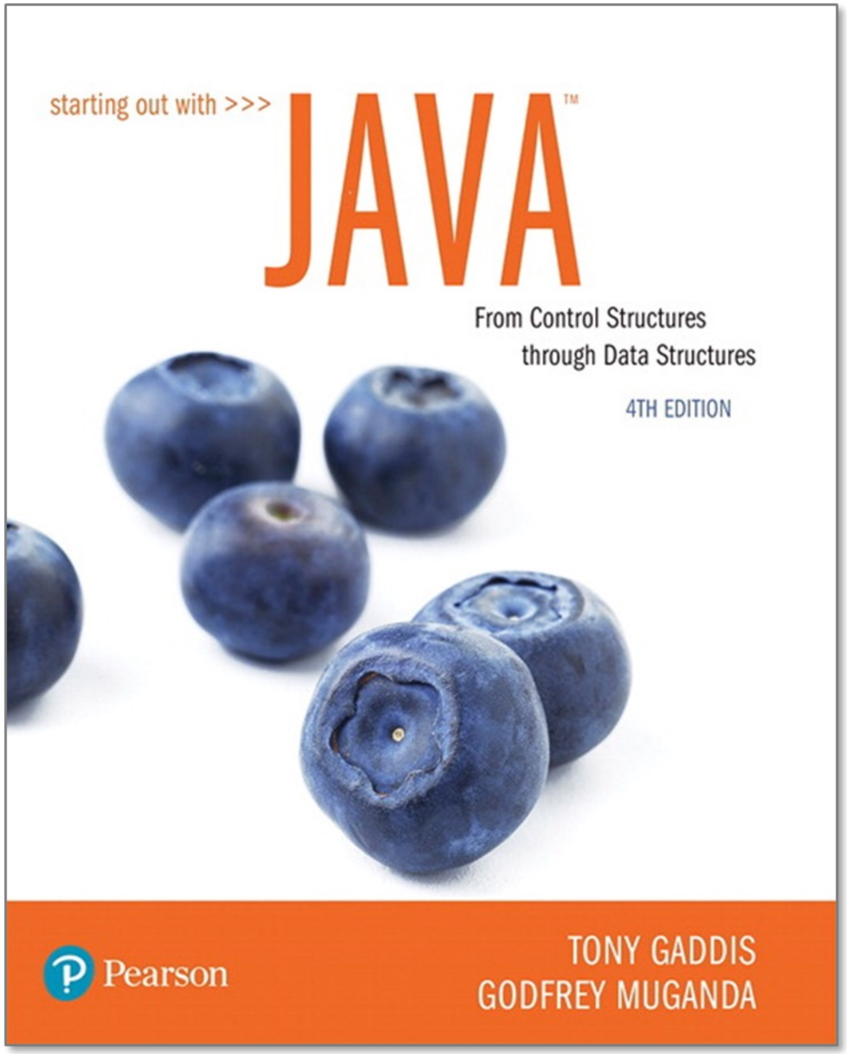
STARTING OUT WITH JAVA™

4th Edition


starting out with >>> JAVA™
From Control Structures through Data Structures
4TH EDITION
Pearson
TONY GADDIS
GODFREY MUGANDA

Chapter 12

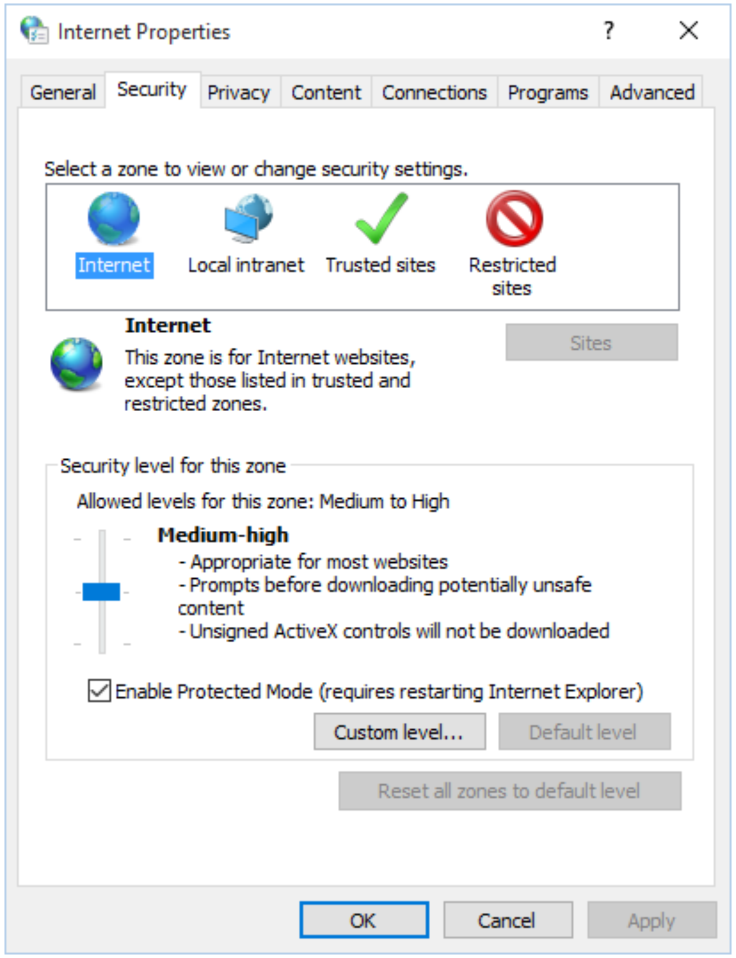
JavaFX: GUI Programming and Basic Controls

