ECE25100: OBJECT ORIENTED PROGRAMMING

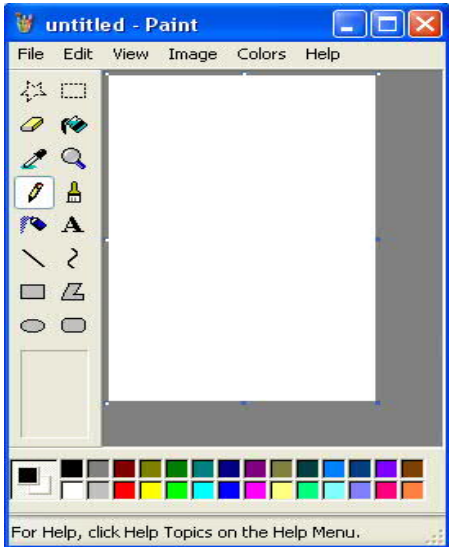
Note 9 _ User Interfaces

Instructor: Xiaoli Yang

User Interface Terminology

- All applications require some kind of user interface which allows the user to interact with the underlying program/software.
- Most user interfaces have the ability to take-in information from the user and also to provide visual or audible information back to the user.
- Sometimes the interface has physical/hardware interactive components (e.g., ATM machine) and sometimes the components are software-based (e.g., menus, buttons, text fields).

User Interface Terminology



- We will consider software-based user interfaces.
- In all our programs, we did not get to interact much at all with it, except to say "Go!" by running it.
- Something was fundamentally common :
  - they all had underlying classes representing objects that were specific to the application (e.g., the Team and League classes)
  - the testing code (which was our "user interface") always made use of these underlying objects.

User Interface Terminology

- A Model:
  - Consists of all classes representing the "business logic" part of the application.
  - represents the underlying system on which a user interface is attached.
  - is developed separately from the user interface.
  - should not assume any knowledge about the user interface.

User Interface Terminology

- The User Interface:
  - is "attached to" a model, so we often develop the model first
  - handles user interaction and does NOT deal with the business logic
  - uses" the model classes" methods
  - often causes the model to change according to user interaction and these model changes are often reflected back on the user interface
- A Graphical User Interface (GUI):
  - is a user interface that has one or more windows
  - is often preferred over text-based interfaces

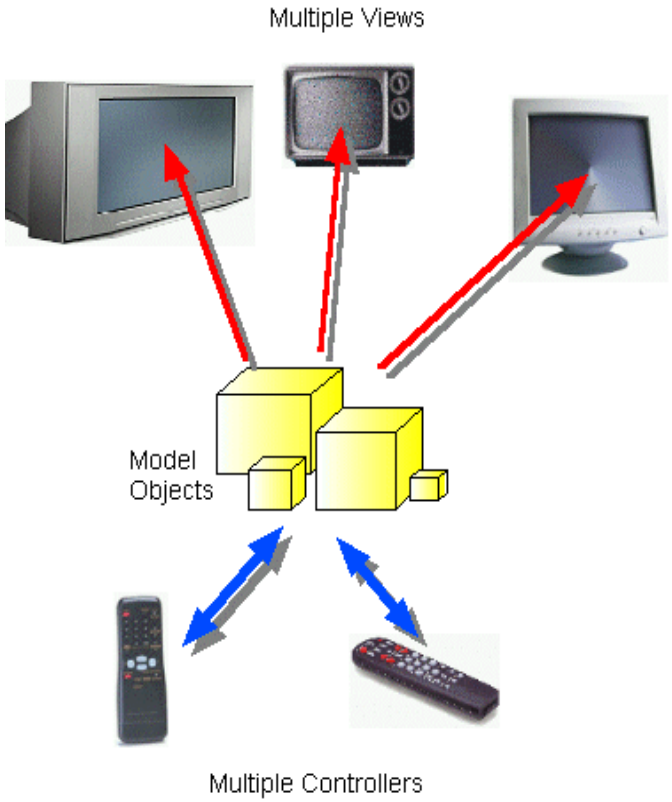
User Interface Terminology

- it is important to understand that there should always be a separation in your code between the model classes and the user interface classes.
- Such a separation allows you to share the same model classes with different interfaces.

User Interface Terminology

- We often split up the user interface as well into two separate pieces called the view and the controller:
- A View is:
  - the portion of the user interface code that specifies how the model is shown visually (i.e., its appearance).
- A Controller is:
  - the portion of the user interface code that specifies how the model interacts with the view (i.e., its behavior).
  - code that serves as a "mediator" between the model and the view.

User Interface Terminology



- It is ALWAYS a good idea to separate the model, view and controller (MVC):
  - code is cleaner and easier to follow when the view and controller are separated
  - we may want to have multiple views or controllers on the same model.
  - we may want to have multiple models for the same user interface

A Simple Text-Based User Interface

- Let us now look at an example of how to separate our model from our user interface. We will create an application that allows us to maintain a small database to store the DVDs that we own. We must think of what we want to be able to do with the application. These are the requirements:
  - We want to keep the DVD's title, year and duration (i.e., how long the DVD lasts) when played (e.g., 120 minutes).
  - We should be able to add new DVDs to our collection as well as remove them (if we lose or sell them).
  - We should be able to list our DVDs sorted by title.

A Simple Text-Based User Interface

- So, that's it. It will be a simple application. Let us begin by creating the model.
- What objects do we need to define ?
  - a DVD object (maintains title, year, duration)
  - an object to represent the collection of DVDs
- We can make a DVD collection class to list, add, or remove DVDs.
- Example: DVDCollectionTest

A Simple Text-Based User Interface

- Let us now make a user interface for this model. We should make a new class for this. We will make a simple text-based interface. It should perhaps bring up a menu and repeatedly prompt for a user choice. Here is what we will make appear on the screen: Example: DVDUI1

```
Welcome to the Dvd Collection User Interface
-------------------------------------------

1.  Add DVD
2.  Delete DVD
3.  List DVDs
4.  Exit

Please make a selection:
```

- So we now have:
  - The Model: which is the DVD collection class and the DVD class
  - The View: which is the menu system

A Simple Text-Based User Interface

- Improvements:
  - If the user makes an invalid selection, we print an error message out. This is considered to be our main menu. Once we make a selection and the action has been performed, this menu will be displayed again. Here is the user interface process:
    1. Display a menu with choices
    2. Wait for the user to make a selection
    3. Perform what needs to be done from that selection
       1. possibly more input is required
       2. the user may want to quit the program
       3. computations may be made, output may be displayed
    4. Go back up to step 1.
- Example: DVDUI2

