ALGONQUIN COLLEGE

MODULE 04: INTRODUCTION TO JAVAFX

Professor : Dave Houtman

Office: T323

Office Hrs: Wednesday 14:15 – 15:30

Wednesday (after lecture*)

* confirm beforehand

Email: houtmad@algonquincollege.com

4.0 Java and GUIs – A Brief History

Back when Java was first introduced (in 1995), it included a small graphics library called **AWT**, the *Abstract Windows Toolkit*. This allowed developers to build basic graphical operations.

Complicating matters somewhat, IBM released its own GUI library, the *Standard Widget Toolkit* (**SWT**), around the same time. This continues to be supported by Eclipse.

With the J2SE release of 1998, Sun Microsystems officially replaced AWT with **Swing**, which gave developers a more powerful set of graphical tools. It quickly took over from AWT as the preferred API of choice for GUI developers.

In 2008, Sun Microsystems announced that Swing would be replaced by **JavaFX**. At present, while Swing and JavaFX continue to coexist, Swing will eventually be phased out, leaving JavaFX as (presumably) the dominant software platform for GUI development in the Java world...

4.0 Java and GUIs – A Brief History

Of note to developers: while AWT, Swing, and JavaFX can be mixed, this is inadvisable, and should be avoided, unless you know exactly what you're doing.

Students seem to be particularly susceptible to including AWT or Swing components in their JavaFX code, presumably because they saw it recommended in a 'quickfix' popup. Be advised that accepting such a fix to solve a JavaFX problem is usually a recipe for disaster, one which will cost you far more time to fix than you would have spent if you'd resolved the initial problem correctly. Therefore, *under no circumstances* should you include AWT or Swing objects in the code you submit with your assignments. And avoid quickfixes altogether unless you know *exactly what you're doing*, since they usually provide bad advice.

Additionally, using a layout builder to design your code will also be considered a serious breach of protocol for this course, one which will lose you marks.

4.1 Application

An ***API***, or ***Application Program Interface***, is defined (somewhat broadly) as:

"a set of routines, protocols, and tools for building software applications"

(www.webopedia.com/TERM/A/**API**.html, downloaded Sept. 18, 2016).

The JavaFX API is contained in the various libraries needed to build a JavaFX application. Therefore, each JavaFX application will typically begin by importing the libraries that are needed by the code, including the following common components:

```
import javafx.application.Application;
import javafx.application.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
```

Liang 14.3

4.1 Application

To build your JavaFX program, you begin by loading a new java file in a package and then extending `Application`, which is association with the `javafx.application` library:

```
import javafx.application.Application;

public class MyJavaFXApp extends Application {
    …
}
```

As always, the name of the subclass of `Application` can be any valid identifier. For this example, we'll use `myJavaFXApp`.

4.1 Application

According to the JavaFX documentation*, we can determine that the `Application` class *must* contain at least the following four public methods (the exact details need not concern us):

```
public class Application extends Object{
    public static void launch(String[] args){…}
    public void init(){…};     // set up the window
    public abstract void start(Stage stage);
    public void stop(){…};     // stop execution
}
```

When we extend `Application`, we inherit these classes and all the code that comes with them, except, of course, for the `start()` method, which we must override in any subclass that extends `Application`.

*at https://docs.oracle.com/javase/8/javafx/api/javafx/ application/Application.html

4.1 Application

According to the documentation, `Application.launch()` is the starting point for JavaFX execution. As stated earlier, JavaFX applications do not, strictly speaking, require a `main()` method, since most IDEs, like Eclipse, will automatically begin JavaFX execution with the `launch` method, which, like `main()`, is a declared as `public static void`, and which takes an array of strings as its argument.

4.1 Application

However, some legacy IDE's don't 'understand' JavaFX. Therefore, it's considered good practice to *always* include a `main()` routine in your code, and call `Application.launch` inside `main()`. This ensures your JavaFX code will always execute, regardless of the IDE:

```
import javafx.application.Application;

public class MyJavaFXApp extends Application{

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

    ... //etc

}
```

1a. Older IDEs start here

1b. Newer IDEs run `launch` automatically

4.1 Application

Also, while Eclipse recognizes `launch` as the starting point for your JavaFX applications, it may have trouble recognizing *which* JavaFX application to run if you do not include a `main()` method in each program. Therefore, *always* include a `main()` method, or Eclipse may execute your *last* JavaFX program, rather than the current one.

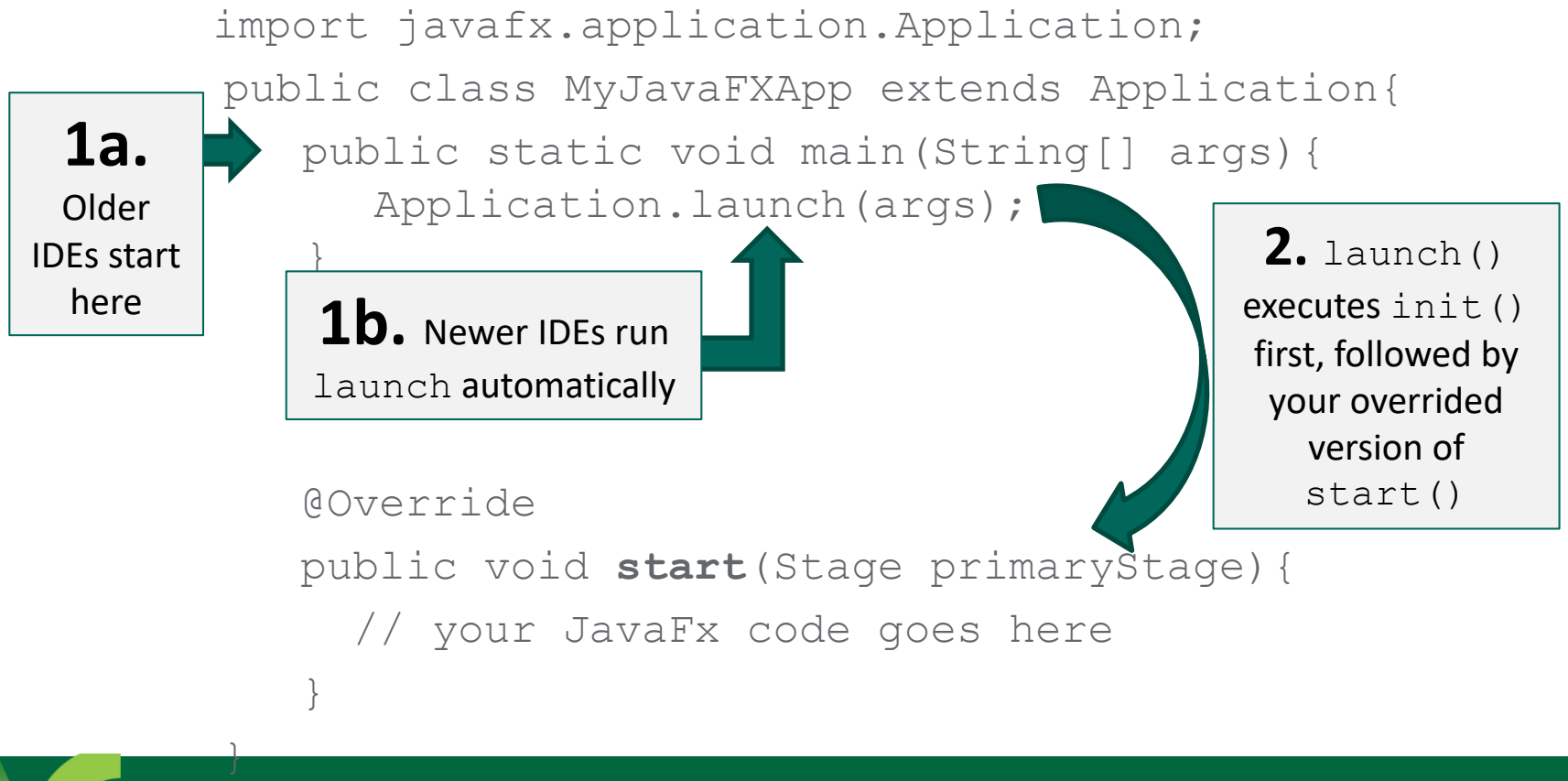
4.1 Application

When `launch()` begins, it performs the following operations:

1. It constructs an instance of the `Application` subclass
2. It calls the `init()` method
3. It calls the `start()` method.
4. It finishes when either
   a) the application calls `Platform.exit()`
   b) the last window has been closed
5. It calls the `stop()` method

4.1 Application

Your code must override the `start()` method, which was declared abstract in the `Application` superclass. `start()` is effectively the true starting point of for your JavaFX application, since at this point that your code controls GUI execution.



4.2 Stage

- Note that the `start()` method takes a `Stage` object as its parameter. `Application` supplies you with this object, called `primaryStage` in the documentation (although you can use any legitimate identifier you want…). The `Stage` object is your handle to the application window, which is the outer window that you will eventually fill with the graphics objects that form your GUI application. Therefore, your overridden `start()` method mostly deals with the 'big picture' features of your application window via the `primaryStage`. (The `Application` object itself has no visible manifestation.)

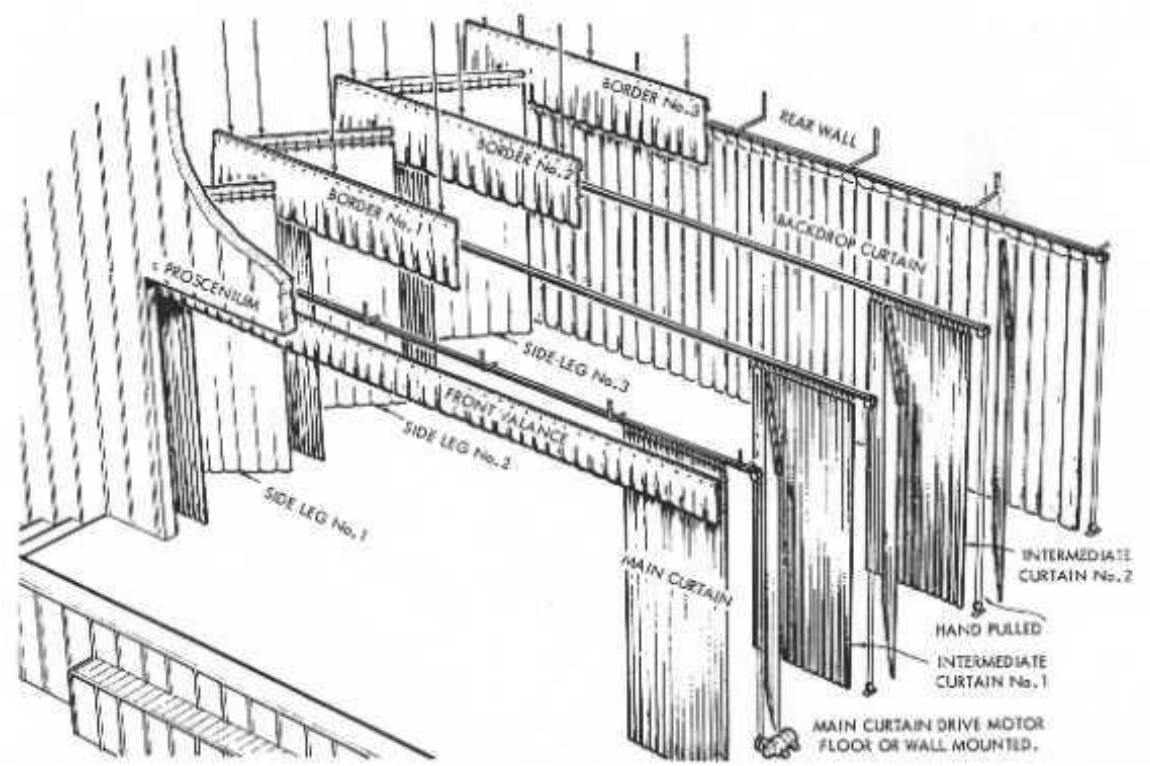
  So treat `start()` in much the same way as you normally treat `main()` for your console-based applications, i.e. `start()` acts as the 'control panel' that (mostly) instantiates other classes and calls on their methods (or it may call on static methods inside a class) to get the job done. As before, most of the real work gets done inside the instance methods.

4.2 Stage

- Since the `primaryStage` object consists of the window itself—including the outer frame, caption bar, title, and the minimize, maximize, and close icons (seen in the top right corner of every GUI application window)—it has relatively few methods of interest to us. But one method `is` crucial. Each `Stage` object is normally invisible on startup. In order to make a `Stage` visible, you must execute the `Stage`'s `show()` method. But this happens only after you've loaded up all the other objects inside the `Stage` object first…

4.2 Stage

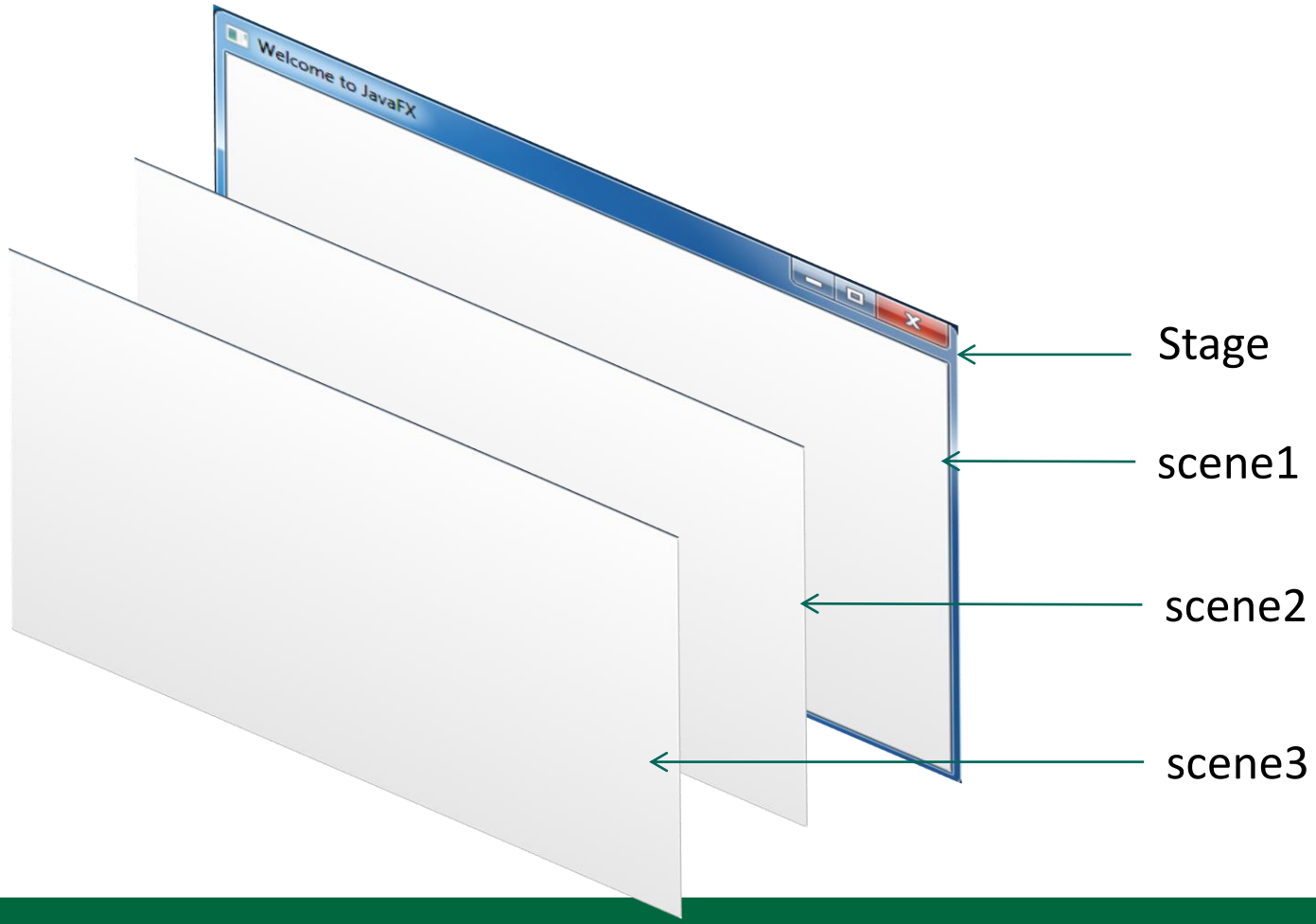
JavaFX uses a theatre metaphor for some of its terminology at this level. In theatre, a *stage* is the location for one or more *scenes*…



From: https://stagecraftsmath.wikispaces.com/, downloaded Sept. 18, 2016

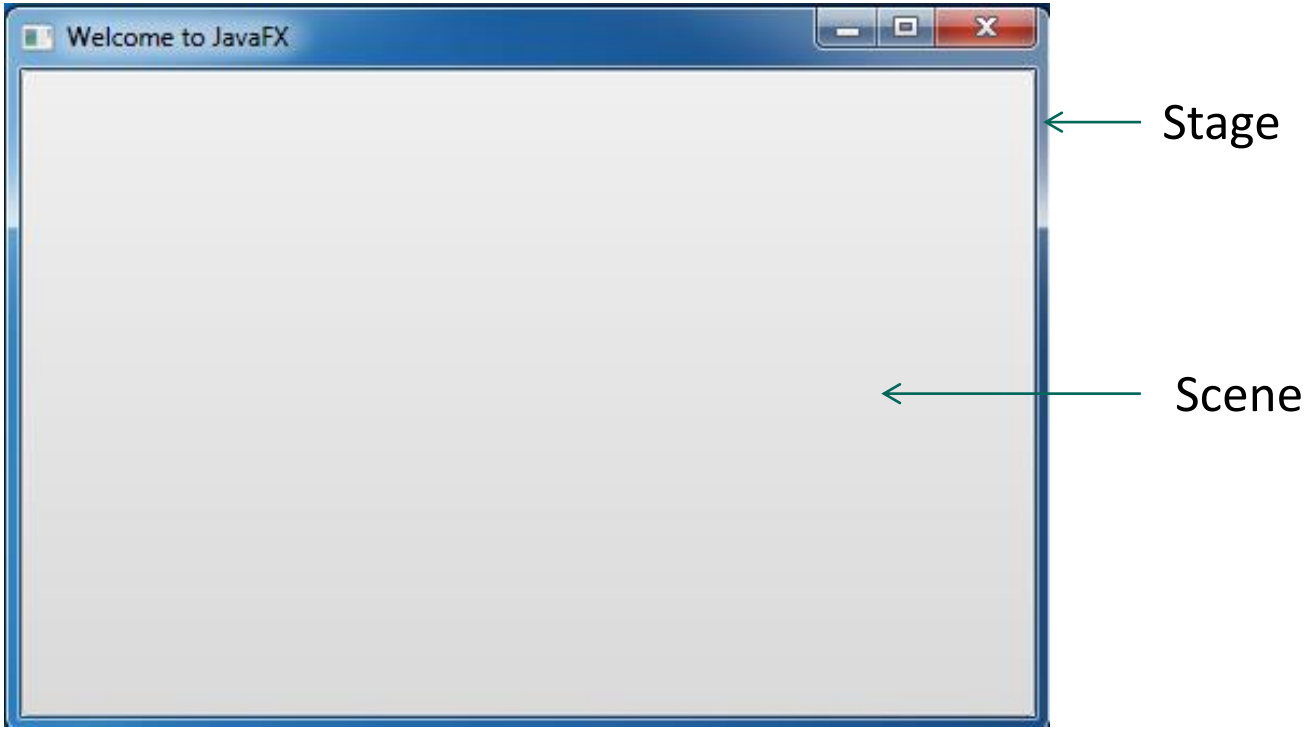
4.2 Stage

…so in JavaFX, you construct different presentations 'behind the scenes' and display them in succession by loading a `Stage` with one or more completed `Scene` objects in succession:



4.3 Scene

So JavaFX treats the window as a stage for graphical displays, and the scene as the container for each presentation. Thus every `Stage` must have at least *one* `Scene`. And this is typically done during the setup in the `start()` method.