Topics

- Graphical User Interfaces
- Introduction to JavaFX
- Creating Scenes
- Displaying Images
- More About the `HBox`, `VBox`, and `GridPane` Layout Containers
- `Button` Controls and Events
- Reading Input with `TextField` Controls
- Using Lambda Expressions to Handle Events
- The `BorderPane` Layout Container
- The `ObserveableList` Interface

Graphical User Interfaces (1 of 3)

- Many Java application use a *graphical user interface* or *GUI* (pronounced “gooey”).
- A GUI is a graphical window or windows that provide interaction with the user.


Internet Properties

Select a zone to view or change security settings.

Internet

This zone is for Internet websites, except those listed in trusted and restricted zones.

Security level for this zone

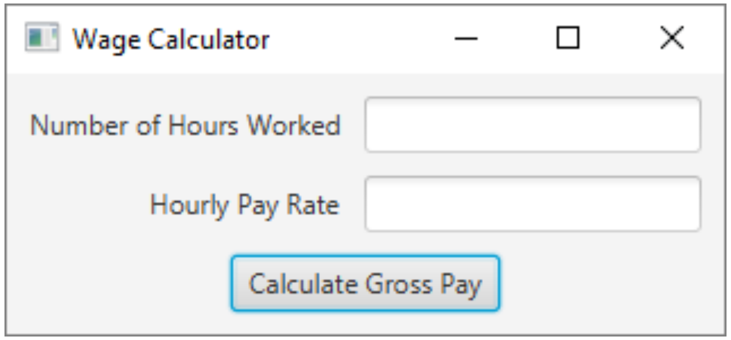
Allowed levels for this zone: Medium to High

Medium-high
- Appropriate for most websites
- Prompts before downloading potentially unsafe content
- Unsigned ActiveX controls will not be downloaded

☑ Enable Protected Mode (requires restarting Internet Explorer)

Graphical User Interfaces (2 of 3)

- A window in a GUI commonly consists of several *controls* that present data to the user and/or allow interaction with the application.
- Some of the common GUI *controls* are buttons, labels, text fields, check boxes, and radio buttons.



Graphical User Interfaces (3 of 3)

- Programs that operate in a GUI environment must be *event-driven*.
- An *event* is an action that takes place within a program, such as the clicking of a button.
- Part of writing a GUI application is creating event listeners.
- An *event listener* is a method that automatically executes when a specific event occurs.

Introduction to JavaFX (1 of 7)

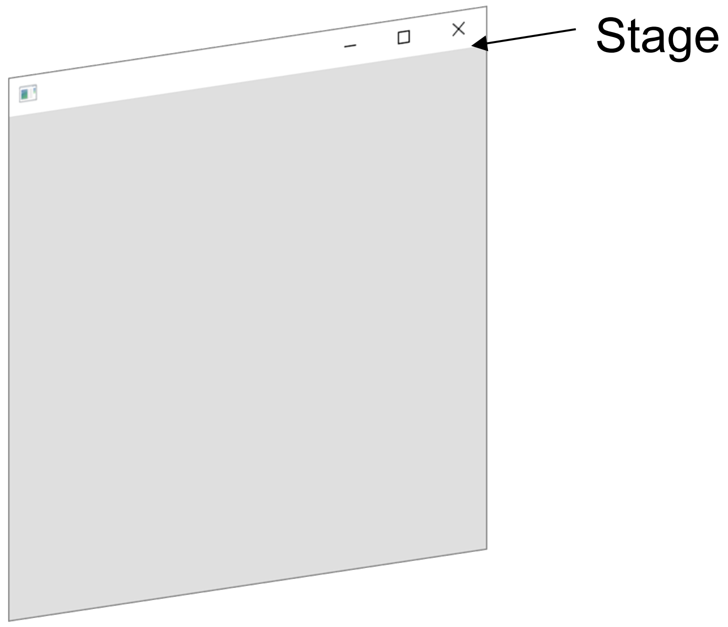
- *JavaFX* is a Java library for developing rich applications that employ graphics.
- You can use it to create:
  - GUI applications, as well as applications that display 2D and 3D graphics
  - standalone graphics applications that run on your local computer
  - applications that run from a remote server
  - applications that are embedded in a Web page

Introduction to JavaFX (2 of 7)

- JavaFX uses a theater metaphor to describe the structure of a GUI.
- A theater has a stage
- On the stage, a scene is performed by actors

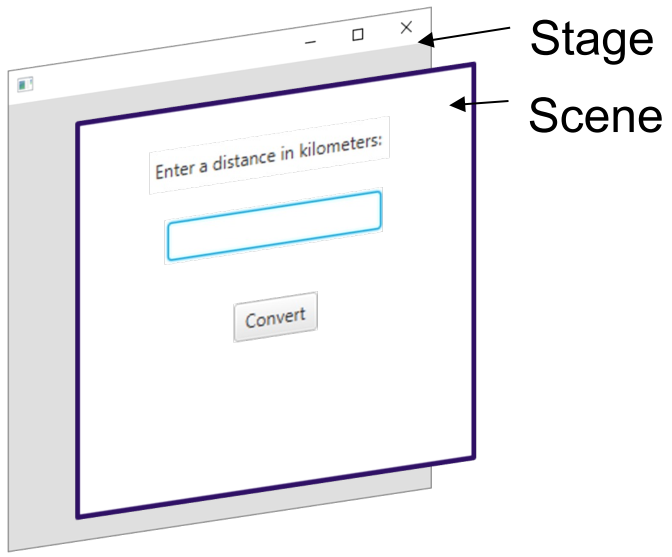
Introduction to JavaFX (3 of 7)