Graphical User Interfaces

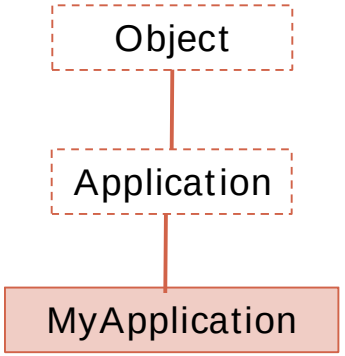
- We will develop JAVA applications that bring up windows for user to interact with.
- How do we develop our own Graphical User Interfaces to represent main windows for our JAVA applications.
- We learn to use the JavaFX package

JavaFX

- It is a software platform that allows us to create desktop applications as well as rich internet applications that can run across a variety of devices and web browsers.
- It supports desktop computers and web browsers on Microsoft Windows, Linux, and Mac OS X.

JavaFX

- When using JavaFX, all of our main windows for our applications will be instances of the Application class (i.e., we will extend that class) which is in the javafx package.
- To launch (i.e., start) the application, we call a launch() method from our main method.
- Upon startup, the application will call a start() method. Typically, that method will create and display a window of some sort.



JavaFX

- Here is a template of the bare minimum that you need to start a JavaFX application.

```
import javafx.application.Application;
import javafx.stage.Stage;
public class MyApplication extends Application {
        // Called automatically to start the application (you must write this)
        public void start(Stage primaryStage) { ... }
        public static void main(String[] args) {
                launch(args);// Get everything going (call this exactly once)
        }
}
```

JavaFX

- You may also write init() and stop() methods.
- The init() method is automatically called by the application to initialize things before the application starts. You may write code there to perform calculations, read file data, configure things, etc.
- When the application terminates (usually as a result of the user closing the window), the application will automatically call a stop() method, which you may write. This could be used to close files, release resources, notify other applications, etc.
- If ever you want to quit the application at any time within your code, you should use Platform.exit(), which will also call the stop() method before the application quits.

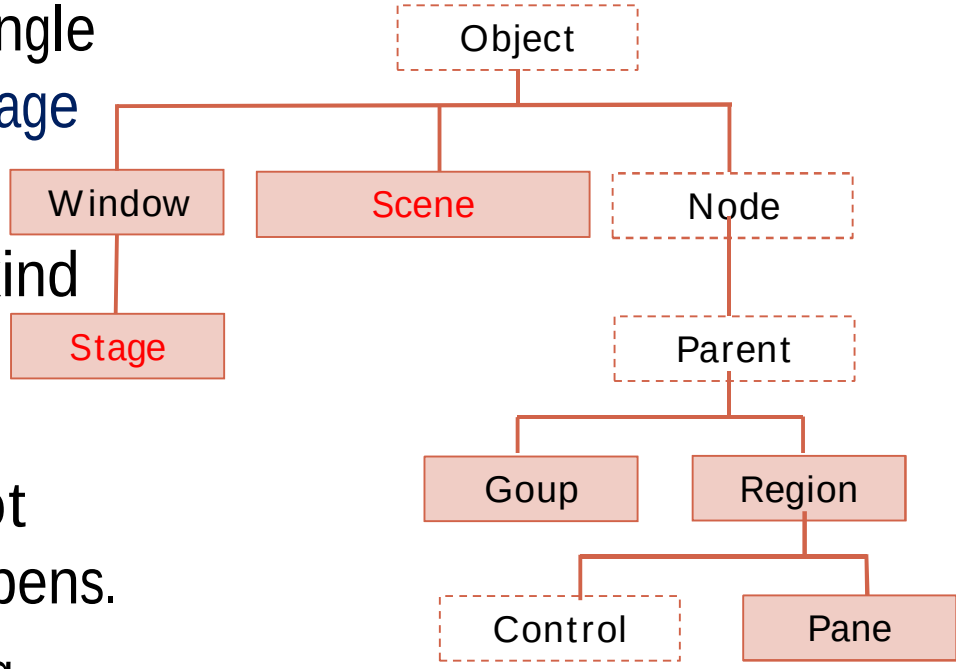
JavaFX

- Here is an expanded (optional) template:

```
import javafx.application.Application;
import javafx.stage.Stage;
public class MyApplication extends Application {
    // Initialize things before the app starts (optional)
    public void init() { ... }
    public void start(Stage primaryStage) { ... }
    // Clean up things just before the app stops (optional)
    public void stop() { ... }
    public static void main(String[] args) {
            launch(args);
    }
}
```

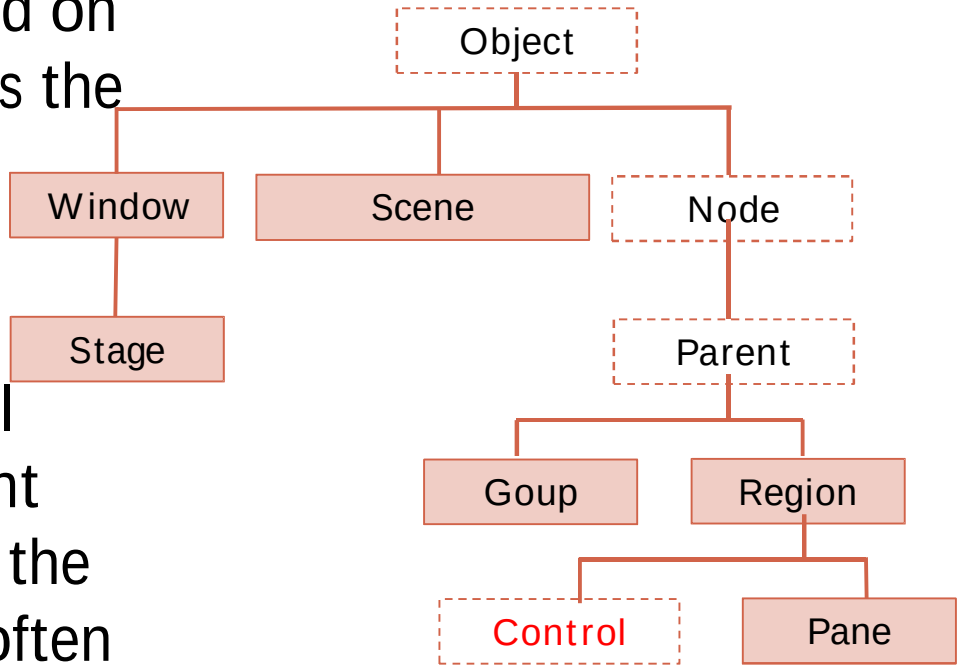
JavaFX

- The start() method has a single parameter called primaryStage which is a Stage object.
- A Stage object is a special kind of Window object. The primaryStage is set up automatically and is the root upon which everything happens.
- In order to make something visible, we must create a Scene object. This is where we define what the application will look like.



