5

UI Controls, Layouts, Charts, and CSS

Objectives

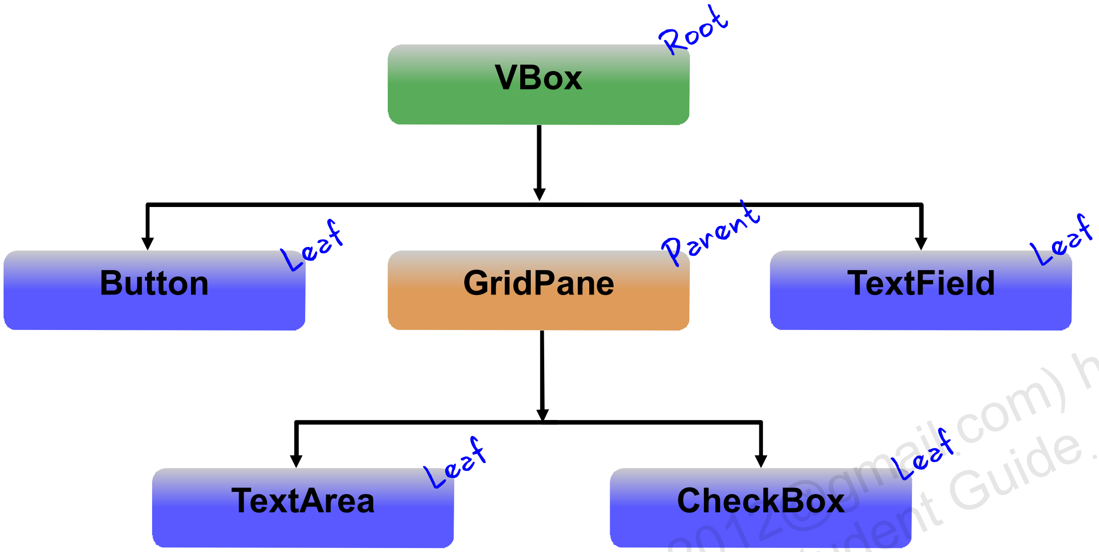
After completing this lesson, you should be able to:

- Relate UI components to the scene graph
- Describe and implement JavaFX UI components such as controls, images, shapes, and layout containers
- Use CSS
- Add events to JavaFX controls
- Create charts

Topics

- Relating UI components to the scene graph
- Describing and implementing JavaFX UI components such as controls, images, shapes, and layout containers
- Using CSS
- Adding events to JavaFX components
- Creating charts

JavaFX Scene Graph and UI Elements



Root
VBox

Leaf
Button

Parent
GridPane

Leaf
TextField

Leaf
TextArea

Leaf
CheckBox

The JavaFX UI components that are available through the API are built by using nodes in the scene graph. Therefore, the components can use the visually rich features of the JavaFX platform. Because UI controls from the `javafx.scene.control` package are all extensions of the Node class, they can be integrated with the scene graph rendering, animation, transformations, and animated transitions. Scene graphs can become much larger than the example shown in the slide, but the basic organization (that is, the way in which parent nodes contain child nodes) is a pattern that repeats in all applications.

In the graphic in the slide, VBox is the root element and is from the `javafx.scene.layout` package. Button and TextField are leaf elements from the Control class, and GridPane is a parent node from the `javafx.scene.layout` package with the children TextArea and CheckBox, both from the `javafx.scene.control` package.

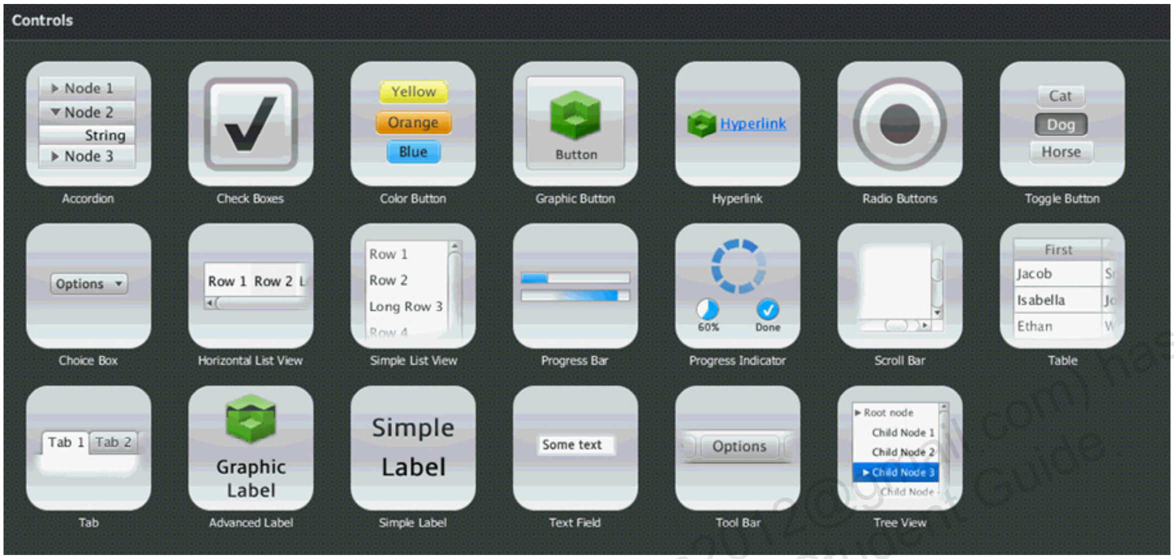
Throughout the rest of this lesson, you will see various applications of the scene graph. You will also learn how to manipulate nodes in the graph.

Topics

- Relating UI components to the scene graph
- Describing and implementing JavaFX UI components such as controls, images, shapes, and layout containers
- Using CSS
- Adding events to JavaFX components
- Creating charts

ORACLE

UI Controls

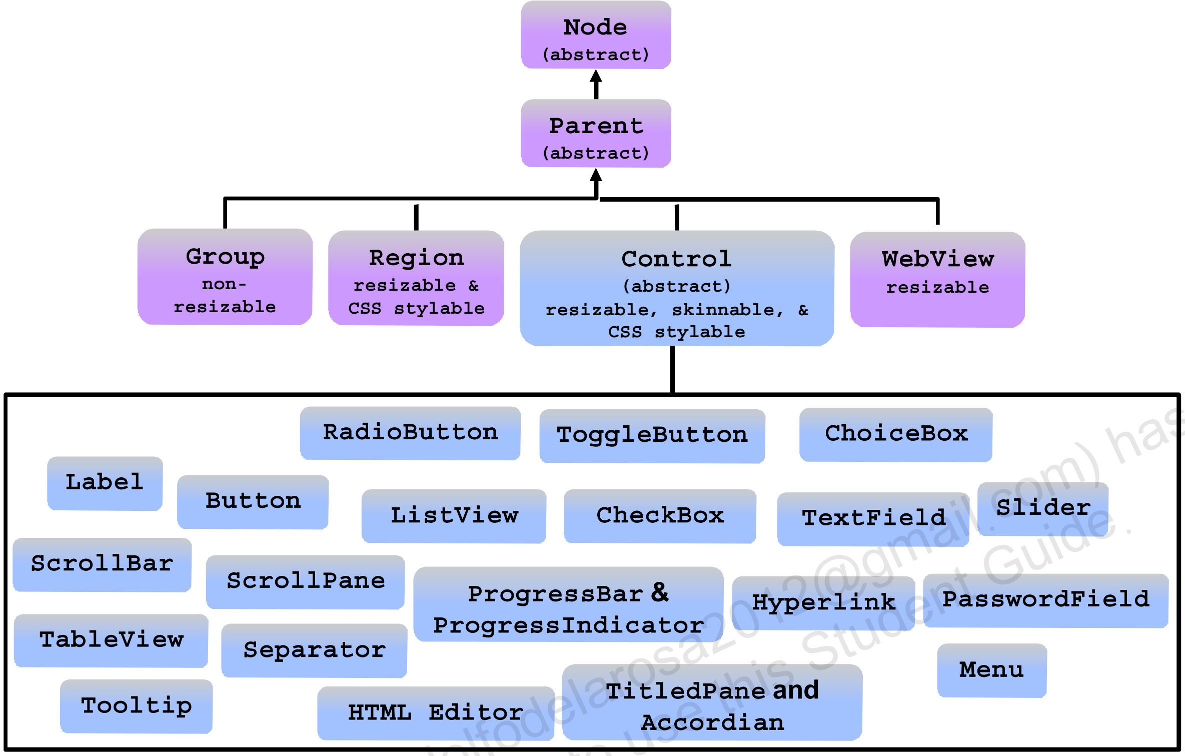


Some of the UI controls are:

- Label
- Button
- Radio Button
- Toggle Button
- Check Box
- Choice Box
- Text Field
- Password Field
- Scroll Bar
- Scroll Pane
- List View
- Table View
- Tree View
- Slider
- Progress Bar and Progress Indicator
- Hyperlink
- Tooltip
- HTML Editor
- Titled Pane and Accordion
- Menu
- Separator

See the Ensemble application for a complete list of UI controls.

Control Is a Node



Node
(abstract)

Parent
(abstract)

Group
non-resizable

Region
resizable &
CSS stylable

Control
(abstract)
resizable, skinnable, &
CSS stylable

WebView
resizable

RadioButton, ToggleButton, ChoiceBox, Label, Button, ListView, CheckBox, TextField, Slider, ScrollBar, ScrollPane, ProgressBar & ProgressIndicator, Hyperlink, PasswordField, TableView, Separator, Menu, Tooltip, HTML Editor, TitledPane and Accordian

The JavaFX user interface controls (*UI controls,* or just *controls*) are specialized nodes in the JavaFX scene graph especially suited for reuse in many different application contexts. They are designed to be highly customizable visually by designers and developers. They are designed to work well with layout mechanisms. Examples of prominent control nodes include Button, Label, ListView, and TextField. The graphic in the slide shows the available controls.

Because controls are nodes in the scene graph, they can be freely mixed with images, media, text, and basic geometric shapes, and they can be combined into groups.

The UI controls are resizable, skinnable, and CSS-stylable. Although writing new UI controls is not trivial, using and styling them are very easy—especially for web developers.

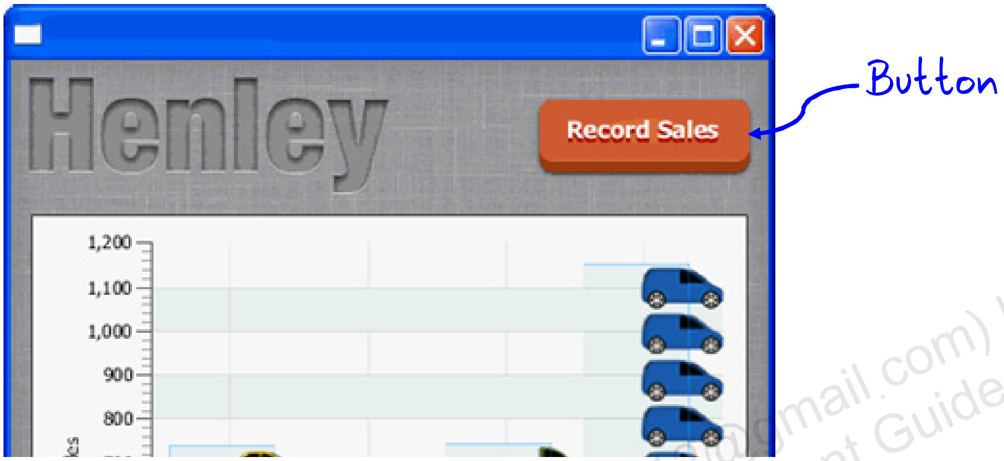
A control extends the Parent class and, as such, is not a leaf node. From the perspective of a developer or designer, the control can be thought of as if it were a leaf node in many cases. For example, the developer or designer can consider a button as if it were a rectangle or other simple leaf node.

