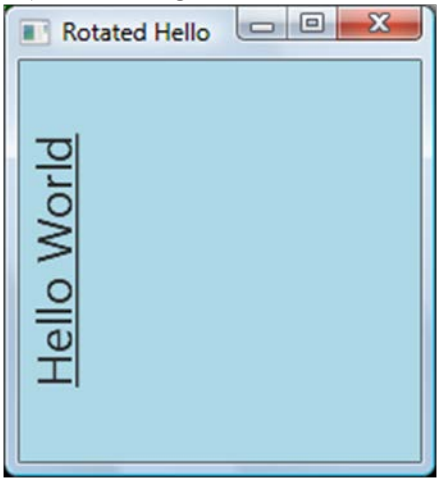
13. [after §17.15] Modify the code in the latter part of the `DialogDriver` class in Figure 17.16b so that the argument of the `setOnHiding` method call is an anonymous instance of an anonymous class.

14. [after §17.15] Modify the code in FactorialButton's refined `Handler` class in Figure 17.11 so that if the user tries to complete the entry by a keyboard `Enter` with the cursor in the text box, instead of completing the entry, the program responds by displaying the following message dialog box:



To do this, put the present body of the `handle` method in the "else" part of an "if, else" statement, and put a `createMessageDialog` method call in the "if" part of that statement.

15. [after §17.16] Write a CSS class file that prescribes the five special style properties needed to produce the display shown below. (The color is light blue and the font size is 24.)



Then write a Java program that generates this display, using the following code, with a chained `getRoot` method call in the second scene method-call statement.

```
public void start(Stage stage)
{
  Label message = new Label("Hello World");
  Scene scene = new Scene(message, 200, 200);
```

<two-scene-method-call-statements>

```
  stage.setTitle("Rotated Hello");
  stage.setScene(scene);
  stage.show();
} // end start
```

16. [after §17.17] What do the letters in "awt" stand for?

Review Question Solutions

1. An event is an object that tells your program that something has happened.

2. An event handler is the part of your program that responds to an event.

3. The `launch` method is a class method in your program's extension of the `Application` class. It creates an object of that `Application` extension. Then it has that object create a `Stage` object and call its `start` method with that stage as an argument.

4. A scene must contain exactly one item. That item could be one component, or it could be one container. That container could itself contain any number of components.

5. Possible components and containers in a scene graph:

class	component?	container?
`Image`	no	no
`ImageView`	yes	no
`Text`	yes	no
`FlowPane`	yes	yes
`Label`	yes	yes
`Button`	yes	yes
`TextField`	yes	yes

6. `Label hello = new Label("Hello World!");`

7. `nameBox.setPrefColumnCount(nameBox.getPrefColumnCount() - 2);`

8. `component.setOnAction(new Responder());`

9. When you define a class to handle an event, append something like this to the class's heading:

```
implements EventHandler<ActionEvent>
```

10. The heading of the one method an event handler must implement is:

```
public void handle(ActionEvent e)
```

11. An inner class is completely defined within another class.

12. The argument to give to a `setOnAction` method to register an anonymous handler class is:

```
new EventHandler<ActionEvent>()
{
  <implementation-of-EventHandler-interface>
}
```

13. It's ambiguous. It could be either `xBox` or `btn`.

14. To identify the firing component, call the `getSource` method.

15. What `DialogDriver`'s `Scene` argument method calls return:

`create` returns a new `FlowPaneBuilder` object
`children` returns `this` (the calling `FlowPaneBuilder` object)
`prefWidth` returns `this` (the calling `FlowPaneBuilder` object)
`prefHeight` returns `this` (the calling `FlowPaneBuilder` object)
`hgap` returns `this` (the calling `FlowPaneBuilder` object)
`padding` returns `this` (the calling `FlowPaneBuilder` object)
`build` returns a new `FlowPane` object

16. The package that contains the `WindowEvent` class is `javafx.stage`.

17. This statement creates a secondary stage called `notice` with a banner that includes the "X" (close) button only:

```
Stage notice = new Stage(StageStyle.UTILITY);
```

18. To make `button1`'s text 20 points and white, use:

```
button1.setStyle("-fx-font-size: 20; -fx-color-text: white");
```

19. False. Cascading is automatic for font, alignment, and cursor properties, but to inherit any other property, the component must call `setStyle` and give that other property an `inherit` value.

20. Statement that registers an instance of an inner class called `DragHandler` with a `Node` called `node` for mouse dragging events:

```
node.setOnMouseDragged(new DragHandler());
```

Header for that `DragHandler` class:

```
class DragHandler implements EventHandler<MouseEvent>
```