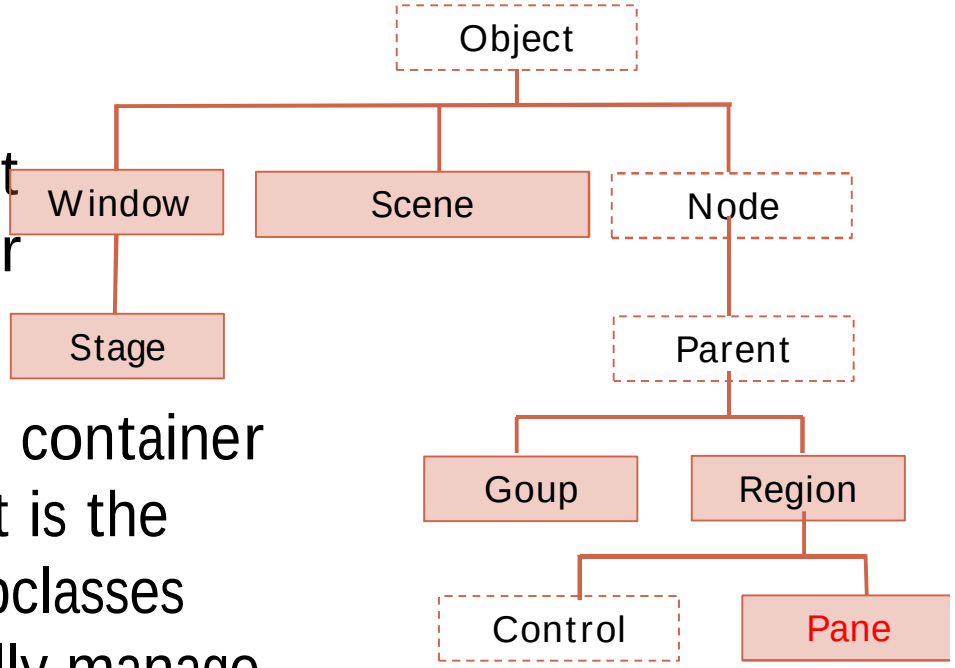
JavaFX

- A window component is an object with a visual representation that is placed on a window and usually allows the user to interact with it.
- In JavaFX, typical window components (e.g., buttons, textFields, lists) are Control objects ... and they represent the various components of the window. Components are often grouped together, much like adding elements to an array.



JavaFX

- A container is an object that contains components and/or other containers.
- The most easily understood container in JavaFX is the Pane class. It is the base class for its various subclasses that are used to automatically manage the layout of the window components. Its variety of subclasses allow the components to be resized and repositioned automatically, while maintaining a particular arrangement/layout on the window.



JavaFX

- The following code creates a simple JavaFX window. You can use this program as a template for all of your window-based applications:

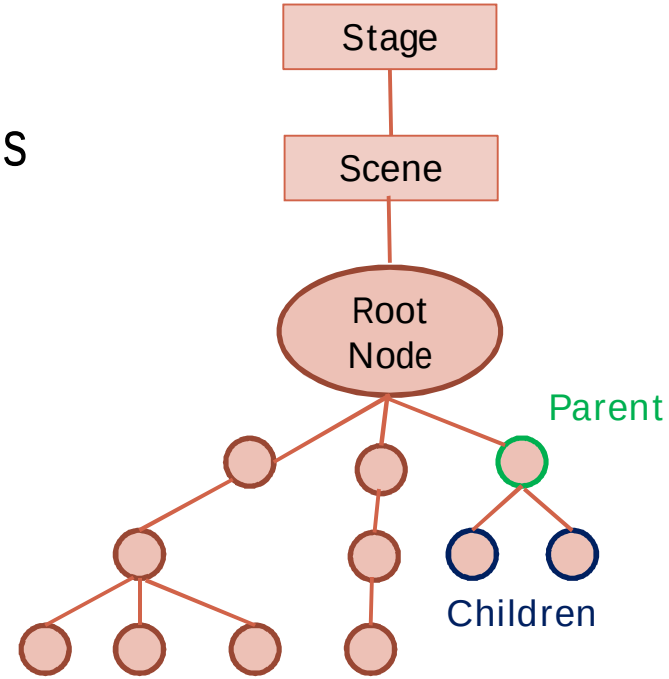
```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;
public class MyApplication extends Application {
    public void start(Stage primaryStage) {
            Scene scene = new Scene(new Pane(), 300, 100); // Set window size
            primaryStage.setTitle("My Window"); // Set window title
            primaryStage.setScene(scene);
            primaryStage.show(); // Show window
   }
   public static void main(String[] args) {
            launch(args); // Initialize/start
 } }
```

JavaFX

- Steps creating your own JavaFX application that uses a main window:
  1. Create a new class (separate from model classes) to represent your application.
  2. Have this class extend Application. Make sure to import the packages shown at the top of the program.
  3. The start() method is automatically called from the launch() method to initialize and show the window. Normally, we set the size of the window as well as the title of the window within this start() method.