Because a control is resizable, it is auto-sized to its preferred size on each scene graph pulse. Setting the width and height of the control does not affect its preferred size. When a control is used in a layout container, the layout constraints imposed on the control (or manually specified on the control) determine how it is positioned and sized.

The skin of a control can be changed at any time. Doing so marks the control as needing to be laid out because changing the skin is likely to change the preferred size of the control. If no skin is specified at the time that the control is created, a default CSS-based skin is provided for all of the built-in controls.

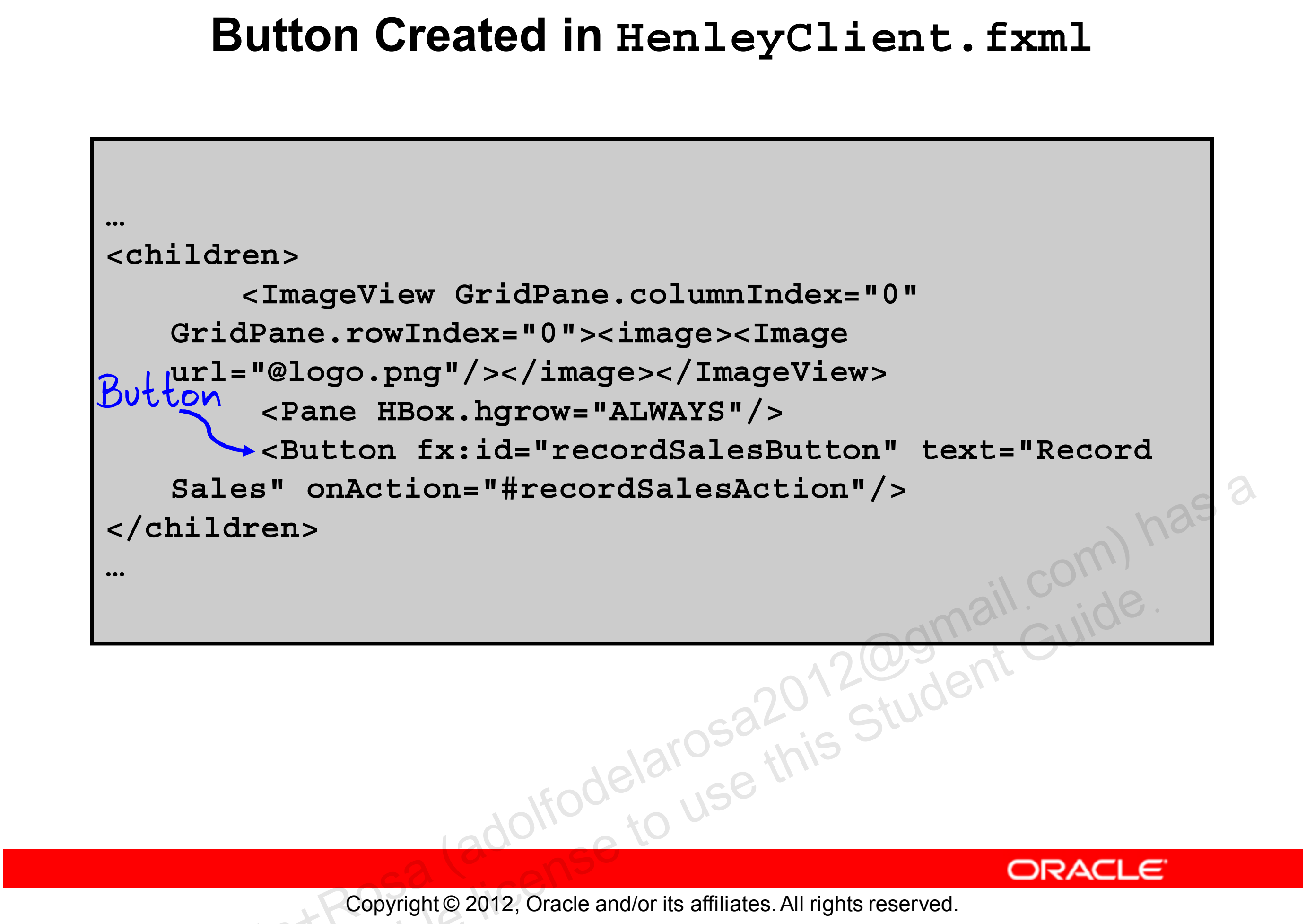
Each control may have an optional tooltip specified. The tooltip is a component that displays some (usually textual) information about a control when the cursor moves over the control for some period of time. As with other controls, a tooltip can be styled from CSS.

Henley Car Sales Button



UI Control: Example

The button is created in `HenleyClient.FXML`, and events are controlled in `HenleyClient.java`.



Button Created in HenleyClient.fxml

Button

```
…
<children>
       <ImageView GridPane.columnIndex="0"
   GridPane.rowIndex="0"><image><Image
   url="@logo.png"/></image></ImageView>
        <Pane HBox.hgrow="ALWAYS"/>
        <Button fx:id="recordSalesButton" text="Record
   Sales" onAction="#recordSalesAction"/>
</children>
…
```

In this example of an FXML application, the button is created in `HenleyClient.fxml`. This example represents the code for the button displayed in the slide titled “Henley Car Sales Button.”

Button Example in a Java Class

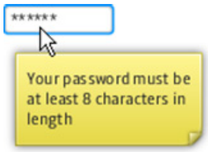
```
@Override
   public void start(Stage primaryStage) {
       primaryStage.setTitle("Hello World!");
       Button btn = new Button();
       btn.setText("Say 'Hello World'");
       btn.setOnAction(new EventHandler<ActionEvent>()
   {

           @Override
           public void handle(ActionEvent event) {
               System.out.println("Hello World!");
           }
       });
```

In this example of a JavaFX application, the button and event handles are created in the same class. Button is created using the `Button()` constructor. You can set the text for the button because the Button class inherits its properties and methods from the Labeled class.

This example represents the code for the button shown in the slide titled “Henley Car Sales Button.”

Tooltip

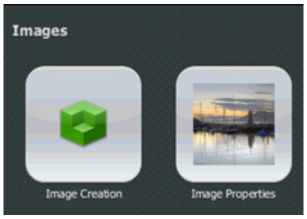
The Tooltip class represents a common UI component that is typically used to display additional information about the UI control. The tooltip is shown when the cursor is placed over the control.

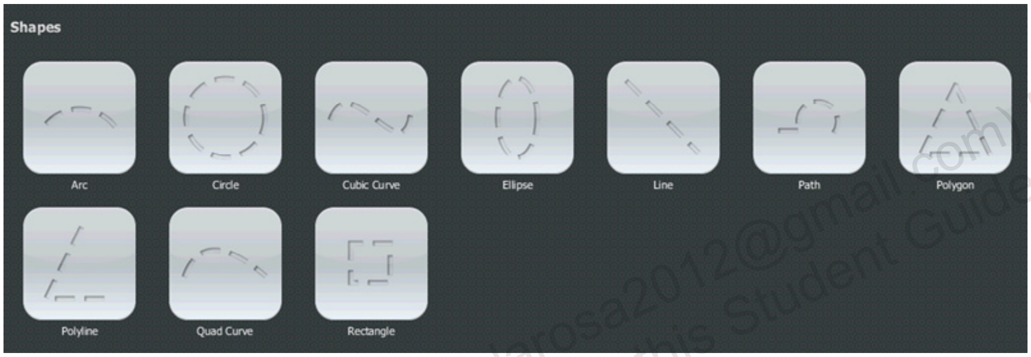


```
final PasswordField pf = new
PasswordField(); final Tooltip tooltip
= new Tooltip(); tooltip.setText(
    "\nYour password must be\n" + "at
    least 8 characters in length\n" +
);
pf.setTooltip(tooltip);
```

The tooltip can be set on any control by calling the `setTooltip()` method.

Images and Shapes

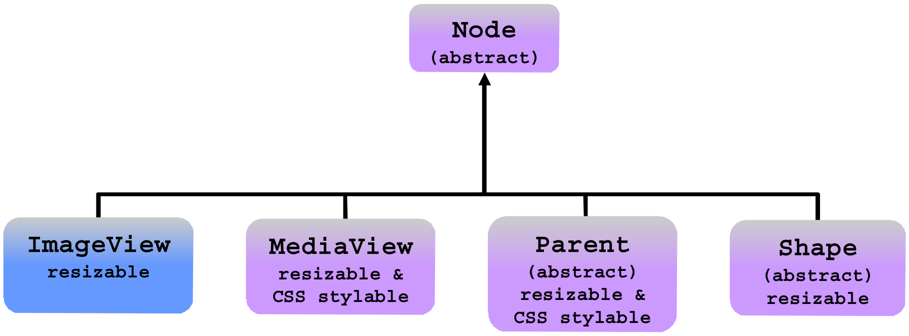




The `Image` class `javafx.scene.image` represents graphical images and is used for loading images from a specified URL. Images can be resized as they are loaded (for example, to reduce the amount of memory consumed by the image). The application can specify the quality of filtering used when scaling, and whether or not to preserve the original image's aspect ratio. Use `ImageView` to display images loaded with this class. The same `Image` instance can be displayed by multiple `ImageViews`.

The `Shape` class `javafx.scene.shape` is a set of 2D classes for defining and performing operations on objects related to two-dimensional geometry.

ImageView Is a Node



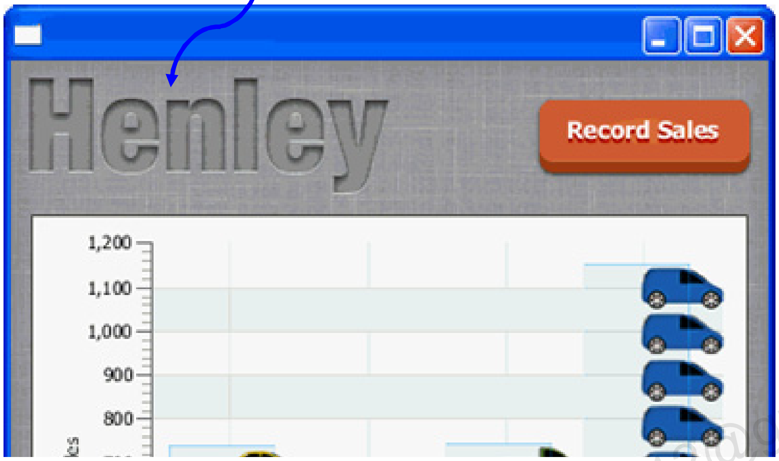
```
public class ImageView
extends Node
```

ImageView is a node used for painting images loaded with the Image class. This class allows resizing the displayed image (with or without preserving the original aspect ratio) and specifying a viewport into the source image for restricting the pixels displayed by this ImageView. Because the ImageView class is an extension of the Node class, you can apply visual effects, animations, and transformation to images in JavaFX.

The Image class is used to load images (synchronously or asynchronously). Image can be resized as it is loaded. The resizing can be performed with specified filtering quality and with the option of preserving the original aspect ratio of the image.

Henley Car Sales Image

Image



Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

The image is created in `HenleyClient.FXML`.