- In JavaFX, the stage is an empty window



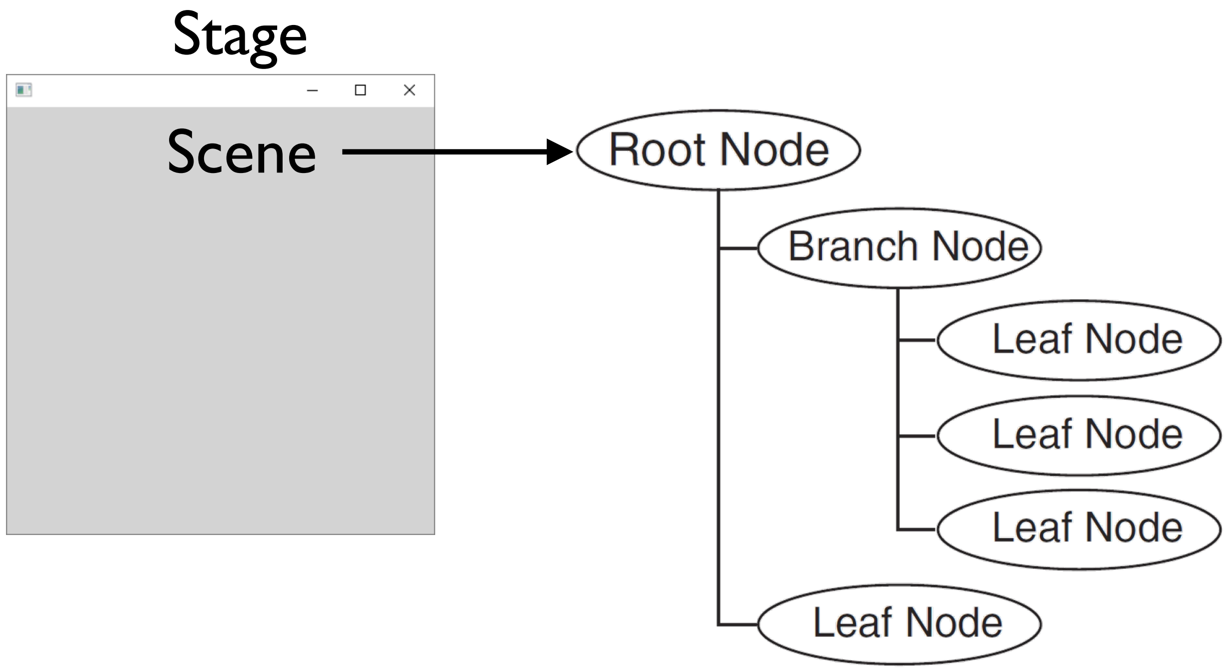
Introduction to JavaFX (4 of 7)

- The scene is a collection of GUI objects (controls) that are contained within the window.
- You can think of the GUI objects as the actors that make up the scene.



Introduction to JavaFX (5 of 7)

- In memory, the GUI objects in a scene are organized as nodes in a *scene graph*, which is a tree-like hierarchical data structure.



Introduction to JavaFX (6 of 7)

- The scene graph has three types of nodes:
  - **Root Node:** There is only one root node, which is the parent of all the other nodes in the scene graph.
  - **Branch Node:** A node that can have other nodes as children
  - **Leaf Node:** A node that cannot have children

Introduction to JavaFX (7 of 7)

- The `Application` Class
  - An abstract class that is the foundation of all JavaFX applications
  - JavaFX applications must extend the `Application` class
  - The `Application` class has an abstract method named `start`, which is the entry point for the application
  - Because the `start` method is abstract, you must override it

General Layout of a JavaFX Program

- Various `import` statements
- A class that extends the `Application` class
- A `start` method
- A `main` method

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
Other import statements…

public class ClassName extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Insert startup code here.
   }
}
```

$\}$ Necessary import statements

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
```
Other import statements…

Necessary import statements

```
public class ClassName extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Insert startup code here.
   }
}
```

A class that extends the `Application` class

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
```
Other import statements…

```
public class ClassName extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Insert startup code here.
   }
}
```

Necessary `import` statements

A static `main` method that calls the inherited `launch` method

A class that extends the `Application` class

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
Other import statements…

public class ClassName extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Insert startup code here.
   }
}
```

