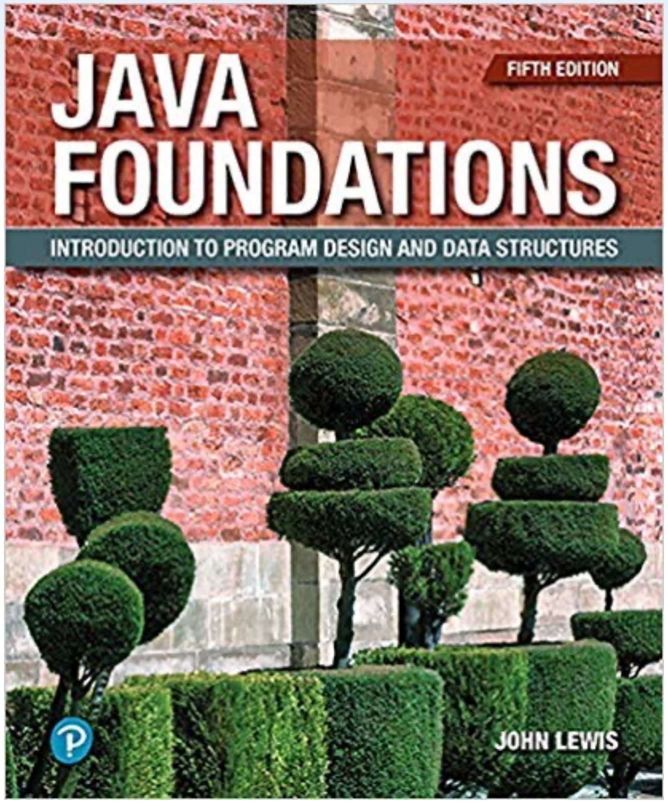


Chapter 6

Graphical User Interfaces (largely abbreviated)

Extended with additional material by K.J. Kochut

Event-Driven System

- An *event-driven system* is a software system which is designed around responding to various events.
- Events may include a key pressed on a keyboard, mouse pointer movement, an arrival of a message, or timer clock running out.
- A system is composed of code fragments that are invoked in response to events. These code fragments are called *event-handlers*.