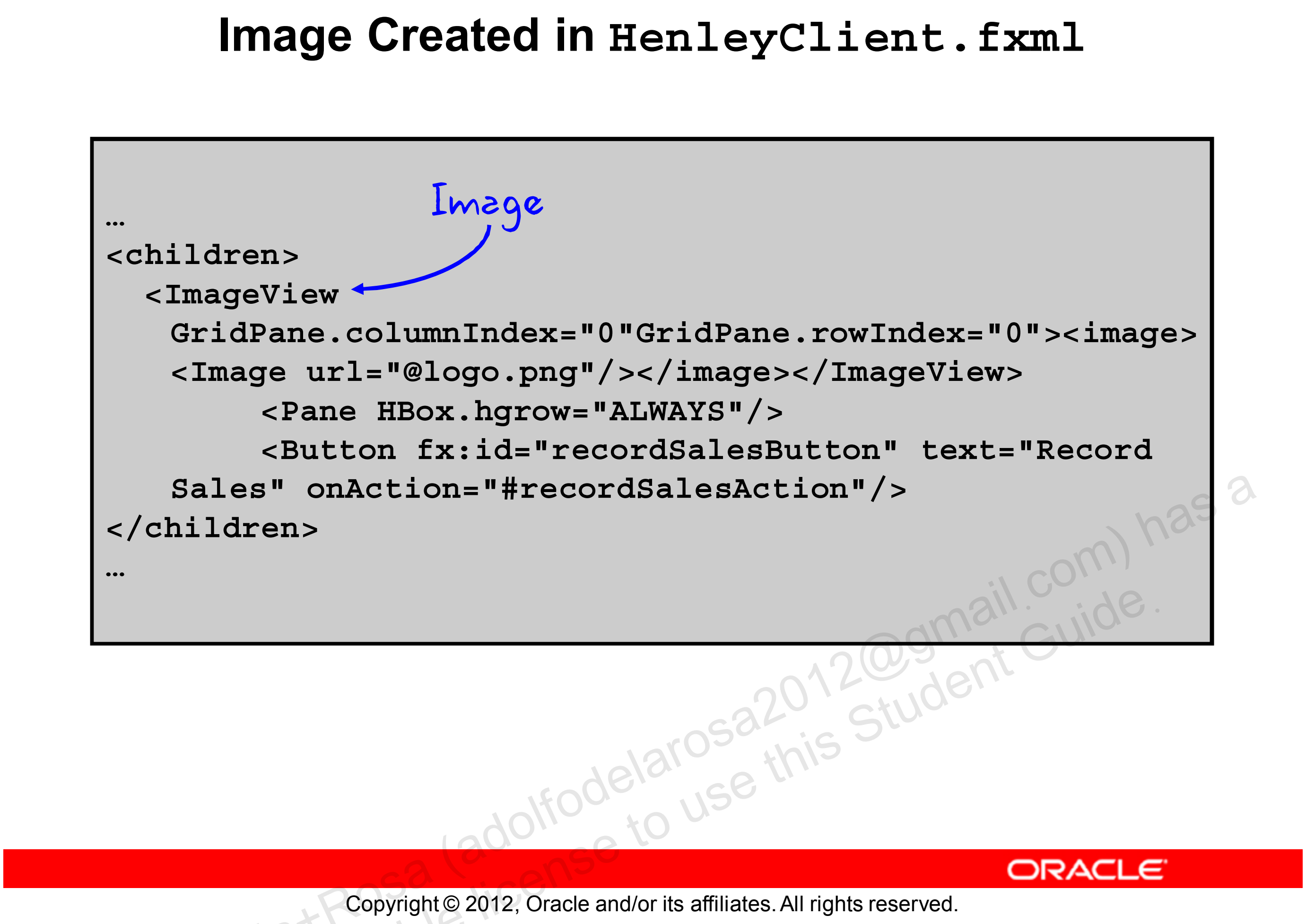


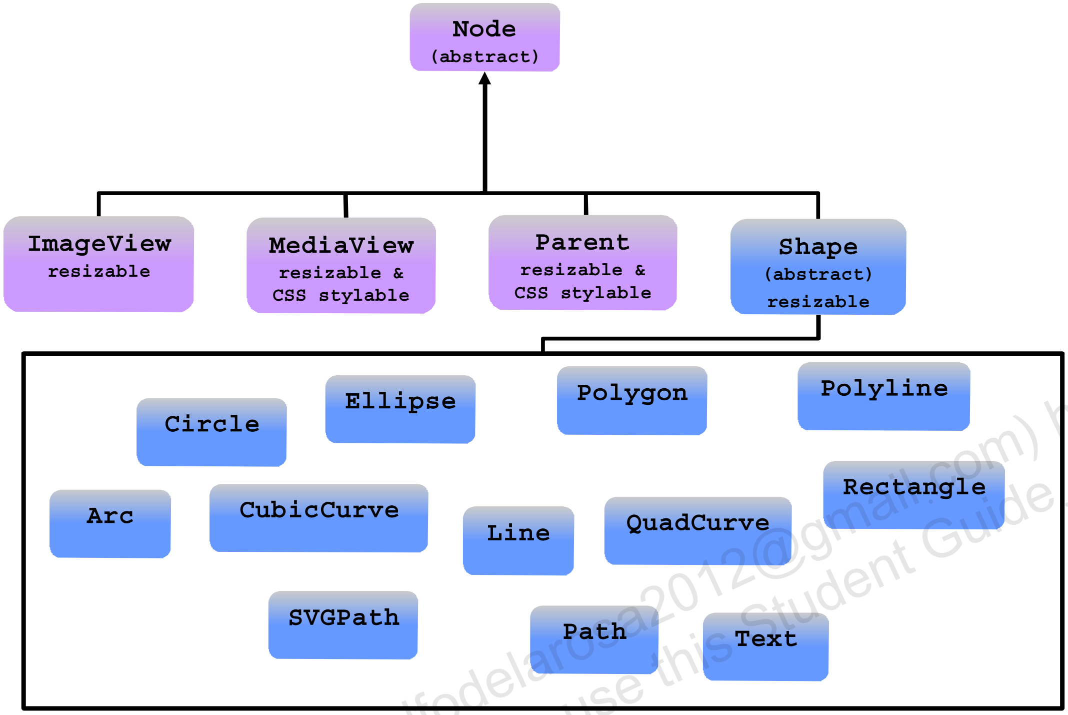
Image Created in `HenleyClient.fxml`

```
…
<children>
  <ImageView
    GridPane.columnIndex="0"GridPane.rowIndex="0"><image>
    <Image url="@logo.png"/></image></ImageView>
         <Pane HBox.hgrow="ALWAYS"/>
         <Button fx:id="recordSalesButton" text="Record
    Sales" onAction="#recordSalesAction"/>
</children>
…
```

The code example `HenleyClient.fxml` shows the `ImageView` for the Henley logo, `logo.png`.

This example represents the code for the button shown in the slide titled “`ImageView` Is a `Node`.”

Shape Is a Node



Node (abstract)

ImageView resizable

MediaView resizable & CSS stylable

Parent resizable & CSS stylable

Shape (abstract) resizable

Circle

Ellipse

Polygon

Polyline

Arc

CubicCurve

Line

QuadCurve

Rectangle

SVGPath

Path

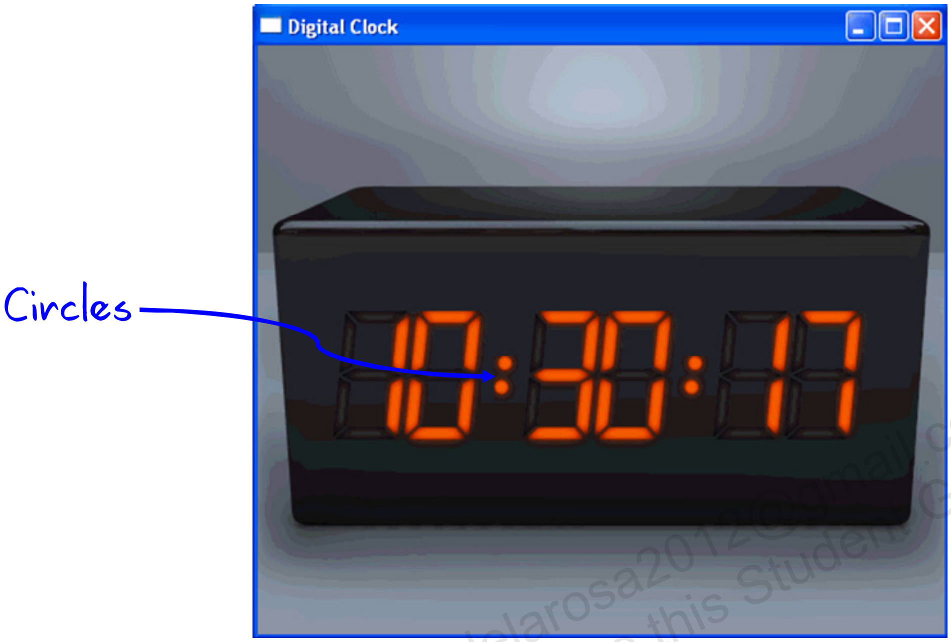
Text

```
public abstract class Shape
extends Node
```

The `Shape` class provides definitions of common properties for objects that represent some form of geometric shape. These properties include:

- The Paint to be applied to the interior of the shape (see `setFill`)
- The Paint to be applied to stroke the outline of the shape (see `setStroke`)
- The decorative properties of the stroke, including:
  - The width of the border stroke
  - Whether the border is drawn as an exterior padding to the edges of the shape (as an interior edging that follows the inside of the border) or as a wide path that follows along the border straddling it equally both inside and outside (see `StrokeType`)
  - Decoration styles for the joins between path segments and the unclosed ends of paths
  - Dashing attributes

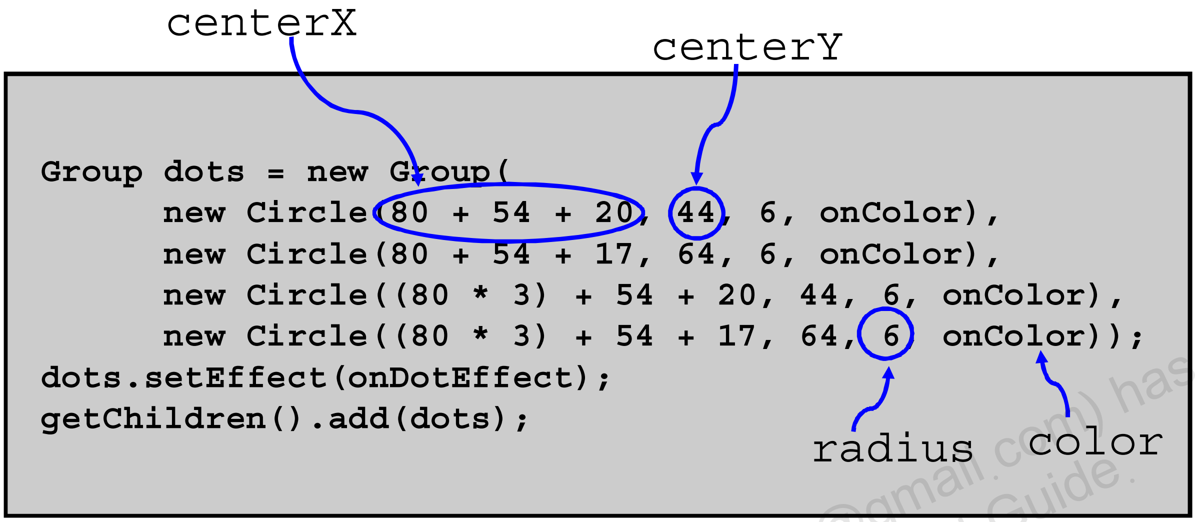
Clock Example Uses Circles



In the DigitalClock sample application, the dots that separate the hours, minutes, and seconds are created using circles.