Components and Containers

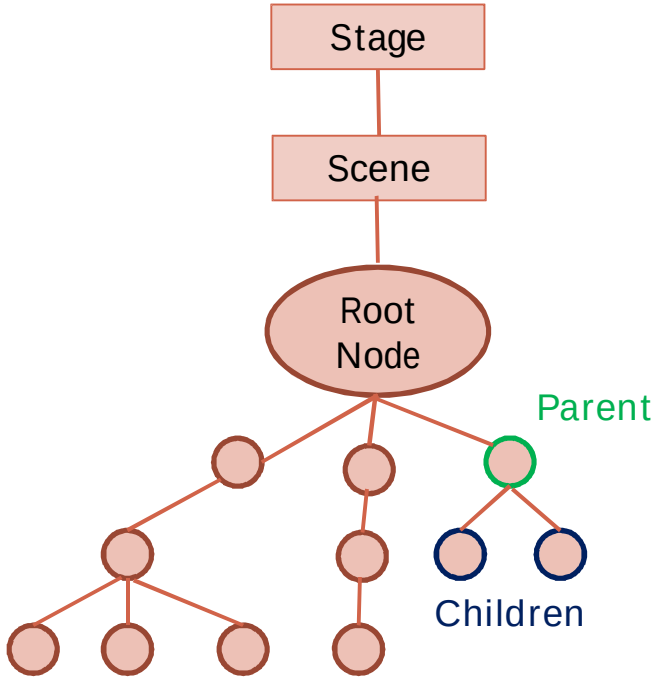
- The scene is arranged as a tree-like graph containing Node objects, which is called the scene graph.
- The scene graph begins with a root node. The nodes of the graph are basically the components of the window as well as the layout regions and the components within them. If node c is contained within another node p, then c is called a child of p and p is the parent of c.



Components and Containers

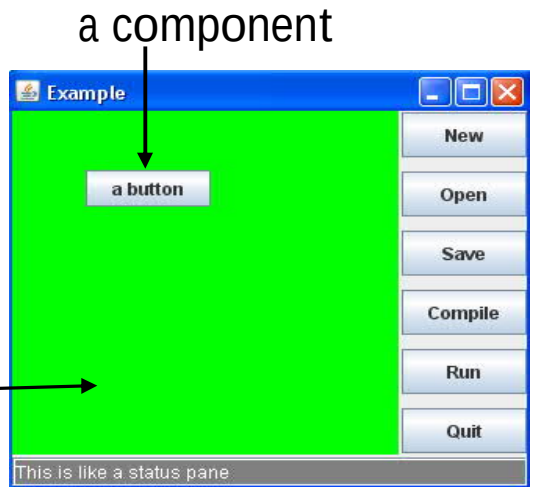
- All window components actually keep pointers to their parent. Parents of nested components are stored recursively:

```
Control aComponent;
Parent p;
Parent parentOfParent;
aComponent = ... ;
p = aComponent.getParent();
parentOfParent = p.getParent();
```



Components and Containers

- Parents keep pointers to their children and we can access these using the parent's getChildren() method:

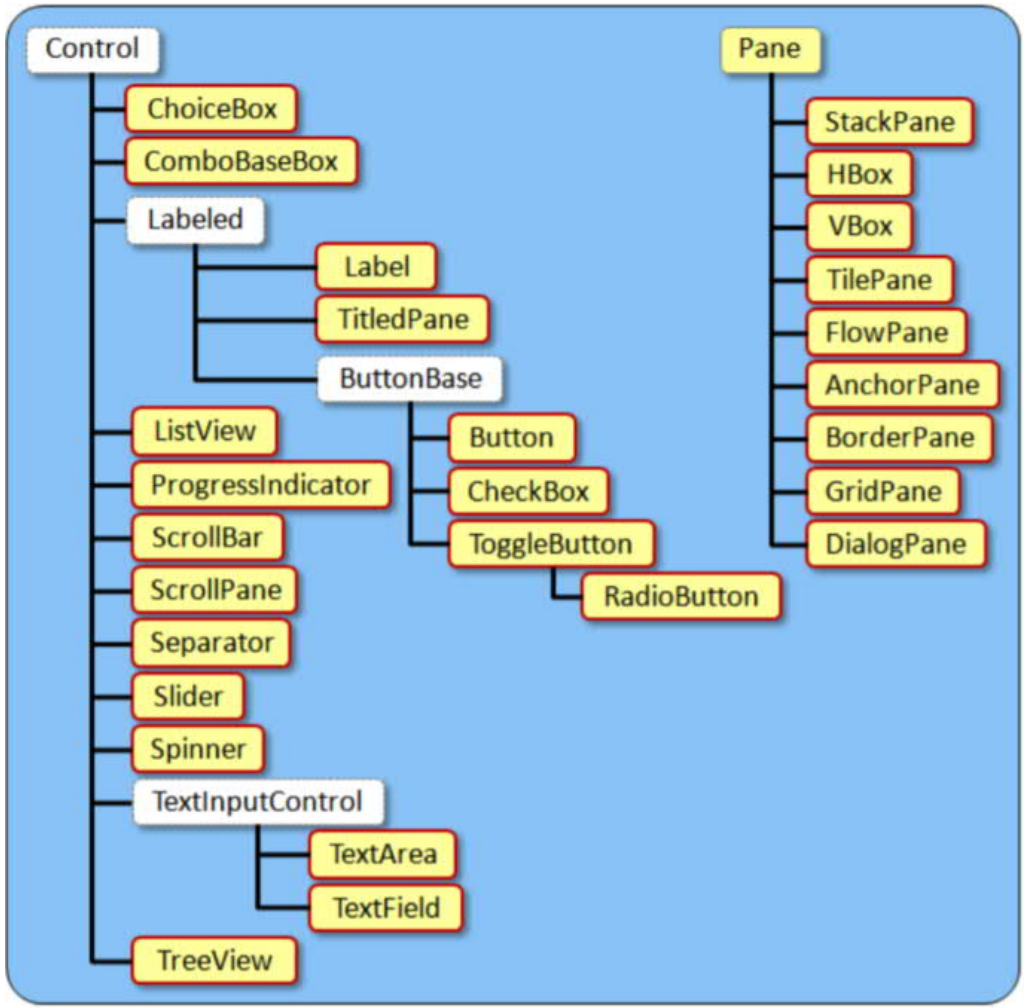



```
Parent aParent = ... ;
ObservableList<Node> children= aParent.getChildren();
Node c1 = children.get(0);
Node c2 = children.get(1);
Node c3 = children.get(2);
```

- The class hierarchy shown earlier shows the Control and Pane classes. Each of these classes has many subclasses representing various window components and layout managers.