4.3 Scene

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
...
public class MyJavaFXApp extends Application {
   @Override     //Override Application's start() method
   public void start (Stage primaryStage){
      ...
      Scene myNewScene = new Scene(…);
      primaryStage.setTitle("Welcome to JavaFX");
      primaryStage.setScene(myNewScene);
      primaryStage.show();
   }

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

4.3 Scene

However, the `Scene` object itself must first be populated, otherwise it's just a gray block of empty real estate inside the `Stage`. However, a `Scene` must first be loaded with a `Parent` object. This is done via the `Scene()` constructor, e.g.

```
Scene myNewScene = new Scene(someParentObject);
```

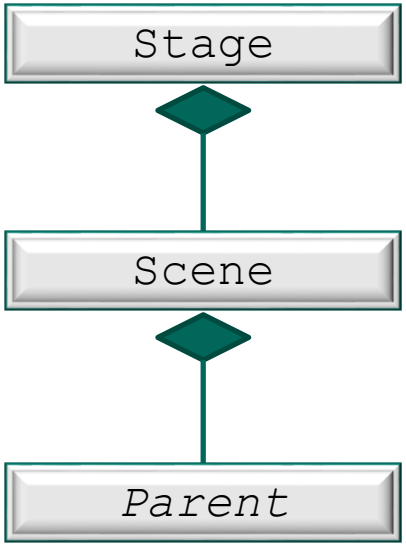
Or, since `Scene()` is an overloaded constructor, you can supply the dimensions of the `Scene` window:

```
Scene myNewScene = new Scene(someParentObject, 500, 300);
```

4.3 Scene

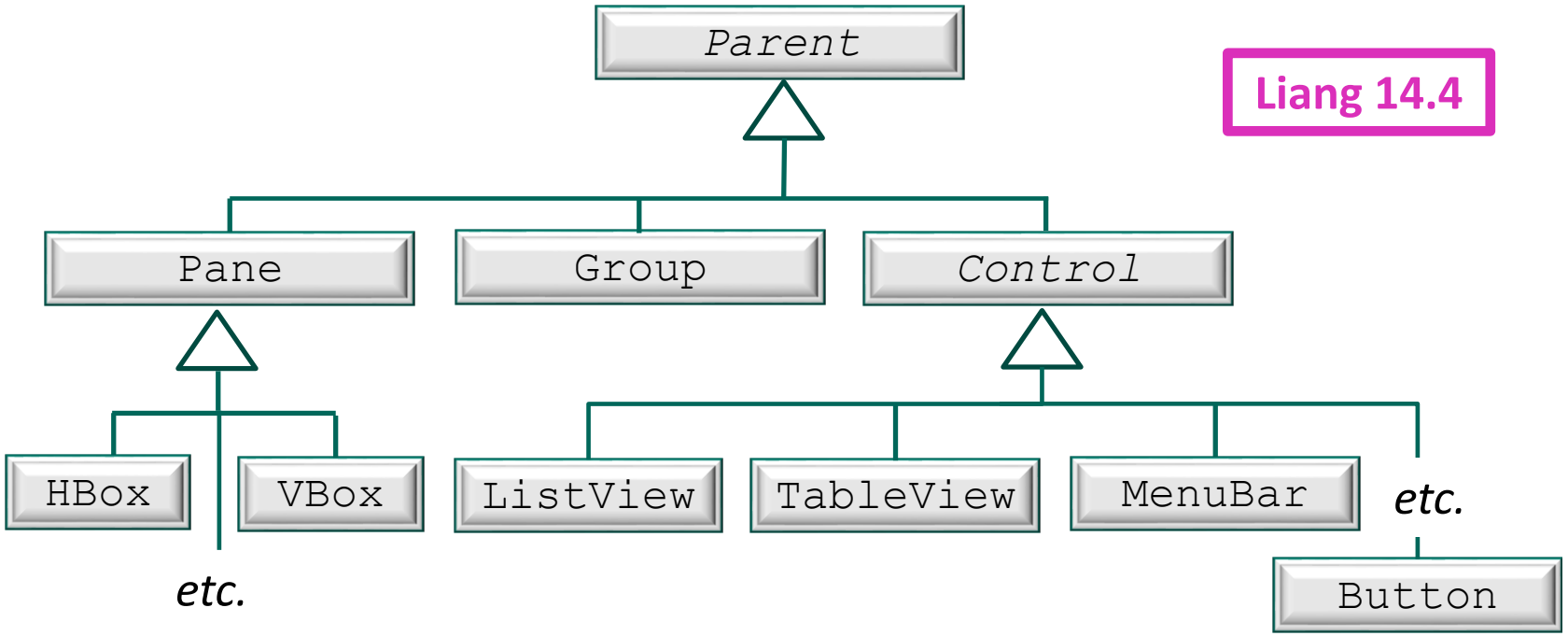
We'll get to `Parent` in a moment. For now, note that a `Stage` *has a* `Scene`, and a `Scene` *has a* `Parent`, as shown in the UML diagram at right above.

So before we can `show()` the `Stage` (the `primaryStage` object we were given by `Application`) we must first load the `Scene` with its `Parent` 'actors'.



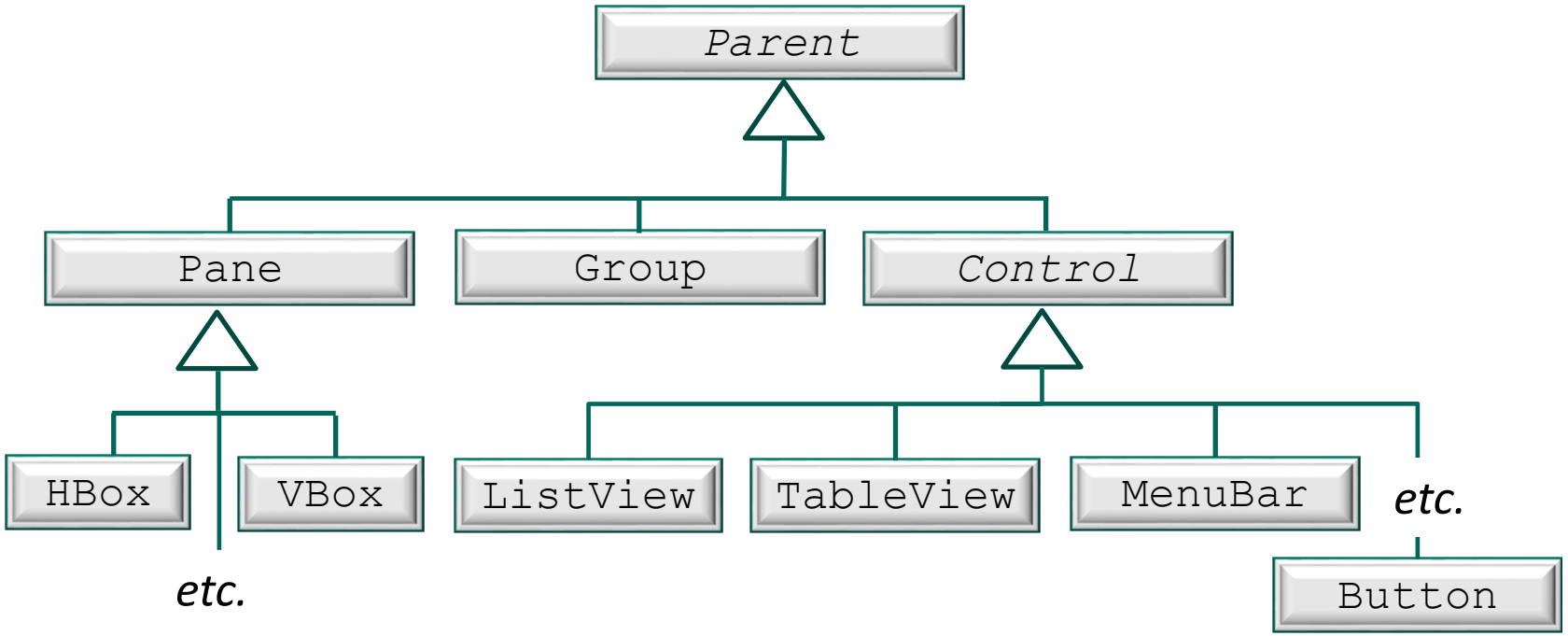
4.4 Parent

`Parent` itself is an abstract class—note the italic font in the UML diagram. So you can't load a `Parent` object directly (it's abstract!), you must load an instance of one of its *subclasses* into the `Scene()` constructor—*via polymorphism*. The `Parent` superclass essentially acts as an abstract container that (1) contains the minimum set of instance members needed to satisfy `Scene`'s requirements, and (2) acts as the 'parent' for many of the standard objects used in GUI programming.



4.4 Parent

From this diagram we see that `Button` is a concrete subclass of `Parent`. Or rather, it is a sub-sub-sub…class. The screenshot on the next page shows how distant a descendant `Button` is to `Parent`…

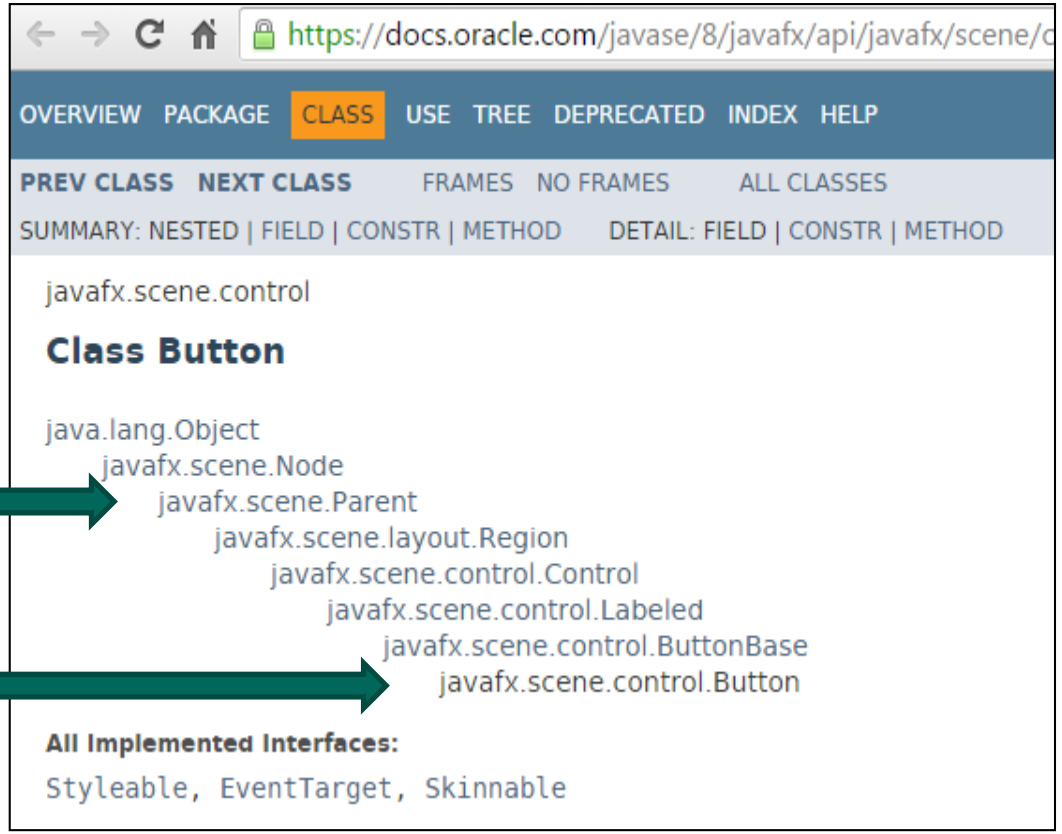


4.4 Parent

Despite the separation, a `Button` *is a* `Parent`. So we could load a new `Button` object directly into the `Scene`, thus satisfying the requirement that a scene must have a `Parent`.

`Parent` →

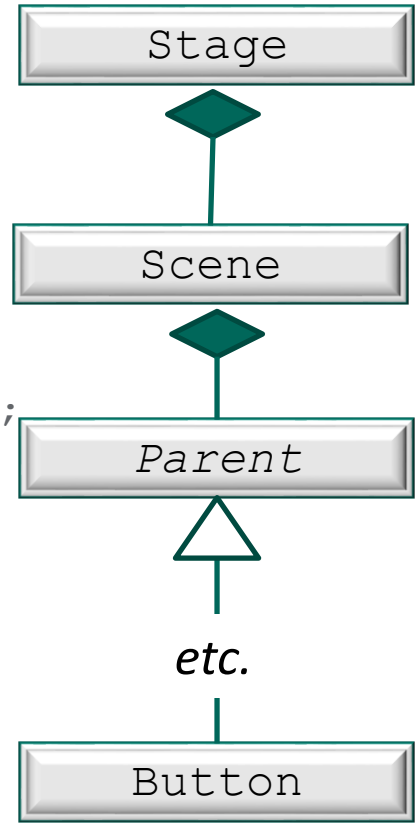
`Button` →



We can then load into the `primaryStage` object. This is done in the code on the next slide…

4.4 Parent

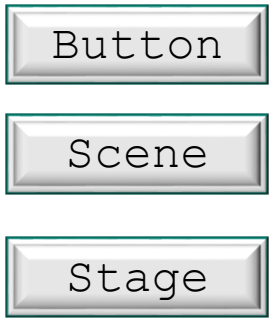
```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.control.Button;

public class MyJavaFXApp extends Application {
    @Override
    public void start (Stage primaryStage){
        Button btOK = new Button("OK");
        Scene scene = new Scene(btOK, 500, 300);
        primaryStage.setTitle("Welcome to JavaFX");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

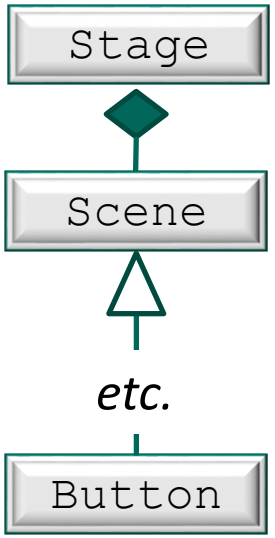


4.4 Parent

Notice that the UML diagram displays the dependency information from top to bottom, in the opposite order in which it is instantiated in the code. Why? When an object *has a* 'something', that something needs to be loaded before its owner can be used. This guarantees that the information at the end of the *has a* chain must be created first, in reverse order to the way it is displayed in the UML diagram.

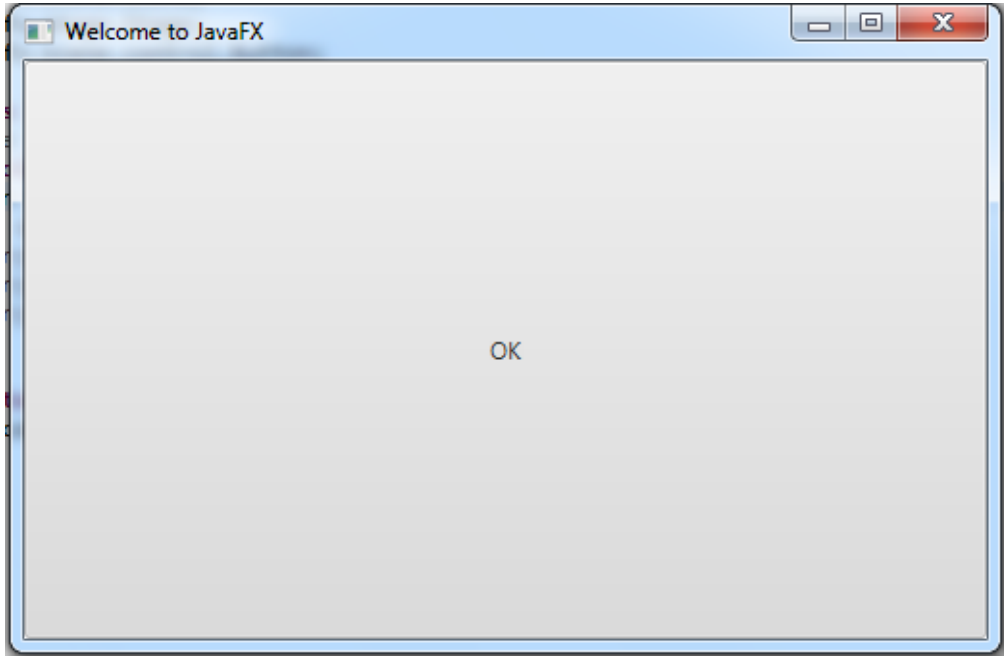
```
public class MyJavaFXApp extends Application{
    @Override
    public void start (Stage primaryStage){
        Button btOK = new Button("OK");
        Scene scene = new Scene(btOK…);
        primaryStage.setTitle(…);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```