This Digital Clock example is in `D:\labs\06-CreateUI\examples\DigitalClock`.

Clock Example: Circles

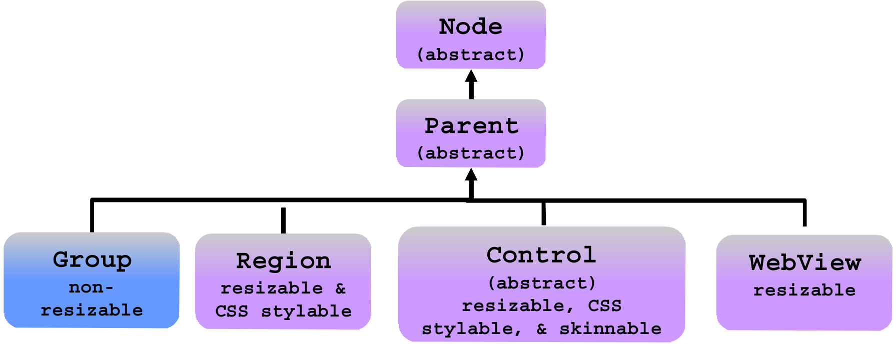


```
Group dots = new Group(
      new Circle(80 + 54 + 20, 44, 6, onColor),
      new Circle(80 + 54 + 17, 64, 6, onColor),
      new Circle((80 * 3) + 54 + 20, 44, 6, onColor),
      new Circle((80 * 3) + 54 + 17, 64, 6, onColor));
dots.setEffect(onDotEffect);
getChildren().add(dots);
```

In the DigitalClock sample application shown in the previous slide, the dots that separate the hours, minutes, and seconds are created using circles. The four instances of Circle are created in a group. The size, position, and color of the circles are set.

- The first set of numbers is double `centerX`, which is the horizontal position of the center of the circle in pixels.
- The second set of numbers is double `centerY`, which is the vertical position of the center of the circle in pixels.
- The last number is the radius of the circle.
- The last entry sets the color.

Group Is a Node



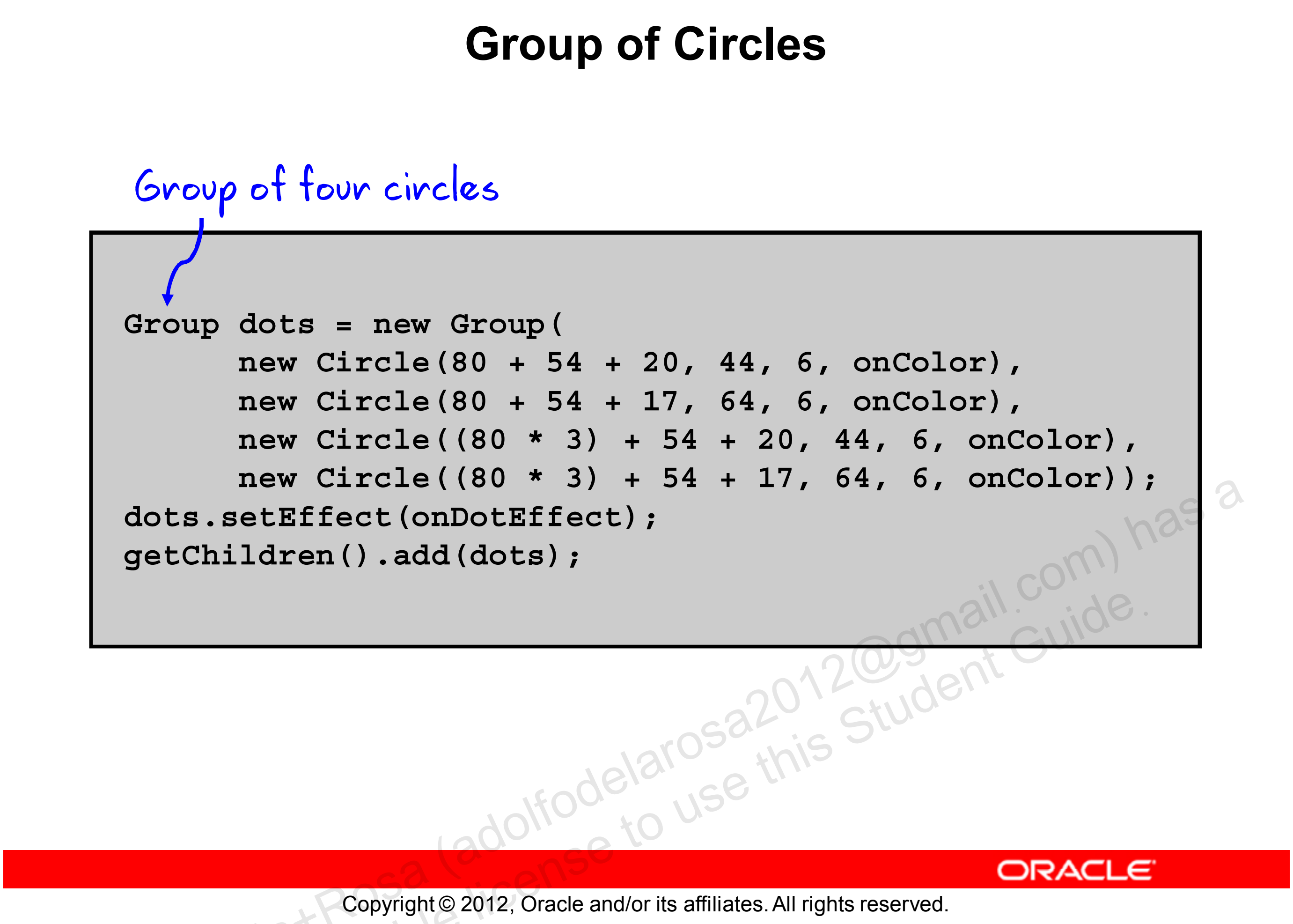
```
public class Group
extends Parent
```

A Group node contains an `ObservableList` of children that are rendered in order whenever this node is rendered. A group will take on the collective bounds of its children and is not directly resizable.

Any transform, effect, or state applied to a group will be applied to all children of that group. Such transforms and effects will *not* be included in this group's layout bounds. However, if transforms and effects are set directly on children of this group, those will be included in this group's layout bounds.

By default, a group will "auto-size" its managed resizable children to their preferred sizes during the layout pass to ensure that regions and controls are sized properly as their state changes. If an application needs to disable this auto-sizing behavior, it should set `autoSizeChildren` to `false` and understand that if the preferred size of the children changes, they will not automatically resize.

The previous slide of the DigitalClock code shows an example of objects grouped together.

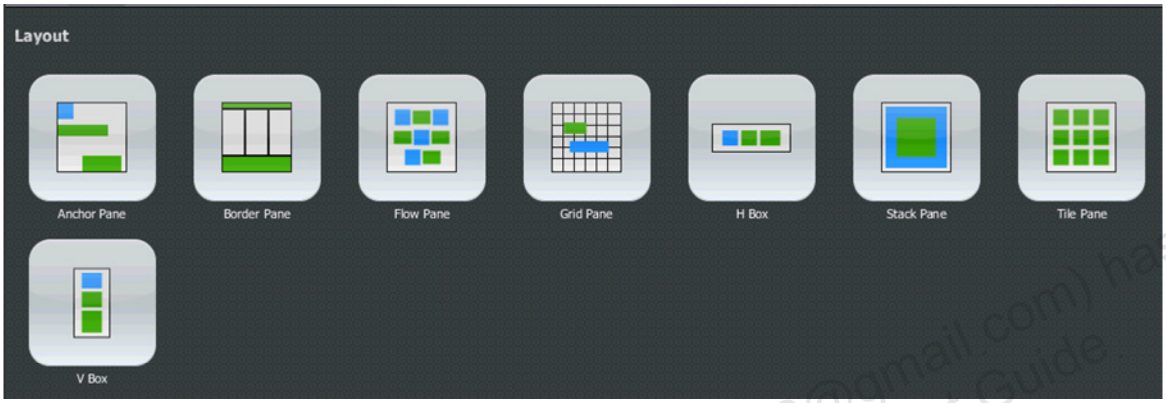


Group of Circles

Group of four circles

```
Group dots = new Group(
      new Circle(80 + 54 + 20, 44, 6, onColor),
      new Circle(80 + 54 + 17, 64, 6, onColor),
      new Circle((80 * 3) + 54 + 20, 44, 6, onColor),
      new Circle((80 * 3) + 54 + 17, 64, 6, onColor));
dots.setEffect(onDotEffect);
getChildren().add(dots);
```

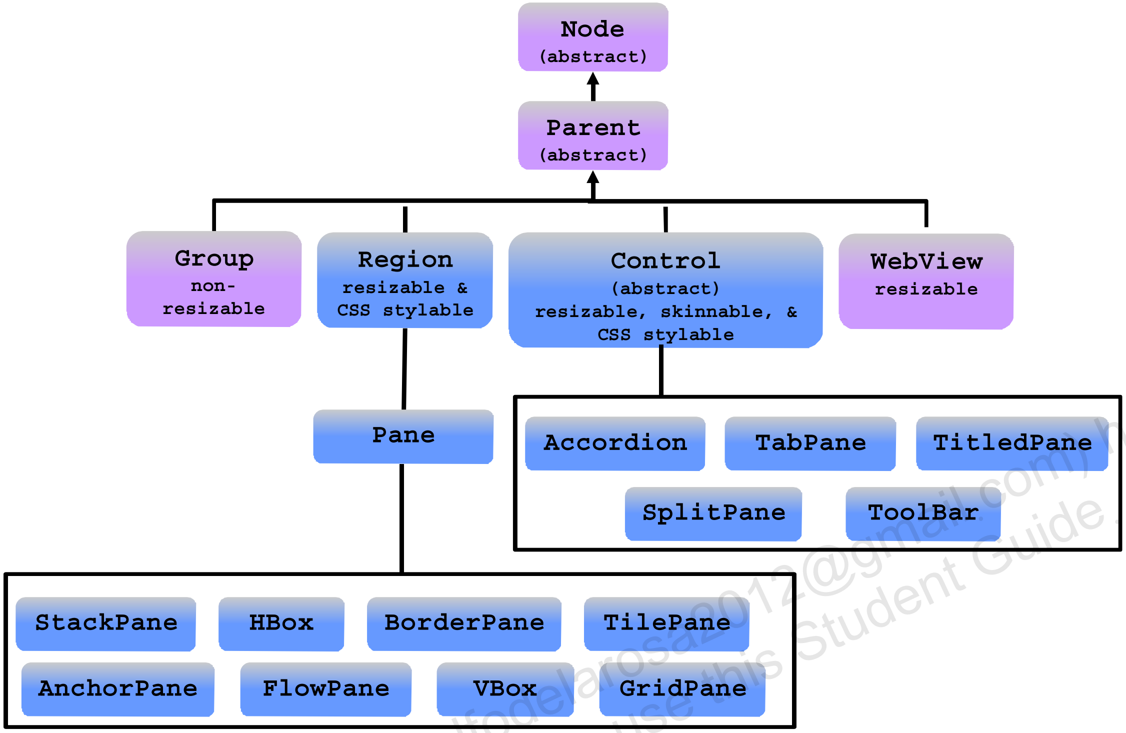
Layout Containers



The JavaFX SDK provides several layout container classes, called *panes*, for the easy setup and management of classic layouts such as rows, columns, stacks, tiles, and others. As a window is resized, the layout pane automatically repositions and resizes the nodes that it contains according to the properties for the nodes.