Components and Containers

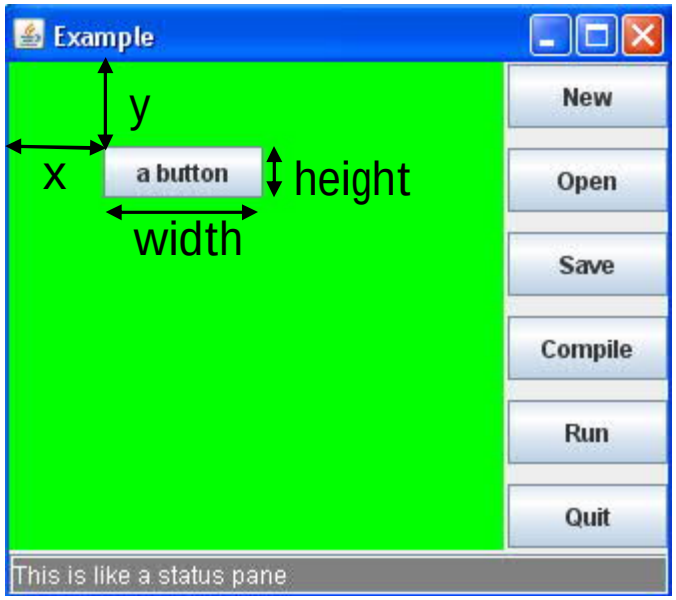
- Hierarchies with some of the commonly used classes.



Components and Containers

All JComponents have the following state:

1. Location, Width and Height (in pixels):
   - location is an (x,y) coordinate with respect to top left corner of parent container:



   - We can access/modify this information from the component at any time.

Components and Containers

2. Text on Component

- We can create a button with some text on it:

```
new Button(“AButton");
or Button b = new Button(" AButton");
```

- You can change the text at any time by calling the setText() method, but depending on the size of the button, the text could be cut off:

```
b.setText(“A Very Long Long Button");
```

Components and Containers

3. Alignment

– You can even adjust the alignment of the text by using the setAlignment() method as follows:

```
b.setAlignment(Pos.CENTER);
```


```
b.setAlignment(Pos.CENTER_LEFT);
```


```
b.setAlignment(Pos.CENTER_RIGHT);
```

– This will require us to import the Pos class at the top of our code:
import javafx.geometry.Pos;

Components and Containers

4. Add an image to a component

```
Image anImage = new
Image(getClass().getResourceAsStream(“dog.gif"));
Button b = new Button(“AButton", new ImageView(anImage));
```

– This will require us to import these classes at the top of our code:

```
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
```

Components and Containers

4. Add an image to a component

So you will want to have the image file in the src folder that contains your code so that it can be found upon startup. If you don't want the text ... just the image, then you can use the setGraphic() method as follows:

```
Image anImage = new Image(getClass().getResourceAsStream("dog.gif"));
Button b = new Button();
b.setGraphic(new ImageView(anImage));
```

Components and Containers

5. Style

Each component also has a variety of style settings that we can set. We will just look at three here: (1) text color, (2) background color and (3) font. We can use the setStyle() method to set any of these styles as follows:


```
b.setStyle("-fx-font: 22 arial; -fx-base: rgb(170,0,0); -fx-text-fill: rgb(255,255,255);");
```

Notice that the method takes a string parameter. In that string we specify a sequence of style tags followed by their values.

The -fx-font: tag allows us to specify a font size followed by a font type.

The -fx-base: and -fx-text-fill: tags allow us to specify the background color and text color, respectively.

Components and Containers

5. Style

In the case of -fx-base:, the button's nice beveled border remains intact. We could instead use -fx-background-color: and then the border will also be colored over with the background color as shown here.

In each case, we specify the color as the amount of red, green and blue (from 0 to 255) that we want. There are also some predefined colors that we can use as well to simplify, but these predefined color names may not be what we prefer:


```
b.setStyle("-fx-font: 22 arial; -fx-base: RED; -fx-text-fill: WHITE;");
```

Components and Containers

6. Ability to be Enabled/Disabled:

– We can enable and disable components at any time on our program.

```
b.setDisable(false);
b.setDisable(true);
```

– While a component is disabled, it is "greyed out", but cannot be used.

Components and Containers

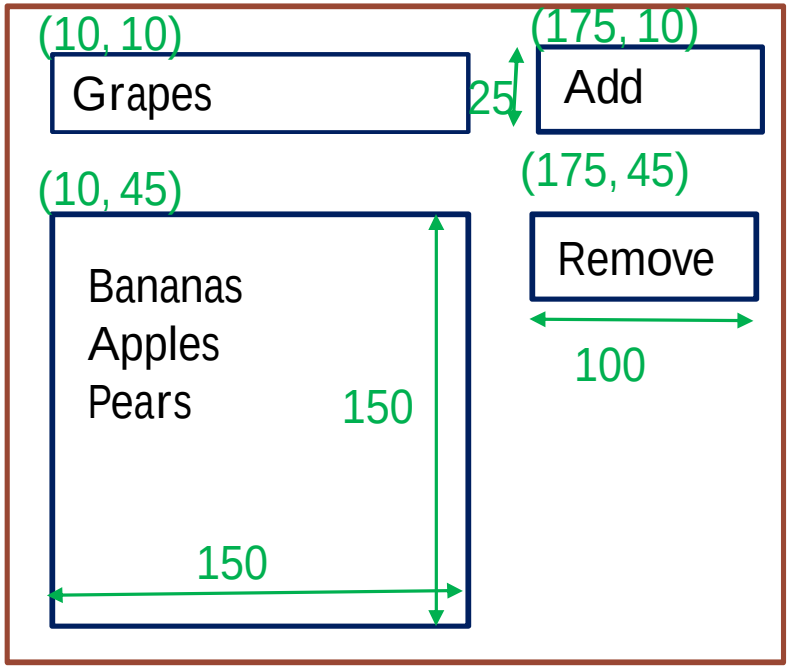
7. Ability to be Hidden/Shown:

```
b.setVisible(false);
```

– By default, all components are enabled when created. At any time, we can also hide a component completely as follows: b.setVisible(false); This will simply make the component invisible ... unable to be seen nor controlled by the user.