4.4 Parent

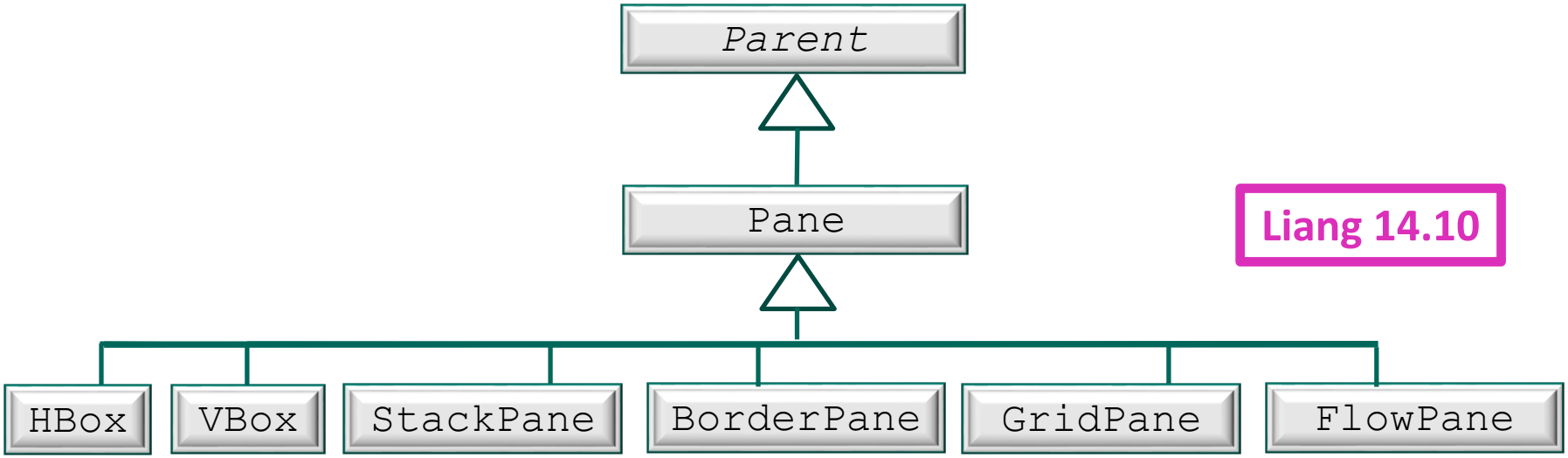
The output is:



Notice that, by default, the `Button` fills the entire `Scene`, and the `Scene` determines the size of the `Stage`, which is, in this case, 500 X 300 pixels…perhaps not what you expected!

4.5 Node

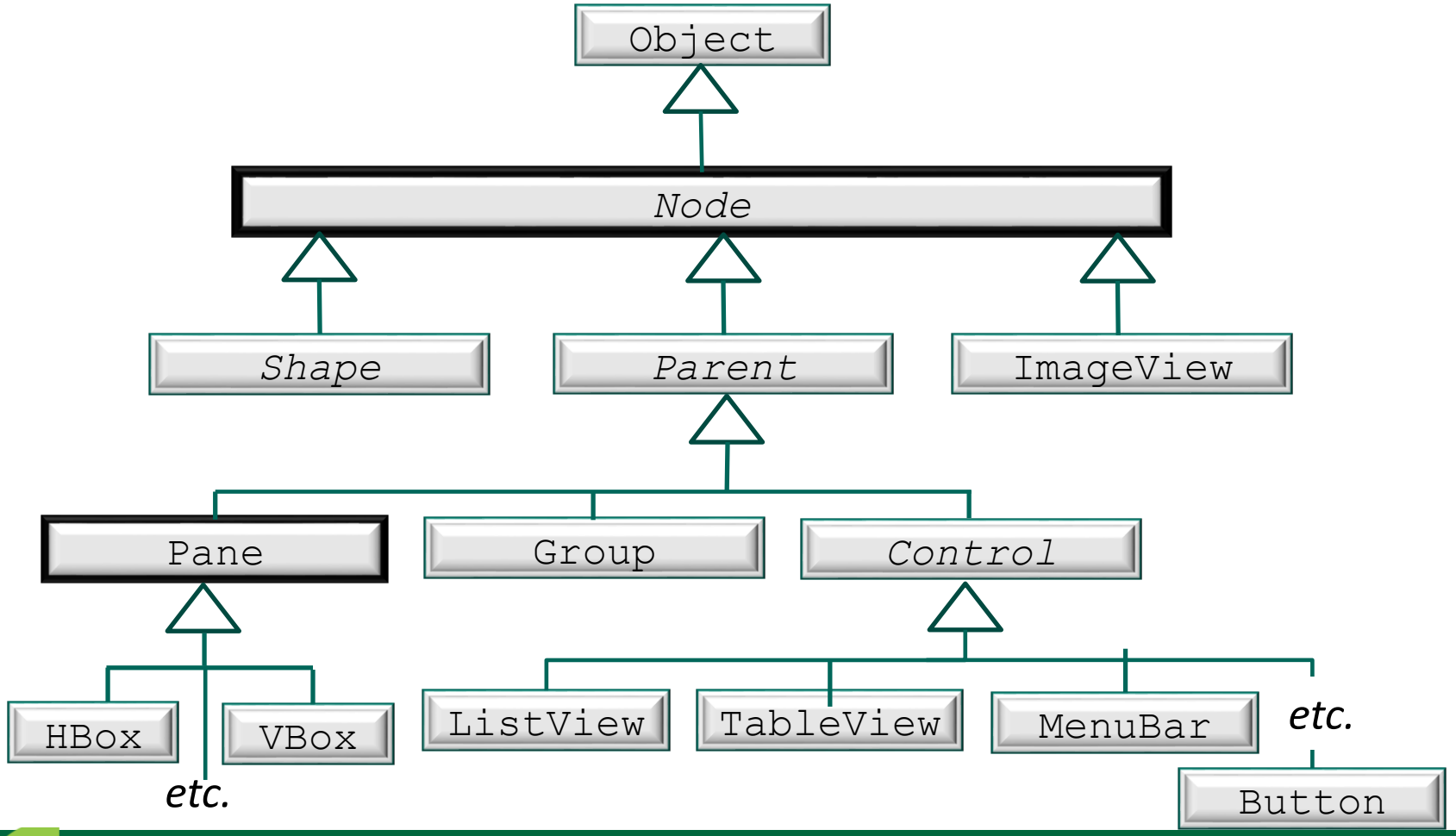
To limit the dimensions of a button so it doesn't fill the entire scene, you could change its height and width properties using the appropriate methods of the `Button` object itself. But the preferred JavaFX way is to use yet another object to structure the space inside the scene first, using an object called a `Pane`. Like a `Button` (and most of the other graphical controls), a `Pane` is a `Parent`. So a `Pane` can be loaded into a `Scene` just as well as a `Button`—but the `Pane` is built to control the workspace inside the `scene`, unlike the `Button`, whose dimensions normally conform to the defaults. (We'll use a `Pane` to structure the `scene's` space in just a few minutes.)



Liang 14.10

4.5 Node

In fact, Buttons, Panes, Menubars, etc—virtually any graphical member of a Scene—are subclasses of a much broader superclass called a Node:



4.5 Node

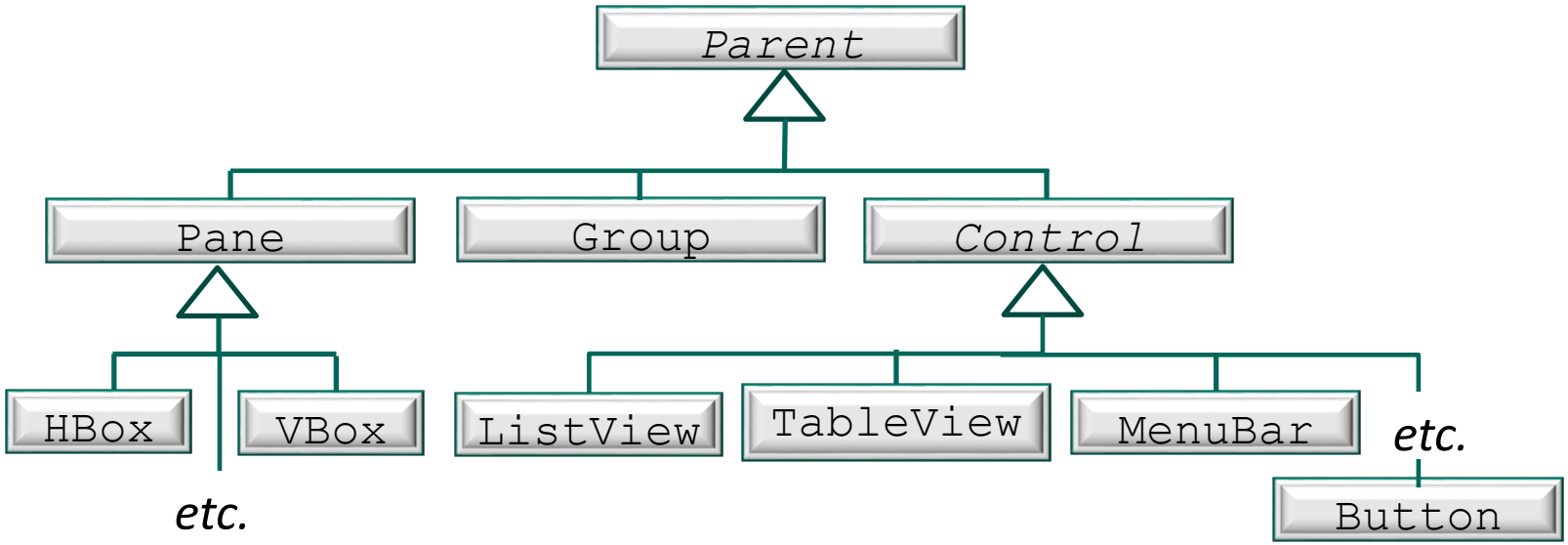
Node is a generic term for any graphical element in JavaFX, in much the same way that `GeometricObject` served as a general abstract container for shape objects in an earlier example. The `Node` class can also handle **events**, such as clicking on an object, or hovering over it, or resizing it—without actually knowing what kind of object it is responsible for—which is exactly what you expect from an abstract superclass.

A `Node` serves as the superclass for several immediate subclasses, of which only three are shown in the UML diagram in the previous slide:

1. The `Shape` class is the base class for several concrete classes, including `Circle`s, `Square`s, `Rectangle`s, `Arc`s, `Line`s, `Polygon`s, etc.
2. The `ImageView` class allows images to be loaded, resized, zoomed into, etc. This is a concrete class that can be instantiated without needing to be extended first.
3. `Parent` is an abstract subclass of `Node` that adds additional children to the `Node`, i.e. it deals with adding new graphical members.

4.6 The JavaFX API

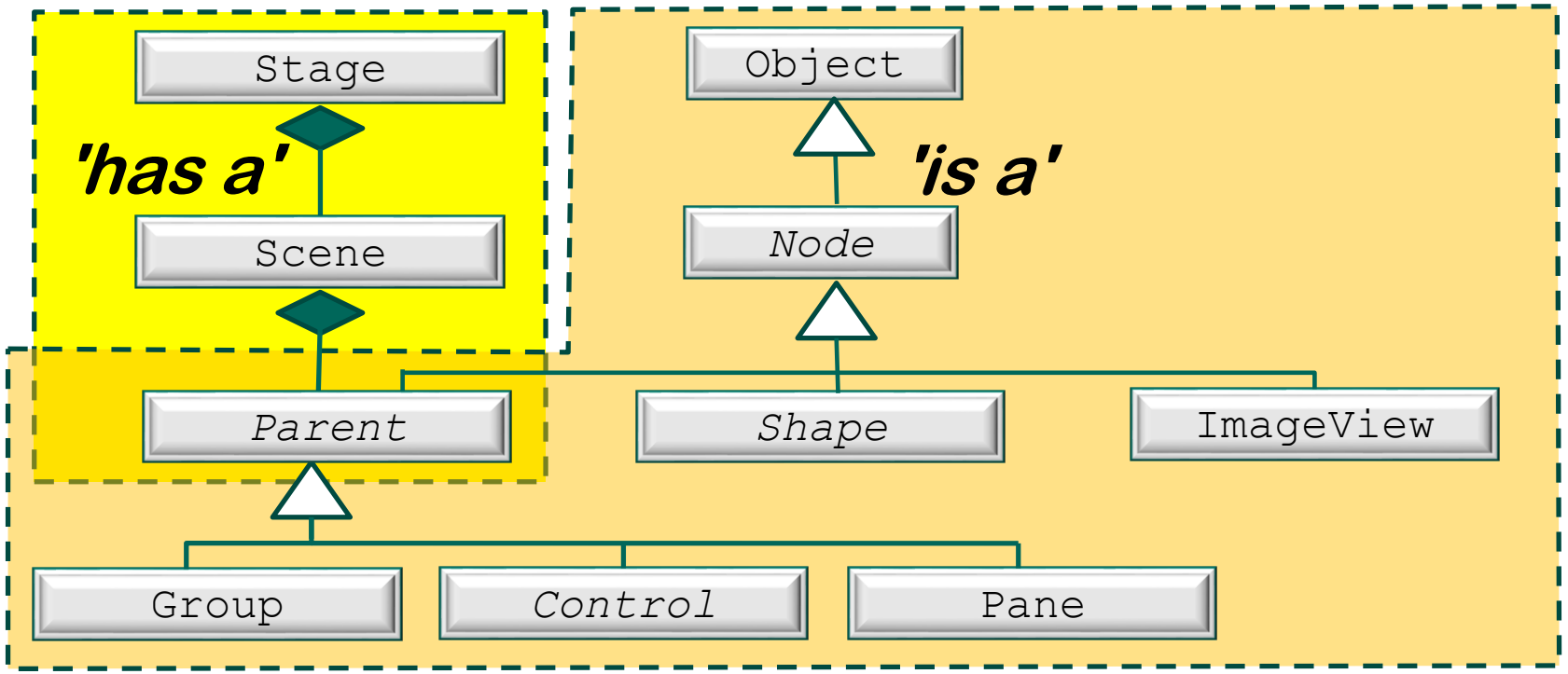
Consider again three of the subclasses of `Parent`: `Pane`, `Group`, and `Control` (which includes such subclasses as `MenuBar`, `ListView`, etc.) Each of these is not a single graphical item, but an item that is used to organize other items. For example, a `MenuBar` contains various menu items. A `Group` contains a collection of objects that are grouped together. And a `Pane`, as we've described, works something like a window frame that determines the space available for objects in the scene. Each of these objects acts like a parent with children. And so each gets its needed organizational capabilities (i.e. needed methods) from the `Parent` superclass.



4.6 The JavaFX API

So a `Parent` is just a `Node` with added functionality designed to handle the children it has to deal with (hence its name).

Recall that the `Parent` class is involved in the *has a* relationship with the `Scene` class; each `Scene` *has a* `Parent` object. But the UML diagram shown in previous slides also indicates that `Parent` *is a* `Node`.