- BorderPane
- HBox
- VBox
- StackPane
- GridPane
- FlowPane
- TilePane
- AnchorPane

A Layout Container Is a `Node`



The packages `javafx.scene.layout` and `javafx.scene.control.Control` provide classes to support user interface layout. Each layout pane class supports a different layout strategy for its children, and applications may nest these layout panes to achieve the needed layout structure in the user interface. When a node is added to one of the layout panes, the pane will automatically manage the layout for the node, so the application should not position or resize the node directly.

The scene graph layout mechanism is driven automatically by the system after the application creates and displays a scene. The scene graph detects dynamic node changes that affect layout (such as a change in size or content), and it calls `requestLayout()`, which marks that branch as needing layout so that on the next pulse, a top-down layout pass is executed on that branch by invoking `layout()` on that branch's root. During that layout pass, the `layoutChildren()` callback method will be called on each parent to lay out its children. This mechanism is designed to maximize layout efficiency by ensuring that multiple layout requests are coalesced and processed in a single pass rather than executing re-layout on each minute change. Therefore, applications should not invoke layout directly on nodes.

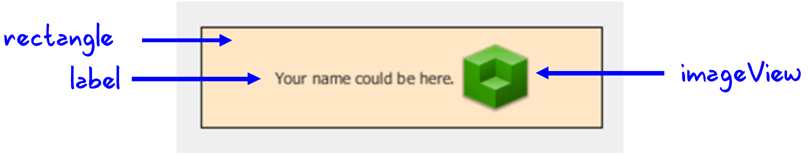
Types of Layout Containers

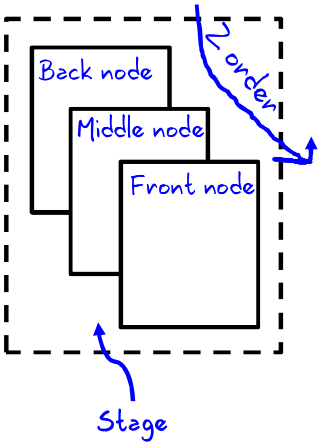
- StackPane
- BorderPane
- HBox
- VBox
- GridPane

Some of the layout containers used in the course application are listed in the slide.

StackPane

The StackPane stacks nodes in a Z order from back to front.





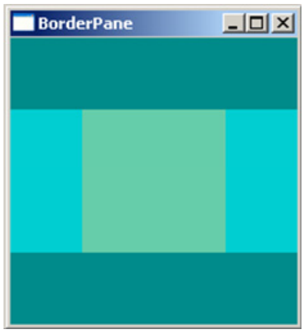
```
    StackPane stackPane = new StackPane();

           Rectangle rectangle = new Rectangle(280, 70,
Color.BISQUE);
           rectangle.setStroke(Color.BLACK);
           ImageView imageView = new ImageView(ICON_48);
           Label label = new Label("Your name could be
here.", imageView);
           label.setContentDisplay(ContentDisplay.RIGHT);

           stackPane.getChildren().addAll(rectangle,
label);

           root.getChildren().add(stackPane);
      }
```

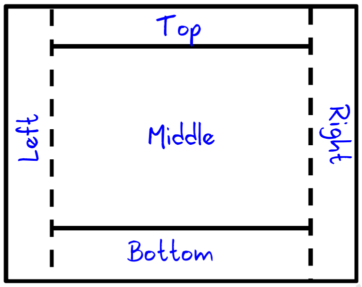
StackPane can be used to:

- Overlay text on an image or shape
- Overlap common shapes to make a complex shape

BorderPane

The BorderPane provides five regions in which to place nodes.





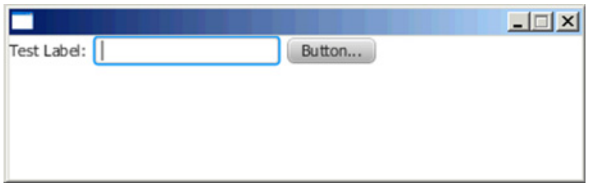
```
BorderPane root = new BorderPane();
root.setTop(new Rectangle(200, 50,
                          Color.DARKCYAN));
root.setBottom(new Rectangle(200, 50,
                             Color.DARKCYAN));
root.setCenter(new Rectangle(100, 100,
                       Color.MEDIUMAQUAMARINE));
root.setLeft(new Rectangle(50, 100,
                          Color.DARKTURQUOISE));
root.setRight(new Rectangle(50, 100,
                          Color.DARKTURQUOISE));

Scene scene = new Scene(root);
```

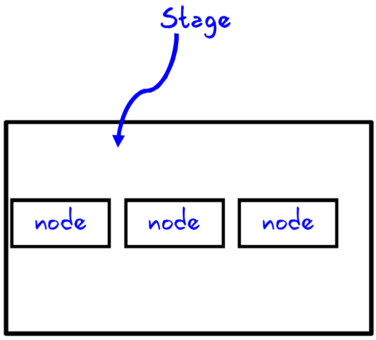
In the BorderPane, regions can be any size and do not have to be defined if they are not needed. A BorderPane can be used for the following:

- Menu across the top
- Status bar across the bottom
- Navigation bar along the left
- More information along the right
- Working area in the middle

HBox

The HBox arranges nodes in a horizontal row.





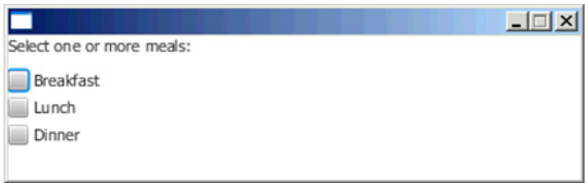
```
//Controls to be added to the HBox
    Label label = new Label("Test Label:");
    TextField tb = new TextField();
    Button button = new Button("Button...");

//HBox with spacing = 5
     HBox hbox = new HBox(5);
     hbox.getChildren().addAll(label, tb,button);
     hbox.setAlignment(Pos.CENTER);
     root.getChildren().add(hbox);
```

An HBox is easily modified:

- Padding attributes can be set to manage the distance between the nodes and the edges of the HBox pane.
- Spacing attributes can be set to manage the distance between the nodes.
- The style can be set to change the background color.

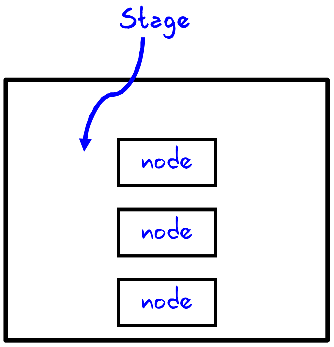
VBox

The VBox arranges nodes in a vertical column.



Select one or more meals:

□ Breakfast
□ Lunch
□ Dinner



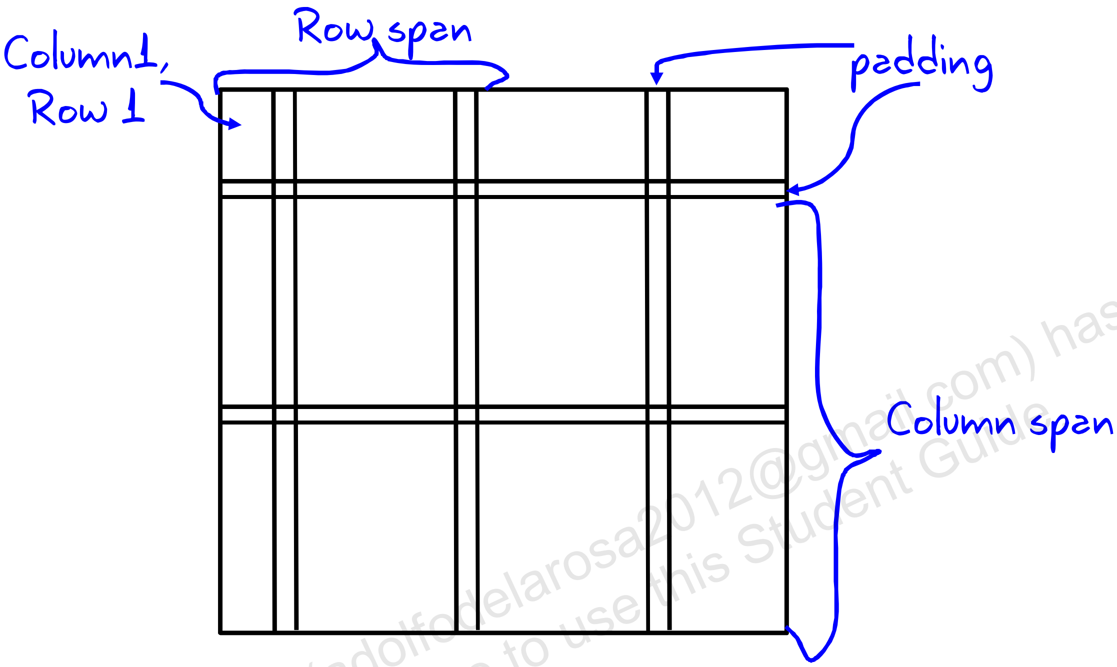
```
VBox vboxMeals = new VBox(5);
vboxMeals.getChildren().addAll(cb1, cb2, cb3);

Label label = new Label("Select one or more meals:");
VBox vboxOuter = new VBox(10);

vboxOuter.getChildren().addAll(label, vboxMeals);
vboxOuter.setAlignment(Pos.CENTER_LEFT);

root.getChildren().add(vboxOuter);
}
```

A VBox is easily modified:

- Padding attributes can be set to manage the distance between the nodes and the edges of the VBox pane.
- Spacing attributes can be set to manage the distance between the nodes.
- The style can be set to change the background color.

GridPane

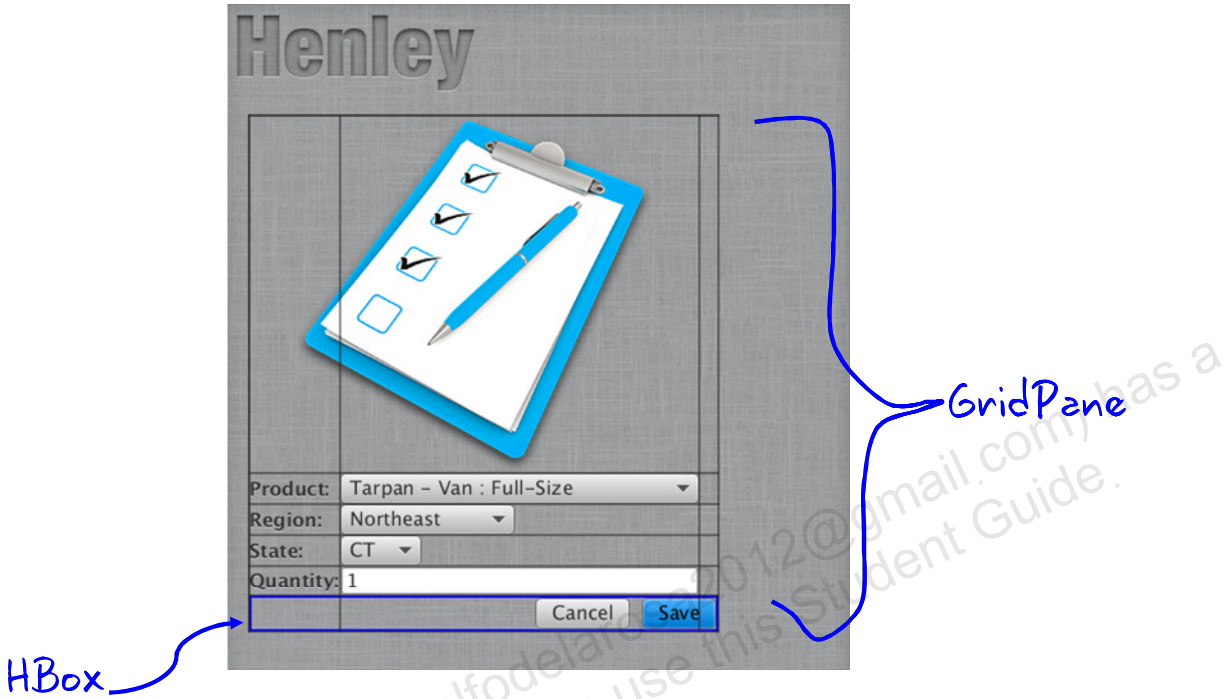
The GridPane is a flexible grid of rows and columns.



