CSCI 2120:

Software Design & Development II

UNIT4: UI management

GUI framework

JavaFX Controls: Control elements

Overview

Introduction

- Package: javafx.scene.control
- 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 systems. Examples of prominent controls include Button, Label, ListView, and TextField.
- Since Controls are Nodes in the scenegraph, they can be freely mixed with Groups, Images, Media, Text, and basic geometric shapes.
- While writing new UI Controls is not trivial, using and styling them is very easy, especially to existing web developers.

JavaFX Architecture for Control class

- Controls follow the classic MVC design pattern. The Control is the "model". It contains both the state and the functions which manipulate that state.

- The Control class itself does not know how it is rendered or what the user interaction is. These tasks are delegated to the Skin ("view"), which may internally separate out the view and controller functionality into separate classes, although at present there is no public API for the "controller" aspect.

- All Controls extend from the Control class, which is in turn a Parent node, and which is a Node. Every Control has a reference to a single Skin, which is the view implementation for the Control.

- The Control delegates to the Skin the responsibility of computing the min, max, and pref sizes of the Control, the baseline offset, and hit testing (containment and intersection). It is also the responsibility of the Skin, or a delegate of the Skin, to implement and respond to all relevant key events which occur on the Control when it contains the focus.

Control class

- Control extends from Parent, 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.

- Since a Control is resizable, a Control will be **auto-sized to its preferred size** on each scenegraph pulse. Setting the width and height of the Control does not affect its preferred size. When used in a layout container, the layout constraints imposed upon the Control (or manually specified on the Control) will determine how it is positioned and sized.

- The Skin of a Control can be changed at any time. Doing so will mark the Control as needing to be laid out since changing the Skin likely has changed the preferred size of the Control. If no Skin is specified at the time that the Control is created, then a default CSS-based skin will be provided for all of the built-in Controls.

Control class

- Each Control may have an optional tooltip specified. The Tooltip is a Control which displays some (usually textual) information about the control to the user when the mouse hovers over the Control from some period of time. It can be styled from CSS the same as with other Controls.

- focusTraversable is overridden in Control to be true by default, whereas with Node it is false by default. Controls which should not be focusable by default (such as Label) override this to be false.

- The getMinWidth, getMinHeight, getPrefWidth, getPrefHeight, getMaxWidth, and getMaxHeight functions are delegated directly to the Skin. The baselineOffset method is delegated to the node of the skin. It is not recommended that subclasses alter these delegations.

Control class declaration

Control is the base class for all user interface controls. A "Control" is a node in the scene graph which can be manipulated by the user. Controls provide additional variables and behaviors beyond those of Node to support common user interactions in a manner which is consistent and predictable for the user.

```
public abstract class Control extends Parent implements Skinnable
```

Additionally, controls support explicit skinning to make it easy to leverage the functionality of a control while customizing its appearance.

Most controls have their focus Traversable property set to true by default, however read-only controls such as Label and ProgressIndicator, and some controls that are containers ScrollPane and ToolBar do not.

See specific Control subclasses for information on how to use individual types of controls.

Styling Controls

- There are two methods for customizing the look of a Control. The most difficult and yet most flexible approach is to write a new Skin for the Control which precisely implements the visuals which you desire for the Control. Consult the Skin documentation for more details.

- The easiest and yet very powerful method for styling the built in Controls is by using CSS. Please note that in this release the following CSS description applies only to the default Skins provided for the built in Controls. Subsequent releases will make this generally available for any custom third party Controls which desire to take advantage of these CSS capabilities.

- Each of the default Skins for the built in Controls is comprised of multiple individually styleable areas or regions. This is much like an HTML page which is made up of <div>'s and then styled from CSS. Each individual region may be drawn with backgrounds, borders, images, padding, margins, and so on. The JavaFX CSS support includes the ability to have multiple backgrounds and borders, and to derive colors. These capabilities make it extremely easy to alter the look of Controls in JavaFX from CSS.

Styling Controls

- The colors used for drawing the default Skins of the built in Controls are all derived from a base color, an accent color and a background color. Simply by modifying the base color for a Control you can alter the derived gradients and create Buttons or other Controls which visually fit in with the default Skins but visually stand out.

- As with all other Nodes in the scenegraph, Controls can be styled by using an external stylesheet, or by specifying the style directly on the Control. Although for examples it is easier to express and understand by specifying the style directly on the Node, it is recommended to use an external stylesheet and use either the styleClass or id of the Control, just as you would use the "class" or id of an HTML element with HTML CSS.

- Each UI Control specifies a styleClass which may be used to style controls from an external stylesheet. For example, the Button control is given the "button" CSS style class. The CSS style class names are hyphen-separated lower case as opposed to camel case, otherwise, they are exactly the same. For example, Button is "button", RadioButton is "radio-button", Tooltip is "tooltip" and so on.

- The class documentation for each Control defines the default Skin regions which can be styled. For further information regarding the CSS capabilities provided with JavaFX, see the CSS Reference Guide.

Control subclasses

- Control elements (externals nodes) → focus for this lecture
- Control containers (internal nodes)

MenuButton → Introduction

The JavaFX MenuButton control works like a regular **JavaFX Button** except it provides a list of options which the user can choose to click. Each of these options function like a separate button - meaning your application can listen for clicks and respond individually to each option. In a way, a JavaFX MenuButton works a bit like a **JavaFX MenuBar**.

The JavaFX MenuButton can show or hide the menu items. The menu items are usually shown when a little arrow button is clicked in the MenuButton. The JavaFX MenuButton control is represented by the class javafx.scene.control.MenuButton.

MenuButton vs. ChoiceBox and ComboBox

The MenuButton looks similar to a **ChoiceBox** and **ComboBox**, but the difference is, that the MenuButton is designed to trigger an action when you select one of its menu options, whereas ChoiceBox and ComboBox are designed to just note internally what option was selected so it can be read later.

MenuButton → Creating a MenuButton

You create a JavaFX MenuButton by creating an instance of the MenuButton class. The MenuButton constructor takes a button text and a button graphic. You can pass null for the text and / or the graphic, in case you want a MenuButton without either text or graphic.

Here is an example of creating a JavaFX MenuButton with only a text label:

```
MenuItem menuItem1 = new MenuItem("Option 1");
MenuItem menuItem2 = new MenuItem("Option 2");
MenuItem menuItem3 = new MenuItem("Option 3");

MenuButton menuButton = new MenuButton("Options", null, menuItem1, menuItem2, menuItem3);
```

First 3 MenuItem instances are created, each with a different text. Then a MenuButton instance is created, passing a button text, a graphic icon (null) and the 3 MenuItem instances as parameter to the MenuButton constructor.

MenuButton → Adding a MenuButton to the Scene Graph

To make a MenuButton visible you must add it to the JavaFX scene graph. This means adding it to a Scene, or as child of a layout which is attached to a Scene object.

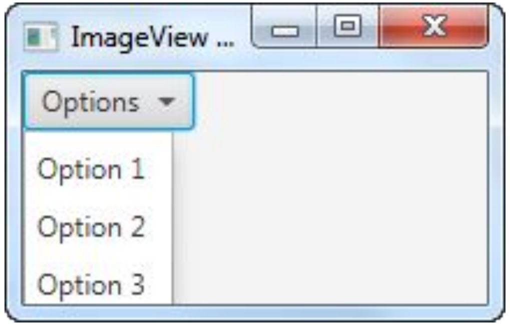
Here is an example that attaches a JavaFX MenuButton to the scene graph:

MenuButton → Adding a MenuButton to the Scene Graph

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.MenuItem;
import javafx.scene.control.MenuButton;
import java.io.FileInputStream;

public class MenuButtonExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("ImageView Experiment 1");
       MenuItem menuItem1 = new MenuItem("Option 1");
       MenuItem menuItem2 = new MenuItem("Option 2");
       MenuItem menuItem3 = new MenuItem("Option 3");
       MenuButton menuButton = new MenuButton("Options", null, menuItem1, menuItem2, menuItem3);
       HBox hbox = new HBox(menuButton);
       Scene scene = new Scene(hbox, 200, 100);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

MenuButton → Adding a MenuButton to the Scene Graph

Here is how the application resulting from the above example looks:



MenuButton → MenuButton Font

You can specify what font the text label on a JavaFX MenuButton should be rendered with. You set the font via the MenuButton setFont() method. Here is an example of setting the font of a JavaFX MenuButton via setFont():

```
MenuItem menuItem1 = new MenuItem("Option 1");
MenuItem menuItem2 = new MenuItem("Option 2");

MenuButton menuButton = new MenuButton("Options", null, menuItem1, menuItem2);

Font font = Font.font("Courier New", FontWeight.BOLD, 36);
menuButton.setFont(font);
```

MenuButton → MenuButton Icon

The JavaFX MenuButton enables you to add a graphical icon which is then displayed next to the menu text - just like you can do with a regular **JavaFX Button**.

The second example in the "Create a MenuButton" section shows how to create a MenuButton and pass the graphical icon via the constructor. However, it is also possible to set the graphical icon of a MenuButton via its setGraphic() method.