Necessary `import` statements

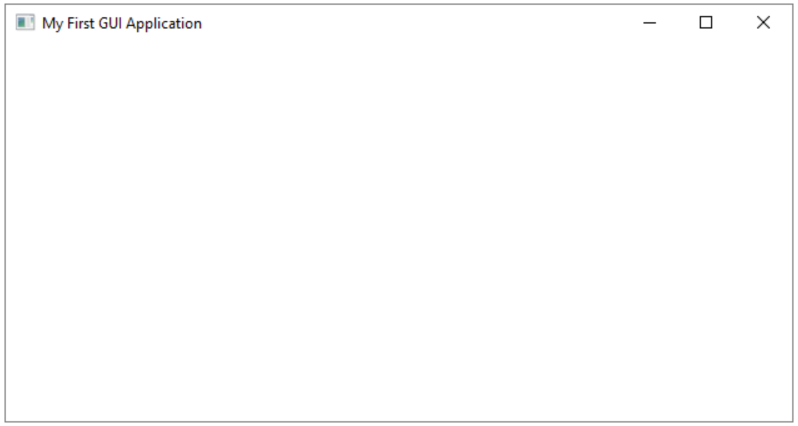
A static `main` method that calls the inherited `launch` method

A class that extends the `Application` class

A `start` method that accepts a Stage argument. This method is called by the inherited `launch` method.

MyFirstGUI.java

```
import javafx.application.Application;
import javafx.stage.Stage;

/**
 * A simple JavaFX GUI application
 */

public class MyFirstGUI extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Set the stage title.
      primaryStage.setTitle("My First GUI Application");

      // Show the window.
      primaryStage.show();
   }
}
```



Creating Controls (1 of 2)

- Process for creating a control:
  - Import the class for the control from the necessary `javafx` package. Example:

    ```
    import javafx.scene.control.Label;
    ```

  - Instantiate the class, calling the desired constructor. Example:

    ```
    Label messageLabel = new Label("Hello World");
    ```

Creating Controls (2 of 2)

- Another example: Creating a Button
  - Import the `Button` class from the necessary `javafx` package:

    ```
    import javafx.scene.control.Button;
    ```

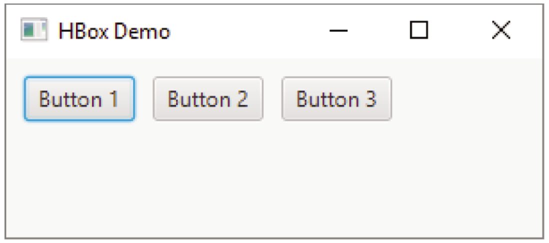
  - Instantiate the class, calling the desired constructor:

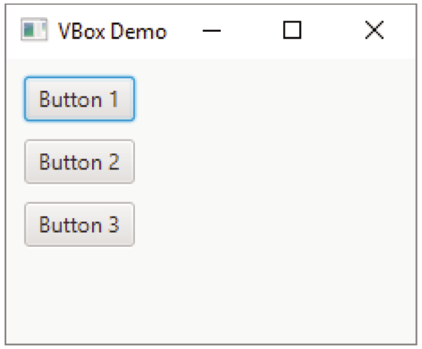
    ```
    Button mybutton = new Button("Click Me");
    ```

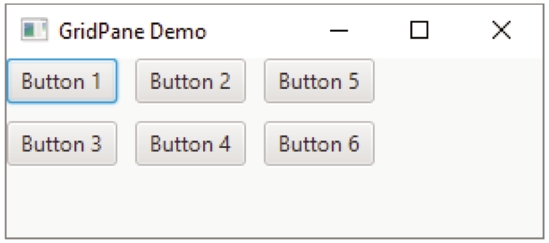
Layout Containers (1 of 2)

- You use layout containers to arrange the positions of controls on the screen.
- JavaFX provides many layout containers.
- We will start with these:
  - `HBox`: Arranges controls in a single horizontal row.
  - `VBox`: Arranges controls in a single vertical row.
  - `GridPane`: Arranges controls in a grid with rows and columns.

Layout Containers (2 of 2)



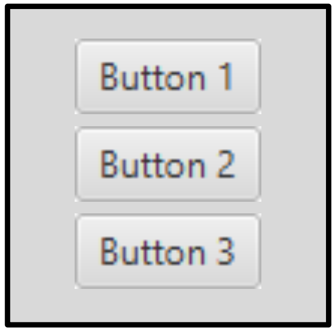




The layout container classes are in the `javafx.scene.layout` package.

Adding Controls to a Layout Container

VBox



```
Button b1 = new Button("Button 1");
Button b2 = new Button("Button 2");
Button b3 = new Button("Button 3");

VBox vbox = new VBox(b1, b2, b3);
```

Creating a Scene (1 of 2)

- To create a scene, you instantiate the `Scene` class (in the `javafx.scene` package)
- Then, you add your root node to the scene

Creating a Scene (2 of 2)

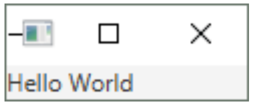
```
// Create a Label control.
Label messageLabel = new Label("Hello World");

// Create an HBox container and add the Label.
HBox hbox = new HBox(messageLabel);

// Create a Scene and add the HBox as the root node.
Scene scene = new Scene(hbox);
```

Adding the Scene to the Stage

- Once you've created a `Scene` object, you add it to the application's stage.
- The stage is an instance of the `Stage` class (from the `javafx.stage` package)
- You do not have to instantiate the `Stage` class, however. It is created automatically, and passed as an argument to the `start` method.

```
@Override
public void start(Stage primaryStage)
{

}
```