A GridPane enables you to create a flexible grid of rows and columns in which to lay out nodes. Nodes can be placed in any cell in the grid and can span cells as needed. A grid pane is useful for creating forms or any layout that is organized in rows and columns. Gap properties can be set to manage the spacing between the rows and columns. The padding property can be set to manage the distance between the nodes and the edges of the grid pane.

There is more than one approach to using a GridPane. First, the code can specify which rows and/or columns should contain the content. Second, the code can alter the constraints of the rows and/or columns themselves, either by specifying the preferred minimum or maximum heights or widths, or by specifying the percentage of the GridPane that belongs to certain rows or columns.

Henley Car Sales Form Layout



Product: Tarpan - Van : Full-Size

Region: Northeast

State: CT

Quantity: 1

In the Henley Car Sales Form, a GridPane is the main layout class, and it contains a nested HBox layout container. All elements within the GridPane are child nodes of the GridPane. The image in the slide shows the GridPane with visible grid lines.

The GridPane has three columns and six rows, and the component position is specified by the row and column numbers. For example, the image is in Row 1, Column 1, and it spans two columns, as shown in the following code:

```
<ImageView GridPane.columnIndex="1" GridPane.rowIndex="1"
GridPane.columnSpan="2" GridPane.halignment="center"><image><Image
url="@clipboard.png"/></image></ImageView>
```

The `SalesForm.fxml` class is shown below:

```
<GridPane xmlns:fx="http://javafx.com/fxml"
fx:controller="henleyclient.SalesForm" fx:id="salesForm" hgap="15"
vgap="12">
    <padding><Insets top="10" right="10" bottom="10"
left="10"/></padding>
    <children>
        <ImageView GridPane.columnIndex="1" GridPane.rowIndex="1"
GridPane.columnSpan="2" GridPane.halignment="center"><image><Image
url="@clipboard.png"/></image></ImageView>
        <Label text="Product:" GridPane.columnIndex="1"
GridPane.rowIndex="2"/>
        <ChoiceBox fx:id="productsChoiceBox"
GridPane.columnIndex="2" GridPane.rowIndex="2" prefWidth="300"/>
        <ProgressIndicator fx:id="productsProgress" prefWidth="16"
prefHeight="16" GridPane.columnIndex="3" GridPane.rowIndex="2"/>
        <Label text="Region:" GridPane.columnIndex="1"
GridPane.rowIndex="3"/>
        <ChoiceBox fx:id="regionsChoiceBox" GridPane.columnIndex="2"
GridPane.rowIndex="3"/>
        <ProgressIndicator fx:id="regionsProgress" prefWidth="16"
prefHeight="16" GridPane.columnIndex="3" GridPane.rowIndex="3"/>
        <Label text="State:" GridPane.columnIndex="1"
GridPane.rowIndex="4"/>
        <ChoiceBox fx:id="statesChoiceBox" GridPane.columnIndex="2"
GridPane.rowIndex="4" GridPane.columnSpan="2"/>
        <Label text="Quantity:" GridPane.columnIndex="1"
GridPane.rowIndex="5" minWidth="-Infinity"/>
        <TextField fx:id="quantityTextField" text="1"
GridPane.columnIndex="2" GridPane.rowIndex="5"/>
        <HBox spacing="10" alignment="center_right"
GridPane.columnIndex="1" GridPane.rowIndex="6"
GridPane.columnSpan="3">
            <children>
                <Button text="Cancel" onAction="#cancelAction"
cancelButton="true"/>
                <Button text="Save" onAction="#saveAction"
defaultButton = "true"/>
            </children>
        </HBox>
    </children>
</GridPane>
```

Resizability

JavaFX has two mechanisms for achieving resizing:

- Layout container classes
- Resizing and layout at the node level

Nodes in the scene graph have the potential to be *resizable*. This means that a node expresses its minimum, preferred, and maximum size constraints and allows its parent to resize it during layout (hopefully within that range, if the parent is honorable).

Each layout pane implements its own policy for resizing a resizable child, given the child's size range. For example, a `StackPane` will attempt to resize all children to fill its width/height (honoring their max limits) while a `FlowPane` always resizes children to their preferred sizes, which is what we mean by the term *auto-size* ("resize-to-preferred").

Not all scene-graph node classes are resizable:

- **Resizable:** Region, Control, WebView, MediaView, ImageView
- ***Not* resizable:** Group, Text, Shape

If a Node class is not resizable, note the following results:

`isResizable()` returns `false`.

`minWidth()`, `minHeight()`, `prefWidth()`, `prefHeight()`, `maxWidth()`, and `maxHeight()` all return their current sizes.

`resize()` is a no-op.

See Amy Fowler's blog at http://amyfowlersblog.wordpress.com/2011/06/02/javafx2-0-layout-a-class-tour/ for information about resizability, including the following discussion:

- The size of a non-resizable node is affected by its various properties (width/height for Rectangle, radius for Circle, and so on) and it is the application's responsibility to set these properties (and remember that scaling is *not* the same as resizing, because a scale also transforms the stroke, which is not usually what you want for layout). When non-resizable nodes are placed into layout panes, they will be positioned but not resized.
- Why not make shapes resizable? Although it is fairly easy to do the math for basic shapes (Rectangle, Circle, and so on), it becomes a bit more difficult with Paths, Polygons, and 3D. Also, keeping the core graphics shapes non-resizable creates certain efficiencies, such as ensuring that a Group with non-resizable descendents pays no penalty for layout.

Note: For additional information about layout containers and resizing nodes, see the tutorial "Working with Layouts" at http://docs.oracle.com/javafx/2.0/layout/jfxpub-layout.htm.

Quiz

When a layout container is resized, its nodes are resized according to their properties.

a. True

b. False

Answer: a

Topics

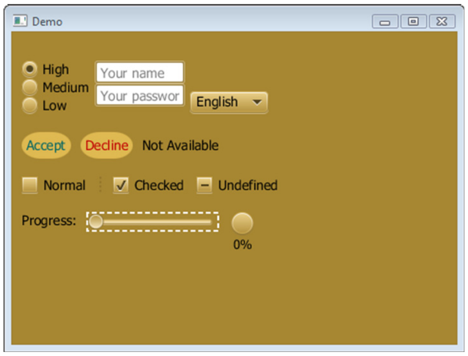
- Relating UI components to the scene graph
- Describing and implementing JavaFX UI components such as controls, images, shapes, and layout containers
- Using CSS
- Adding events to JavaFX components
- Creating charts

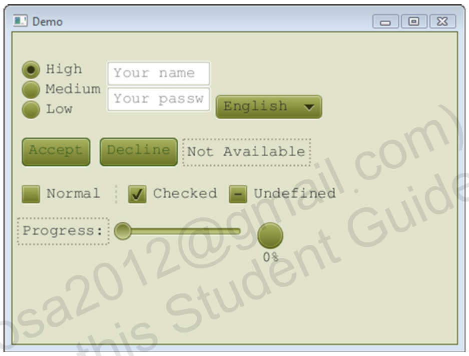
ORACLE

Skinning UI Controls

UI controls can be skinned using two methods:

- Applying CSS
- Overriding default styles on individual components





You can create a custom appearance (also called a *skin*) for your JavaFX application with cascading style sheets (CSS). CSS contain style definitions that control the appearance of user interface elements. Using CSS in JavaFX applications is similar to using CSS in HTML. JavaFX CSS are based on the W3C CSS version 2.1 specification (available at http://www.w3.org/TR/CSS21/) with some additions from current work on version 3 of the specification and some extensions that support specific JavaFX features.

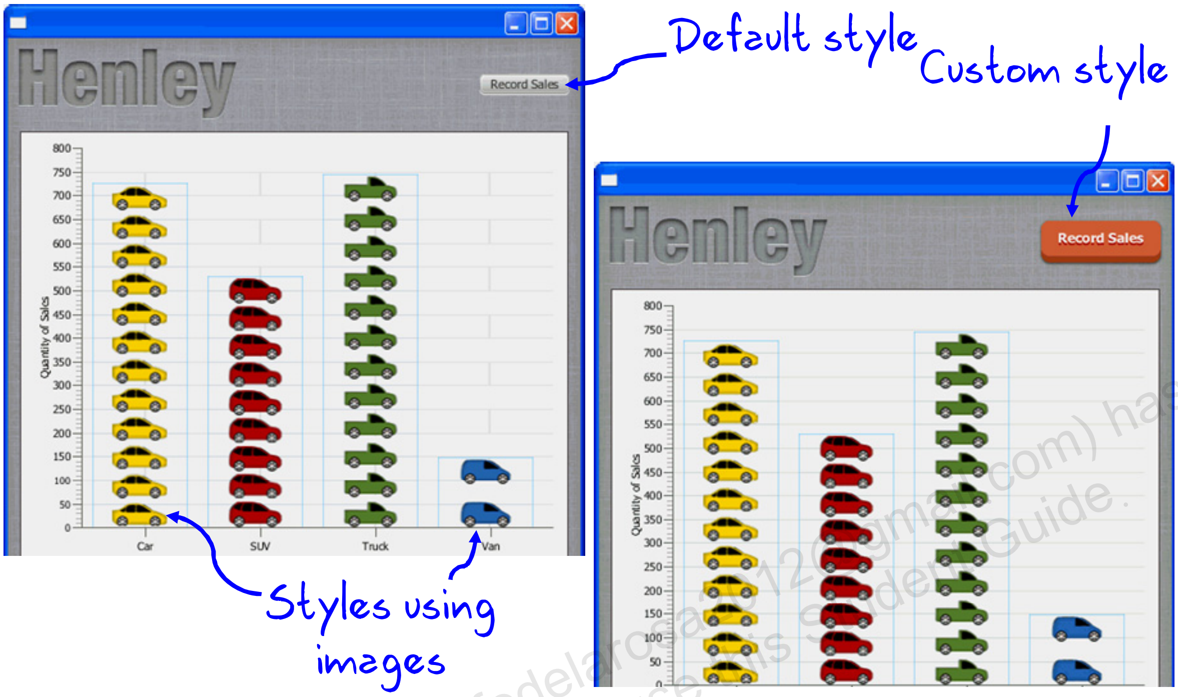
You can further customize your UI by defining styles for the different controls that you are using. You can override the definitions in the default style sheet or create new class or ID styles. You can also define the style for a node within your code.