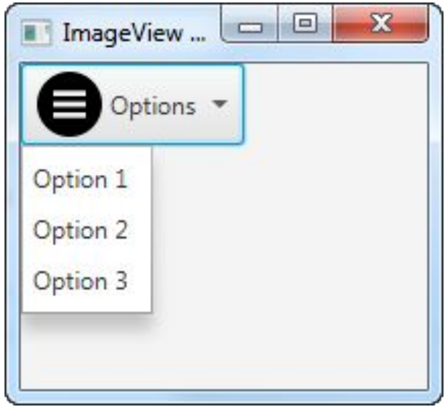
Here is how the example from the previous section would look with a graphic icon added to the MenuButton via its setGraphic() method:

MenuButton → MenuButton Icon

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.layout.HBox;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.control.MenuButton;
import javafx.scene.control.MenuItem;
import java.io.FileInputStream;

public class MenuButtonExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("ImageView Experiment 1");
       MenuItem menuItem1 = new MenuItem("Option 1");
       MenuItem menuItem2 = new MenuItem("Option 2");
       MenuItem menuItem3 = new MenuItem("Option 3");
       MenuButton menuButton = new MenuButton("Options", null, menuItem1, menuItem2, menuItem3);
       FileInputStream input = new FileInputStream("resources/images/iconmonstr-menu-5-32.png");
       Image image = new Image(input);
       ImageView imageView = new ImageView(image);
       menuButton.setGraphic(imageView);
       HBox hbox = new HBox(menuButton);
       Scene scene = new Scene(hbox, 200, 160);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

MenuButton → MenuButton Icon

Here is how the application resulting from the above example looks:



MenuButton → Responding to Menu Item Selection

To respond to when a user selects a menu item, add an "on action" event listener to the corresponding MenuItem object. Here is an example showing you how to add an action event listener to a MenuItem object:

```
MenuItem menuItem3 = new MenuItem("Option 3");

menuItem3.setOnAction(new EventHandler<ActionEvent>() {
@Override
public void handle(ActionEvent event) {
       System.out.println("Option 3 selected");
   }
});
```

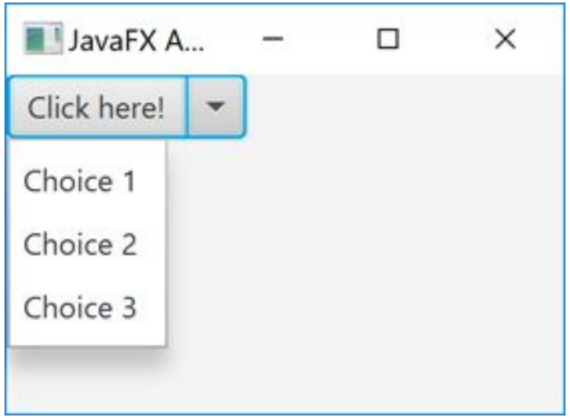
You can also use a **Java Lambda expression** instead of an anonymous implementation of the EventHandler interface. Here is how that looks:

```
menuItem3.setOnAction(event -> {
   System.out.println("Option 3 selected via Lambda");
});
```

SplitMenuButton → Introduction

The *JavaFX SplitMenuButton* control can show a list of menu options which the user can choose from, as well as a button which the user can click on when a menu option has been chosen. The JavaFX *SplitMenuButton* can show or hide the menu items. The menu items are usually shown when a little arrow button is clicked in the *SplitMenuButton*. The JavaFX SplitMenuButton control is represented by the class javafx.scene.control.SplitMenuButton.

Here is a screenshot of a JavaFX SplitMenuButton:



SplitMenuButton → Create SplitMenuButton

Before you can use the JavaFX SplitMenuButton you must create an instance of it.

Here is an example of creating a JavaFX SplitMenuButton:

```
SplitMenuButton splitMenuButton = new SplitMenuButton();
```

SplitMenuButton → SplitMenuButton Text

You can set the SplitMenuButton's button text via its setText() method.

Here is an example of setting the button text of a JavaFX SplitMenuButton:

```
splitMenuButton.setText("Click here!");
```

SplitMenuButton → SplitMenuButton Font

The JavaFX SplitMenuButton enables you to set the font used to render the text of the SplitMenuButton..

Here is an example of setting a font on a JavaFX SplitMenuButton:

```
SplitMenuButton splitMenuButton = new SplitMenuButton();

Font font = Font.font("Courier New", FontWeight.BOLD, 36);
splitMenuButton.setFont(font);
```

SplitMenuButton → Set SplitMenuButton Menu Items

You can set the menu items to display in the menu part of a JavaFX SplitMenuButton via its MenuItem collection returned by getItems(). Each menu item is represented by a MenuItem object.

Here is an example of setting three menu items on a JavaFX SplitMenuButton:

```
MenuItem choice1 = new MenuItem("Choice 1");
MenuItem choice2 = new MenuItem("Choice 2");
MenuItem choice3 = new MenuItem("Choice 3");

button.getItems().addAll(choice1, choice2, choice3);
```

SplitMenuButton → Respond to Menu Item Selection

The JavaFX SplitMenuButton works similarly to the **JavaFX MenuButton** when it comes to responding to selected menu items. To respond to selection of a menu item in a JavaFX *SplitMenuButton* you must set an action listener on each MenuItem added to the SplitMenuButton.

Here is an example of responding to menu item selection in a JavaFX SplitMenuButton by setting action listeners on its MenuItem instances:

```
MenuItem choice1 = new MenuItem("Choice 1");
MenuItem choice2 = new MenuItem("Choice 2");
MenuItem choice3 = new MenuItem("Choice 3");

choice1.setOnAction((e)-> {
   System.out.println("Choice 1 selected");
});
choice2.setOnAction((e)-> {
   System.out.println("Choice 2 selected");
});
choice3.setOnAction((e)-> {
   System.out.println("Choice 3 selected");
});
```

Code Explanation:

In this example the action listeners simply print a text to the console. In a real application you would probably want to store information about what action was selected, or take some other action, rather than just printing a text out to the console.

SplitMenuButton → Respond to Button Click

You can respond to JavaFX SplitMenuButton button clicks by setting an action listener on it.

Here is an example of setting an action listener on a JavaFX SplitMenuButton:

```
splitMenuButton.setOnAction((e) -> {
   System.out.println("SplitMenuButton clicked!");
});
```

This example uses a **Java Lambda Expression** as action listener. When the button is clicked, the text SplitMenuButton clicked! will be printed to the console.

SplitMenuButton → SplitMenuButton vs. other Controls

SplitMenuButton vs. MenuButton, ChoiceBox and ComboBox

You might be wondering what the difference is between a ***JavaFX SplitMenuButton*** and a **JavaFX MenuButton**, **JavaFX ChoiceBox** and a **JavaFX ComboBox**. I will try to explain that below.

The SplitMenuButton and MenuButton controls are *buttons*. That means, that they are intended for your application to respond to clicks on either one of the menu items, or in the case of the SplitMenuButton - the primary button or one of the menu items. Use one of these two controls when you want an immediate action to follow when the user clicks / selects a menu item. Use the SplitMenuButton when one of the choices is done more often than the rest. Use the button part for the most selected choice, and the menu items for the less often selected choices.

The ChoiceBox and ComboBox merely store internally what choices the user has made among their menu items. They are not designed for immediate action upon menu item selection. Use these controls in forms where the user has to make several choices before finally clicking either an "OK" or "Cancel" button. When on of these buttons are clicked, you can read what menu item is chosen from the ChoiceBox or ComboBox.

ToggleButton → Introduction

- A JavaFX ToggleButton is a button that can be selected or not selected. Like a button that stays in when you press it, and when you press it the next time it comes out again. Toggled - not toggled.

- The JavaFX ToggleButton is represented by the class javafx.scene.control.ToggleButton .

ToggleButton → Creating a ToggleButton

You create a JavaFX ToggleButton by creating an instance of the ToggleButton class.

Here is an example of creating a JavaFX ToggleButton instance:

```
ToggleButton toggleButton1 = new ToggleButton("Left");
```

This example creates a ToggleButton with the text Left on.

ToggleButton → Adding a ToggleButton to Scene Graph

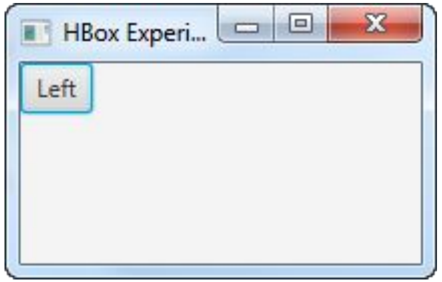
To make a ToggleButton visible you must add it to the JavaFX scene graph. This means adding it to a Scene, or as child of a layout which is attached to a Scene object.

Here is an example that attaches a JavaFX ToggleButton to the scene graph:

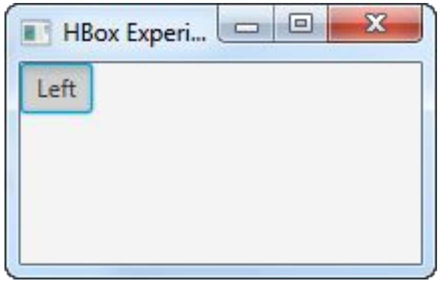
```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.ToggleButton;

public class ToggleButtonExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("HBox Experiment 1");
       ToggleButton toggleButton1 = new ToggleButton("Left");
       HBox hbox = new HBox(toggleButton1);
       Scene scene = new Scene(hbox, 200, 100);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

ToggleButton → Adding a ToggleButton to Scene Graph

The application resulting from running the above example code is illustrated in the following two screenshots.



← The first screenshot shows a `ToggleButton` which is not pressed



← The second screenshot shows the same `ToggleButton` pressed (selected, activated etc.):

ToggleButton → ToggleButton Text

You can set or change the text of a JavaFX ToggleButton via its setText() method.

Here is an example of changing the text of a JavaFX ToggleButton via setText():

```
ToggleButton toggleButton = new ToggleButton("Toggle This!");

toggleButton.setText("New Text");
```

ToggleButton → ToggleButton Font

You can set the font to use to render the button text of a JavaFX ToggleButton via its setFont() method.

Here is an example of setting the font of a JavaFX ToggleButton:

```
ToggleButton toggleButton = new ToggleButton("Toggle This!");

Font arialFontBold36  = Font.font("Arial", FontWeight.BOLD, 36);

toggleButton.setFont(arialFontBold36);
```

ToggleButton → Reading Selected State

The ToggleButton class has a method named isSelected which lets you determine if the ToggleButton is selected (pressed) or not. The isSelected() method returns a boolean with the value true if the ToggleButton is selected, and false if not.

Here is an example:

```
boolean isSelected = toggleButton1.isSelected();
```

ToggleButton → ToggleGroup

You can group JavaFX ToggleButton instances into a ToggleGroup. A ToggleGroup allows at most one ToggleButton to be toggled (pressed) at any time. The ToggleButton instances in a ToggleGroup thus functions similarly to radio buttons.

Here is a JavaFX ToggleGroup example:

```
ToggleButton toggleButton1 = new ToggleButton("Left");
ToggleButton toggleButton2 = new ToggleButton("Right");
ToggleButton toggleButton3 = new ToggleButton("Up");
ToggleButton toggleButton4 = new ToggleButton("Down");
ToggleGroup toggleGroup = new ToggleGroup();

toggleButton1.setToggleGroup(toggleGroup);
toggleButton2.setToggleGroup(toggleGroup);
toggleButton3.setToggleGroup(toggleGroup);
toggleButton4.setToggleGroup(toggleGroup);
```

ToggleButton → ToggleGroup

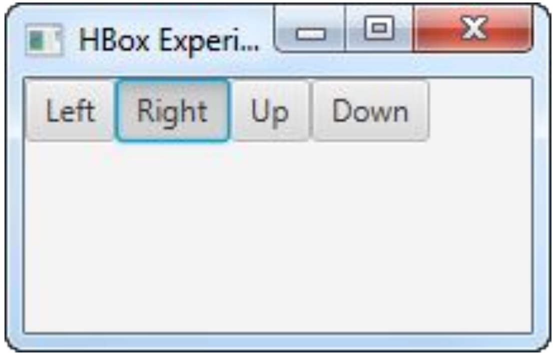
Here is a full example that adds the 4 ToggleButton instances to a ToggleGroup, and adds them to the scene graph too:

ToggleButton → ToggleGroup

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ToggleButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class ToggleButtonExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("HBox Experiment 1");
       ToggleButton toggleButton1 = new ToggleButton("Left");
       ToggleButton toggleButton2 = new ToggleButton("Right");
       ToggleButton toggleButton3 = new ToggleButton("Up");
       ToggleButton toggleButton4 = new ToggleButton("Down");
       ToggleGroup toggleGroup = new ToggleGroup();
       toggleButton1.setToggleGroup(toggleGroup);
       toggleButton2.setToggleGroup(toggleGroup);
       toggleButton3.setToggleGroup(toggleGroup);
       toggleButton4.setToggleGroup(toggleGroup);
       HBox hbox = new HBox(toggleButton1, toggleButton2, toggleButton3, toggleButton4);
       Scene scene = new Scene(hbox, 200, 100);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

ToggleButton → ToggleGroup

The resulting applications looks like this:



ToggleButton → Reading Selected State of a ToggleGroup

You can read which ToggleButton of a ToggleGroup is selected (pressed) using the getSelectedToggle() method, like this:

```
ToggleButton selectedToggleButton = (ToggleButton) toggleGroup.getSelectedToggle();
```

If no ToggleButton is selected the getSelectedToggle() method returns null .

RadioButton → Introduction

- A JavaFX RadioButton is a button that can be selected or not selected.
- The RadioButton is very similar to the **JavaFX ToggleButton**, but with the difference that a RadioButton cannot be "unselected" once selected. If RadioButtons are part of a ToggleGroup, then once a RadioButton has been selected for the first time, there must be one RadioButton selected in the ToggleGroup .
- The JavaFX RadioButton is represented by the class javafx.scene.control.RadioButton.
- The RadioButton class is a subclass of the ToggleButton class.

RadioButton → Creating a RadioButton

You create a JavaFX RadioButton using its constructor.

Here is a JavaFX RadioButton instantiation example:

```
RadioButton radioButton1 = new RadioButton("Left");
```

The String passed as parameter to the RadioButton constructor is displayed next to the RadioButton.

RadioButton → Adding a RadioButton to the Scene Graph

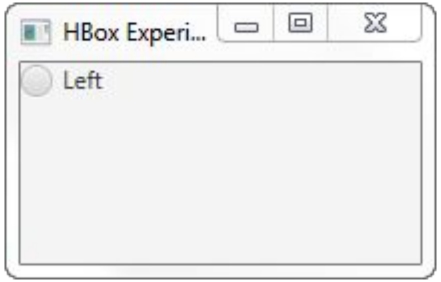
To make a RadioButton visible you must add it to the scene graph of your JavaFX application. This means adding the RadioButton to a Scene, or as child of a layout which is attached to a Scene object.

Here is an example that attaches a JavaFX RadioButton to the scene graph:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.RadioButton;

public class RadioButtonExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("HBox Experiment 1");
       RadioButton radioButton1 = new RadioButton("Left");
       HBox hbox = new HBox(radioButton1);
       Scene scene = new Scene(hbox, 200, 100);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

RadioButton → Adding a RadioButton to the Scene Graph

The application resulting from running this example looks like this:



RadioButton → Reading Selected State

The RadioButton class has a method named isSelected which lets you determine if the RadioButton is selected or not. The isSelected() method returns a boolean with the value true if the RadioButton is selected, and false if not.

Here is an example:

```
boolean isSelected = radioButton1.isSelected();
```

RadioButton → ToggleGroup

You can group JavaFX RadioButton instances into a ToggleGroup. A ToggleGroup allows at most one RadioButton to be selected at any time.

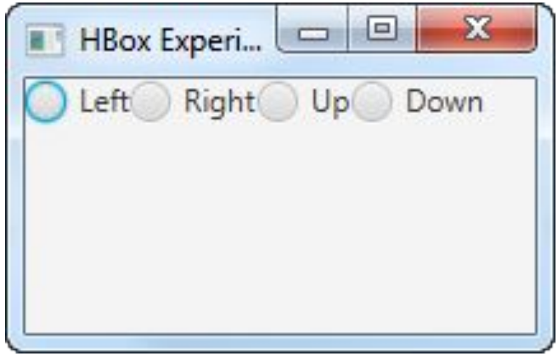
Here is a JavaFX ToggleGroup example:

RadioButton → ToggleGroup

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;

public class RadioButtonExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("HBox Experiment 1");
       RadioButton radioButton1 = new RadioButton("Left");
       RadioButton radioButton2 = new RadioButton("Right");
       RadioButton radioButton3 = new RadioButton("Up");
       RadioButton radioButton4 = new RadioButton("Down");
       ToggleGroup radioGroup = new ToggleGroup();
       radioButton1.setToggleGroup(radioGroup);
       radioButton2.setToggleGroup(radioGroup);
       radioButton3.setToggleGroup(radioGroup);
       radioButton4.setToggleGroup(radioGroup);
       HBox hbox = new HBox(radioButton1, radioButton2, radioButton3, radioButton4);
       Scene scene = new Scene(hbox, 200, 100);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

RadioButton → ToggleGroup

The application resulting from running this example looks like this:



RadioButton → Reading Selected State of a ToggleGroup

You can read which RadioButton of a ToggleGroup is selected using the getSelectedToggle() method, like this:

```
RadioButton selectedRadioButton = (RadioButton) toggleGroup.getSelectedToggle();
```

If no RadioButton is selected the getSelectedToggle() method returns null .

CheckBox → Introduction

- A JavaFX CheckBox is a button which can be in three different states: Selected, not selected and unknown (indeterminate).
- The JavaFX CheckBox control is represented by the class javafx.scene.control.CheckBox.

CheckBox → Creating a CheckBox

You create a JavaFX CheckBox control via the CheckBox constructor.

Here is a JavaFX CheckBox instantiation example:

```
CheckBox checkBox1 = new CheckBox("Green");
```

The String passed to the CheckBox constructor is displayed next to the CheckBox control.

CheckBox → Adding a CheckBox to the Scene Graph

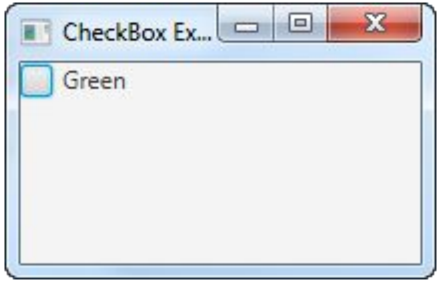
To make a JavaFX CheckBox control visible you must add it to the scene graph of your JavaFX application. That means adding the CheckBox control to a Scene object, or to some layout component which is itself added to a Scene object.

Here is an example showing how to add a CheckBox to the scene graph:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.CheckBox;

public class CheckBoxExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("CheckBox Experiment 1");
       CheckBox checkBox1 = new CheckBox("Green");
       HBox hbox = new HBox(checkBox1);
       Scene scene = new Scene(hbox, 200, 100);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

CheckBox → Adding a CheckBox to the Scene Graph

The application resulting from running this code looks like this:



CheckBox → Reading Selected State

You can read the selected state of a CheckBox via its method isSelected(). Here is an example of how calling isSelected() looks:

```
boolean isSelected = checkBox1.isSelected();
```

CheckBox → Allowing Indeterminate State

As mentioned earlier a JavaFX `CheckBox` can be in an ***indeterminate*** *state* which means that is is neither selected, nor not selected. The user simply has not interacted with the `CheckBox` yet.

By default a `CheckBox` is not allowed to be in the indeterminate state. You can set if a `CheckBox` is allowed to be in an indeterminate state using the method `setAllowIndeterminate()`.

Here is an example of allowing the indeterminate state for a `CheckBox`:

```
checkBox1.setAllowIndeterminate(true);
```

CheckBox → Reading Indeterminate State

You can read if a CheckBox is in the indeterminate state via its isIndeterminate() method. Here is an example of checking if a CheckBox is in the indeterminate state:

```
boolean isIndeterminate = checkBox1.isIndeterminate();
```

Note, that if a CheckBox is not in the indeterminate state, it is either selected or not selected, which can be seen via its isSelected() method shown earlier.

ChoiceBox → Introduction

- The JavaFX ChoiceBox control enables users to choose an option from a predefined list of choices.
- The JavaFX ChoiceBox control is represented by the class javafx.scene.control.ChoiceBox .
- This JavaFX ChoiceBox section will explain how to use the ChoiceBox class.

ChoiceBox → Creating a ChoiceBox

You create a ChoiceBox simply by creating a new instance of the ChoiceBox class.

Here is a JavaFX ChoiceBox instantiation example:

```
ChoiceBox choiceBox = new ChoiceBox();
```

ChoiceBox → Adding Choices to a ChoiceBox

You can add choices to a ChoiceBox by obtaining its item collection and add items to it.

Here is an example that adds choices to a JavaFX ChoiceBox :

```
choiceBox.getItems().add("Choice 1");
choiceBox.getItems().add("Choice 2");
choiceBox.getItems().add("Choice 3");
```

ChoiceBox → Adding a ChoiceBox to the Scene Graph

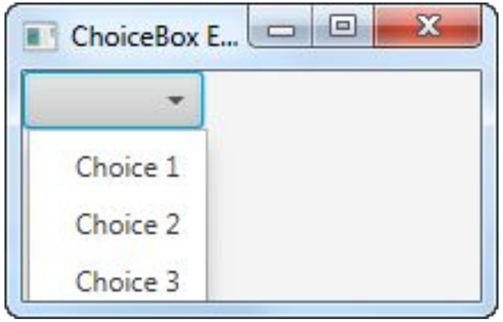
To make a ChoiceBox visible you must add it to the scene graph. This means that you must add the ChoiceBox to a Scene object or to some layout component which is then attached to the Scene object.

Here is an example showing how to add a JavaFX ChoiceBox to the scene graph:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.ChoiceBox;

public class ChoiceBoxExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("ChoiceBox Experiment 1");
       ChoiceBox choiceBox = new ChoiceBox();
       choiceBox.getItems().add("Choice 1");
       choiceBox.getItems().add("Choice 2");
       choiceBox.getItems().add("Choice 3");
       HBox hbox = new HBox(choiceBox);
       Scene scene = new Scene(hbox, 200, 100);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

ChoiceBox → Adding Choices to a ChoiceBox

The application resulting from running this example would look similar to this:



ChoiceBox → Reading the Selected Value

You can read the selected value of a ChoiceBox via its getValue() method. If no choice is selected, the getValue() method returns null.

Here is an example of calling getValue():

```
String value = (String) choiceBox.getValue();
```

ChoiceBox → Listening for Selection

It is possible to listen for selection changes in a JavaFX ChoiceBox by setting an action listener on the ChoiceBox via its setOnAction() method.

Here is an example of setting an action listener on a ChoiceBox which reads what value was selected in the ChoiceBox:

```
ChoiceBox choiceBox = new ChoiceBox();

choiceBox.getItems().add("Choice 1");
choiceBox.getItems().add("Choice 2");
choiceBox.getItems().add("Choice 3");

choiceBox.setOnAction((event) -> {
   int selectedIndex = choiceBox.getSelectionModel().getSelectedIndex();
   Object selectedItem = choiceBox.getSelectionModel().getSelectedItem();

   System.out.println("Selection made: [" + selectedIndex + "] " + selectedItem);
   System.out.println("   ChoiceBox.getValue(): " + choiceBox.getValue());
});
```

ComboBox → Introduction

- The JavaFX ComboBox control enables users to choose an option from a predefined list of choices, or type in another value if none of the predefined choices matches what the user want to select.
- The JavaFX ComboBox control is represented by the class javafx.scene.control.ComboBox .
- This JavaFX ComboBox tutorial will explain how to use the ComboBox class.

ComboBox → Creating a ComboBox

You create a ComboBox simply by creating a new instance of the ComboBox class.

Here is a JavaFX ComboBox instantiation example:

```
ComboBox comboBox = new ComboBox();
```

ComboBox → Adding Choices to a ComboBox

You can add choices to a ComboBox by obtaining its item collection and add items to it.

Here is an example that adds choices to a JavaFX ComboBox :

```
comboBox.getItems().add("Choice 1");
comboBox.getItems().add("Choice 2");
comboBox.getItems().add("Choice 3");
```

ComboBox → Adding a ComboBox to the Scene Graph

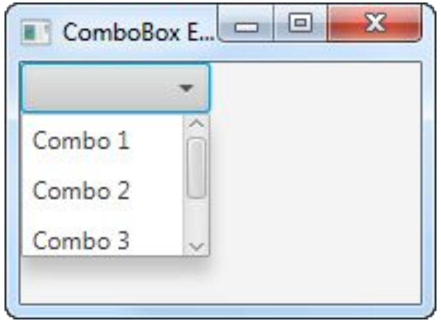
To make a ComboBox visible you must add it to the scene graph. This means that you must add the ComboBox to a Scene object or to some layout component which is then attached to the Scene object.

Here is an example showing how to add a JavaFX ComboBox to the scene graph:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.ComboBox;

public class ComboBoxExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("ComboBox Experiment 1");
       ComboBox comboBox = new ComboBox();
       comboBox.getItems().add("Choice 1");
       comboBox.getItems().add("Choice 2");
       comboBox.getItems().add("Choice 3");
       HBox hbox = new HBox(comboBox);
       Scene scene = new Scene(hbox, 200, 120);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

ComboBox → Adding a ComboBox to the Scene Graph

The application resulting from running this example would look similar to this:



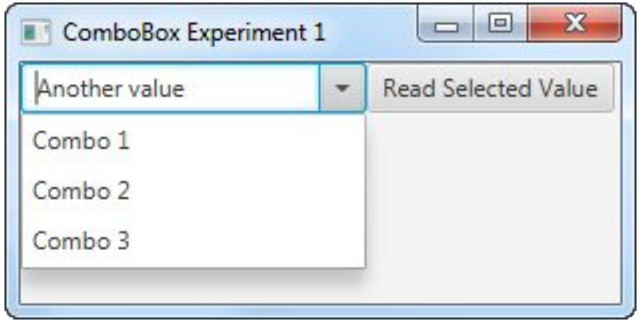
ComboBox → Making the ComboBox Editable

A ComboBox is not editable by default. That means, that by default the user cannot enter anything themselves, but only choose from the predefined list of options. To make a ComboBox editable you must call the setEditable() method of the ComboBox.

Here is an example making a JavaFX ComboBox editable:

```java
comboBox.setEditable(true);
```

Once the ComboBox is editable the user can type in values into the ComboBox. The entered value is also read via the getValue() method as explained earlier.



← This screenshot shows a JavaFX ComboBox which is editable, and with a custom value entered:

ComboBox → Reading the Selected Value

You can read the selected value of a ComboBox via its getValue() method. If no choice is selected, the getValue() method returns null.

Here is an example of calling getValue():

```
String value = (String) comboBox.getValue();
```

ComboBox → Listening for Selection

It is possible to listen for selection changes in a JavaFX ComboBox by setting an action listener on the ComboBox via its setOnAction() method.

Here is an example of setting an action listener on a ComboBox which reads what value was selected in the ComboBox:

```java
ComboBox comboBox = new ComboBox();

comboBox.getItems().add("Choice 1");
comboBox.getItems().add("Choice 2");
comboBox.getItems().add("Choice 3");

comboBox.setOnAction((event) -> {
   int selectedIndex = comboBox.getSelectionModel().getSelectedIndex();
   Object selectedItem = comboBox.getSelectionModel().getSelectedItem();
   System.out.println("Selection made: [" + selectedIndex + "] " + selectedItem);
   System.out.println("   ComboBox.getValue(): " + comboBox.getValue());
});
```

ListView → Introduction

- The JavaFX ListView control enables users to choose one or more options from a predefined list of choices.
- The JavaFX ListView control is represented by the class javafx.scene.control.ListView .
- This JavaFX ListView tutorial will explain how to use the ListView class.

ListView → Creating a ListView

You create a ListView simply by creating a new instance of the ListView class.

Here is a JavaFX ListView instantiation example:

```
ListView listView = new ListView();
```

ListView → Adding Items to a ListView

You can add items (options) to a ListView by obtaining its item collection and add items to it.

Here is an example that adds items to a JavaFX ListView :

```
listView.getItems().add("Item 1");
listView.getItems().add("Item 2");
listView.getItems().add("Item 3");
```

ListView → Adding a ListView to the Scene Graph

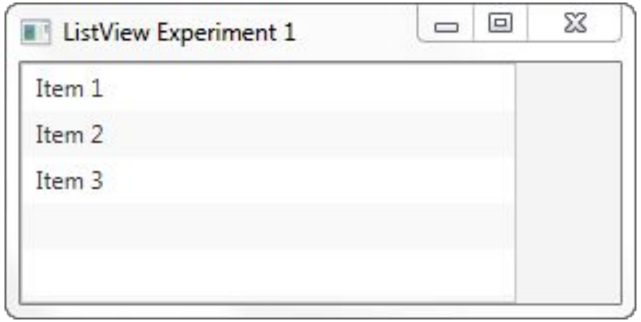
To make a ListView visible you must add it to the scene graph. This means that you must add the ListView to a Scene object or to some layout component which is then attached to the Scene object.

Here is an example showing how to add a JavaFX ListView to the scene graph:

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.ListView;

public class ListViewExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("ListView Experiment 1");
       ListView listView = new ListView();
       listView.getItems().add("Item 1");
       listView.getItems().add("Item 2");
       listView.getItems().add("Item 3");
       HBox hbox = new HBox(listView);
       Scene scene = new Scene(hbox, 300, 120);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

ListView → Adding a ListView to the Scene Graph

The application resulting from running this example would look similar to this screenshot:



Notice how the ListView shows multiple options by default. You can set a height and width for a ListView, but you cannot set explicitly how many items should be visible. The height determines that based on the height of each item displayed.

If there are more items in the ListView than can fit into its visible area, the ListView will add scroll bars so the user can scroll up and down over the items.

ListView → Reading the Selected Value

You can read the selected indexes of a ListView via its SelectionModel.

Here is an example showing how to read the selected indexes of a JavaFX ListView:

```
ObservableList selectedIndices = listView.getSelectionModel().getSelectedIndices();
```

The OberservableList will contain Integer objects representing the indexes of the selected items in the ListView.

ListView → Reading the Selected Value

Here is a full JavaFX example with a button added which reads the selected items of the ListView when clicked:

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.control.Button;
import javafx.scene.control.ListView;
import javafx.scene.control.SelectionMode;
import javafx.collections.ObservableList;

public class ListViewExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("ListView Experiment 1");
       ListView listView = new ListView();
       listView.getItems().add("Item 1");
       listView.getItems().add("Item 2");
       listView.getItems().add("Item 3");
       Button button = new Button("Read Selected Value");
       button.setOnAction(event -> {
           ObservableList selectedIndices = listView.getSelectionModel().getSelectedIndices();
           for(Object o : selectedIndices){
               System.out.println("o = " + o + " (" + o.getClass() + ")");
           }
       });
       VBox vBox = new VBox(listView, button);
       Scene scene = new Scene(vBox, 300, 120);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

ListView → Allowing Multiple Items to be Selected

To allow multiple items in the ListView to be selected you need to set the corresponding selection mode on the ListView selection model.

Here is an example of setting the selection mode on the JavaFX ListView:

```
listView.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
```

Once you have set the SelectionMode.MULTIPLE on the ListView selection model, the user can select multiple items in the ListView by holding down SHIFT or CTRL when selecting additional items after the first selected item.

ListView → Allowing Multiple Items to be Selected

Here is a full JavaFX example that shows how to set a ListView into multiple selection mode, including a button which when clicked will write out the indices of the selected items in the ListView :

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.control.Button;
import javafx.scene.control.ListView;
import javafx.scene.control.SelectionMode;
import javafx.collections.ObservableList;

public class ListViewExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("ListView Experiment 1");
       ListView listView = new ListView();
       listView.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
       listView.getItems().add("Item 1");
       listView.getItems().add("Item 2");
       listView.getItems().add("Item 3");
       Button button = new Button("Read Selected Value");
       button.setOnAction(event -> {
           ObservableList selectedIndices = listView.getSelectionModel().getSelectedIndices();
           for(Object o : selectedIndices){
               System.out.println("o = " + o + " (" + o.getClass() + ")");
           }
       });
       VBox vBox = new VBox(listView, button);
       Scene scene = new Scene(vBox, 300, 120);
       primaryStage.setScene(scene);
       primaryStage.show();
   }
   public static void main(String[] args) {
       Application.launch(args);
   }
}
```

DatePicker → Introduction

- A JavaFX DatePicker control enables the user to enter a date or choose a date from a wizard-like popup dialog.

- The popup dialog shows only valid dates, so this is an easier way for users to choose a date and ensure that both the date and date format entered in the date picker text field is valid.

- The JavaFX DatePicker is represented by the class javafx.scene.control.DatePicker .

- The DatePicker is a subclass of the ComboBox class, and thus shares some similarities with this class.

DatePicker → Creating a DatePicker

You create a DatePicker control via the constructor of the DatePicker class.

Here is a JavaFX DatePicker instantiation example:

```
DatePicker datePicker = new DatePicker();
```

DatePicker → Adding a DatePicker to the Scene Graph

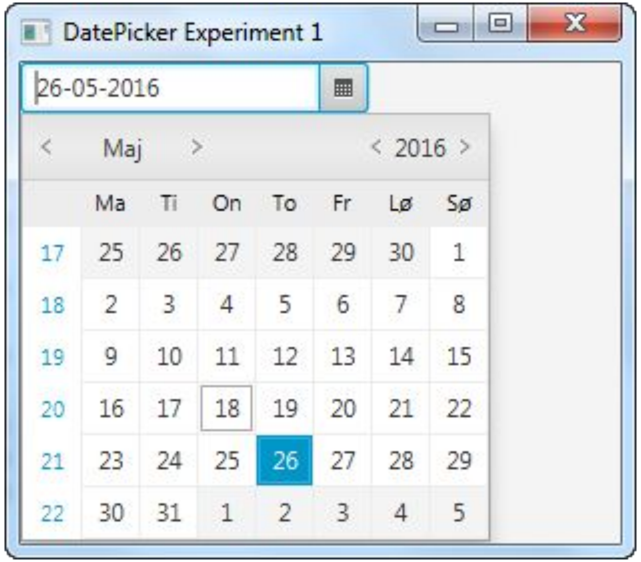
To make a DatePicker visible it must be added to the JavaFX scene graph. This means adding it to a Scene object, or to a layout component which is added to a Scene object.

Here is an example showing how to add a JavaFX DatePicker to the scene graph:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.DatePicker;

public class DatePickerExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("Button Experiment 1");
       DatePicker datePicker = new DatePicker();
       HBox hbox = new HBox(datePicker);
       Scene scene = new Scene(hbox, 200, 100);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

DatePicker → Adding a DatePicker to the Scene Graph

The application resulting from running this example would look similar to this:



DatePicker Experiment 1

26-05-2016

‹ Maj › ‹ 2016 ›

	Ma	Ti	On	To	Fr	Lø	Sø
17	25	26	27	28	29	30	1
18	2	3	4	5	6	7	8
19	9	10	11	12	13	14	15
20	16	17	18	19	20	21	22
21	23	24	25	26	27	28	29
22	30	31	1	2	3	4	5

DatePicker → Reading the Selected Date

Reading the date selected in the DatePicker can be done using its getValue() method.

Here is an example of reading the selected date from a DatePicker:

```
LocalDate value = datePicker.getValue();
```

The getValue() returns a **LocalDate** object representing the date selected in the DatePicker.

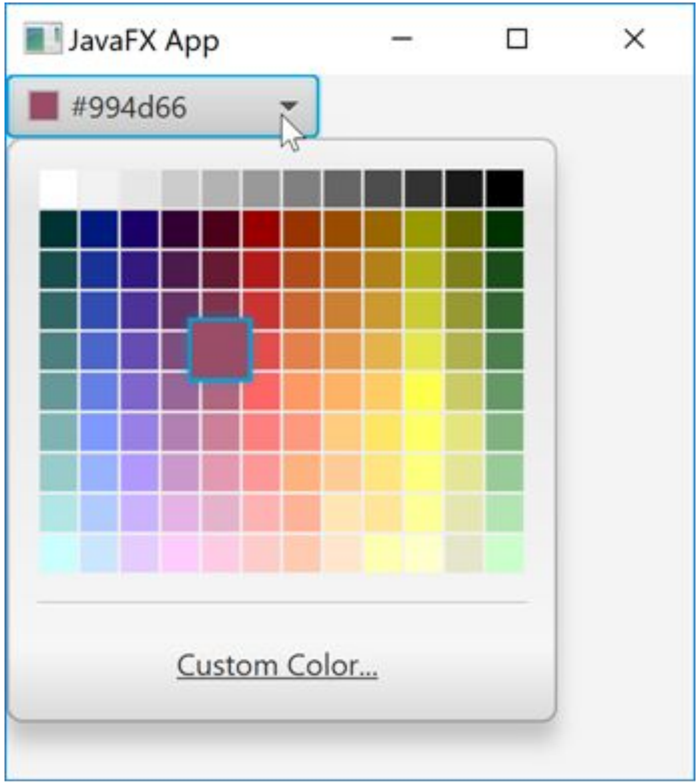
DatePicker → Reading the Selected Date

Here is a full example with a button added to extract the selected date in the DatePicker when the button is clicked:

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.Button;
import javafx.scene.control.DatePicker;
import java.time.LocalDate;

public class DatePickerExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("DatePicker Experiment 1");
       DatePicker datePicker = new DatePicker();
       Button button = new Button("Read Date");
       button.setOnAction(action -> {
           LocalDate value = datePicker.getValue();
       });
       HBox hbox = new HBox(datePicker);
       Scene scene = new Scene(hbox, 300, 240);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

ColorPicker → Introduction

- The *JavaFX ColorPicker* control enables the user to choose a color in a popup dialog.
- The chosen color can later be read from the ColorPicker by your JavaFX application.
- The JavaFX ColorPicker control is represented by the class javafx.scene.control.ColorPicker.
- Here is a screenshot of an opened JavaFX ColorPicker:



ColorPicker → Full ColorPicker Example

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.control.ColorPicker;
import javafx.scene.paint.Color;

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

   @Override
   public void start(Stage primaryStage) {
       primaryStage.setTitle("JavaFX App");
       ColorPicker colorPicker = new ColorPicker();
       Color value = colorPicker.getValue();
       VBox vBox = new VBox(colorPicker);
       Scene scene = new Scene(vBox, 960, 600);
       primaryStage.setScene(scene);
       primaryStage.show();
   }
}
```

Here is a full JavaFX ColorPicker example so you can see what the code looks like:

ColorPicker → Create a ColorPicker

In order to use a JavaFX ColorPicker you must first create an instance of the ColorPicker class.

Here is an example of creating a JavaFX ColorPicker:

```
ColorPicker colorPicker = new ColorPicker();
```

ColorPicker → Get Chosen Color

To read the color chosen in a JavaFX ColorPicker you call its getValue() method.

Here is an example of getting the chosen color in a JavaFX ColorPicker:

```
Color value = colorPicker.getValue();
```

TextField → Introduction

- A JavaFX TextField control enables users of a JavaFX application to enter text which can then be read by the application.
- The JavaFX TextField control is represented by the class javafx.scene.control.TextField .

TextField → Creating a TextField

You create a TextField control by creating an instance of the TextField class.

Here is a JavaFX TextField instantiation example:

```
TextField textField = new TextField();
```

TextField → Adding a TextField to the Scene Graph

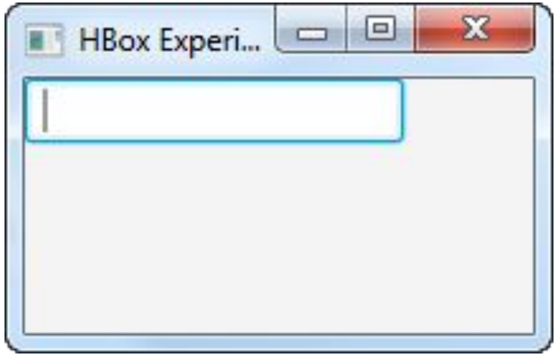
For a JavaFX TextField to be visible the TextField object must be added to the scene graph. This means adding it to a Scene object, or as child of a layout which is attached to a Scene object.

Here is an example that attaches a JavaFX TextField to the scene graph:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class TextFieldExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("HBox Experiment 1");
       TextField textField = new TextField();
       HBox hbox = new HBox(textField);
       Scene scene = new Scene(hbox, 200, 100);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

TextField → Adding a TextField to the Scene Graph

The result of running the above JavaFX TextField example is an application that looks like this:



TextField → Getting the Text of a TextField

You can get the text entered into a TextField using its getText() method which returns a String.

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

public class TextFieldExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("HBox Experiment 1");
        TextField textField = new TextField();
        Button button = new Button("Click to get text");
        button.setOnAction(action -> {
            System.out.println(textField.getText());
        });
        HBox hbox = new HBox(textField, button);
        Scene scene = new Scene(hbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
   }

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

Here is a full example that shows a TextField and a Button and which reads the text entered into the TextField when the button is clicked:

TextField → Setting the Text of a TextField

You can set the text of a TextField using its setText() method. This is often useful when you need to set the initial value for at text field that is part of a form. For instance, editing an existing object or record.

Here is a simple example of setting the text of a JavaFX TextField:

```
textField.setText("Initial value");
```

PasswordField → Introduction

- A JavaFX PasswordField control enables users of a JavaFX application to enter password which can then be read by the application.
- The PasswordField control does not show the texted entered into it.
- Instead it shows a circle for each character entered.
- The JavaFX PasswordField control is represented by the class javafx.scene.control.PasswordField .

PasswordField → Creating a PasswordField

You create a PasswordField control by creating an instance of the PasswordField class.

Here is a JavaFX PasswordField instantiation example:

```
PasswordField passwordField = new PasswordField();
```

PasswordField → Adding PasswordField to Scene Graph

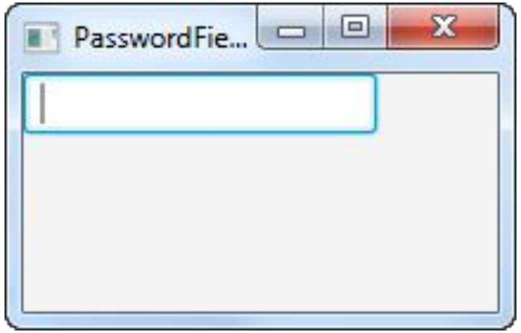
For a JavaFX PasswordField to be visible the PasswordField object must be added to the scene graph. This means adding it to a Scene object, or as child of a layout which is attached to a Scene object.

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.PasswordField;

public class PasswordFieldExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("PasswordField Experiment 1");
       PasswordField passwordField = new PasswordField();
       HBox hbox = new HBox(passwordField);
       Scene scene = new Scene(hbox, 200, 100);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

Here is an example that attaches a JavaFX PasswordField to the scene graph:

PasswordField → Adding PasswordField to Scene Graph

The result of running the above JavaFX PasswordField example is an application that looks like this:



PasswordField → Getting the Text of a PasswordField

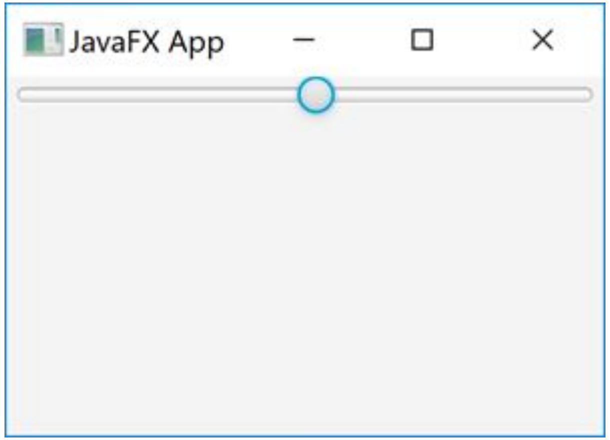
You can get the text entered into a PasswordField using its getText() method which returns a String.

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

public class PasswordFieldExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("PasswordField Experiment 1");
       PasswordField passwordField = new PasswordField();
       Button button = new Button("Click to get password");
       button.setOnAction(action -> {
           System.out.println(passwordField.getText());
       });
       HBox hbox = new HBox(passwordField, button);
       Scene scene = new Scene(hbox, 200, 100);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

Here is a full example that shows a PasswordField and a Button and which reads the text entered into the PasswordField when the button is clicked:

Slider → Introduction

- The *JavaFX Slider* control provides a way for the user to select a value within a given interval by sliding a handle to the desired point representing the desired value.
- The *JavaFX Slider* is represented by the JavaFX class javafx.scene.control.Slider.
- Here is a screenshot of how a JavaFX Slider looks:



Slider → JavaFX Slider Example

Here is a full JavaFX Slider code example:

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

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

   @Override
   public void start(Stage primaryStage) {
       primaryStage.setTitle("JavaFX App");
       Slider slider = new Slider(0, 100, 0);
       VBox vBox = new VBox(slider);
       Scene scene = new Scene(vBox, 960, 600);
       primaryStage.setScene(scene);
       primaryStage.show();
   }
}
```

Slider → Create a Slider

To use a JavaFX Slider you must first create an instance of the Slider class.

Here is an example of creating a JavaFX Slider instance:

```
Slider slider = new Slider(0, 100, 0);
```

The Slider constructor used above takes three parameters: The min value, the max value and the initial value.

- The min value is the value sliding the handle all the way to the left represents.
  - This is the beginning of the interval the user can select a value in.
- The max value is the value sliding the handle all the way to the right represents.
  - This the end of the interval the user can select a value in.
- The initial value is the value that the handle should be located at, when presented to the user at first.

Slider → Reading Slider Value

You can read the value of a Slider as selected by the user via the getValue() method.

Here is an example of reading the selected value of a JavaFX Slider:

```
double value = slider.getValue();
```

Slider → Major Tick Unit

You can set the major tick unit of a JavaFX Slider control. The major tick unit is how many units the value changes every time the user moves the handle of the Slider one tick.

Here is an example that sets the major tick unit of a JavaFX Slider to 8:

```
Slider slider = new Slider(0, 100, 0);

slider.setMajorTickUnit(8.0);
```

This Slider will have its value change with 8.0 up or down whenever the handle in the Slider is moved.

Slider → Minor Tick Count

You can set the minor tick count of a JavaFX Slider via the setMinorTickCount() method. The minor tick count specifies how many minor ticks there are between two of the major ticks.

Here is an example that sets the minor tick count to 2:

```
Slider slider = new Slider(0, 100, 0);

slider.setMajorTickUnit(8.0);

slider.setMinorTickCount(3);
```

The Slider configured here has 8.0 value units between each major tick, and in between each of these major ticks it has 3 minor ticks.

Slider → Snap Handle to Ticks

You can make the handle of the JavaFX Slider snap to the ticks using the Slider setSnapToTicks() method, passing a parameter value of true it.

Here is an example of making the JavaFX Slider snap its handle to the ticks:

```
slider.setSnapToTicks(true);
```

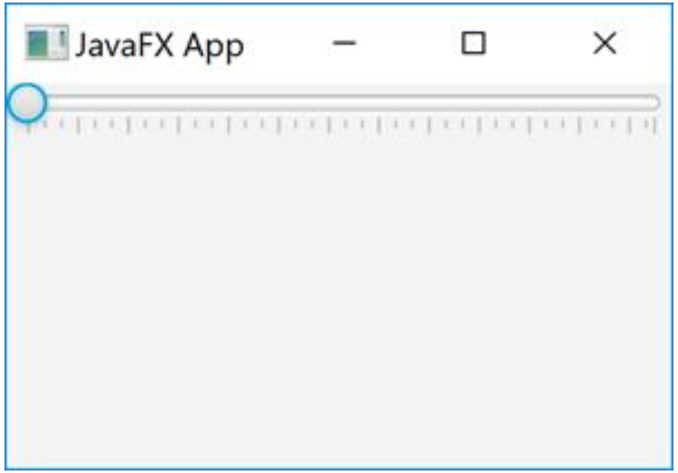
Slider → Show Tick Marks

You can make the *JavaFX Slider* show marks for the ticks when it renders the slider. You do so using its setShowTickMarks() method.

Here is an example of making a JavaFX Slider show tick marks:

```java
slider.setShowTickMarks(true);
```

Here is a screenshot of how a JavaFX Slider looks with tick marks shown:



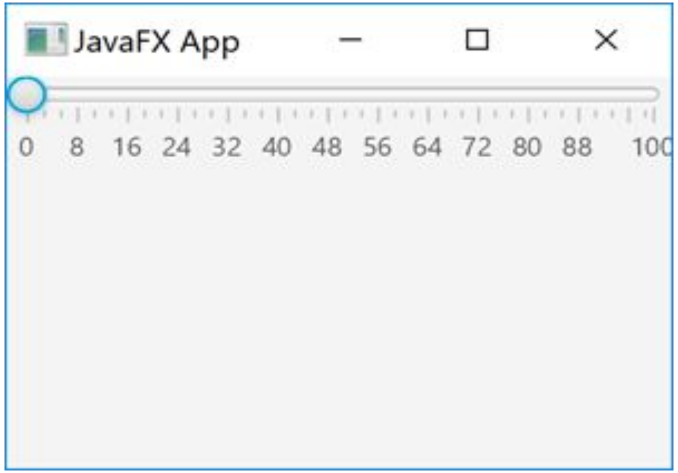
Slider → Show Tick Labels

You can make the JavaFX Slider show tick labels for the ticks when it renders the slider. You do so using its setShowTickLabels() method.

Here is an example of making a JavaFX Slider show tick labels:

```java
slider.setShowTickLabels(true);
```

Here is a screenshot of how a JavaFX Slider looks with tick marks and labels shown:



TextArea → Introduction

A JavaFX TextArea control enables users of a JavaFX application to enter text spanning multiple lines, which can then be read by the application.