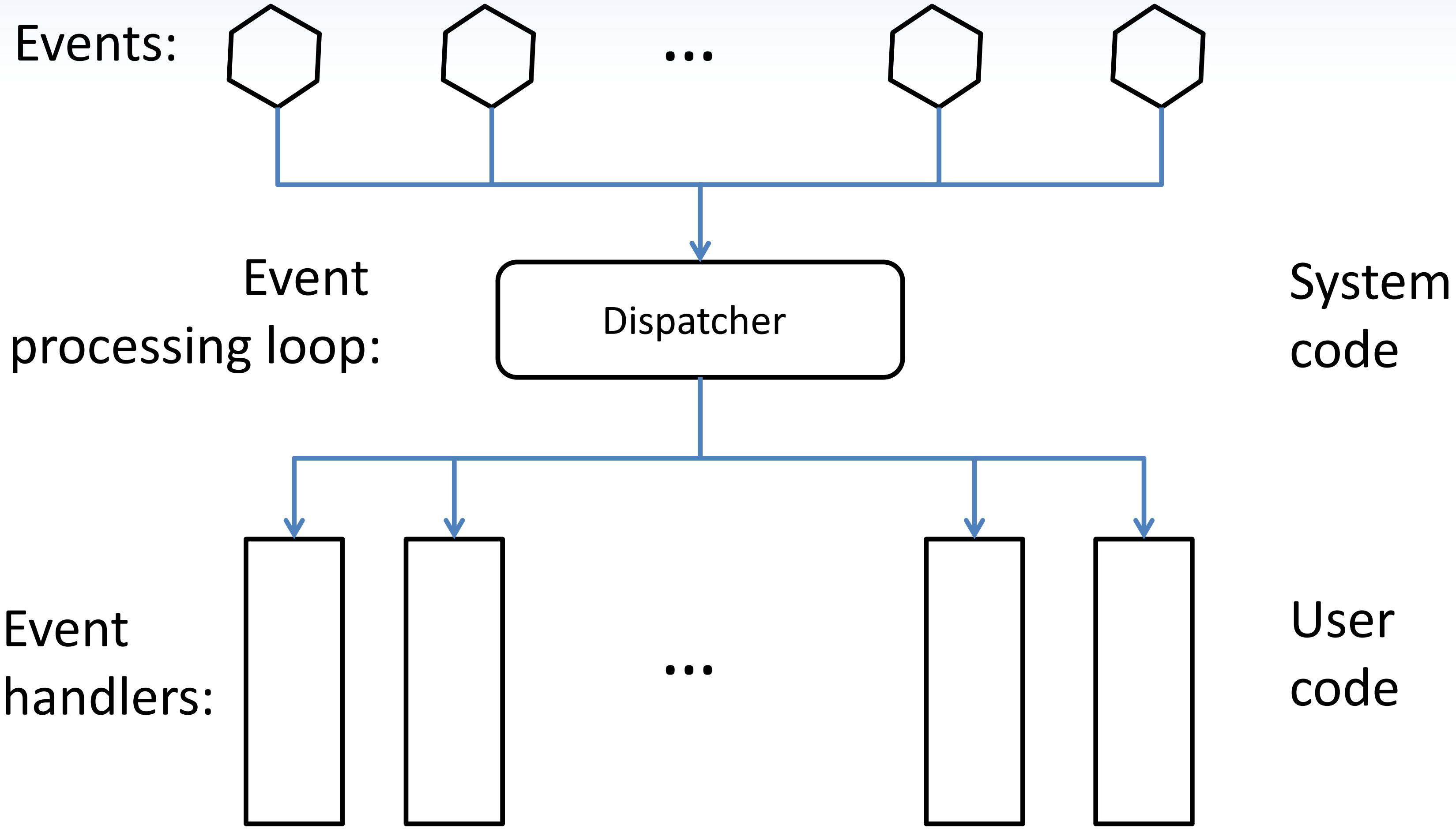
Event-Driven System

- Software control resides in a piece of code called *event-dispatcher*
- Event-dispatcher runs a continuous loop which accepts events and invokes the associated event-handlers
- In that sense, the occurring *events drive the computation*, as which code is executed is determined by the events; hence an *event-driven system*.

Procedure-Driven System

- Software control always resides in *procedures* (or methods, functions, subroutines, etc.), i.e., in user code.
- Procedures call each other and control is passed from the caller to the called procedure.
- Caller is blocked until the called procedure exists
- In that sense, procedures direct the control flow; hence an *procedure-driven system*.

Event-Driven System



Events:

...

Event processing loop:

Dispatcher

System code

Event handlers:

...

User code

KJK additions

Graphical User Interfaces

- A Graphical User Interface (GUI) in Java is created with at least three kinds of objects:
  - controls, events, and event handlers
- A *control* is a screen element that displays information or allows the user to interact with the program:
  - labels, buttons, text fields, sliders, etc.

Graphical User Interfaces

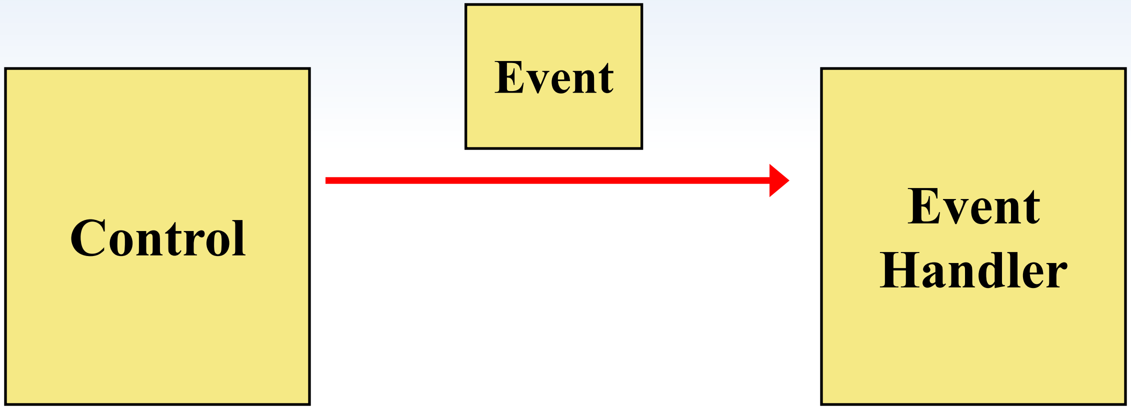
- An *event* is an object that represents some activity to which we may want to respond