Example

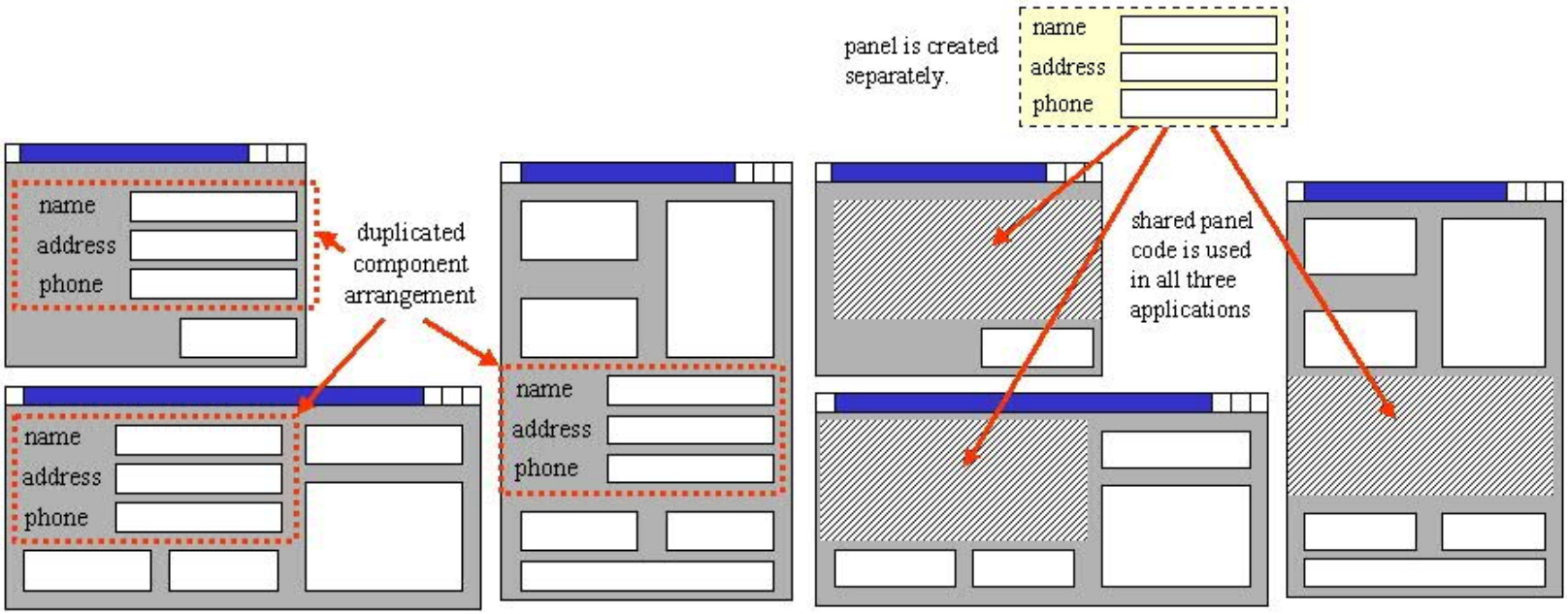
- The top/left component is a TextField. The bottom/left component is a ListView and the right two components are Button objects to add and remove items to the list.
- Example: FruitListApp



Grouping Components Together

- It is often the case that an arrangement of components may be similar (or duplicated) within different windows. For example, an application may require a name, address and phone number to be entered at different times in different windows
- It is a good idea to share component layouts among the similar windows within an application so that the amount of code you write is reduced. To do this, we often lay out components onto a Pane and then place the Pane on our window. We can place the pane on many different windows with one line of code ... this can greatly reduce the amount of GUI code that you have to write.

Grouping Components Together



Grouping Components Together

- You will often want to create separate Pane objects to contain groups of components so that you can move them around (as a group) to different parts of a window or even be shared between different windows.
- The code to do this simply involves creating our Pane with its appropriate component arrangement and then adding the Pane to the window. Pane objects are added to a window just like any other objects. So, we can have a Pane within another Pane.
- Example: AddressPane->OneApp and TwoApp

User Interface Extensions

- Layout panes that automatically lay out and resize the components on the window.
- JAVA was developed for the internet and JAVA applications were initially meant to run as applets within an internet browser.
- Since browsers are often resized, it is often desirable to allow an application's components to be rearranged so that they ALL fit on the browser window at all times.
- In fact, JAVA FX provides a mechanism called a Layout Pane that allows the automatic arrangement (i.e., "laying out") of the components of an application as the window is resized.

User Interface Extensions

- Why should we use a layout pane ?
  - we would not have to compute locations and sizes for our components
  - our components will resize automatically when the window is resized
  - our interface will appear "nicely" on all platforms
- In JAVA FX, each layout defines methods necessary for a class to be able to arrange Components within a Container.

User Interface Extensions

- We will discuss the following, although there are more available:
  - FlowPane
  - BorderPane
  - HBox,VBox
  - GridPane
- Layout panes are "set" for a pane using the setLayout() method. If set to null, then no layout manager is used .

FlowPane

- The simplest layout pane is the FlowPane. It is commonly used to arrange just a few components on a pane.
- With this pane, components (e.g., buttons, text fields, etc..) are arranged horizontally from left to right. If no space remains on the current line, components flow (or wrap around) to the next "line".
- The height of each line is the maximum height of any component on that line.
- By default, components are centered horizontally on each line, but this can be changed.
- Example: FlowPaneExample

HBox or VBox

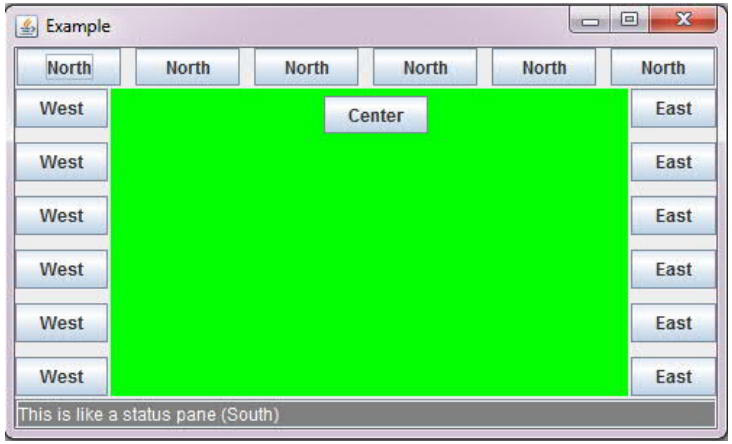
- The HBox and VBox layouts are similar to the FlowPane in that it arranges components one after another, either horizontally or vertically.
- However, it does not have a wrap-around effect. Instead, any components that do not fit on the line are simply not shown.
- If we want to lay the components out horizontally, we use new HBox() as our pane. To lay the components out vertically, we use new VBox().
- As with the FlowPane, we can specify the Insets as well as spacing between components:

```
aPane.setPadding(new Insets(10));
aPane.setSpacing(5);
```

- Example: HBoxExample

BorderPane

- The BorderPane is a very useful layout. Instead of re-arranging components, it allows you to place components at one of five anchored positions on the window (i.e., top, left, bottom, right or center).
- As the window resizes, components stay "anchored" to the side of the window or to its center. The components will grow accordingly.