4.6 The JavaFX API

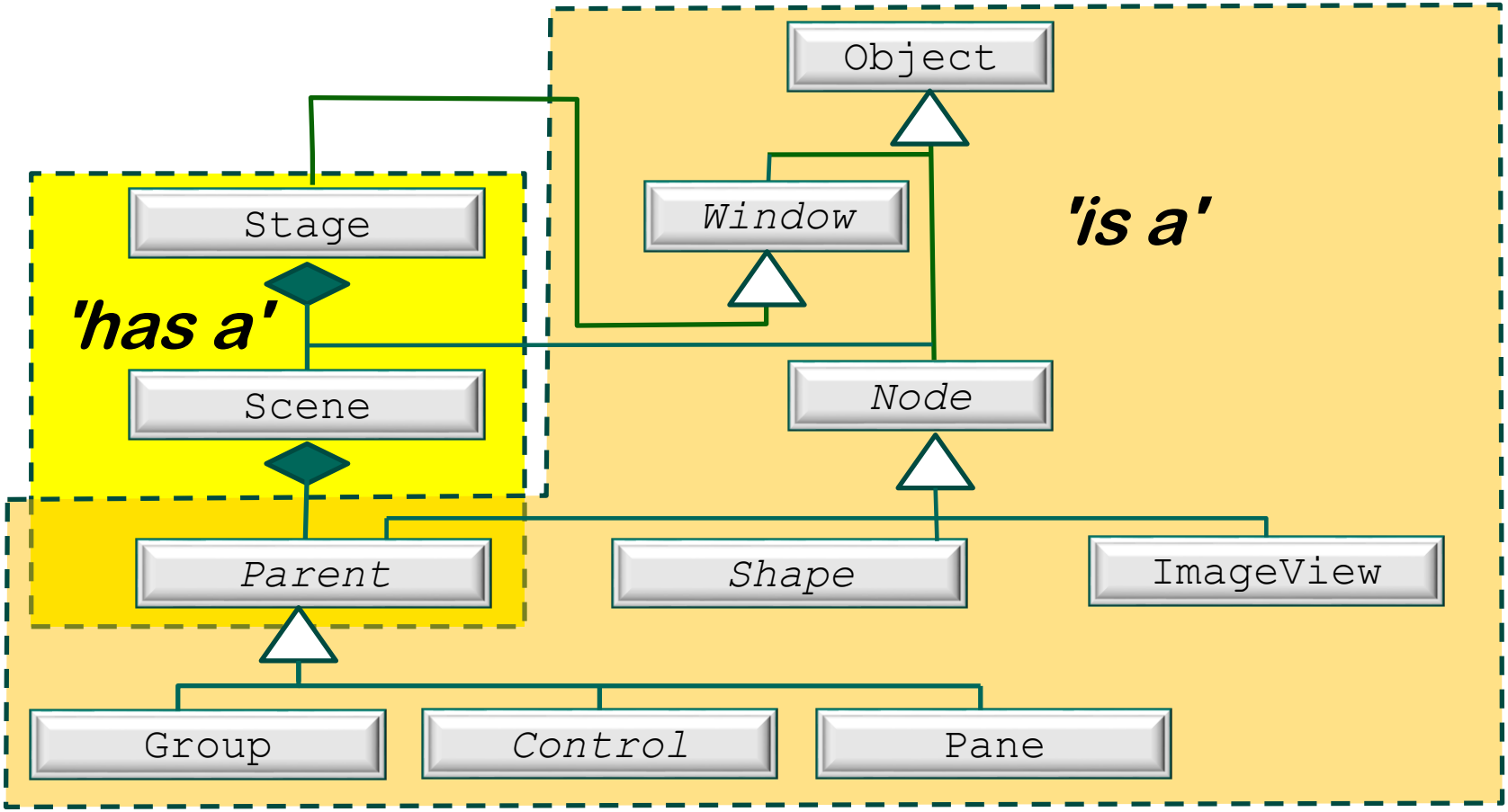
This is not really all that strange. After all, most objects both inherit from a parent class, but belong to (are compositions of) other objects. For example…

A firetruck *is a* truck…

A firehall *has a* (potentially many) firetrucks

4.6 The JavaFX API

Similarly, both `Stage` and `Scene` descend from `Object`. `Stage` itself directly descends from the `Window` class, but we don't normally include this information, since it only serves to clutter and confuse the UML diagram:



4.6 The JavaFX API

So there's nothing unusual about a class participating in both *is a* and *has a* relationships in the same UML diagram. UML was designed to handle *all* types of relationships. However, for simplicity, we sometimes choose to ignore relationships when they are not too important, or when their inclusion would obscure the more important relationships that exist between the classes we are interested in.

4.7 Pane

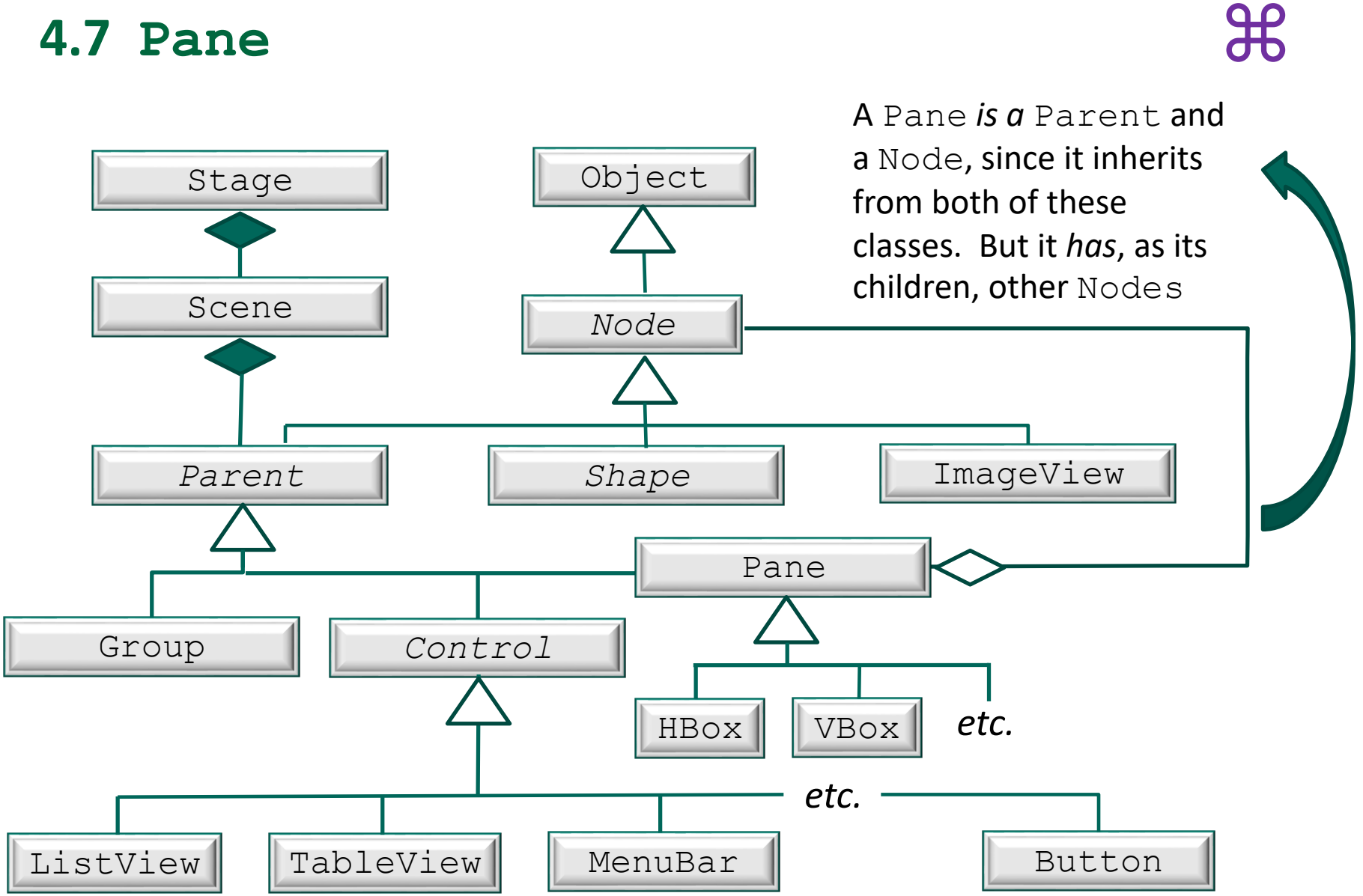
Returning to the `Pane` class, which will be loaded into `Scene` to help organize the placement of other `Nodes`…a `Pane` also participates in both *is a* and *has a* relationships.

As we've already stated above, a `Pane` *is a* `Parent`. But it holds other objects inside it—it is an ***aggregator***, in the language of the relationships we discussed at the start of the previous module. We could indicate this as follows:

This says: a `Pane` has many `Nodes`.

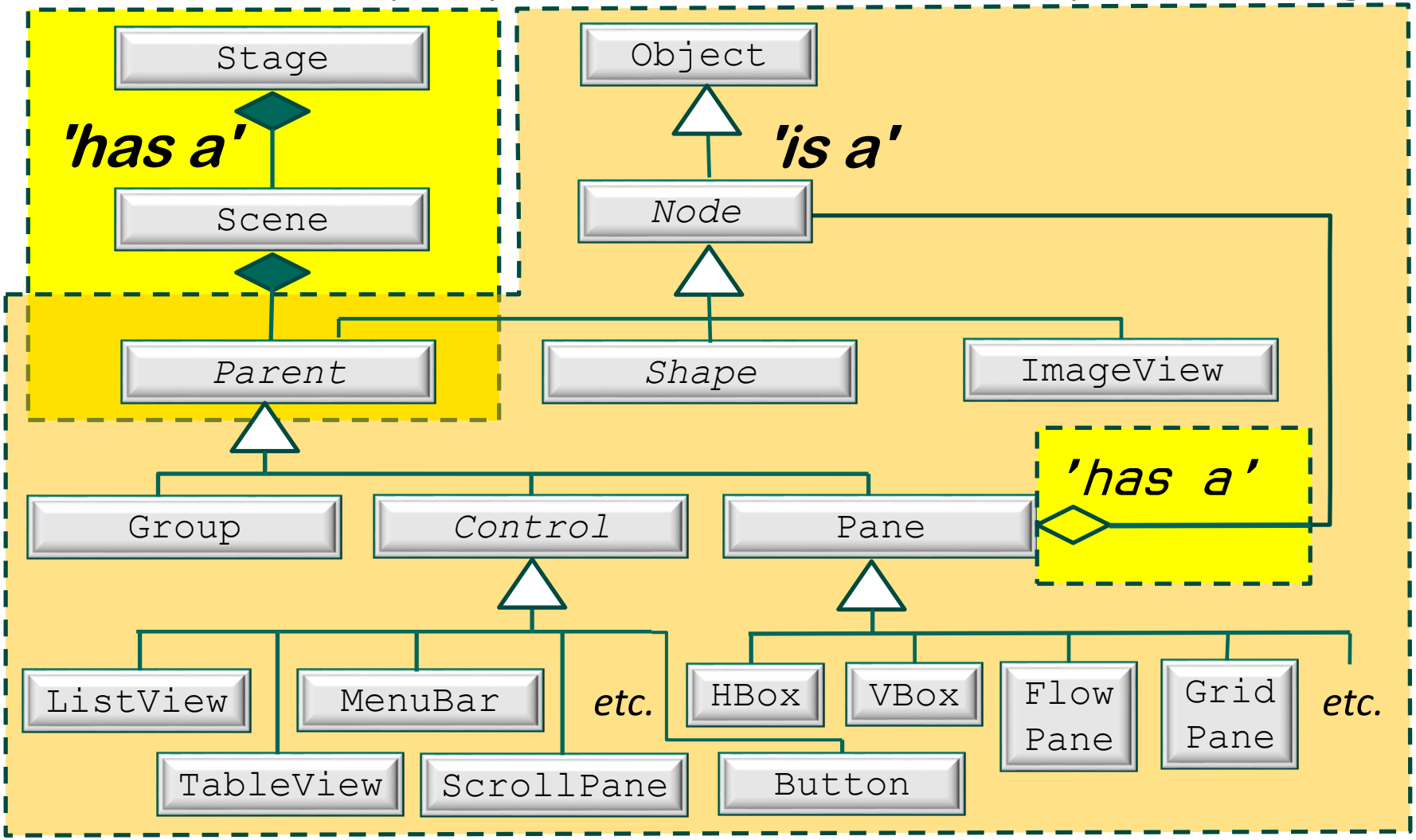
Written into our existing JavaFX UML diagram this relationship appears as:

4.7 Pane


A Pane is a Parent and a Node, since it inherits from both of these classes. But it has, as its children, other Nodes
Stage
Scene
Object
Node
Parent
Shape
ImageView
Pane
Group
Control
HBox
VBox
etc.
etc.
ListView
TableView
MenuBar
Button

4.7 Pane

Panes, like Parents, participate in both *is a* and *has a* relationships in our UML diagram



4.7 Pane

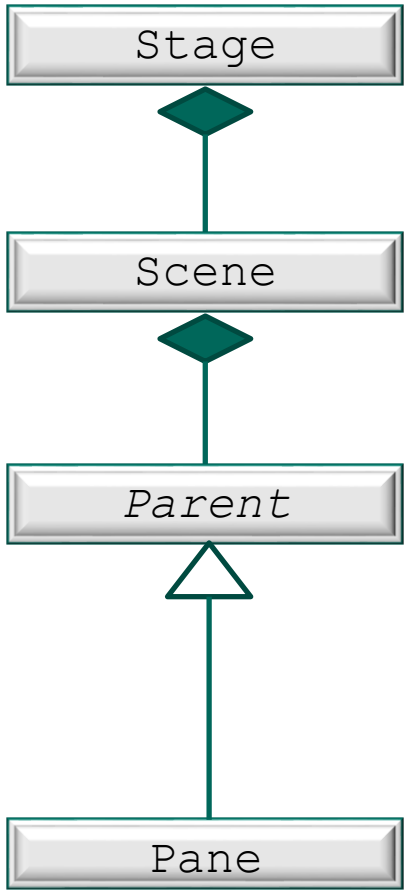
So UML diagrams are not restricted to just telling us where an object comes from (who its parents are) *or* who it belongs to (what role does it play in a composition).

UML diagrams can express both these things at the same time.

4.8 Recap: *is a* and *has a* relations in the JavaFX API

To recap: we need to load some kind of `Parent` object into the `scene`, before we load the `scene` into the `Stage` and `show()` it.

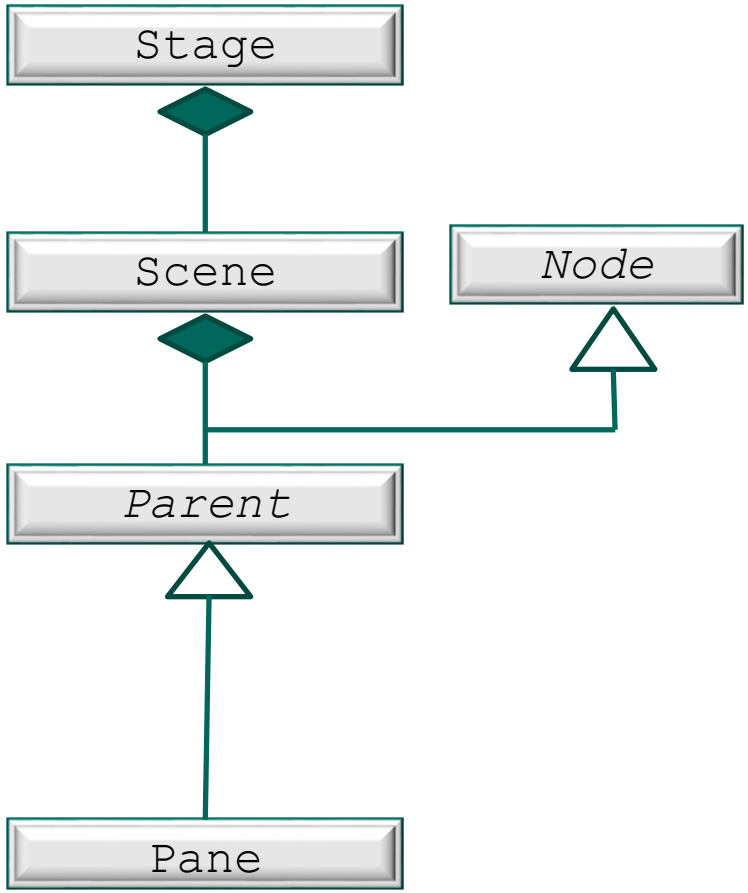
The `Pane` is one such `Parent` object that can be loaded…



4.8 Recap: *is a* and *has a* relations in the JavaFX API

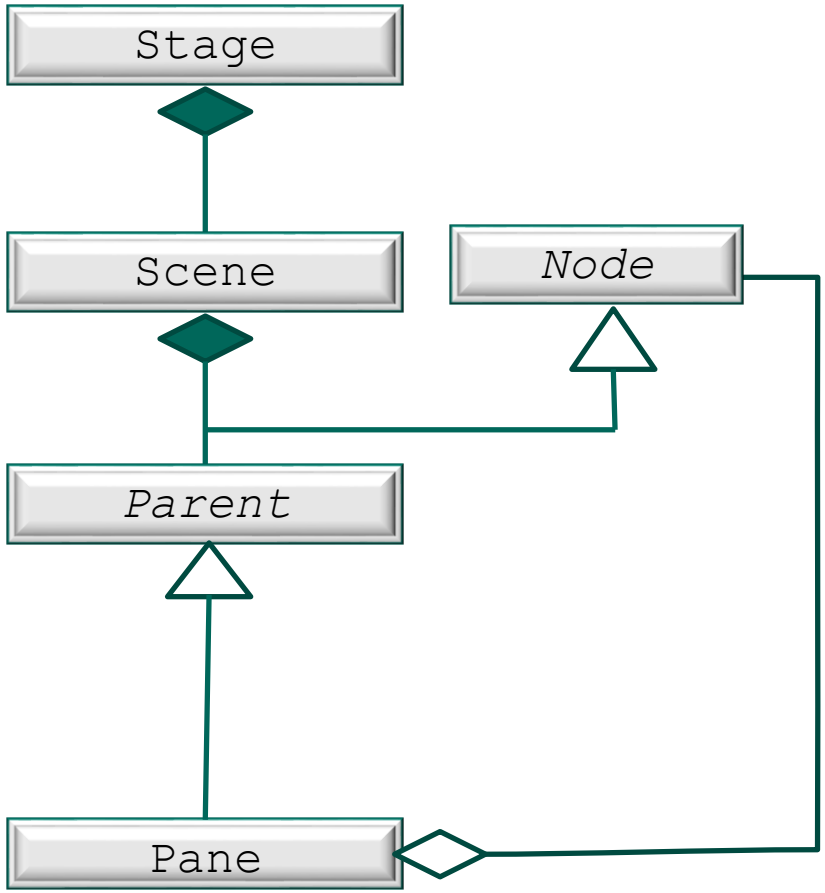
A `Parent` is a kind of `Node`, which holds the generic methods required by any graphical object.