The JavaFX TextArea control is represented by the class javafx.scene.control.TextArea .

TextArea → Creating a TextArea

You create a TextArea control by creating an instance of the TextArea class.

Here is a JavaFX TextArea instantiation example:

```
TextArea textArea = new TextArea();
```

TextArea → Adding a TextArea to the Scene Graph

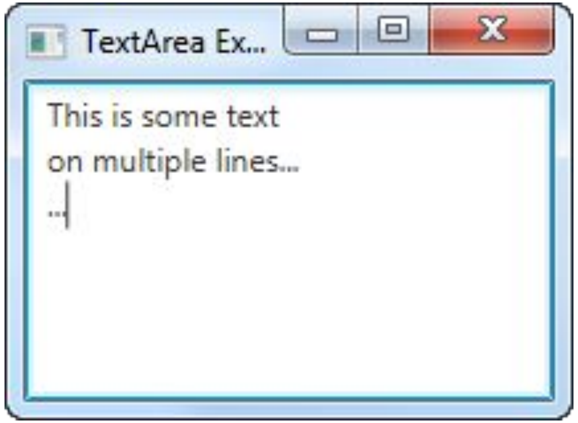
For a JavaFX TextArea to be visible the TextArea object must be added to the scene graph. This means adding it to a Scene object, or as child of a layout which is attached to a Scene object.

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

public class TextAreaExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("TextArea Experiment 1");
       TextArea textArea = new TextArea();
       VBox vbox = new VBox(textArea);
       Scene scene = new Scene(vbox, 200, 100);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

Here is an example that attaches a JavaFX TextArea to the scene graph

TextArea → Adding a TextArea to the Scene Graph

The result of running the above JavaFX TextArea example is an application that looks like this:



```
This is some text
on multiple lines...
...
```

TextArea → Reading the Text of a TextArea

You can read the text entered into a TextArea via its getText() method.

Here is an example of reading text of a JavaFX TextArea control via its getText() method:

```
String text = textArea.getText();
```

TextArea → Reading the Text of a TextArea

Here is a full example that shows a TextArea and a Button and which reads the text entered into the TextArea when the button is clicked:

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.control.Button;
import javafx.scene.control.TextArea;

public class TextAreaExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
       primaryStage.setTitle("TextArea Experiment 1");
       TextArea textArea = new TextArea();
       Button button = new Button("Click to get text");
       button.setMinWidth(50);
       button.setOnAction(action -> {
           System.out.println(textArea.getText());
           textArea.setText("Clicked!");
       });
       VBox vbox = new VBox(textArea, button);
       Scene scene = new Scene(vbox, 200, 100);
       primaryStage.setScene(scene);
       primaryStage.show();
   }

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

TextArea → Setting the Text of a TextArea

You can set the text of a TextArea control via its setText() method.

Here is an example of setting the text of a TextArea control via setText() :

```
textArea.setText("New Text");
```

ToolBar → Introduction

- The *JavaFX ToolBar* class (javafx.scene.control.ToolBar) is a horizontal or vertical bar containing buttons or icons that are typically used to select different tools of a JavaFX application.

- Actually, a JavaFX ToolBar can contain other JavaFX controls than just buttons and icons.

- In fact, you can insert any JavaFX control into a ToolBar.

ToolBar → Creating a ToolBar

In order to create a JavaFX ToolBar you must first instantiate it.

Here is an example of creating a JavaFX ToolBar instance:

```java
ToolBar toolBar = new ToolBar();
```

That is all it takes to create a JavaFX ToolBar.

ToolBar → Adding Items to a ToolBar

Once a JavaFX ToolBar has been created, you can add items (JavaFX components) to it. You add items to a ToolBar by obtaining its collection of items and adding the new item to that collection.

Here is an example of adding an item to a ToolBar:

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

toolBar.getItems().add(button);
```

ToolBar → Adding a ToolBar to the Scene Graph

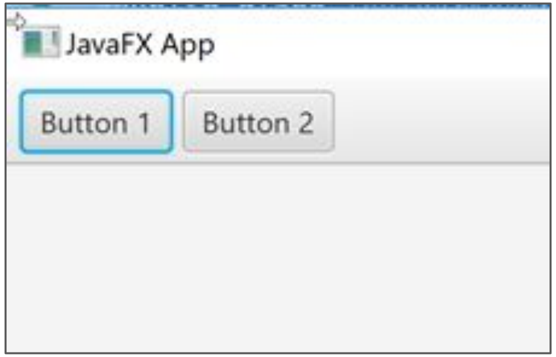
In order to make a JavaFX ToolBar visible, it must be added to the JavaFX scene graph.

Here is a full example that shows the creation of a JavaFX ToolBar and adding it to the JavaFX scene graph:

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

public class ToolBarExample extends Application {
   public static void main(String[] args) {
       launch(args);
   }
   @Override
   public void start(Stage primaryStage) {
       primaryStage.setTitle("JavaFX App");
       ToolBar toolBar = new ToolBar();
       Button button1 = new Button("Button 1");
       toolBar.getItems().add(button1);
       Button button2 = new Button("Button 2");
       toolBar.getItems().add(button2);
       VBox vBox = new VBox(toolBar);
       Scene scene = new Scene(vBox, 960, 600);
       primaryStage.setScene(scene);
       primaryStage.show();
   }
}
```

ToolBar → Adding a ToolBar to the Scene Graph

The JavaFX GUI resulting from this ToolBar example would look similar to this:



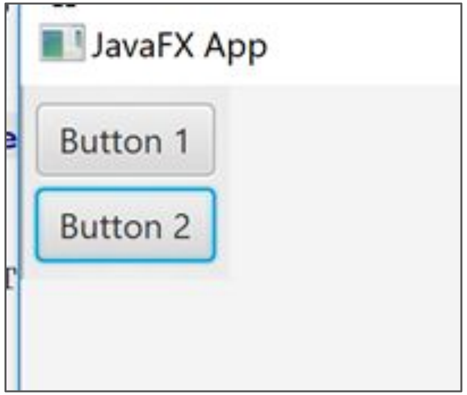
ToolBar → Vertical Oriented ToolBar

By default a JavaFX ToolBar displays the items added to it in a horizontal row. It is possible to get the ToolBar to display the items vertically instead, so the ToolBar becomes a vertical toolbar. To make the ToolBar display its items vertically, you call its setOrientation() method.

Here is an example of setting the orientation of a ToolBar to vertical:

```
toolBar.setOrientation(Orientation.VERTICAL);
```

Here is a screenshot of how the JavaFX ToolBar from the previous section looks in vertical orientation:



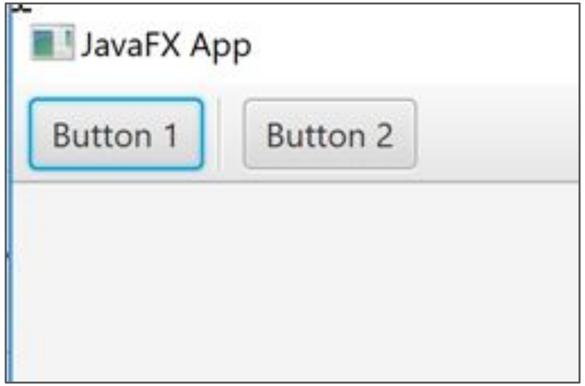
ToolBar → Separating Items in a ToolBar

You can add a visual separator to a JavaFX ToolBar. The visual separator is typically displayed as a vertical or horizontal line between the items in the ToolBar.

Here is an example of adding a separator to a ToolBar:

```
Button button1 = new Button("Button 1");
toolBar.getItems().add(button1);

toolBar.getItems().add(new Separator());

Button button2 = new Button("Button 2");
toolBar.getItems().add(button2);
```

Here is a screenshot of how a visual separator between items in a ToolBar looks:



Tooltip → Introduction

- The *JavaFX Tooltip* class (javafx.scene.control.Tooltip) can display a small popup with explanatory text when the user hovers the mouse over a JavaFX control.
- A Tooltip is a well-known feature of modern desktop and web GUIs.
- A Tooltip is useful to provide extra help text in GUIs where there is not space enough available to have an explanatory text visible all the time, e.g. in the button text.

Tooltip → Creating a Tooltip Instance

To use the JavaFX Tooltip class you must create a Tooltip instance. Here is an example of creating a JavaFX Tooltip instance:

```
Tooltip tooltip1 = new Tooltip("Creates a new file");
```

The text passed as parameter to the Tooltip constructor is the text displayed when the Tooltip is visible.

Tooltip → Adding a Tooltip to a JavaFX Component

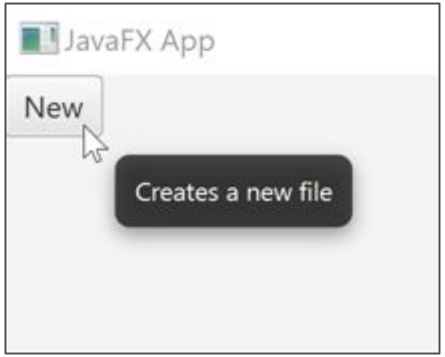
Once you have created a Tooltip instance you need to add it to a JavaFX component to make it active.

Here is an example of adding a Tooltip instance to a JavaFX Button:

```
Tooltip tooltip1 = new Tooltip("Creates a new file");

Button button1 = new Button("New");
button1.setTooltip(tooltip1);
```

Notice the call to Button's setTooltip() method. This is what causes the Tooltip instance to be visible when the mouse is hovered over the button.

Here is a screenshot showing how the resulting Tooltip could look:



Tooltip → Text Alignment

You can set the text alignment of the text inside the Tooltip box via its setTextAlignment() method.