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

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

   @Override
   public void start(Stage primaryStage)
   {
      Label messageLabel = new Label("Hello World"); ← Make a Label control
      VBox vbox = new VBox(messageLabel); ← Put the Label in a VBox
      Scene scene = new Scene(vbox); ← Make the VBox the root node in the scene
      primaryStage.setScene(scene); ← Set the scene to the stage
      primaryStage.show(); ← Show the stage (display it)
   }
}
```

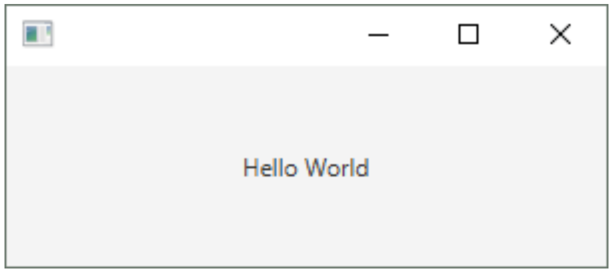


```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.geometry.Pos;

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

   @Override
   public void start(Stage primaryStage)
   {
      Label messageLabel = new Label("Hello World");
      VBox vbox = new VBox(messageLabel);
      vbox.setAlignment(Pos.CENTER);
      Scene scene = new Scene(vbox , 300, 100);
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

Center-align the VBox

Width

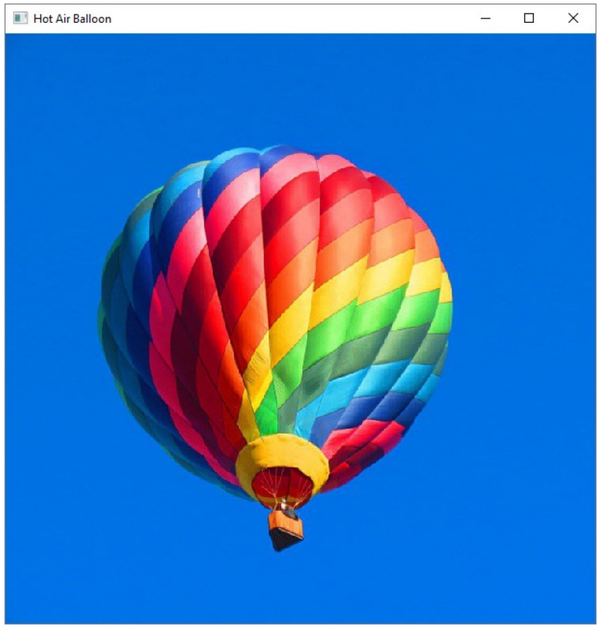
Height


Hello World

Displaying Images

- You need two JavaFX classes:
  - The `Image` class, from the `javafx.scene.image` package
    - Use this class to load an image into memory
  - The `ImageView` class, also from the `javafx.scene.image` package
    - Use this class to create a node that displays the image

```
@Override
public void start(Stage primaryStage)
{
   // Create an Image object.
   Image image = new Image("file:HotAirBalloon.jpg");

   // Create an ImageView object.
   ImageView imageView = new ImageView(image);

   // Put the ImageView in an HBox.
   HBox hbox = new HBox(imageView);

   // Create a Scene with the HBox as its root node.
   Scene scene = new Scene(hbox);

   // Add the Scene to the Stage.
   primaryStage.setScene(scene);

   // Set the stage title.
   primaryStage.setTitle("Hot Air Balloon");

   // Show the window.
   primaryStage.show();
}
```



More About HBox and VBox Containers (1 of 2)

- To add spacing between the items in an HBox or VBox:

```
HBox hbox = new HBox(10, label1, label2, label3);
```

Spacing

```
VBox vbox = new VBox(20, button1, button2, button3);
```

Spacing

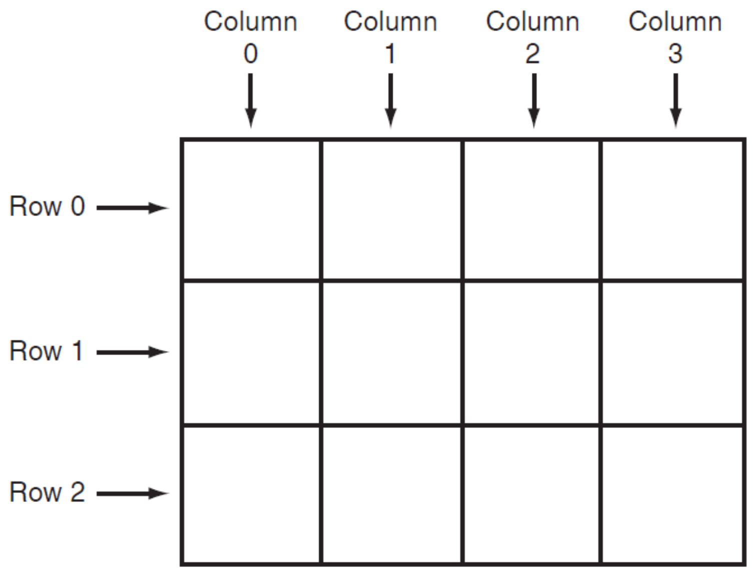
More About HBox and VBox Containers (2 of 2)

- *Padding* is space that appears around the inside edge of a container.
- The `HBox` and `VBox` containers have a `setPadding` method.
- You pass an `Insets` object as an argument to the `setPadding` method.
- The `Insets` object specifies the number of pixels of padding.
- The `Insets` class is in the `javafx.geometry` package.