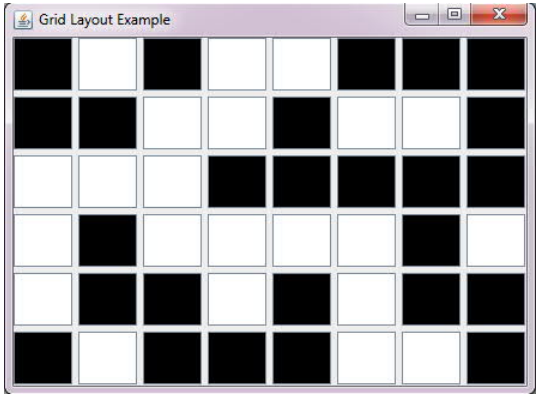


BorderPane

- You may place at most one component in each of the 5 anchored positions. But this one component may be a container such as another Pane that contains other components inside of it.
- Typically, you do NOT place a component in each of the 5 areas, but choose just a few of the areas.
- Example: BorderPanelExample

GridPane

- A GridPane is excellent for arranging a 2-dimensional grid of components (such as buttons on a keypad). It automatically aligns the components neatly into rows and columns. Typically, the components are all of the same size, however you can add different sized components as well. Components are added by specifying their column and row in the grid.



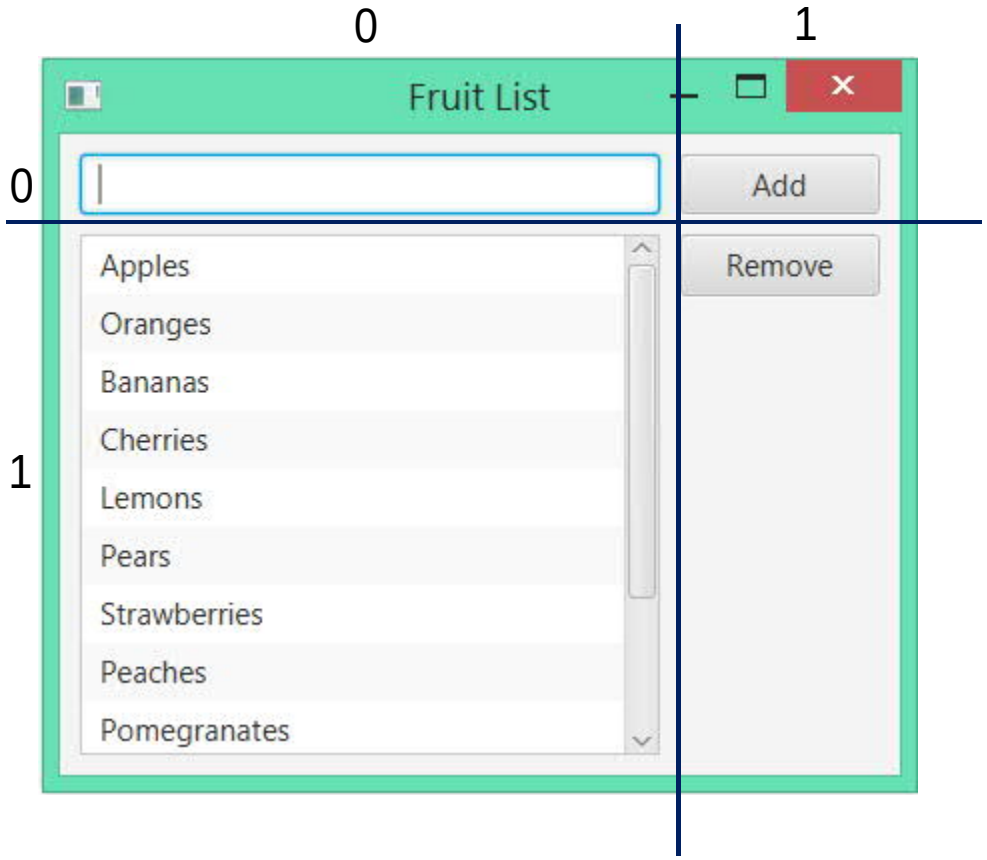
GridPane_Example 1

- JAVA determines the number of rows and columns to use for the grid by considering all of the row and col parameters that you use in these add() method calls.
- The setHgap() and setVgap() specify the horizontal and vertical margin (in pixels) between components and the setPadding() allows you to specify margins around the outside of the pane
- Example: GridPaneExample

GridPane_Example 2

- The GridPane can also be the most flexible of all the layout panes.
- It allows you to be very specific in the placement of all components and to indicate exactly how each component is to resize as the window shrinks or grows. However, due to the flexibility of this layout, it is more complicated to use than any of the other layouts.
- The GridPane can also arrange components in a non-uniform grid where the grid rows and columns are not explicitly defined. It may be non-uniform in that the rows and columns may have variable heights and widths. Also, each component can occupy (i.e., span) multiple rows and columns.