A `Pane` is one of a number of `Parent` classes, each of which has added functionality to handle multiple *children* (since `Panes`, like `Groups` and `Menus`, have to organize aggregates of different types)



4.8 Recap: *is a* and *has a* relations in the JavaFX API

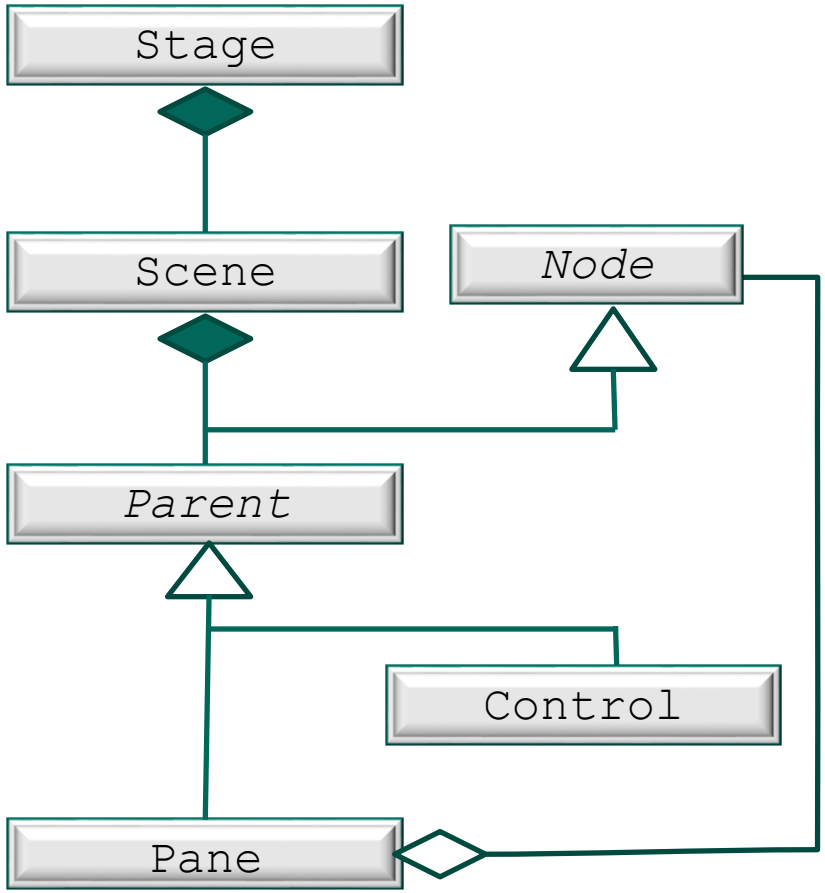
…and these children can include other kinds of `Nodes`. This means that anything which is a subclass of `Node` can be loaded into a `Pane`—including, by the way, other `Panes`.



4.8 Recap: *is a* and *has a* relations in the JavaFX API

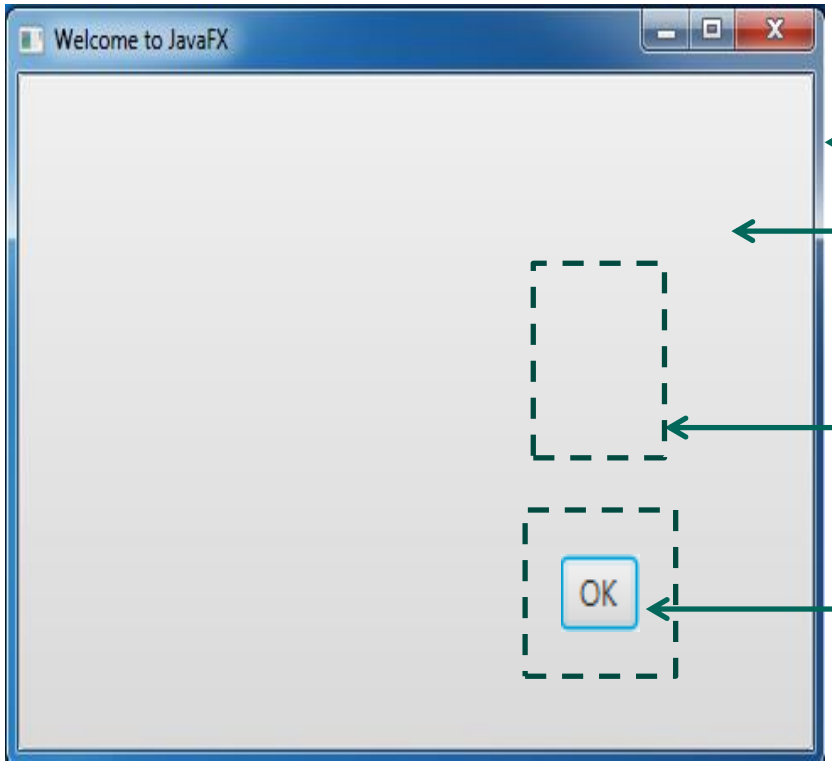
Thus a `Pane` can hold any concrete subclass of `Node`, including any kind of `Control`, including `Buttons`, `Checkboxes`, etc.

So `Panes` can be used to structure and organize other graphical objects in a scene.



4.8 Recap: *is a* and *has a* relations in the JavaFX API

We can summarize this graphically as follows:



The stage is the window where everything happens

The scene is the active area where nodes are placed. The stage has a scene.

A pane is an invisible node that organizes other nodes. A pane *is a* parent; each scene *has a* parent

Each pane contains and helps organize one or more nodes, and may even contain other panes. Thus pane *has* one or more node objects, of which a button is one.

4.9 Using Panes

To continue with our code: rather than load the `Button` directly into the `Scene`, we load the `Pane` into the `Scene`, and `Button` into `Pane`. `Pane` controls the `Button`'s size, limiting the dimensions of the `Button` to a reasonable default size:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.Pane;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class myJavaFXApp extends Application{
    @Override
    public void start(Stage primaryStage){
        StackPane pane = new StackPane();
        pane.getChildren().add(new Button("OK"));
         ...
        Scene scene =  new Scene(pane, 200, 50);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Make the pane…

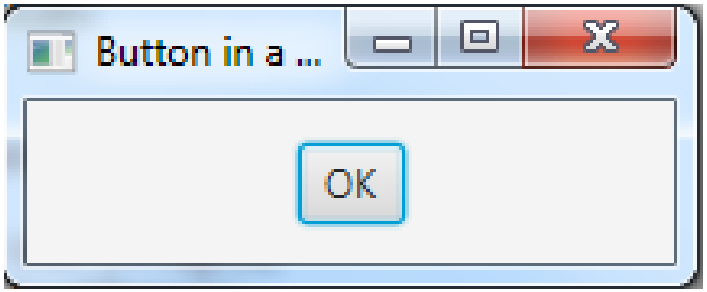
Load the button into the pane…

Load the pane into the scene…

Load the scene into the stage…

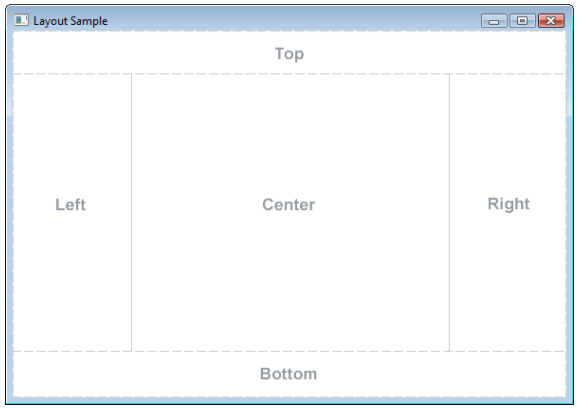
4.9 Using Panes

Now the size of the scene respects the default size of the pane, which contains the button.



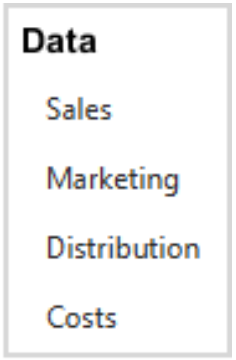
4.9 Using Panes

`Panes` come in different varieties, some of which are indicated below:



`BorderPane`

`FlowPane`



`VBox`

`HBox`

Images retrieved from: http://docs.oracle.com/javafx/2/layout/builtin_layouts.htm, November 27, 2015

4.9 Using Panes

Since panes can contain other panes, we can put, for example, a `Vbox`, a `FlowPane`, and an `HBox` inside a `BorderPane`.

This is how we structure our graphical elements…

`BorderPane`

`HBox`

`FlowPane`

`VBox`

`GridPane`

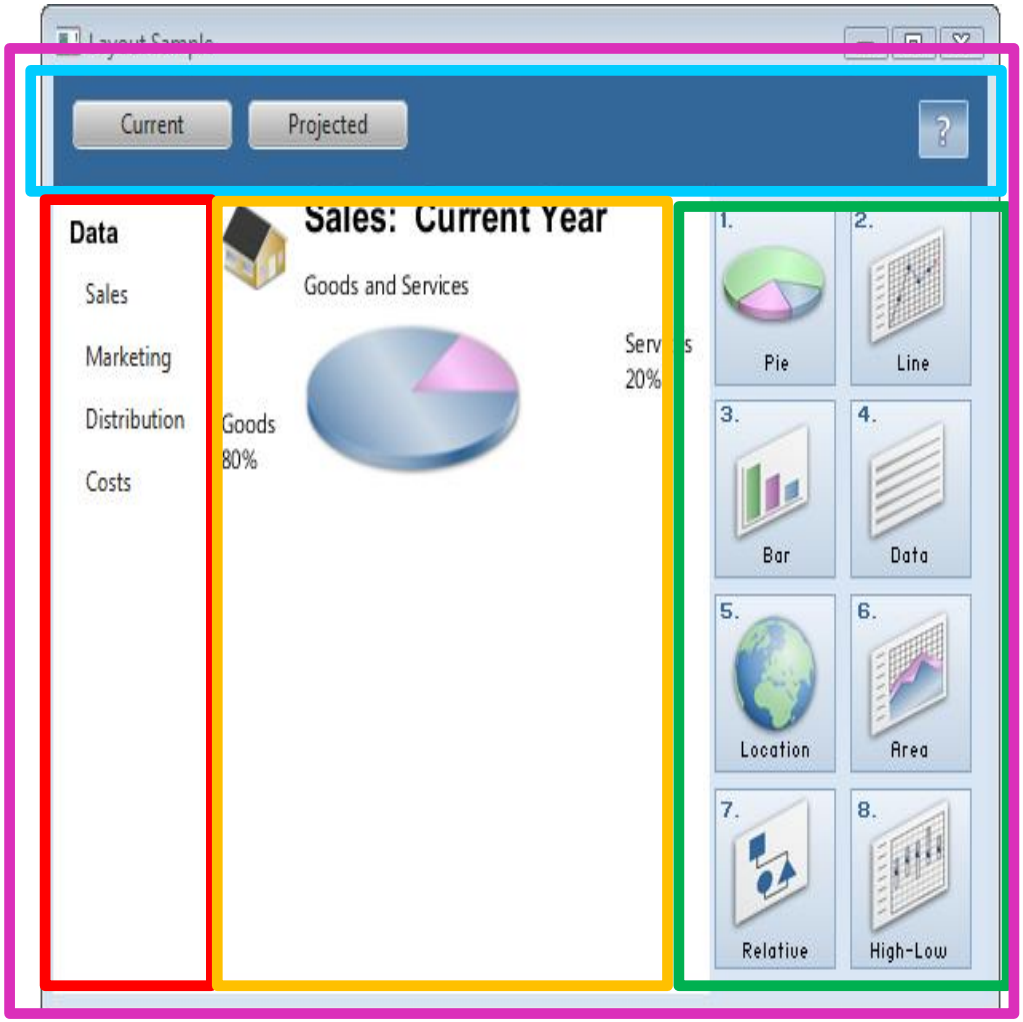


Image retrieved from: http://docs.oracle.com/javafx/2/layout/builtin_layouts.htm, November 27, 2015

4.9 Using Panes

Just as panes can have children added to them, they can also have scroll bar objects added as well. To use these 'clickable' objects, objects need to be able to 'listen' for click events and respond to them when they occur. This is a subject for the end of this module.

Fortunately, there is a special kind of pane that has a scroll bar built in…

4.9 Using Panes

The `ScrollPane` object is an object that (conveniently) comes with scrollbars attached, removing some of the coding that would otherwise be required. For example, we might use

```
ScrollPane sp = new ScrollPane();
sp.setContent(text);
```

to load the `text` control into a scrollable pane. (`setContent()` takes any `Node` object as its sole parameter.) The `ScrollPane` object can be used to scroll any Node, including images, as shown at right.

Source: Alla Redko, http://docs.oracle.com/javafx/2/ui_controls/scrollpane.htm, October 1, 2016

To use the `ScrollPane` object, you must first import `javafx.scene.control.ScrollPane`

4.9 Using Panes

The `ScrollPane` object has both vertical and horizontal scrollbars included automatically. Hence if you load an image like the one shown at right, you get both scrollbars, whether you want them or not.

To remove one or the other of the scrollbars, you must set the vertical scrollbar and vertical scrollbar's *policies*, as follows:

```
sp.setHbarPolicy(ScrollBarPolicy.NEVER);
sp.setVbarPolicy(ScrollBarPolicy.ALWAYS);
Sp.setVbarPolicy(ScrollBarPolicy.AS_NEEDED);
```

The `ALWAYS`, `NEVER` and `AS_NEEDED` constants are part of the `javafx.scene.control.ScrollPane.ScrollBarPolicy` library, which must be imported also.

From: Alla Redko, http://docs.oracle.com/javafx/2/ui_controls/scrollpane.htm, October 1, 2016

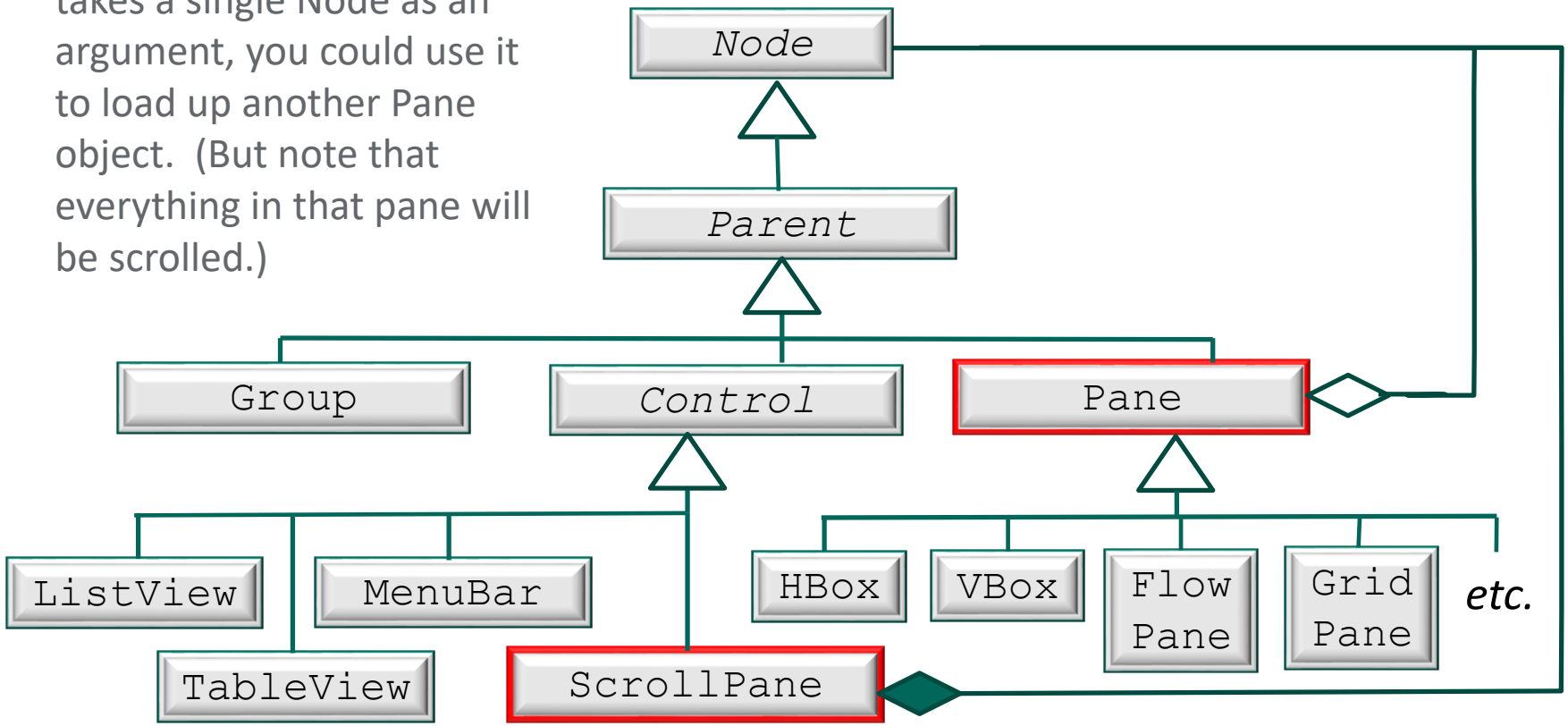
4.9 Using Panes

Note that `ScrollPane` is *not* a child of a `Pane`, it is in fact a child of the `Control` object. However, since it takes a single Node as an argument, you could use it to load up another Pane object. (But note that everything in that pane will be scrolled.)

javafx.scene.control

Class ScrollPane

java.lang.Object
- javafx.scene.Node
  - javafx.scene.Parent
    - javafx.scene.layout.Region
      - javafx.scene.control.Control
        - javafx.scene.control.ScrollPane



4.9 Using Panes

The `Pane` object has one method, and one method only, which it overrides from its `Parent` superclass. This method is `getChildren()`, which returns a list of all the existing objects loaded into the `Pane` itself (which can be an empty list, if nothing is loaded).