- For example, we may want our program to perform some action when the following occurs:
  - a graphical button is pressed
  - a slider is dragged
  - the mouse is moved
  - the mouse is dragged
  - the mouse button is clicked
  - a keyboard key is pressed

Graphical User Interfaces

- The Java API contains several classes that represent typical events
- Controls, such as a button, generate (or fire) an event when it occurs
- We set up an *event handler* object to respond to an event when it occurs
- We design event handlers to take whatever actions are appropriate when an event occurs

Graphical User Interfaces

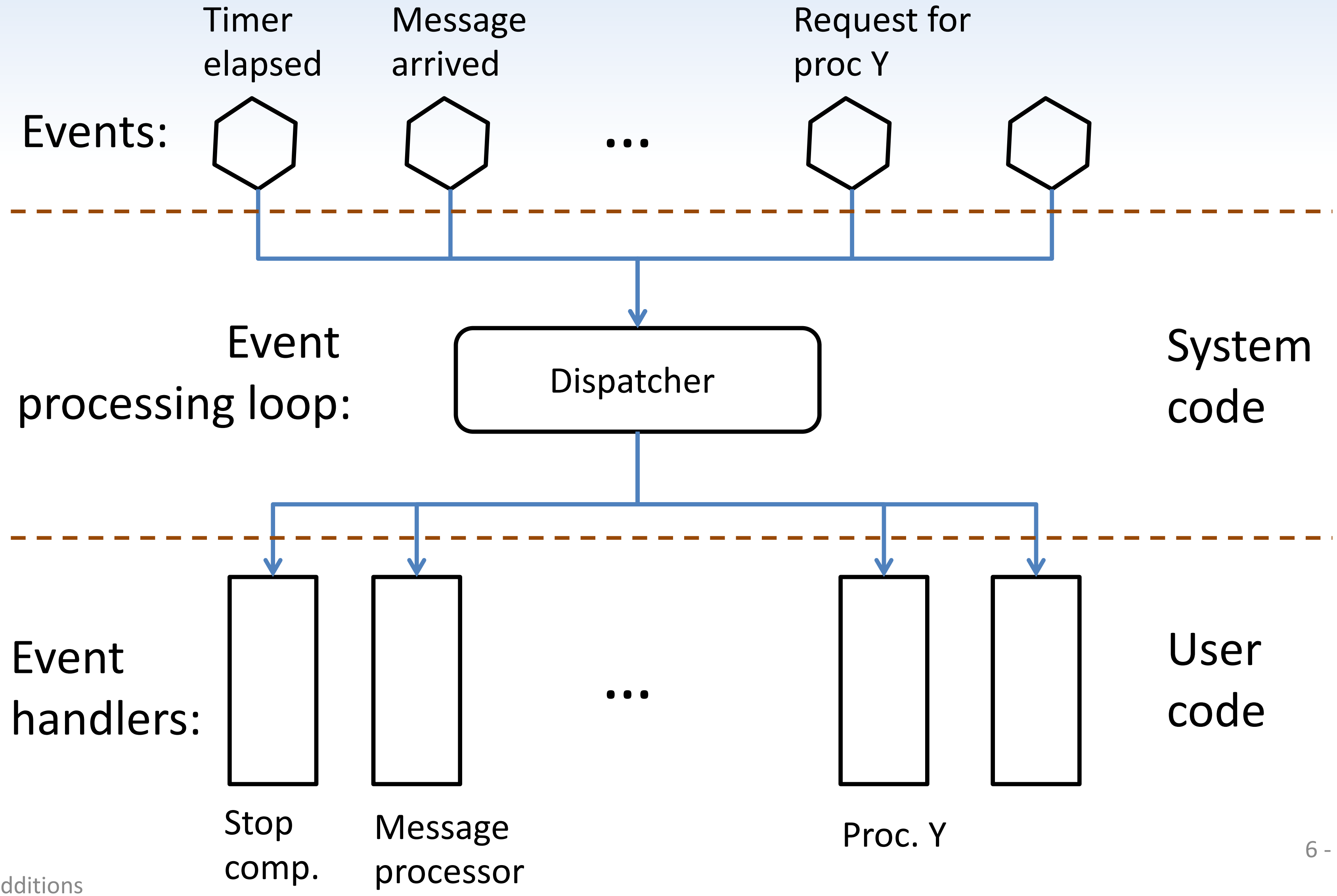


A control object generates an event

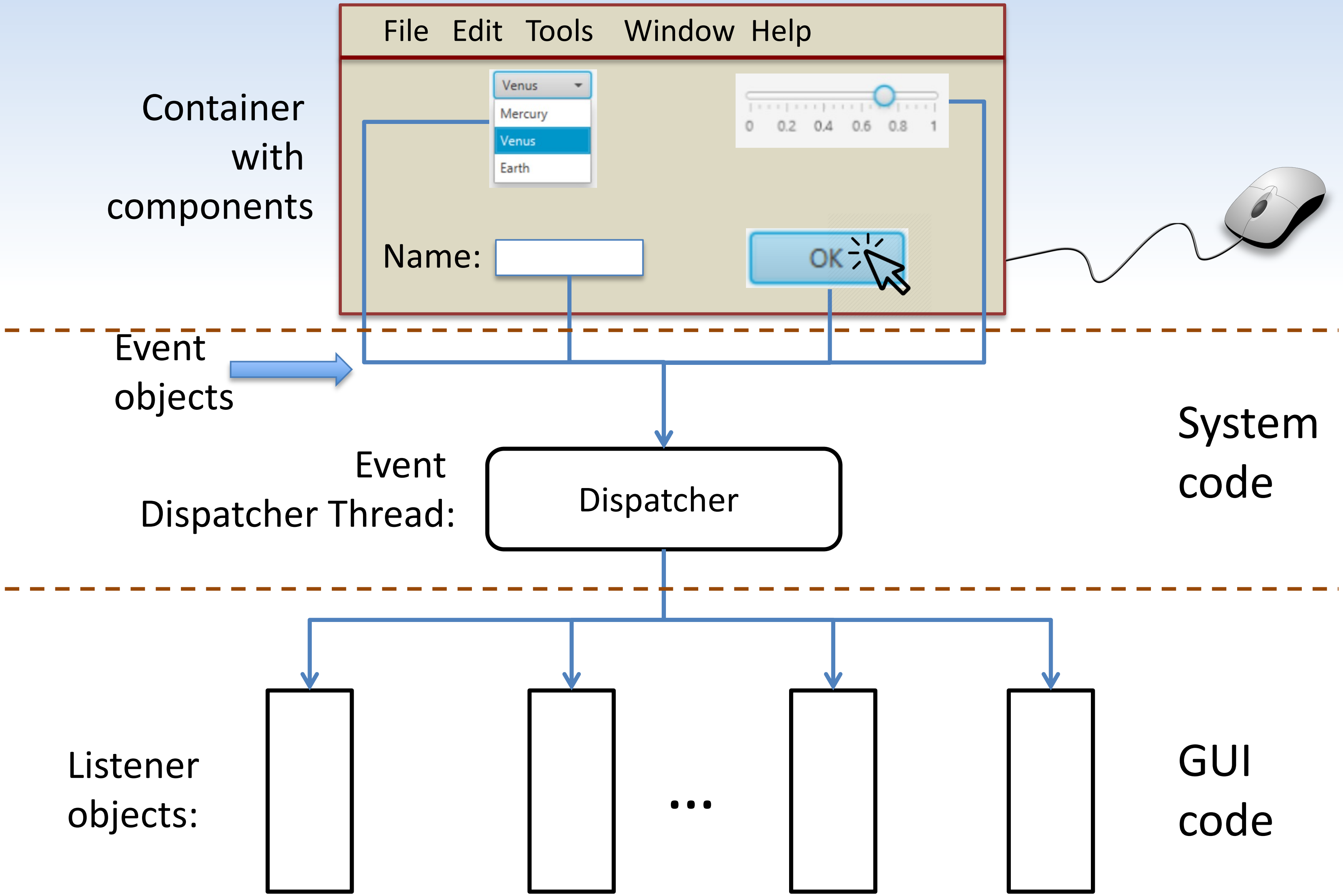
A corresponding event handler is designed to respond to the event

When the event occurs, the control calls the appropriate method of the listener, passing an object that describes the event

Event-Driven System



KJK additions



Container with components

Event objects

Event Dispatcher Thread:

Dispatcher

System code

Listener objects:

…

GUI code

OK Button
save_settings()

Textfields
set_name()

Slider
set_value()

DropDown
set_selection()

KJK additions

Intro to JavaFX

- JavaFX programs extend the `Application` class, inheriting core graphical functionality
- JavaFX embraces a theatre analogy
- A JavaFX program has a `start` method
- The `main` method is only needed to launch the JavaFX application
- The `start` method accepts the primary stage (window) used by the program as a parameter

Intro to JavaFX

- The scene is displayed on the primary `Stage` (window)
- The scene should contain the necessary controls arranged as desired
- Controls should have event handlers (listeners), as needed

Graphical User Interfaces

- A JavaFX *text* is defined by the Text class
- A JavaFX *button* is defined by the `Button` class
- A *button* can generate an *action event*
- The `PushCounter` example displays a *button* that increments a counter each time it is pushed and displays a *text* with the counter value

```java
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.text.Text;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;

//************************************************************************
//  PushCounter.java       Java Foundations
//
//  Demonstrates JavaFX buttons and event handlers.
//************************************************************************

public class PushCounter extends Application
{
    private int count;
    private Text countText;
```

continue

continue

```
    //--------------------------------------------------------------------
    //  Presents a GUI containing a button and text that displays
    //  how many times the button is pushed.
    //--------------------------------------------------------------------
    public void start(Stage primaryStage)
    {
        count = 0;
        countText = new Text("Pushes: 0");

        Button push = new Button("Push Me!");
        push.setOnAction(this::processButtonPress);

        FlowPane pane = new FlowPane(push, countText);
        pane.setAlignment(Pos.CENTER);
        pane.setHgap(20);
        pane.setStyle("-fx-background-color: cyan");

        Scene scene = new Scene(pane, 300, 100);

        primaryStage.setTitle("Push Counter");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

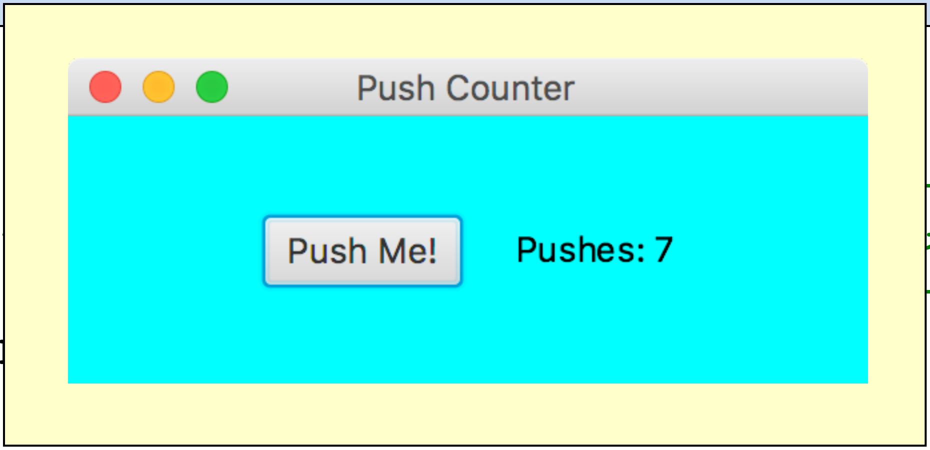
continue

continue

```
    //---------------------------------------------------------------------
    //  Updates the counter and text when the button is pushed.
    //---------------------------------------------------------------------
    public void processButtonPress(ActionEvent event)
    {
        count++;
        countText.setText("Pushes: " + count);
    }
}
```

continue

```
   //----------                                                   -----------------
   //  Updates                                                   ushed.
   //----------                                                   -----------------
   public void 
   {
      count++;
      countText.setText("Pushes: " + count);
   }
}
```



Graphical User Interfaces

- A call to the `setOnAction` method sets up the relationship between the button that generates the event and the event handler that responds to it

- This example uses a *method reference* (using the `::` operator) to specify the event handler method

- The `this` reference indicates that the event handler method is in the same class

- So the `PushCounter` class also represents the event handler for this program

Graphical User Interfaces

- The event handler method can be called whatever you want, but must accept an ActionEvent object as a parmeter

- In this example, the event handler method increments the counter and updates the text object

- The counter and `Text` object are declared at the class level so that both methods can use them

Graphical User Interfaces

- In this example, a `FlowPane` is used as the root node of the scene

- A flow pane is a layout pane, which displays its contents horizontally in rows or vertically in columns

- A gap of 20 pixels is established between elements on a row using the `setHGap` method

Alternate Event Handlers

- Instead of using a method reference, the event handler could be specified using a separate class that implements the `EventHandler` interface:

```
public class ButtonHandler implements EventHandler<ActionEvent>
{
    public void handle(ActionEvent event)
    {
        count++;
        countText.setText("Pushes: " + count);
    }
}
```

Alternate Event Handlers

- The event handler class could be defined as public in a separate file or as a private inner class in the same file
- Either way, the call to the `setOnAction` method would specify a new event handler object:

```
push.setOnAction( new ButtonHandler() );
```

Alternate Event Handlers

- Another approach would be to define the event handler using a *lambda expression* in the call to `setOnAction`:

```
push.setOnAction( (event) -> {
     count++;
     countText.setText("Pushes: " + count);
} );
```

- A lambda expression is defined by a set of parameters, the −> operator and an expression

Alternate Event Handlers

- A lambda expression can be used whenever an object of a *functional interface* is required
- A functional interface contains a single method
- The `EventHandler` interface is a functional interface
- The method reference approach is equivalent to a lambda expression

More Controls

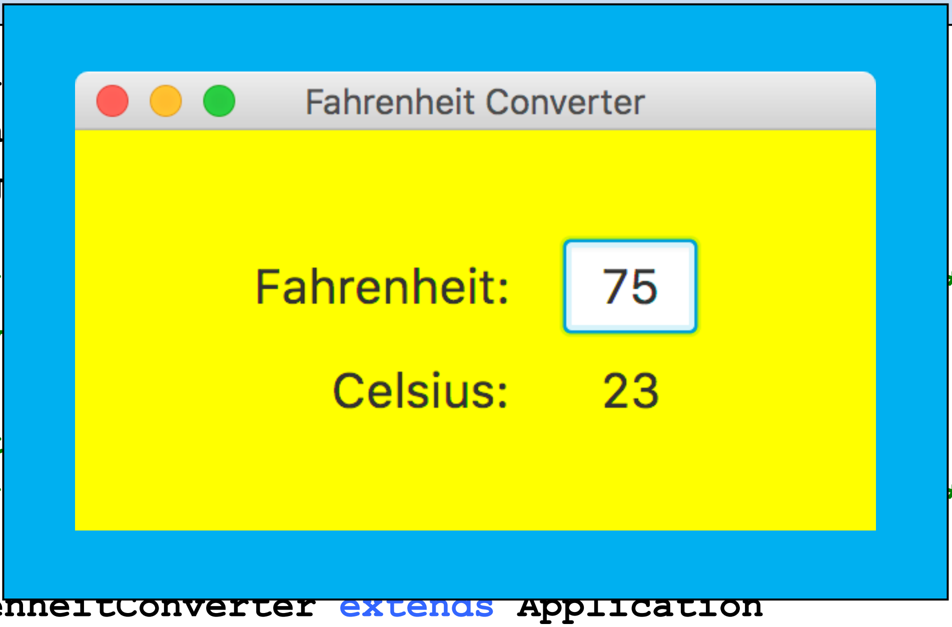
- In addition to push buttons, there are variety of other interactive controls
  - *text fields* – allow the user to enter typed input from the keyboard
  - *check boxes* – a button that can be toggled on or off using the mouse (indicates a boolean value is set or unset)
  - *radio buttons* – used with other radio buttons to provide a set of mutually exclusive options
  - *sliders* – allow the user to specify a numeric value within a bounded range
  - *combo boxes* – allow the user to select one of several options from a “drop down” list
  - *date* and *time pickers* – allow the user to select a specific date or time

Text Fields

- Let's look at a GUI example that uses another type of control
- A *text field* allows the user to enter one line of input
- If the cursor is in the text field, the text field object generates an action event when the enter key is pressed

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

//************************************************************************
//  FahrenheitConverter.java       Java Foundations
//
//  Demonstrates the use of a TextField and a GridPane.
//************************************************************************

public class FahrenheitConverter extends Application
{
    //--------------------------------------------------------------------
    //  Launches the temperature converter application.
    //--------------------------------------------------------------------
    public void start(Stage primaryStage)
    {
        Scene scene = new Scene(new FahrenheitPane(), 300, 150);

        primaryStage.setTitle("Fahrenheit Converter");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