The images in the slide show an application that uses two different style sheets. The default style sheet for JavaFX applications is `caspian.css`, which is found in the JavaFX Runtime JAR file, `jfxrt.jar`. This style sheet defines styles for the root node and the UI controls. To view this file, go to the `\rt\lib` directory under the directory in which the JavaFX SDK is installed. Use the following command to extract the style sheet from the JAR file:

```
jar -xf jfxrt.jar
com/sun/javafx/scene/control/skin/caspian/caspian.css
```

The JavaFX scene graph provides the capability of styling nodes by using CSS. The Node class contains the styleClass id, and style variables are used by CSS selectors to find nodes to which styles should be applied. The Scene class contains the stylesheets variable, which is a sequence of URLs that reference CSS style sheets that are to be applied to the nodes within that scene.

CSS in Henley Sales Application



The screenshots in the slide show two versions of the Record Sales button in the Henley Sales application. The first image shows the button using the default style. The second image shows the button with custom styles applied.

Note: For additional information about CSS, applying CSS styles to nodes, and the properties that are available for styling, see the “CSS Reference Guide” at `D:\labs\resources\JavaFX-CSS-Reference-Guide.pdf`.

CSS Syntax

The following example of JavaFX CSS syntax changes the color of the button text from the default black to white. It also includes a drop shadow.

```
 #recordSalesButton .text {
    -fx-fill: white;
    -fx-effect: dropshadow( gaussian , #a30000 , 0,0,0,2
   );
}
```

An `id` for the element must be defined and then referenced in the associated Java file so that the style can be applied to the element.

```
<ChoiceBox fx:id="customerChoiceBox"/>
```

Each node in the scene graph has an `id` variable (a string). This is analogous to the `id="..."` attribute that can appear in HTML elements. Supplying a string for a node's `id` variable causes style properties for this node to be looked up using that `id`. Styles for specific `id`s can be specified using the `#nodeid` selector syntax in a style sheet.

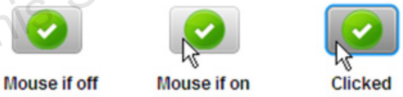
Note: For more information, see the "CSS Reference Guide" at `D:\labs\resources\JavaFX-CSS-Reference-Guide.pdf`.

Topics

- Relating UI components to the scene graph
- Describing and implementing JavaFX UI components such as controls, images, shapes, and layout containers
- Using CSS
- Adding events to JavaFX components
- Creating charts

Events in JavaFX

- JavaFX uses event handlers and event filters for events such as:
  - Mouse events
  - Keyboard events
  - Drag-and-drop events
  - Window events
  - Action events
- Every event includes
  - Event type
  - Event source
  - Event target

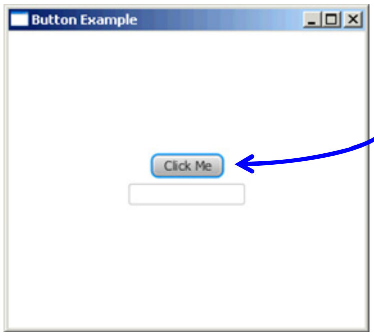


An event represents an occurrence of something of interest to the application, such as moving a cursor or pressing a key. In JavaFX, an event is an instance of the `javafx.event.Event` class or any subclass of Event. JavaFX provides several events, including `DragEvent`, `KeyEvent`, `MouseEvent`, `ScrollEvent`, and others. You can define your own event by extending the Event class.

The event filters and event handlers are implementations of the `EventHandler` interface. If an application wants to be notified when an event occurs, a filter or handler is registered for the event. The primary difference between a filter and a handler is when each one is executed. Event handlers are executed as the event traverses the event dispatch chain from the event target. Event filters are executed during the event capture phase as the event traverses the event dispatch chain from the stage to the event target.

- **Event type:** Type of event that occurred. An event type is an instance of the `EventType` class.
- **Source:** Origin of the event, with respect to the location of the event in the event dispatch chain. The source changes as the event is passed along the chain.
- **Target:** Node on which the action occurred and the end node in the event dispatch chain. The target does not change. The target of an event can be an instance of any class that implements the `EventTarget` interface.

Button Event

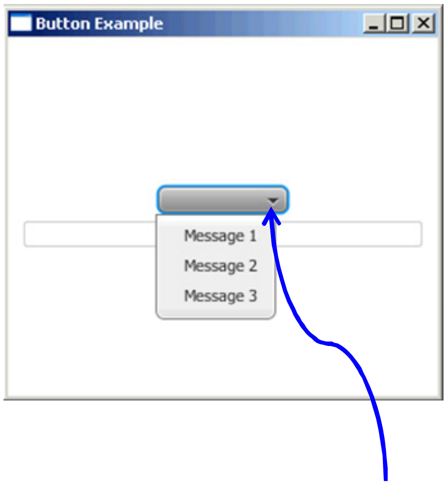


```
// Click event handler
   btn.setOnAction(new EventHandler<ActionEvent>() {
     @Override
       public void handle(ActionEvent event) {
       System.out.println("Button Clicked");
                clickCtr++;

       clickFld.setText(Integer.toString(clickCtr));
     }
   });
```

In this example, the button event prints `Button Clicked` each time the user clicks the button. The example also uses a click counter to count each button click.

Choice Box

A Choice Box enables users to choose one of several options.



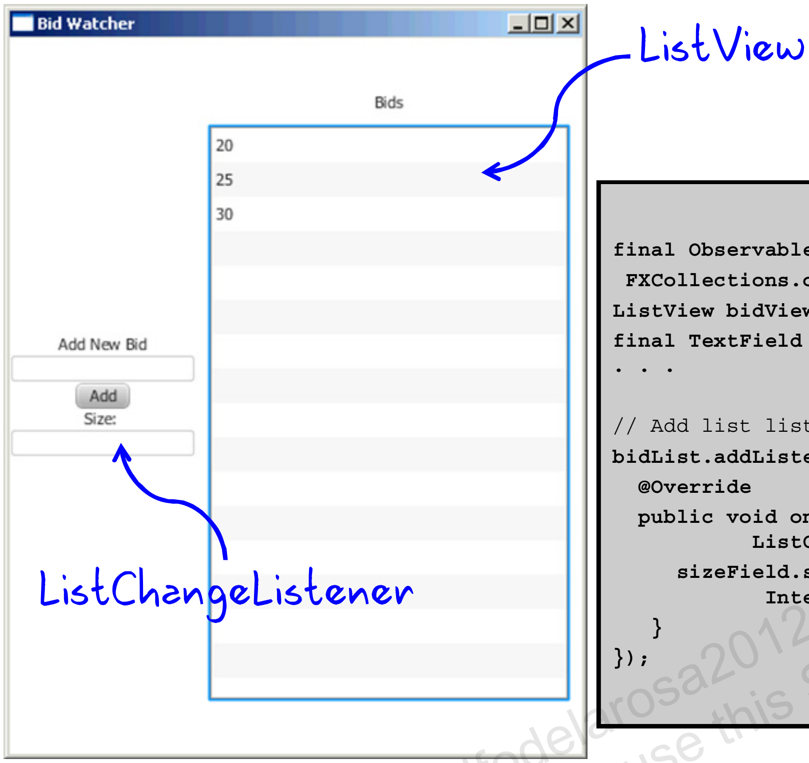
Choice Box

```
// ChoiceBox event handler
   choiceBox.getSelectionModel().selectedIndexProperty().
                                  addListener(
      new ChangeListener<Number>() {

        @Override
         public void changed(ObservableValue ov, Number
                                  value, Number newValue) {
            int index = newValue.intValue();
            messageFld.setText(messages.toArray()[
                                        index].toString());
            System.out.println("Message set to: " +
                     messages.toArray()[index].toString());
        }
     });
```

List items are created and populated within a constructor of the `ChoiceBox` class. The list items are specified by using an observable array. Alternatively, you can use an empty constructor of the class and set the list items by using the `setItems` method.

A Choice Box can contain not only text elements but also other objects.

Listener



```
final ObservableList<Integer> bidList =
 FXCollections.observableArrayList(20, 25, 30);
ListView bidView = new ListView(bidList);
final TextField bidField = new TextField();
. . .

// Add list listener
bidList.addListener(new ListChangeListener(){
  @Override
  public void onChanged(
             ListChangeListener.Change change){
     sizeField.setText(
              Integer.toString(bidList.size()));
   }
});
```

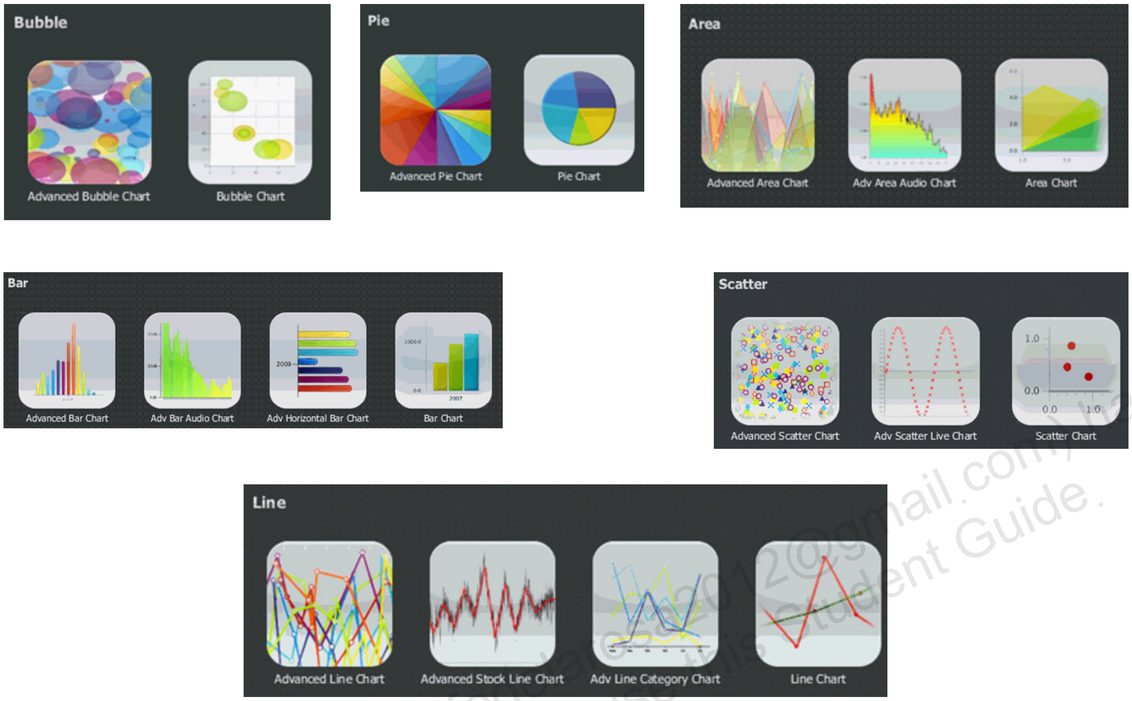
The `ListView` is set up as one listener and automatically updates each time the list is updated.

The `ListChangeListener` event handler is a second listener. It displays the size of the `bidList` every time you add a new bid.