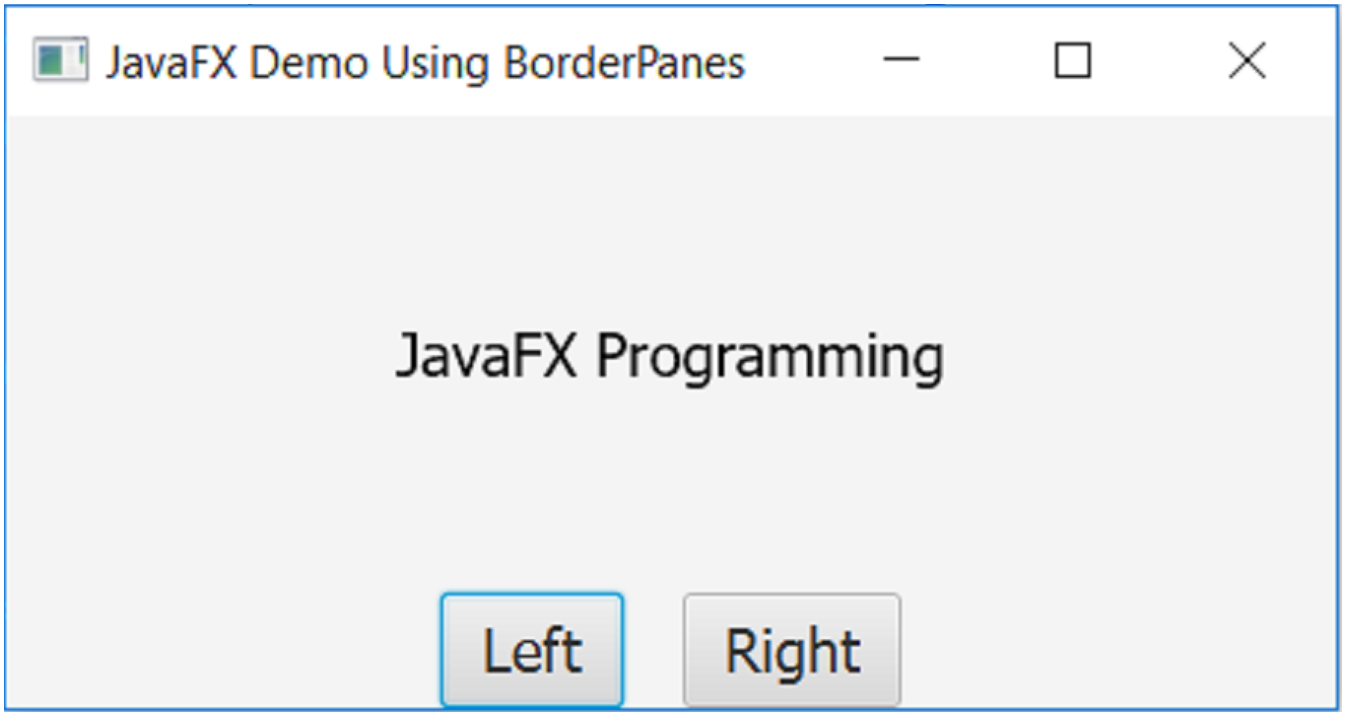
This list, which is a special kind of list called an `ObservableList` ('observable' in the sense that it can respond to any mouse clicks) has two useful methods that we will use to load objects into each pane:

`add(`*`node`*`)`	Add a single node object
`addAll(`*`node1, node2,…`*`)`	Add several nodes objects

So all the descendants of `Pane`, including `HBox`, `VBox`, `BorderPane`, etc (but not `ScrollPane`) will have these methods available to load up nodes into a `Pane` object.

4.9 Using Panes

As an example of "Panes in Action", in the screenshot below, a `BorderPane` has been used to split the scene vertically into two sections, a top and a bottom. The top contains text, the bottom, two buttons.



The example used in this section is taken from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pp. 633 - 634.

4.9 Using Panes

To load the centre panel of the `BorderPane` with text, we first need to create a text object…

```
Text text = new Text("JavaFX Programming");
```

…which we then add to a basic `Pane` object:

```
StackPane centerPane = new StackPane();
centerPane.getChildren().add(text);
```

See: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/text/Text.html for details on the Text object

4.9 Using Panes

Similarly, in the bottom half of the scene, we will load two buttons, which we must instantiate first:

```
Button btLeft = new Button("Left");
Button btRight = new Button("Right");
```

Once the buttons are created, they can be loaded into an `Hbox` pane:

```
HBox bottomPane = new HBox(20);
bottomPane.getChildren().addAll(btLeft, btRight);
bottomPane.setAlignment(Pos.CENTER);
bottomPane.setStyle("-fx-border-color: green");
```

4.9 Using Panes

Finally, the two panes are loaded into a `BorderPane`:

```
BorderPane rootPane = new BorderPane();
rootPane.setCenter(centerPane); // load centre pane
rootPane.setBottom(bottomPane); // load bottom pane
```

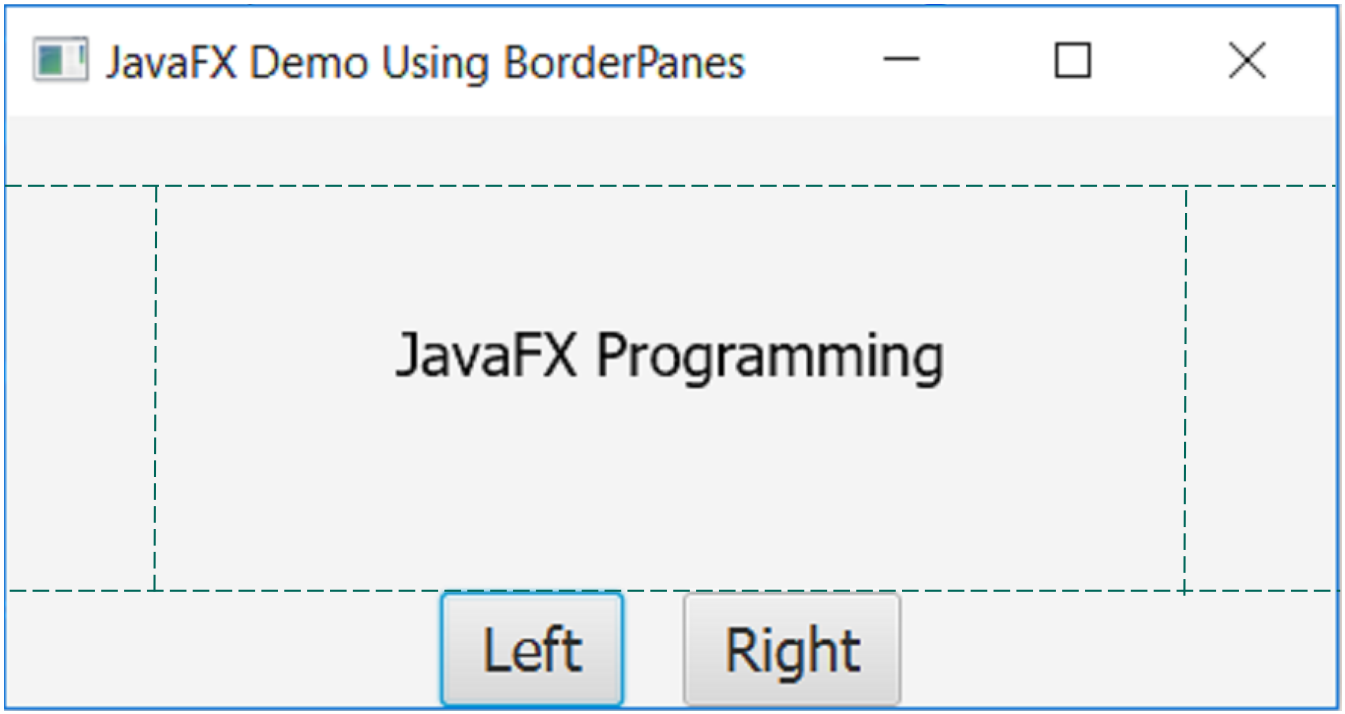
Now that the `pane` is loaded, we can set the `Scene` and `Stage`:

```
Scene scene = new Scene(rootPane, 450, 200);
primaryStage.setScene(scene);
primaryStage.show();
```

Note: the three other panes in a `BorderPane` can be loaded using the `setTop()`, `setLeft()`, and `setRight()` methods.

4.9 Using Panes

Here again is the result, with the border pane regions indicated with dashed lines. The `centerPane`, holding the `text`, is loaded in the centre of the five `BorderPane` regions, while the `bottomPane`, holding the `HBox` with the two buttons, is loaded in the bottom pane of the `BorderPane` region.



4.9 Using Panes

A second example demonstrates the use of the gridpane. With gridpanes you specify the position of controls by giving their coordinates on a grid, in the form (column, row). The top left corner of the gridpane has coordinate (0,0). The following example demonstrates the use of a gridpane:

```
public class GridPaneDemo extends Application{
   public void start(Stage pStage){
     TextField tfName = new TextField();
     tfName.setMaxWidth(120);
     TextField tfPhone = new TextField();
     tfPhone.setMaxWidth(120);
     TextField tfEmail = new TextField();
     tfEmail.setMaxWidth(120);
     …
```

* From Schildt, Herb. Introducing JavaFX Programming, McGraw-Hill publishing, 2015. Pg. 128-129

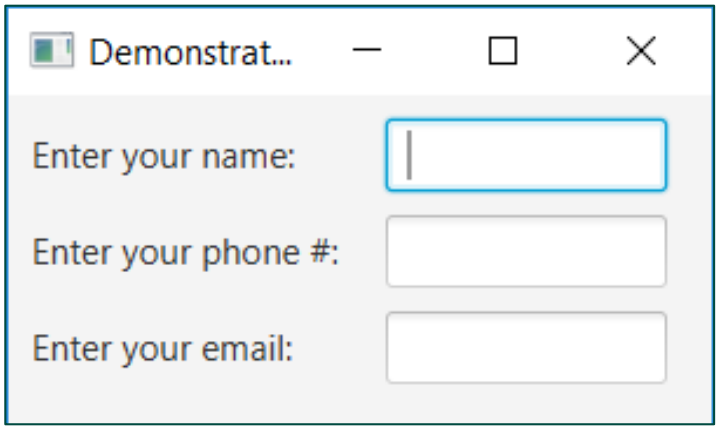
4.9 Using Panes

```
Label lblName = new Label("Enter your name:");
Label lblPhone = new Label("Enter your phone #:");
Label lblEmail = new Label("Enter your email:");

GridPane rootNode = new Gridpane();
rootNode.setPadding(new Insets(10, 10, 10, 10);

rootNode.setVgap(10);
rootNode.setHgap(20);

rootNode.add(lblName, 0, 0);
rootNode.add(lblPhone, 0, 1);
rootNode.add(lblEmail, 0, 2);
```

…

4.9 Using Panes

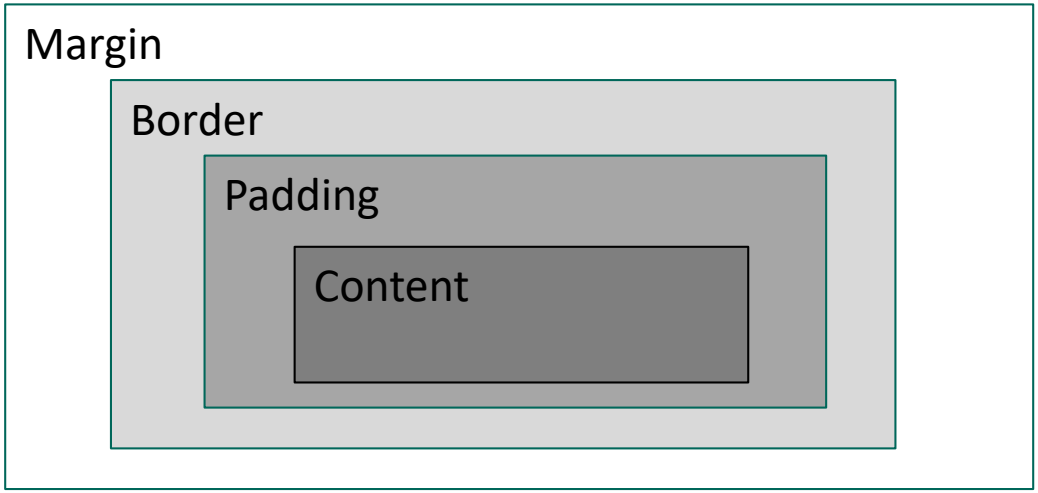
```
        rootNode.add(tfName, 1, 0);
        rootNode.add(tfPhone, 1, 1);
        rootNode.add(tfEmail, 1, 2);

        Scene myScene = new Scene(rootNode, 300, 140);

        pStage.setScene(myScene);
        pStage.show();
        }
}
```



4.9 Using Panes

Note the following features common to add controls used in panes:

- Each region in a pane has both a border and padding according to the following diagram. (This layout is similar to that used in CSS.)



Thus the *insets*, which consist of both the Border plus the Padding, determine the spacing between components

4.9 Using Panes

- Additionally, individual panes may have methods that control additional features of the layout (e.g. Vgap and Hgap in gridpane)
- The methods setMinWidth() and setMaxWidth() can be used to determine the minimum and maximum width of a control. Similarly setMinHeight() and setMaxHeight() determine the height of controls
- Each Node has a setStyle() method that determines the style for all child objects. The parameter is a CSS string. For example

```
defaultText.setStyle("-fx-font: 40px Tahoma; -fx-stroke: black;
-fx-stroke-width: 1;");
```

Applies a black 40 pt. Tahoma font to the contents of a textbox. If applied to a pane, the contents of the pane would all be required to use the same font and style.

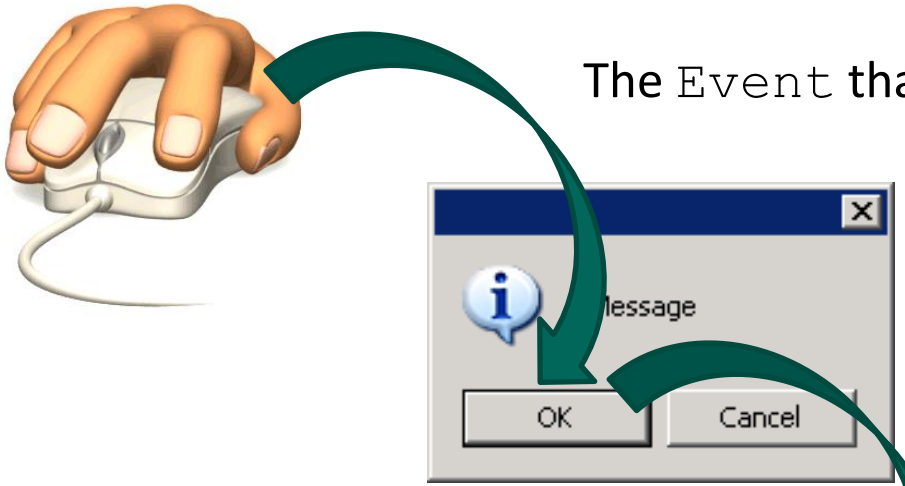
4.10 Dealing with Events

In JavaFX, an **event** refers to anything that results from an interaction with the user. Thus a keypress and mouseclick are events, as are resizing a window, or finishing a printout. Since we are interested in adding functionality to our two buttons, we'll be interested mainly in mouseclick events. But everything that follows can be adapted to events in general.

Liang 15

4.10 Dealing with Events

When a button object is clicked, it must have some way to respond to the mouse click event. This requires the interaction of three separate objects:



The `Event` that occurred…

…the target `Node` that received the `Event`, (for example, a `Button`), and…

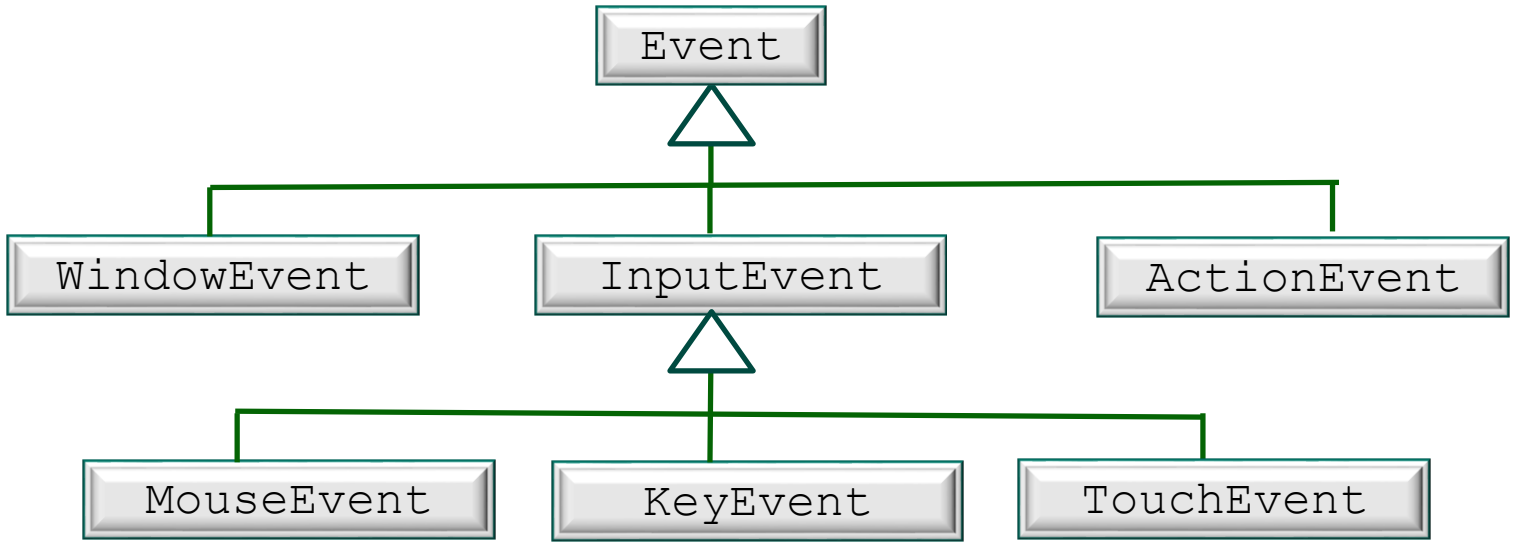
…the code that gets executed as a result, which is contained in an object called an `EventHandler`

```
import javafx.event.EventHandler;
import javafx.event.ActionEvent;

public class HdlResponse implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent e){
        text.setX(e.getX());
        text.setY(e.get(Y));
    }

}
```

4.10 Dealing with Events

Let's look at these three objects in more detail:

1. **`Event`** – an event is an object that includes information such as what the source is (e.g. the mouse), what the event type is (a click), and what the intended target is (a button control). The javafx `Event` class is stored in `javafx.event.Event`. The actual nature of the event can be broken down into various subclasses, of which three are shown below:



4.10 Dealing with Events

1. `Event` (con't) .