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

The GridPane Layout Container (1 of 5)

- Arranges its contents in a grid with columns and rows.
- The columns and rows are identified by indexes.



The `GridPane` Layout Container (2 of 5)

- The `GridPane` class is in the `javafx.scene.layout` package.
- First, you instantiate the `GridPane` class, using the no-arg constructor:

```
GridPane gridpane = new GridPane();
```

- Then, you add controls to the container using the add method:

```
gridPaneObject.add(control, column, row);
```

The GridPane Layout Container (3 of 5)

```
// Create some Label controls.
Label label1 = new Label("This is label1");
Label label2 = new Label("This is label2");
Label label3 = new Label("This is label3");
Label label4 = new Label("This is label4");

// Create a GridPane.
GridPane gridpane = new GridPane();

// Add the Labels to the GridPane.
gridpane.add(label1, 0, 0);  // Column 0, Row 0
gridpane.add(label2, 1, 0);  // Column 1, Row 0
gridpane.add(label3, 0, 1);  // Column 0, Row 1
gridpane.add(label4, 1, 1);  // Column 1, Row 1
```

The GridPane Layout Container (4 of 5)

- By default, there is no space between the rows and columns in a GridPane.
- To add horizontal spacing between the columns in a GridPane, call the container's setHgap method.
- To add vertical spacing between the rows in a GridPane, call the container's setVgap method.

```
GridPane gridpane = new GridPane();
gridpane.setHgap(10);
gridpane.setVgap(10);
```

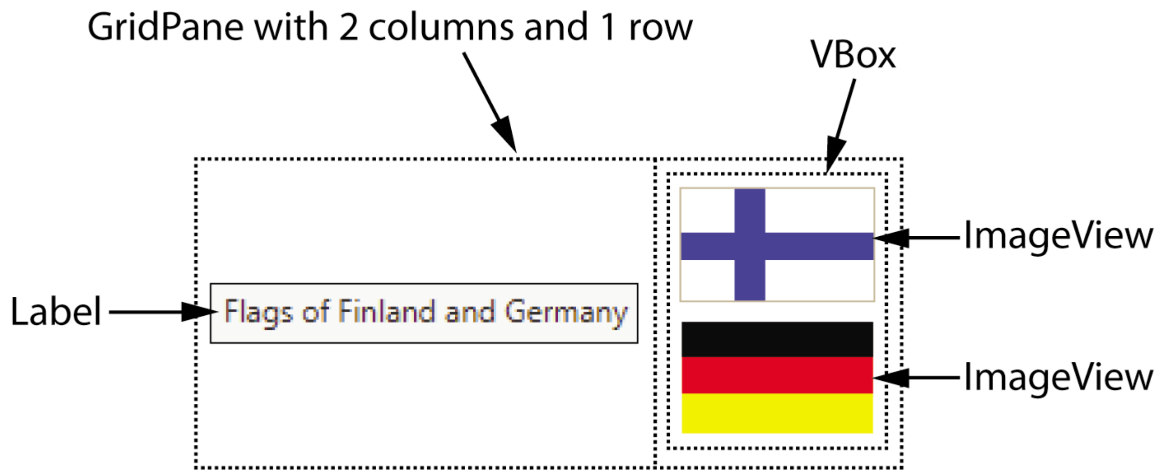
The `GridPane` Layout Container (5 of 5)

- The `GridPane` container also has a `setPadding` method to set the padding around the container's inside edge:

```
GridPane gridpane = new GridPane();
gridpane.setPadding(new Insets(10));
```

Using Multiple Layout Containers

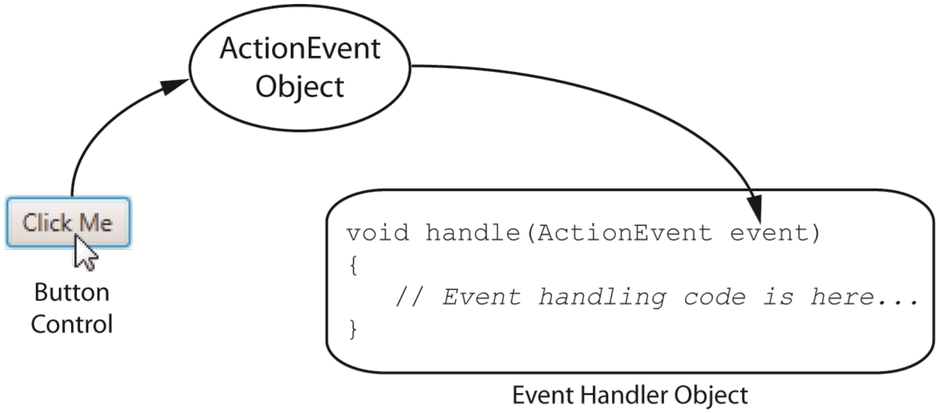
- To get the particular screen layout that you desire, you will sometimes have to nest layout containers.


GridPane with 2 columns and 1 row
VBox
ImageView
Label
Flags of Finland and Germany
ImageView

Handling Events (1 of 2)

- An *event* is an action that takes place within a program, such as the clicking of a button.
- When an event takes place, the control that is responsible for the event creates an *event object* in memory.
- The event object contains information about the event.
- The control that generated the event object is know as the *event source*.
- It is possible that the event source is connected to one or more event listeners.