Topics

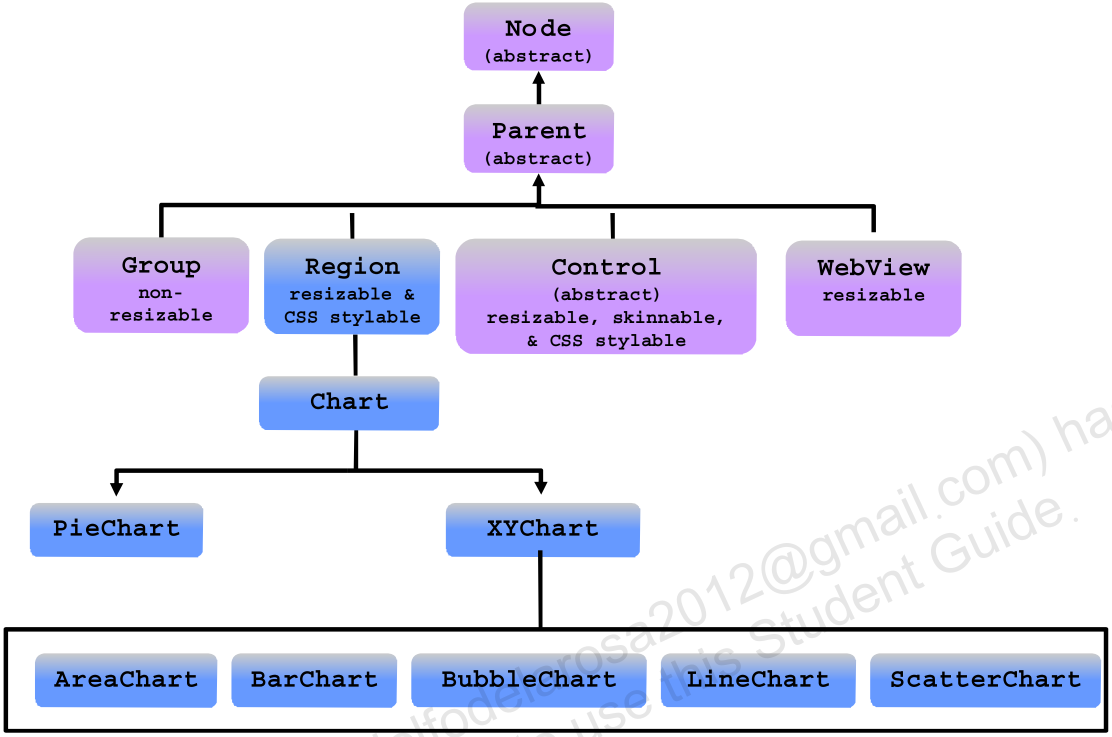
- Relating UI components to the scene graph
- Describing and implementing JavaFX UI components such as controls, images, shapes, and layout containers
- Using CSS
- Adding events to JavaFX components
- Creating charts

Charts



In addition to the typical elements of a user interface, the JavaFX SDK provides charts in the `javafx.scene.chart` package. The types of charts currently supported are bar, area, line, bubble, scatter, and pie.

Chart Is a Node



The JavaFX User Interface provides a set of chart components that are very convenient for displaying data. Application developers can make use of the graphical charts provided by the SDK to illustrate a wide variety of data.

Common types of charts such as Bar, Line, Area, Pie, Scatter, and Bubble are provided. These charts are easy to create and are customizable. The JavaFX Charts API is a visual-centric (rather than model-centric) API.

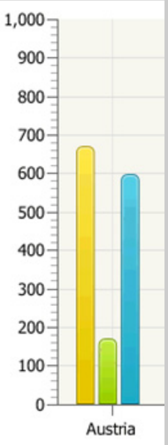
JavaFX charts support animation of chart components as well as auto-ranging of the chart axes:

```
javafx.scene.chart.Chart
```

```
public abstract class Chart
extends Region
```

To create a chart, you need to instantiate the specific chart class, define the data, assign the data items to the specific chart class object, and add the chart to the application.

Chart Example in a Java Class



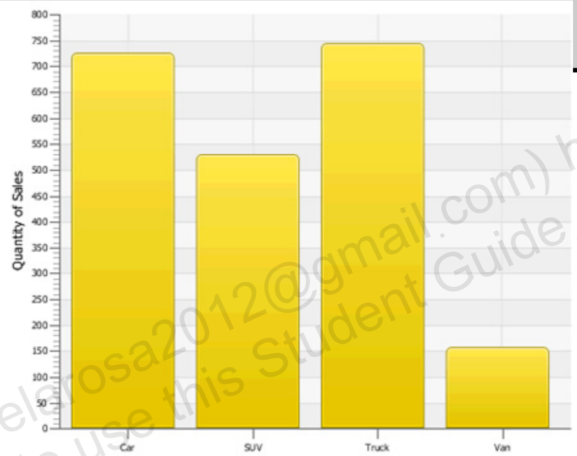
```
 @Override public void start(Stage stage) {
        stage.setTitle("Bar Chart Sample");
        final CategoryAxis xAxis = new CategoryAxis();
        final NumberAxis yAxis = new NumberAxis();
        final BarChart<String,Number> bc =
            new BarChart<String,Number>(xAxis,yAxis);
        bc.setTitle("Country Summary");
        xAxis.setLabel("Country");
        yAxis.setLabel("Value");

        XYChart.Series series1 = new XYChart.Series();
        series1.setName("2003");
        series1.getData().add(new XYChart.Data(austria,
   25601.34));
…
```

ORACLE

The code example shows a BarChart created in a Java class. To build a BarChart in your JavaFX application, you create two axes, instantiate the BarChart class, define the series of data, and assign the data to the chart.

Chart Example in FXML

```
<?import javafx.scene.chart.*?>
…
 <BarChart fx:id="salesChart" legendVisible="false">
  <xAxis><CategoryAxis/></xAxis>
  <yAxis><NumberAxis label="Quantity of Sales"/></yAxis>
 </BarChart>
…
```

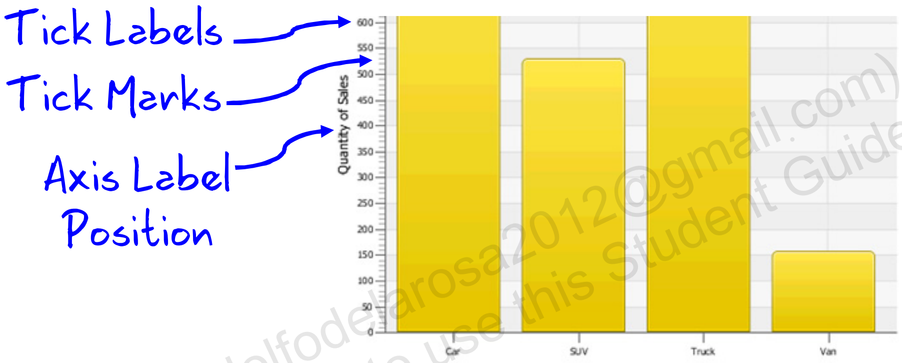


In the FXML example code, the BarChart is named salesChart, and the legend is hidden. The X axis is defined as CategoryAxis and the Y axis is "Quantity of Sales." The X axis labels will be defined using a style sheet.

Defining the X Axis and Y Axis

Change the default appearance of each chart by defining:

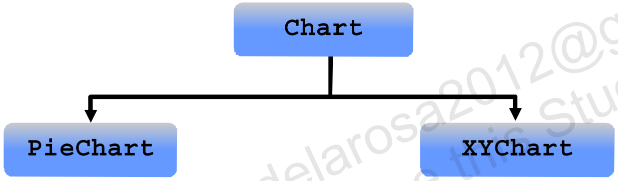
- The axis label
- The axis position relative to the chart plot
- The upper and lower boundaries of the axis range
- The minimum and maximum tick marks, tick units, the gap between two tick marks, and tick labels



You can also specify animation for any changes to an axis and its range, or you can enable the axis to automatically determine its range from the data.

Chart Data

- The XYChart.Data class specifies the data model:
  - xValue defines the value of the chart element plotted on the X axis.
  - yValue defines the value of the chart element plotted on the Y axis.
- The PieChart.Data class specifies values for each slice of the pie.
- The XYChart.Series class defines several series of data.



The XYChart class, a super class for all two-axis charts, provides basic capabilities for building area, line, bar, scatter, and bubble charts. Use the XYChart.Data class to specify the data model for these types of charts.

The xValue property defines the value of a chart element to be plotted on the X axis, and the yValue property defines the value for the Y axis.

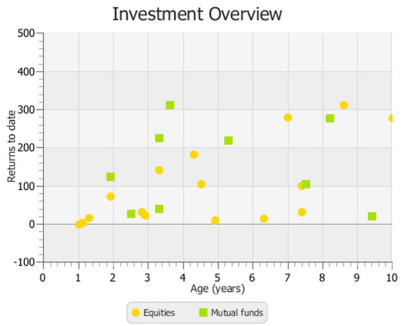
You can also set the extra value for each chart element. This value can be plotted in any way the chart needs, or it can be used to store additional information about the chart element. For example, it can be used to define a radius for bubble charts.

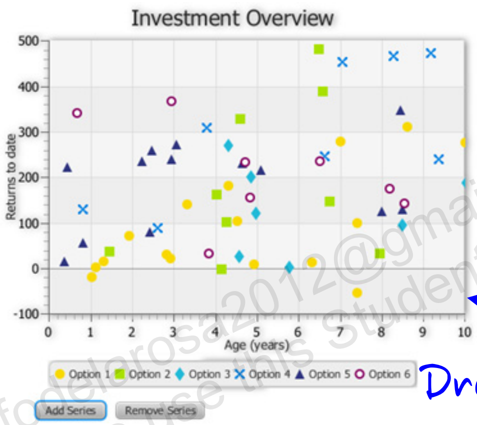
Use the XYChart.Series class to define as many sets of data as you need to represent on your graph. You can also assign a particular name to each series to display in the chart legend.

Unlike a two-axis chart, the pie chart does not require defining values for X and Y axes. You use the PieChart.Data class to specify values for each slice in the pie.

Apply Effects

```
final DropShadow shadow = new DropShadow();
shadow.setOffsetX(2);
shadow.setColor(Color.GREY);
sc.setEffect(shadow);
```





Drop Shadow

ORACLE

All the chart classes available in the `javafx.scene.chart` are extensions of the `Node` class. Therefore, you can apply visual effects or transformation to every type of chart.

The ScatterChart example shown in this slide is in `D:\labs\05-UIControls\examples\ScatterChartSample`.

Style Charts

```
 .default-color0.chart-area-symbol { -fx-background-
   color: #e9967a, #ffa07a; }
.default-color1.chart-area-symbol { -fx-background-
   color: #f0e68c, #fffacd; }
.default-color2.chart-area-symbol { -fx-background-
   color: #dda0dd, #d8bfd855; }

.default-color0.chart-series-area-line { -fx-stroke:
   #e9967a; }
.default-color1.chart-series-area-line { -fx-stroke:
   #f0e68c; }
.default-color2.chart-series-area-line { -fx-stroke:
   #dda0dd; }
```

The colors of the chart elements are defined by the implementation of each particular chart class. However, you can alter these colors as well as chart symbols by applying CSS styles.

Animate Charts

The Chart class has the following:

- animated property
- setAnimated method

With the JavaFX SDK, you can make your chart change dynamically as the data changes. Use the animated property and the setAnimated method of the Chart class to toggle this functionality for the chart. You can use the animated property and the setAnimated method of the Axis class to animate changes to either axis and its range.

Quiz

Which class do you use to specify a series of data in a chart?

a. `XYChart.Data`

b. `PieChart.Data`

c. `PieChart.Series`

d. `XYChart.Series`

Answer: d

Summary

In this lesson, you should have learned how to:

- Relate UI components to the scene graph
- Describe and implement JavaFX UI components such as controls, images, shapes, and layout containers
- Use CSS
- Add events to JavaFX controls
- Create charts

Practice 5: Overview

- 5-1: Creating an Order Form by Using FXML
- 5-2: Adding Events to the Order Form
- 5-3: Creating an Interactive Pie Chart