- A `WindowEvent` is an event which affects the `Stage` itself, such as hiding or closing the window;
- An `InputEvent` is one in which a key is pressed, a mouse is clicked, or the screen is touched; these will be of interest to us shortly;
- Finally, an `ActionEvent` is a high-level event that says: "some action just happened; I'll pass on the information and you decide what it was, and how to deal with it".

Liang 15.8 | Liang 15.9

4.10 Dealing with Events

1. `Event` (con't) .

Note that an `ActionEvent` is a generic event designed to respond to any sort of input. For example, an item in a menu may be triggered by selecting it with the mouse, touching it on a touchscreen, or pressing a combination of keys. Any of these would constitute an `ActionEvent`.

This is in contrast to `InputEvents` such as `MouseEvent`. In this case, specific actions are returned indicating a mouse click, mouse move, or mouse over event. The `MouseEvent` object also includes information such as the X,Y location of the mouse on the screen, and which button has been pressed.

So an `ActionEvent` is a high-level generic event, while a `MouseEvent`, `TouchEvent` or `KeyEvent` is more specific kind of `InputEvent`, usually tied to a the properties of a particular piece of hardware.

4.10 Dealing with Events

2. **`EventHandler`** – this contains a single (non-abstract) method called `handle()`. If an event occurs and the default `handle()` method is used, then *nothing* happens—which is the case most of the time. To respond to an event with your handler, you override `handle()`, so that your code gets executed instead (much as you did with the `start()` method in `Application`.) A typical example looks like this:

```
class myEvtHandler implements EventHandler<Event e>{
   @Override
   public void handle(Event e){
     // deal with the event here
   }
}
```

The meaning of `implements` will be explained in a few weeks. For now, you can safely treat `implements` like `extends`, and so `myEvtHandler` acts like a class inherited from `EventHandler`.

4.10 Dealing with Events

2. **`EventHandler`** (con't) – Note that the `EventHandler` uses the **Generic** notation to indicate that it can handle any type of `Event` object, including and `Event` itself, or any of its subclasses (as seen in the UML diagram four slides ago). This `Event`—be it an `ActionEvent` or a `MouseClick`—is parameterized as the object `e`, which can be used to determine possible outcomes

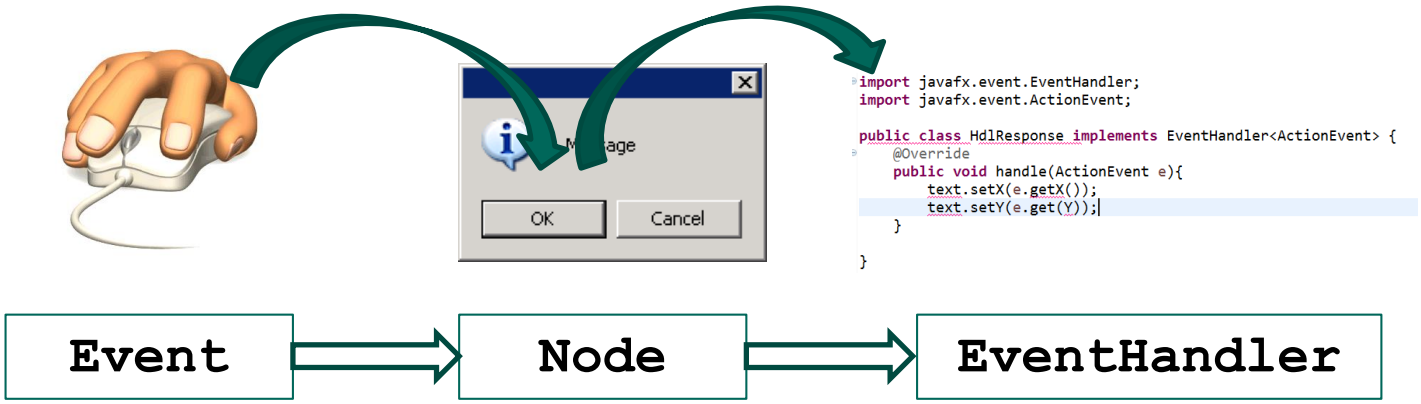
```
class myEvtHandler implements EventHandler<Event e>{
   @Override
   public void handle(Event e){
     System.out.println(e.toString() + " just happened");
   }
}
```

Note: a **handler** is sometimes referred to as a **listener**

Liang 15.10 | D&D 12.6 | D&D 12.7 | D&D 12.8

4.10 Dealing with Events

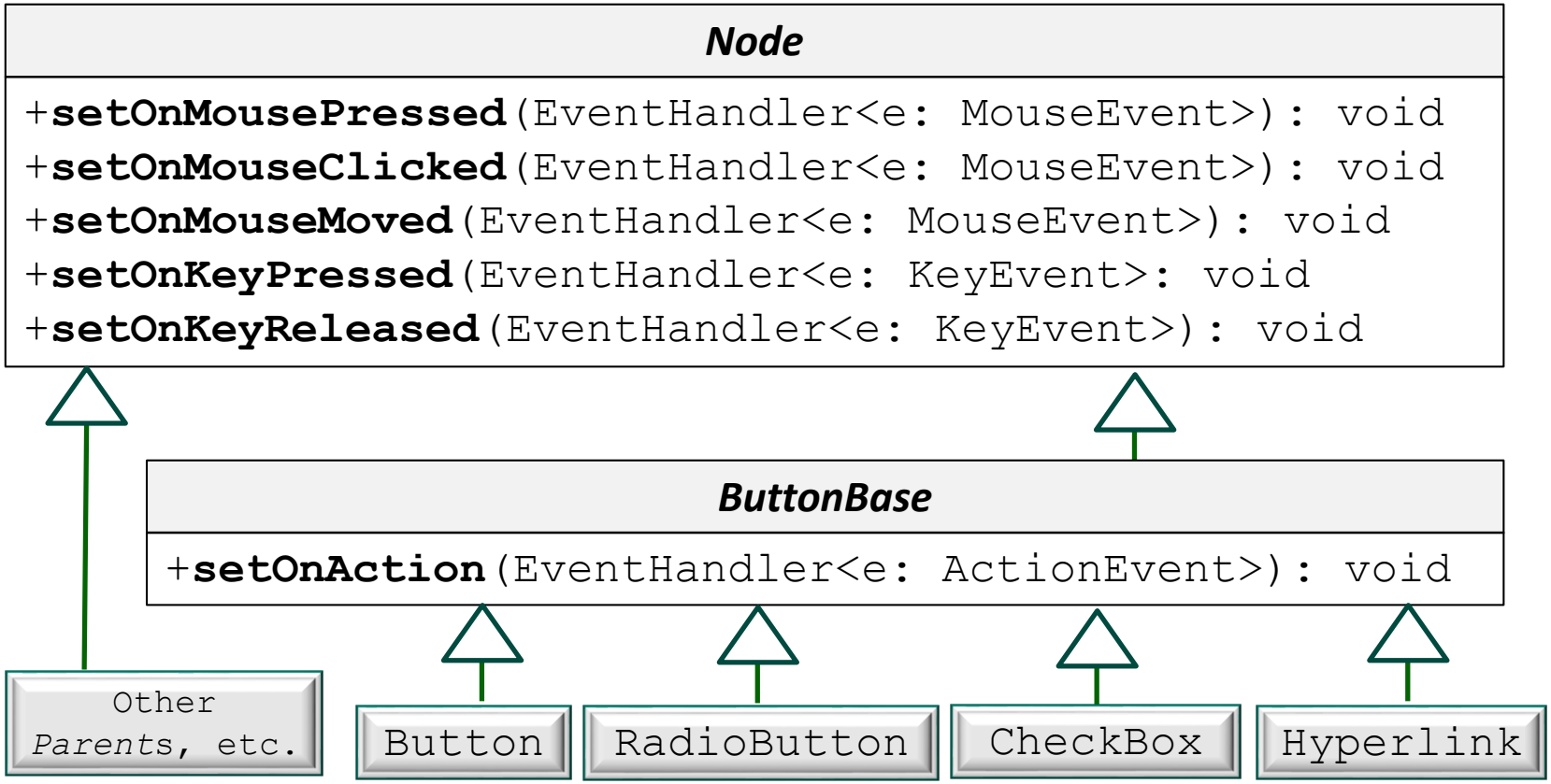
3. Node – Even though the `Event` 'knows' its target, the target itself needs to be set up to respond to events. All `Node` objects contain methods that allow the user to load an event handler so that, when some event happens on a `Node`, that `Node` will execute the appropriate handler code. So to repeat our earlier diagram…



…the `Node` only responds to a click with the appropriate response *only* if the `EventHandler` has been loaded correctly into the `Node` before the event actually occurs.

4.10 Dealing with Events

3. Node – (con't) To load `Node` with your `EventHandler`, you pass the instantiated `EventHandler` object using one of the methods available:



4.10 Dealing with Events

3. Node – (con't) For example, since a `Pane` is a `Node`, any `Pane` should be able to intercept keypresses. Assuming have declared a new event handler class called `evtKeyPressed` which implements `EventHandler<KeyEvent>`, we can then initialize the `HPane` to respond to key press events using

```
HBox HPane = new Hbox();
HPane.setOnKeyPressed(new evtKeyPressed);
```

This loads the new `evtKeyPressed` object into the method designed to respond to key presses. Assuming we overroad the handle method, when a key is pressed on the `HPane` object, the `handle()` method gets executed.

4.10 Dealing with Events

3 . Node (con't)

`Buttons` and `CheckBoxes` (which usually don't need to know about things like X,Y coordinates, or which button was clicked) also respond directly to basic `ActionEvents`. Hence a `new Button()` like the one in the above example, can have an `ActionEvent` attached to it via:

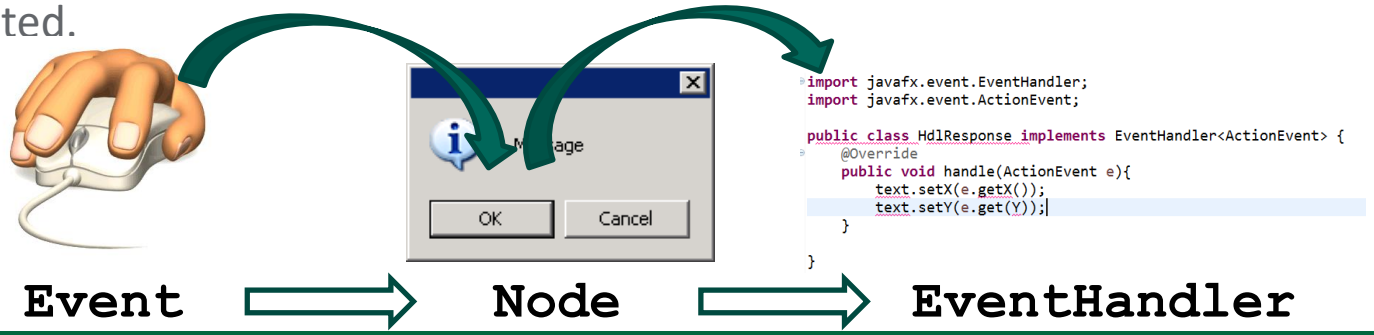
```
btnLeft.setOnAction(hdlrBtnClicked);
```

where **`hdlrBtnClick`** is assumed to implement a `new EventHandler<ActionEvent>`

4.10 Dealing with Events

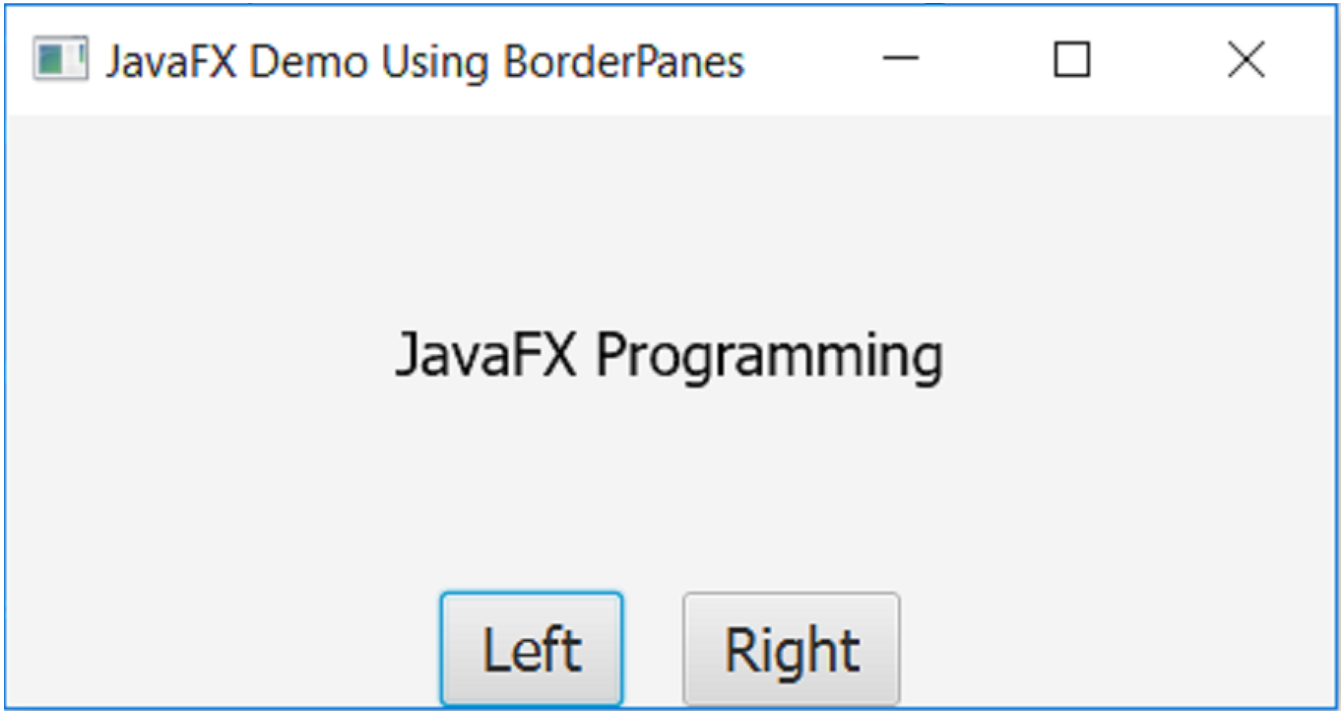
To summarize:

1. Implement a new `EventHandler` appropriate to the event you wish to capture, which overrides the `handle()` method with the code you wish to have executed;
2. Load the `Node` that is to be the recipient of that event with an instance of the above handler class using `setOnAction()` or one of the `setOnMouse…()`/`setOnKey…()` methods, as shown in the `Event` hierarchy several slides back;
3. Trigger the appropriate event on the above object (i.e. a mouse click), which causes the `Event` object to be created and passed to the `EventHandler` object you created in step 1, which causes the code in `handle()` to be executed.



4.10 Dealing with Events

Returning to our earlier example…



We can assign code to the two buttons as follows

4.10 Dealing with Events

Assuming we have created an EventHandler to deal with button clicks—call it `ButtonClickHandler`–it must first be instantiated. The code is straightforward:

```
ButtonClickHandler hdlrBtnClick = new ButtonClickHandler();
```

and then loaded into the control:

```
btRight.setOnAction(hdlrBtnClick);
```

4.10 Dealing with Events

Now, whenever an event occurs on that button, the handler captures the event and deals with it appropriately, according to the code in the overridden `handle()` class.

Consider again our `EventHandler()` class. It has two curious features, which may have escaped your notice:

1. It implements the `EventHandler` *interface*;
2. It is written as an **inner class**, a class that exists *inside* another class

The code for one such event handler is shown on the next slide. Note that this class exists within the same class as the other *methods* used in this program.

4.10 Dealing with Events

```
class ButtonClickHandler
    implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent e) {
        // handle the event here,
        // based on the ActionEvent object e
    }
}
```

Once the button is click, the code in `handle()` is executed. Once this code is finished, execution returns to the last point of execution prior to the interruption.

Thus, an 'event-driven' program might look like the code given on the following pages. (Note that this simple program forms the basis for most of the code you'll ever write in JavaFX.)