```
import javafx.appl
import javafx.scen
import javafx.stag

//***************                                   **************
//  FahrenheitConv
//
//  Demonstrates t
//***************                                   **************

public class FahrenheitConverter extends Application
{
    //--------------------------------------------------------------------
    //  Launches the temperature converter application.
    //--------------------------------------------------------------------
    public void start(Stage primaryStage)
    {
        Scene scene = new Scene(new FahrenheitPane(), 300, 150);

        primaryStage.setTitle("Fahrenheit Converter");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```



Fahrenheit Converter

Fahrenheit: 75

Celsius: 23

Text Fields

- The details of the user interface are set up in a separate class that extends `GridPane`
- `GridPane` is a JavaFX layout pane that displays nodes in a rectangular grid
- The GUI elements are set up in the constructor of `FahrenheitPane`
- The event handler method is also defined in `FahrenheitPane`

```java
import javafx.event.ActionEvent;
import javafx.geometry.HPos;
import javafx.geometry.Pos;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.text.Font;

//************************************************************************
//  FahrenheitPane.java       Java Foundations
//
//  Demonstrates the use of a TextField and a GridPane.
//************************************************************************

public class FahrenheitPane extends GridPane
{
    private Label result;
    private TextField fahrenheit;
```

continue

continue

```
   //--------------------------------------------------------------------
   //  Sets up a GUI containing a labeled text field for converting
   //  temperatures in Fahrenheit to Celsius.
   //--------------------------------------------------------------------
   public FahrenheitPane()
   {
      Font font = new Font(18);

      Label inputLabel = new Label("Fahrenheit:");
      inputLabel.setFont(font);
      GridPane.setHalignment(inputLabel, HPos.RIGHT);

      Label outputLabel = new Label("Celsius:");
      outputLabel.setFont(font);
      GridPane.setHalignment(outputLabel, HPos.RIGHT);

      result = new Label("---");
      result.setFont(font);
      GridPane.setHalignment(result, HPos.CENTER);
```

continue

continue

```
        fahrenheit = new TextField();
        fahrenheit.setFont(font);
        fahrenheit.setPrefWidth(50);
        fahrenheit.setAlignment(Pos.CENTER);
        fahrenheit.setOnAction(this::processReturn);

        setAlignment(Pos.CENTER);
        setHgap(20);
        setVgap(10);
        setStyle("-fx-background-color: yellow");

        add(inputLabel, 0, 0);
        add(fahrenheit, 1, 0);
        add(outputLabel, 0, 1);
        add(result, 1, 1);
    }
```

continue

continue

```
   //-------------------------------------------------------------------
   //  Computes and displays the converted temperature when the user
   //  presses the return key while in the text field.
   //-------------------------------------------------------------------
   public void processReturn(ActionEvent event)
   {
      int fahrenheitTemp = Integer.parseInt(fahrenheit.getText());
      int celsiusTemp = (fahrenheitTemp - 32) * 5 / 9;
      result.setText(celsiusTemp + "");
   }
}
```

Text Fields

- Through inheritance, a `FahrenheitPane` is a `GridPane` and inherits the `add` method
- The parameters to `add` specify the grid cell to which to add the node
- Row and column numbering in a grid pane start at 0
- When the user presses return, the event handler method is called, which converts the value and updates the text result

Layouts

- A *layout* is a way to arrange controls (nodes) within a scene (a scene graph, to be exact).
- In other words, a way to *place/arrange UI elements within a window*.
- JavaFX provides a number of *Panes*, which are layout container classes that hold children nodes (UI controls and/or other Panes).
- Each of the specific Panes arranges its children nodes according to specific *layout rules*.

Layouts

- Layout Panes include:
  - BorderPane – children are in top, left, right, bottom and center areas
  - VBox – children are in a single vertical row
  - HBox – children are in a single horizontal row
  - Stack – children are on top of one another just like in a stack

Layouts

- GridPane – children are in grid of rows and columns
- TilePane – children are in the form of uniformly sized tiles
- FlowPane – children are in the form of rows (or columns) that wrap around at the end of the predetermined pane boundary
- a few others

- Layouts *can be nested* to achieve a specific, more involved arrangement.