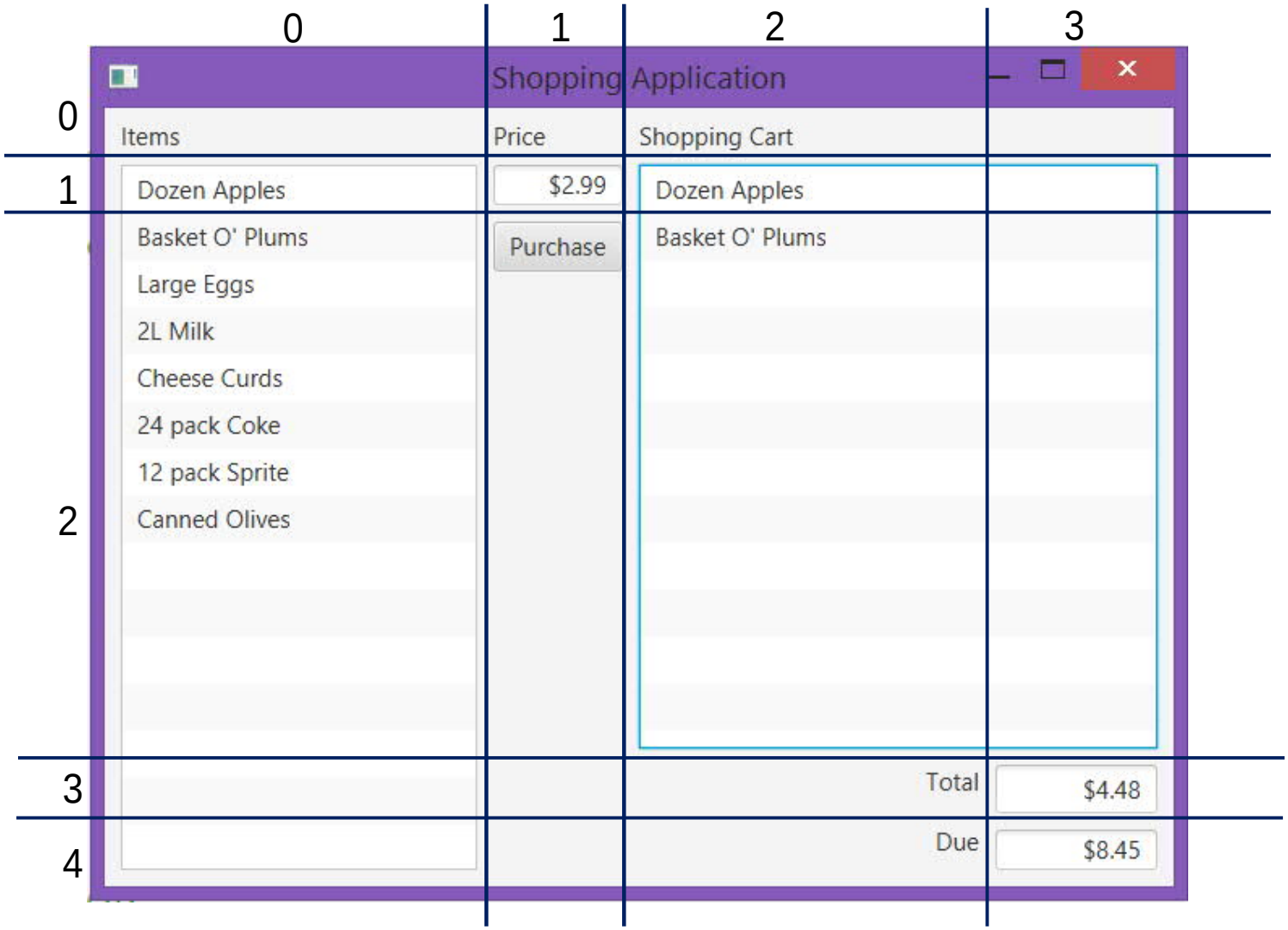
GridPane_Example 2



GridPane_Example 2

- Version 1:
  - We can lay out all the components by specifying the grid location (i.e., column and row) that each component lies in when we add it to the pane:
- Version 2:
  - the Remove button is centered vertically. We can change this by using: aPane.setValignment(removeButton, VPos.TOP);
  - We need to make the buttons the same size. The simplest way to do this is to specify the width and height that we want the buttons to have.
  - we set the fruitlist's preferred width and height to the largest possible values.
- Version 3:
  - Adjust the spacing around the components. We can use the setMargin() method for our components.

GridPane_Example 3



GridPane_Example 3

- Version 1:

add(Node child, int columnIndex, int rowIndex, int colspan, int rowspan)	Adds a child to the gridpane at the specified column,row position and spans.

- Version 2:
  - aPane.setValignment(purchaseButton,VPos.TOP);
  - priceField.setMinHeight(25);
  - priceField.setMinWidth(80);
  - totalField.setMinHeight(25);
  - totalField.setMinWidth(80);
  - dueField.setMinHeight(25);
  - dueField.setMinWidth(80);
  - purchaseButton.setMinHeight(25);
  - purchaseButton.setMinWidth(80);

GridPane_Example 3

- Version 3: In order to get proper resizing behavior, we will need to specify how we want each grid column and row to grow. To do this, we will make use of the ColumnConstraints and RowConstraints objects.
- To make the Items list grow, we need column 0 to grow. To make the Shopping Cart list grow, we need either column 2 or 3 to grow (or both). However, notice that column 3 contains TextFields ... which we likely do not want to grow, since the price field is wide enough to show any reasonable amount already. So, we need columns 0 and 2 to grow.

GridPane_Example 3

- So, Here is the code that we need to use:
  - ColumnConstraints col0 = new ColumnConstraints(50, 300, Integer.MAX_VALUE);
  - ColumnConstraints col1 = new ColumnConstraints(100);
  - ColumnConstraints col2 = new ColumnConstraints(50, 300, Integer.MAX_VALUE);
  - ColumnConstraints col3 = new ColumnConstraints(100);
  - col0.setHgrow(Priority.ALWAYS);
  - col2.setHgrow(Priority.ALWAYS);
  - aPane.getColumnConstraints().addAll(col0, col1, col2, col3);

GridPane_Example 3

- The ColumnConstraints object allows you to pass in 1 parameter (i.e., the width of the column) or you can supply 3 parameters which specify the minimum column width, the preferred column width and the maximum column width. For the columns that will grow, we can set the minimum to 50 so that the lists don't get too narrow. The preferred size of 300 is reasonable in size. The maximum value is set to the largest Integer value (around 2 billion). It tells JavaFX to allow it to grow as large as possible without limit.
- The setHgrow() method allows us to specify that we want certain columns to grow, otherwise they will not. Finally, we add these column constraints to the pane.

GridPane_Example 3

- In addition, we need to set the RowConstraints in the same manner. Here is the code:
  - RowConstraints row0 = new RowConstraints(25);
  - RowConstraints row1 = new RowConstraints(35);
  - RowConstraints row2 = new RowConstraints(50, 300, Integer.MAX_VALUE);
  - RowConstraints row3 = new RowConstraints(40);
  - RowConstraints row4 = new RowConstraints(35);
  - row2.setVgrow(Priority.ALWAYS);
  - aPane.getRowConstraints().addAll(row0, row1, row2, row3, row4);
- We need to specify some margins around some of the components.

References

- Carleton University COMP 1005 and 1006:
  http://www.scs.carleton.ca/~lanthier/teaching/COMP1405/Notes/
  http://www.scs.carleton.ca/~lanthier/teaching/COMP1406/Notes/
- Armstrong Atlantic State University
  http://www.cs.armstrong.edu/liang/intro10e/
- Oracle Java tutorial:
  http://docs.oracle.com/javase/tutorial/
- MIT opencourseware
  https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/