4.10 Dealing with Events

```
// load all the relevant libraries you'll be
// needing.  This can be substantial.

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.scene.text.Text;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
...
```

4.10 Dealing with Events

```
// Make sure there's a main routine for older IDEs
// and overload Application's start method

public class ButtonDemo extends Application {

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

   public void start(Stage primaryStage) {
      Scene scene = new Scene(getPane(), 450, 200);
      primaryStage.setTitle(
          "JavaFX Demo Using BorderPanes");
      primaryStage.setScene(scene);
      primaryStage.show();
   }   ...
```

4.10 Dealing with Events

```
// In this case, the pane is returned by a method
// called getPane().  This allows us to load
// different pane objects depending on what needs
// to be displayed in the scene

public BorderPane getPane() { // return a pane object
   Text text = new Text("JavaFX Programming");
   Button btLeft = new Button("Left");
   Button btRight = new Button("Right");
   ...
```

4.10 Dealing with Events

```
// Now load up different panes, including
// panes within panes

 HBox bottomPane = new HBox(20);
 bottomPane.getChildren().addAll(btLeft, btRight);
 bottomPane.setAlignment(Pos.CENTER);

 rootNode = new BorderPane();
 rootNode.setStyle("-fx-font: 20px Tahoma; -fx-stroke:
     black; -fx-stroke-width: 1;");

 rootNode.setCenter(text);
 rootNode.setBottom(bottomPane);
 ...
```

4.10 Dealing with Events

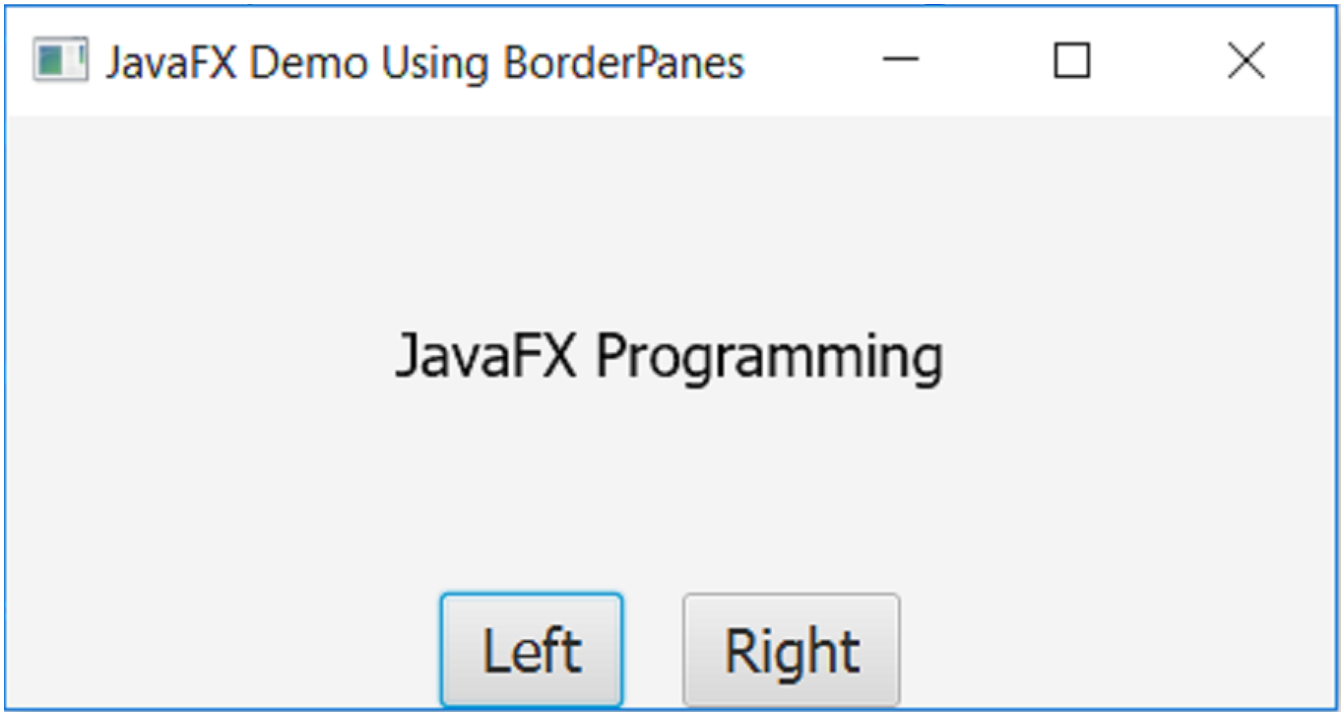
```
        // Create new event handler and load up them
        // up into any affected controls.
        // When the outmost pane is completed, return it
        // from the getPane() method

         LeftHandler btLeftClick = new LeftHandler();
         btLeft.setOnAction(btLeftClick);

         RightHandler btRightClick = new RightHandler();
         btRight.setOnAction(btRightClick);

        return rootPane;
    }   // end of getPane() method to construct panes
```

4.10 Dealing with Events

```
// Now include, in the same class, the event handlers,
// including the code they need to implement when they
// are called upon… these are inner classes

class RightHandler implements EventHandler<ActionEvent> {
   @Override
   public void handle(ActionEvent e) {
    rootNode.setCenter(new Text("Right Button Clicked"));
   }
}

class LeftHandler implements EventHandler<ActionEvent> {
   @Override
     public void handle(ActionEvent e) {
     rootNode.setCenter(new Text("Left Button Clicked"));
     }
}
```

4.10 Dealing with Events

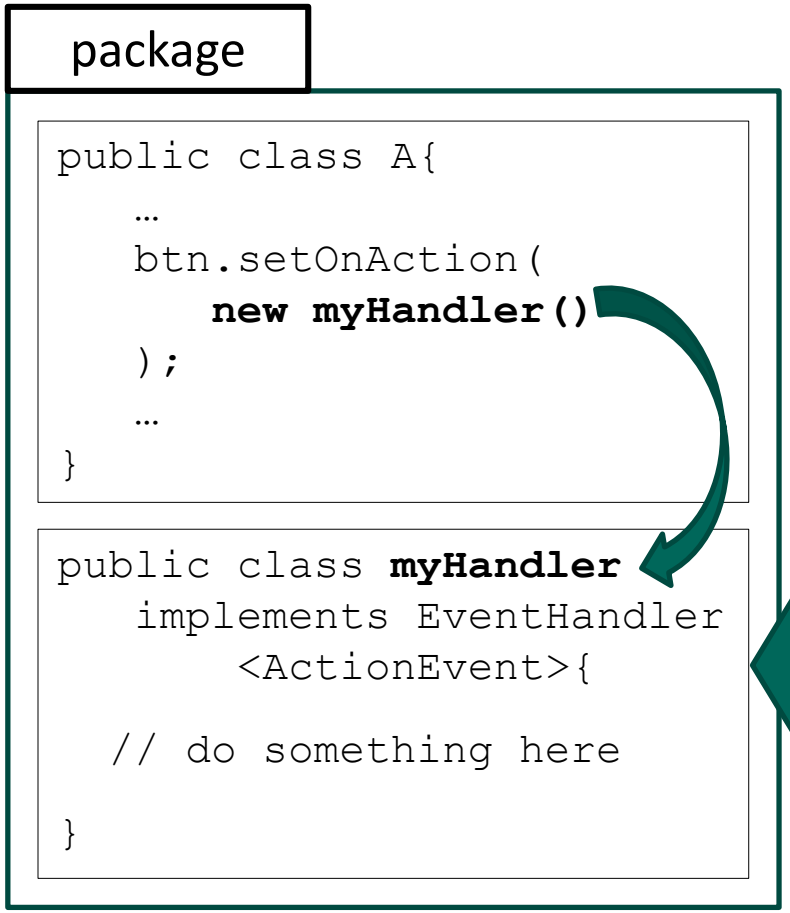
Again, the final product is:



The example used in this section is taken from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pp. 633 - 634.

4.11 Inner Classes

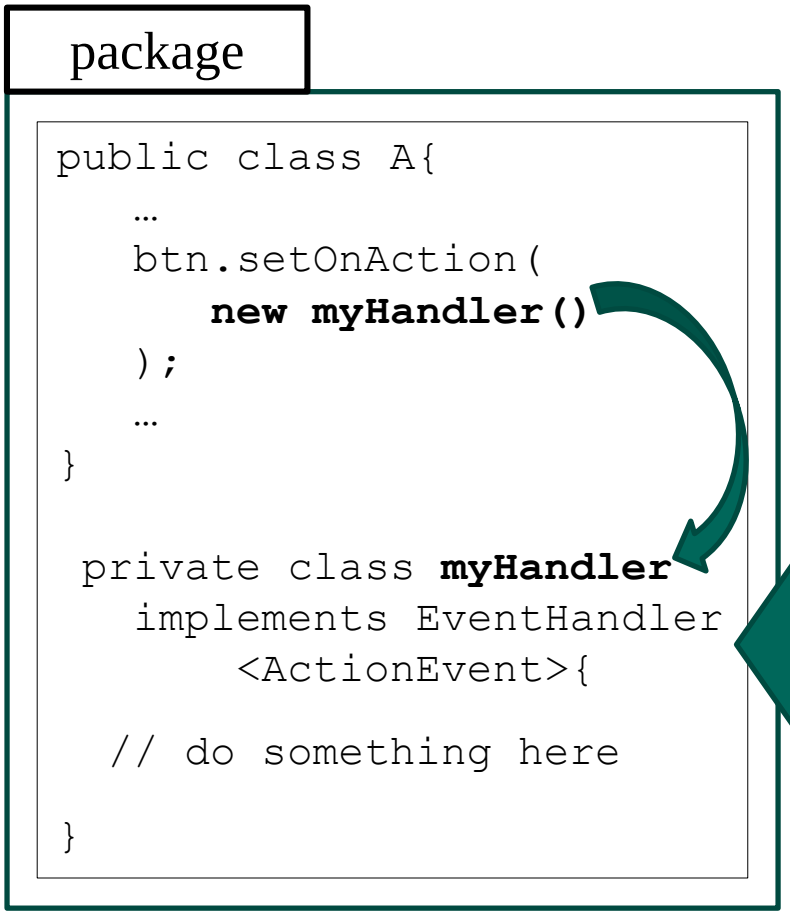
To treat code as an object, you must store that code in a class. In the code above we used an **inner class** to store the EventHandler code. Consider what would have happened if we'd used an external class instead for this purpose:



1. *We'd have to make a handler class for each separate event, no matter how short the handling code was; this would lead to extra classes cluttering up the package*
2. *The class would need to be public, otherwise the calling class would not be able to instantiate a new EventHandler object. But then everyone would be able to instantiate and use it.*

4.11 Inner Classes

Instead, we use an inner class (also sometimes called a *nested class*), which is literally a class within another, **outer class**. Inner classes have certain advantages:



1. *The inner class can be defined in the same class as the source that triggered the event, thus easing code maintenance;*
2. *The inner class can be made `private`, hence its visibility is limited to the outer class alone;*
3. *The package contains fewer classes overall, since each of the EventHandler declarations will all be contained in the outer class*

4.11 Inner Classes

The format of the inner class is identical to that of any other class. It looks just like another method inside the outer class—except that, being a class, it must be instantiated to an object first using the `new` keyword. For example:

```
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
public class myJavaFXCode extends Application{
   @Override
   public void start(Stage primaryStage){
      Button btLeft = new Button();
      btLeft.setOnAction(new lfBtnHndlr);
   }

   private class lfBtnHndlr implements EventHandler<ActionEvent>{
      @Override
      public void handle(ActionEvent e){…// do something }
   }
}
```

Liang 15.4

4.11 Inner Classes

Inner classes are mainly used as event handlers, but they have other potentially powerful uses as well:

- Inner classes can access outer class members transparently (including `private` members) so you do not need to pass this information in to the inner class via an inner class constructor.
- Inner classes can have `private`, `protected`, and `public` access modifiers, the same as outer classes. Additionally, inner classes can be defined as `static`.

When an inner class is compiled, it will appear in the `bin` directory in the form

```
OuterClassIdentifier$InnerClassIdentifier.class
```

So the inner class in the above example will appear in `bin` as:

```
myJavaFXCode$1fBtnHndlr.class
```

Examples from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed. Toronto, ON: Pearson.* pp. 594.

4.11 Inner Classes

Note also:

- Inner class objects are usually instantiated inside the outer class in which they are declared. However, you can instantiate a non-private, non-static inner class object inside another class, *provided* the outer class is instantiated first. For example, to instantiate an `innerObject` based on `InnerClass` inside an instantiated `OuterClass` object (called `outerObject`), you'd use:

```
OuterClass.InnerClass innerObject =
                              outerObject.new InnerClass();
```

- Inner classes may be `static` (unlike their outer classes, which can never be `static`). If so, then the syntax is slightly different:

```
OuterClass.InnerClass innerObject =
                              new OuterClass.InnerClass();
```

4.11 Inner Classes

Note that the name of the inner class itself is superfluous: it serves no real purpose, since it is usually going to be used only inside the `setOnAction()` method (or one of its '`set()`' mouse cousins). Therefore, it is frequently preferable to use an **anonymous inner class**; that is, we load the new EventHandler object directly into the button's `setOnAction()` method.

```
leftButton.setOnAction(new lfBtnHndlr);
…
private class lfBtnHndlr implements
    EventHandler<ActionEvent>{
    @Override
    public void handle()… }
}
```

```
leftButton.setOnAction(
    new EventHandler<ActionEvent>() {
    @Override
    public void handle(){…}
});
```

Liang 15.5

4.11 Inner Classes

When using anonymous inner class handlers, two things are especially important to remember (they are highlighted in yellow below), since they are easily missed:

1. The parentheses after `EventHandler<ActionEvent>`, which do not normally appear when the event handler is implemented as part of a named class. The parentheses are part of the `EventHandler` no-arg constructor;
2. The order of the braces/parentheses at the end of the statement; the anonymous EventHandler class is being passed as an argument to the `setOnAction()` method, so the entire class, including the `{}`'s, is inside the braces. Therefore the statement terminates with `});` This closes the class (first, with the `}`) and then the method (second), as indicated below. (Note that the `handle` method also has its own parentheses and braces as well).

```
leftButton.setOnAction(
    new EventHandler<ActionEvent>(){
    @Override
    public void handle(){…}
});
```

4.12 Lambda Expressions

Event handling may be further simplified through the use of **lambda expressions**, which are new to Java 8. Lambda expressions represent the most concise form of event handling possible. As with inner classes, while lambda expressions are not limited just to input events, they do tend to find their most common use in that particular capacity.

Liang 15.6

4.12 Lambda Expressions

In a lambda expression, the entire anonymous class declaration—the argument passed in the () of the method—is replaced with an expression in the form

```
(Type e) -> expression
```

This tells the Java compiler to infer the actual type used *by* the object parameter (the type is called `e` in this case), and execute (`->`) the following `expression`. This all goes inside the parentheses normally used to pass an anonymous class. In other words, a lambda expression replaces the anonymous inner class with the single, concise expression as shown above.

4.12 Lambda Expressions

How does this apply to the `setOnAction()` method? When the Java compiler sees the lambda expression inside the parentheses of this method, it tells it to:

1. Infer the data type passed to the method; in this case, `EventHandler<ActionEvent>`
2. Call a constructor of the appropriate type (in this case, a new `EventHandler<ActionEvent>()`)
3. Determine the *type* of the parameter used by that constructor. For `EventHandler` this means an `ActionEvent`. The user will need a way to reference the `ActionEvent` to address 'what happened', which was `e` in the example above.
4. Execute the code that would normally be executed by the handler.

Note that, as previously stated, the `EventHandler` only has a single method, the `handle()` method. This is an essential feature of any method that gets passed via a lambda expression: *the interface used can only have a single method*—the one that gets executed by default. The type used by that method will be the type passed by the `EventHandler`—an `ActionEvent` in this case.

4.12 Lambda Expressions

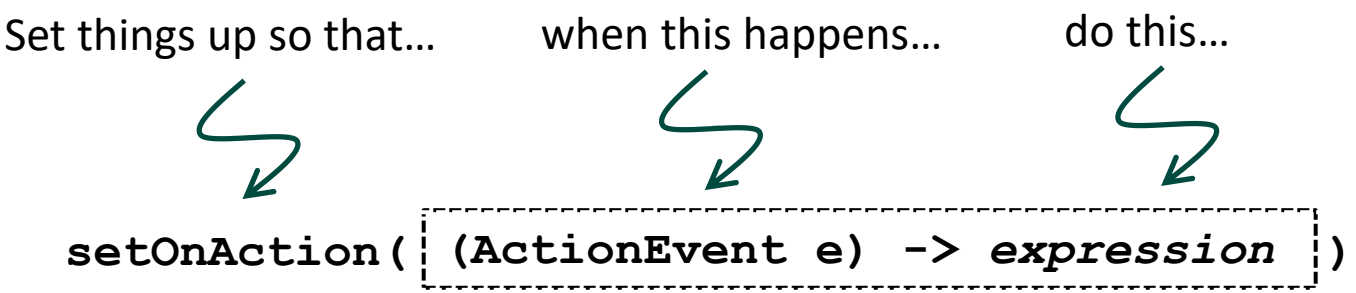
```
new EventHandler<ActionEvent>(){
    @Override
    public void handle(ActionEvent e){
        // expression
    }
}
```

```
(ActionEvent e) -> expression
```

So the lambda expression strips away anything that doesn't explicitly need to be written. The only things that truly need to be known are contained in the lambda expression itself: a handle to the type returned (`e`), and the code (*expression*) that gets executed.

4.12 Lambda Expressions

Or, to set things in a more easily memorizable form, think of the lambda expression as performing the following:



4.12 Lambda Expressions

There are some minor variations on the basic form of the lambda expression. First, If more than a single line of code is needed, we can insert several expressions inside a `{…}` block:

```
(Type e) -> {expressions}
```

If the type is already known (for example, if an `ActionEvent` can be inferred), we don't need to mention the type explicitly:

```
(e) -> expression
```

In this case, we can drop the parentheses around the type:

```
e -> expression
```

Finally, if there is only one type (say an `ActionEvent`), and it isn't actually used in the expression, we can drop the type identifier altogether. But we still need the parentheses as a placeholder:

```
() -> expression
```

4.12 Lambda Expressions

The following sample code shows how an event handler might be used in different ways, according to the variations seen on the previous slide. (We'll assume the appropriate libraries have already been imported.)

```
public class LamdbaHandle extends Application{
    @Override
    public void start(Stage primaryStage){
    // Set up the Hbox pane and add three buttons to it
        Hbox hBox = new Hbox();
        hBox.setSpacing(1);
        hBox.setAlignment(Pos.CENTER);
        Button btNew = new Button("New");
        Button btClick = new Button("Click");
        Button btOpen = new Button("OpenFile");
        hBox.getChildren().addAll(btNew, btClick, btOpen);
    // Now set up the button handlers
```

4.12 Lambda Expressions

```
  btnNew.setOnAction((ActionEvent e) ->
       System.out.println("Clicked on by " +
              e.getSource.toString());
  );

  btnClick.setOnAction(e -> ButtonResource.incrCtr());

  btnOpen.setOnAction(() -> {
     File file = new File ("EmployeeID.txt");
     Scanner input = new Scanner(file);
     while (input.hasNext()){ ID[i++] = input.nextInt(); }
  });

  …// load hBox into scene, scene into stage, and show()
}
```

Adapted From: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed. Toronto, ON: Pearson.* pp. 598-599.