Handling Events (2 of 2)



```
void handle(ActionEvent event)
{
   // Event handling code is here...
}
```

Event Objects

- Event objects are instances of the `Event` class (from the `javafx.event` package), or one of its subclasses.
- For example, a `Button` click generates an `ActionEvent` object. `ActionEvent` is a subclass of the `Event` class.

Event Handlers (1 of 2)

- Event handlers are objects.
- You write event handler classes that implement the `EventHandler` interface (from the `javafx.event` package).
- The `EventHandler` interface specifies a `void` method named `handle` that has a parameter of the `Event` class (or one of its subclasses).

Event Handlers (2 of 2)

```
class ButtonClickHandler implements EventHandler<ActionEvent>
{
    @Override
    void handle(ActionEvent event)
    {
        // Write event handling code here.
    }
}
```

Registering an Event Handler

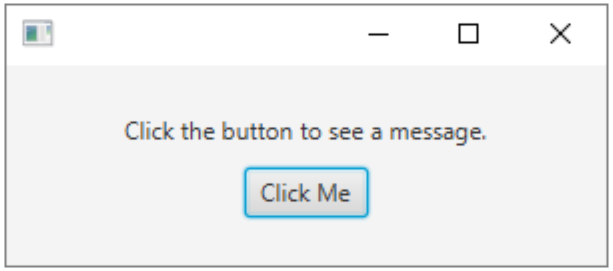
- The process of connecting an event handler object to a control is called *registering* the event handler.
- `Button` controls have a method named `setOnAction` that registers an event handler:

```
mybutton.setOnAction(new ButtonClickHandler());
```

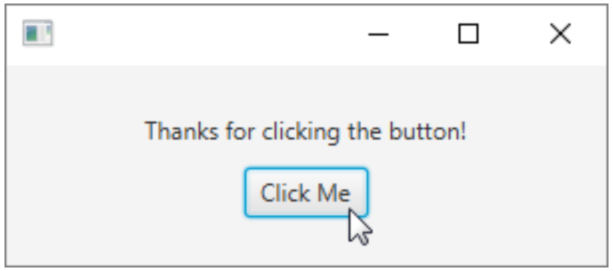
- When the user clicks the button, the event handler object's `handle` method will be executed.

Event Handling Example

- Let's look at an application that initially displays this screen:


Click the button to see a message.

- When the user clicks the button, the screen changes to:


Thanks for clicking the button!

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.geometry.Pos;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;

public class EventDemo extends Application
{
   private Label myLabel;

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

Continued…

```
      @Override
      public void start(Stage primaryStage)
      {
         // Create a Label and a Button.
         myLabel = new Label("Click the button to see a message.");
         Button myButton = new Button("Click Me");

         // Register an event handler.
         myButton.setOnAction(new ButtonClickHandler());

         // Put the Label and Button in a VBox with 10 pixels of spacing.
         VBox vbox = new VBox(10, myLabel, myButton);
         vbox.setAlignment(Pos.CENTER);

         // Create a Scene and display it.
         Scene scene = new Scene(vbox, 300, 100);
         primaryStage.setScene(scene);
         primaryStage.show();
      }

      class ButtonClickHandler implements EventHandler<ActionEvent>
      {
         @Override
         public void handle(ActionEvent event)
         {
            myLabel.setText("Thanks for clicking the button!");
         }
     }
   }
```

Reading Input with `TextField` Controls (1 of 2)

- At runtime, the user can type text into a `TextField` control.
- In the program, you can retrieve the text that the user entered.
- The `TextField` class is in the `javafx.scene.control` package.
- To create an empty `TextField`:

```
   TextField myTextField = new TextField();
```

Reading Input with `TextField` Controls (2 of 2)

- To retrieve the text that the user has typed into a `TextField` control, call the control's `getText` method.
- The method returns the value that has been entered, as a `String`.
- Example:

```
String input;
input = myTextField.getText();
```

- See `KiloConverter.java` in your textbook

Anonymous Inner Classes as Event Handlers

- When an event handler class is instantiated only once, you can write it as an anonymous inner class.

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.geometry.Pos;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;

public class EventDemo2 extends Application
{
   private Label myLabel;

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

Continued…

```
@Override
   public void start(Stage primaryStage)
   {
      // Create a Label and a Button.
      myLabel = new Label("Click the button to see a message.");
      Button myButton = new Button("Click Me");

      // Register an event handler.
      myButton.setOnAction(new EventHandler<ActionEvent>()
      {
         @Override
         public void handle(ActionEvent event)
         {
            myLabel.setText("Thanks for clicking the button!");
         }
      });

      // Put the Label and Button in a VBox with 10 pixels of spacing.
      VBox vbox = new VBox(10, myLabel, myButton);
      vbox.setAlignment(Pos.CENTER);

      // Create a Scene and display it.
      Scene scene = new Scene(vbox, 300, 100);
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

Lambda Expressions as Event Handlers

- Recall that a functional interface is an interface that has one, and only one, abstract method.
- The `EventHandler` interface has only one abstract method is a functional interface.
- Any time you are writing Java code to instantiate an anonymous class that implements a functional interface, you should consider using a lambda expression instead.
- A lambda expression is more concise than the code for instantiating an anonymous class.

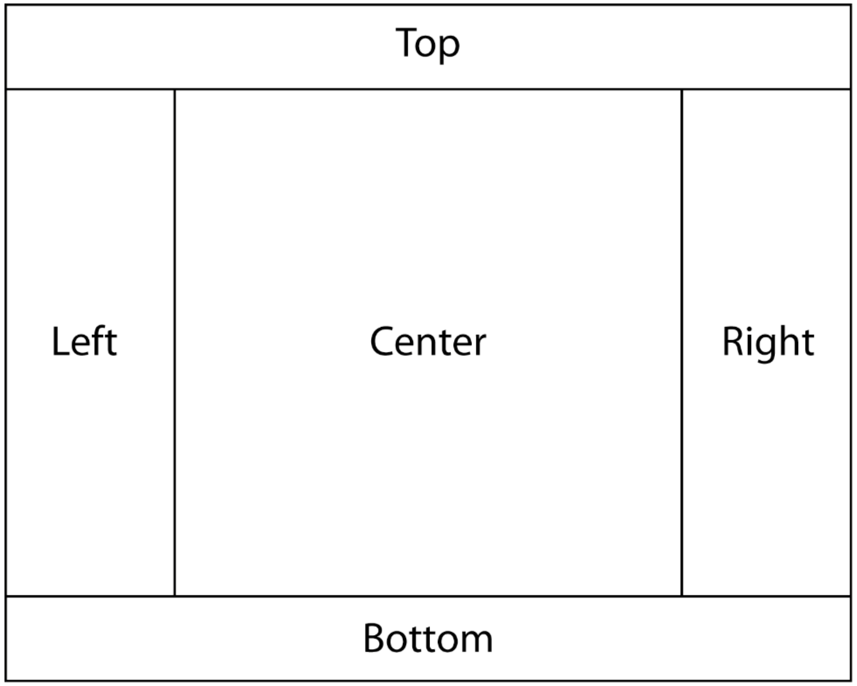
```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.geometry.Pos;

public class EventDemo3 extends Application
{
   private Label myLabel;

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

Continued…

```
@Override
   public void start(Stage primaryStage)
   {
      // Create a Label and a Button.
      myLabel = new Label("Click the button to see a message.");
      Button myButton = new Button("Click Me");

      // Register an event handler.
      myButton.setOnAction(e ->
      {
         myLabel.setText("Thanks for clicking the button!");
      });

      // Put the Label and Button in a VBox with 10 pixels of spacing.
      VBox vbox = new VBox(10, myLabel, myButton);
      vbox.setAlignment(Pos.CENTER);

      // Create a Scene and display it.
      Scene scene = new Scene(vbox, 300, 100);
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

The `BorderPane` Layout Container (1 of 4)

- The `BorderPane` layout container manages controls in five regions:



The `BorderPane` Layout Container (2 of 4)

- Only one object at a time may be placed into a `BorderPane` region.
- You do not usually put controls directly into a `BorderPane` region.
- Typically, you put controls into another type of layout container, then you put that container into one of the `BorderPane` regions.

The BorderPane Layout Container (3 of 4)

- The `BorderPane` class is in the `javafx.scene.layout` package.
- Summary of constructors:

Constructor	Description
BorderPane()	The no-arg constructor creates an empty BorderPane container.
BorderPane(*center*)	This constructor accepts one argument. The node that is passed as the argument is placed in the BorderPane's center region.
BorderPane(*center*, *top*, *right*, *bottom*, *left*)	This constructor accepts five nodes as arguments: one to place in each region.

The BorderPane Layout Container (4 of 4)

- The BorderPane class provides the following methods to add controls to specific regions:
  - setCenter
  - setTop
  - setBottom
  - setLeft
  - setRight
- See BorderPaneDemo1.java in your textbook.

The `ObservableList` Interface (1 of 5)

- Widely used in JavaFX
- Learning a few basic `ObservableList` operations gives you more control over the JavaFX containers with which you will be working.

The `ObservableList` Interface (2 of 5)

- A few `ObservableList` methods:

Method	Description
add(*item*)	Adds a single item to the list. (This method is inherited from the Collection interface.)
addAll(*item*...)	Adds one or more items to the list, specified by the variable argument list.
clear()	Removes all of the items from the list.
remove(*item*)	Removes the object specified by *item* from the list. (This method is inherited from the Collection interface.)
removeAll(*item*...)	Removes one or more items to the list, specified by the variable argument list.
setAll(*item*...)	Clears the current contents of the list and adds all of the items specified by the variable argument list.
size()	Returns the number of items in the list. (This method is inherited from the Collection interface.)

The `ObservableList` Interface (3 of 5)

- For example, layout containers keep their children nodes in an `ObservableList`.
- All layout containers have a method named `getChildren()` that returns their `ObservableList` of nodes.

The `ObservableList` Interface (4 of 5)

- Example: creating an empty `HBox`, then using the `ObservableList`'s `addAll` method to add nodes to the `HBox`:

```
// Create three Label controls.
Label label1 = new Label("This is label1.");
Label label2 = new Label("This is label2.");
Label label3 = new Label("This is label3.");

// Create an empty HBox container.
HBox hbox = new HBox();

// Add the Label controls to the HBox.
hbox.getChildren().addAll(label1, label2, label3);
```

The `ObservableList` Interface (5 of 5)

- Example: removing `label1` from the `HBox`:

```
hbox.getChildren().remove(label1);
```

Copyright