Here is an example of setting the text alignment of a Tooltip:

```
tooltip1.setTextAlignment(TextAlignment.LEFT);
```

The class javafx.scene.text.TextAlignment contains four different constants that represent different kinds of text alignment. The four constants are:

- LEFT
- RIGHT
- CENTER
- JUSTIFY

The first three constants represents the left, right and center justification of text within the popup box. The last constant, JUSTIFY, will align the text with both the left and right edges of the popup box by increasing the space in between the words to make the text fit.

Notice that setting the text alignment may not result in a visible effect on the text alignment. That is because by default width of the popup box around the text is calculated based on the width of the text. If your text is just a single line, the text will almost always appear centered within the popup box. Text alignment first really takes effect when the popup box contains multiple lines of text, or if you set the width of the Tooltip explicitly (manually).

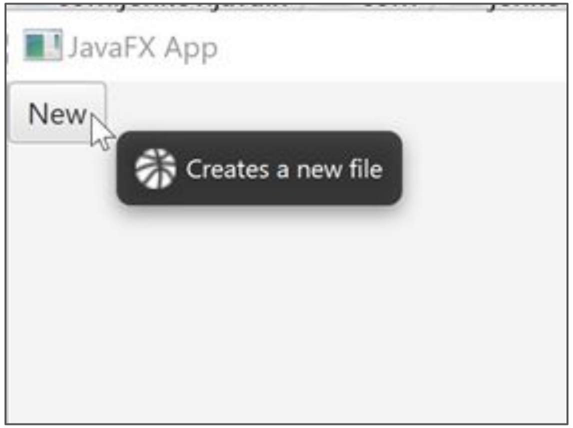
Tooltip → Tooltip Graphics

You can set a graphic icon for a Tooltip via the setGraphic() method.

Here is an example of setting a graphic icon for a Tooltip :

```
tooltip1.setGraphic(new ImageView("file:iconmonstr-basketball-1-16.png"));
```

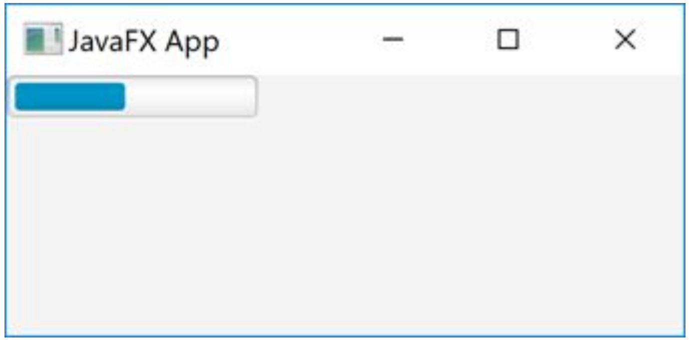
Here is an example screenshot illustrating how a Tooltip graphic could look:



Creates a new file

ProgressBar → Introduction

- The *JavaFX ProgressBar* is a control capable of displaying the progress of some task.
- The progress is set as a double value between 0 and 1, where 0 means no progress and 1 means full progress (task completed).
- The *JavaFX ProgressBar* control is represented by the javafx.scene.control.ProgressBar class.
- Here is a screenshot of how a *JavaFX ProgressBar* looks:



The ProgressBar in the above screenshot has its progress set to 0.5.

ProgressBar → JavaFX ProgressBar Example

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

public class ProgressBarExample extends Application {
   public static void main(String[] args) {
       launch(args);
   }
   @Override
   public void start(Stage primaryStage) {
       primaryStage.setTitle("JavaFX App");
       ProgressBar progressBar = new ProgressBar(0);
       progressBar.setProgress(0.5);
       VBox vBox = new VBox(progressBar);
       Scene scene = new Scene(vBox, 960, 600);
       primaryStage.setScene(scene);
       primaryStage.show();
   }
}
```

Here is a full JavaFX ProgressBar code example:

ProgressBar → Create a ProgressBar

In order to use a JavaFX ProgressBar you must first create an instance of the ProgressBar class.

Here is how you create an instance of a JavaFX ProgressBar:

```
ProgressBar progressBar = new ProgressBar();
```

This example creates a ProgressBar in indeterminate mode, meaning its progress level is not known. In indeterminate mode the JavaFX ProgressBar displays an animation.

You can create a ProgressBar instance with a determinate progress level by passing the progress value as parameter to its constructor, like this:

```
ProgressBar progressBar = new ProgressBar(0);
```

ProgressBar → Setting the Progress Level

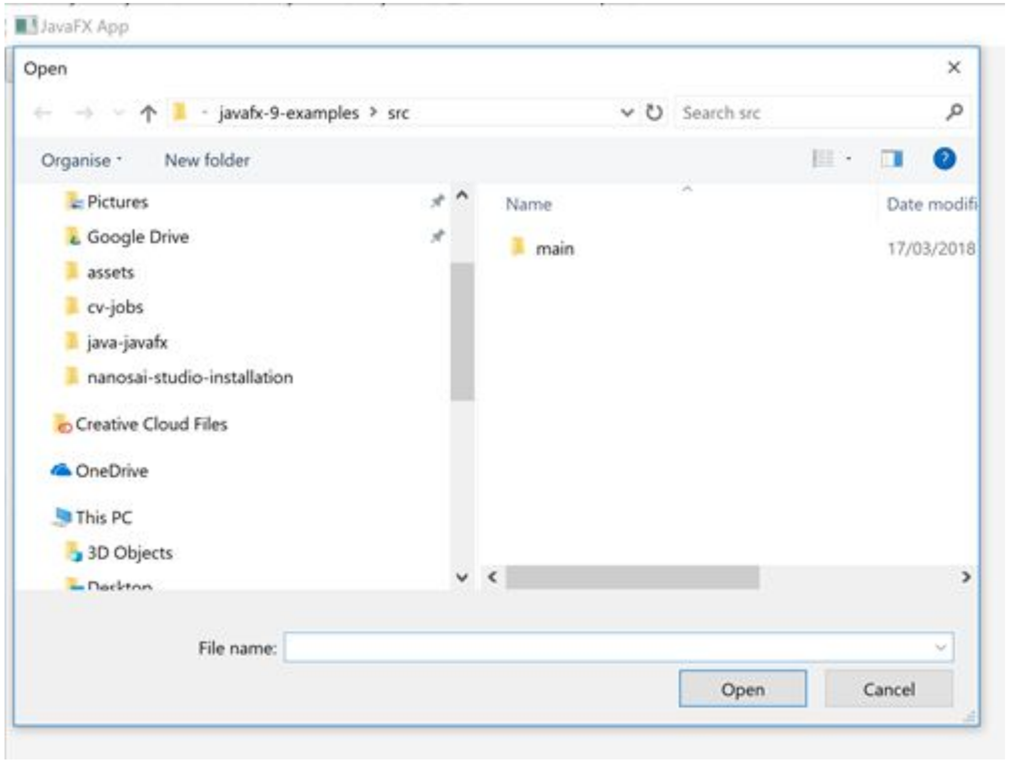
You set the progress level of a ProgressBar via the setProgress() method.

Here is an example of how you set the progress level of a JavaFX ProgressBar:

```
ProgressBar progressBar = new ProgressBar(0);

progressBar.setProgress(0.5);
```

FileChooser → Introduction

- A *JavaFX FileChooser* class is a dialog that enables the user to select one or more files via a file explorer from the user's local computer.
- The JavaFX FileChooser is implemented in the class javafx.stage.FileChooser.
- In this JavaFX FileChooser tutorial I will show you how to use the JavaFX FileChooser dialog.
- Here is an example screenshot of how a JavaFX FileChooser looks:



FileChooser → Creating a FileChooser

In order to use the JavaFX FileChooser dialog you must first create a FileChooser instance.

Here is an example of creating a JavaFX FileChooser dialog:

```
FileChooser fileChooser = new FileChooser();
```

As you can see, it is pretty easy to create a FileChooser instance.

FileChooser → Showing the FileChooser Dialog

Showing the JavaFX FileChooser dialog is done by calling its showOpenDialog() method.

Here is an example of showing a FileChooser dialog:

```
File selectedFile = fileChooser.showOpenDialog(stage);
```

The File returned by the showOpenDialog() method is the file the user selected in the FileChooser.

The stage parameter is the JavaFX Stage that should "own" the FileChooser dialog. By "owning" is meant what Stage from which the FileChooser dialog is shown. This will typically be the Stage in which the button sits that initiates the showing of the FileChooser.

FileChooser → Showing the FileChooser Dialog

Showing a FileChooser is typically done as a result of a click on a button or menu item. Here is a full JavaFX example that shows a button that opens a FileChooser when it is clicked:

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

public class FileChooserExample extends Application {
   public static void main(String[] args) {
       launch(args);
   }
   @Override
   public void start(Stage primaryStage) {
       primaryStage.setTitle("JavaFX App");
       FileChooser fileChooser = new FileChooser();
       Button button = new Button("Select File");
       button.setOnAction(e -> {
           File selectedFile = fileChooser.showOpenDialog(primaryStage);
       });
       VBox vBox = new VBox(button);
       Scene scene = new Scene(vBox, 960, 600);
       primaryStage.setScene(scene);
       primaryStage.show();
   }
}
```

This example creates a full JavaFX application with a Button that when clicked opens a FileChooser . Notice how the primary Stage for the JavaFX application is passed as parameter to the FileChooser showOpenDialog() method.

FileChooser → Setting Initial Directory

You can set the initial directory displayed in the JavaFX FileChooser via its setInitialDirectory() method.

Here is an example of setting the initial directory of a FileChooser dialog:

```
fileChooser.setInitialDirectory(new File("data"));
```

This example sets the initial directory displayed by the FileChooser to data.

FileChooser → Setting Initial File Name

You can set the initial file name to display in the FileChooser . Some platforms (e.g. Windows) may ignore this setting, though.

Here is an example of setting the initial file name of a FileChooser:

```
fileChooser.setInitialFileName("myfile.txt");
```

This example sets the initial file name to myfile.txt .

FileChooser → Adding File Name Filters

It is possible to add file name filters to a JavaFX FileChooser. File name filters are used to filter out what files are shown in the FileChooser when the user browses around the file system.

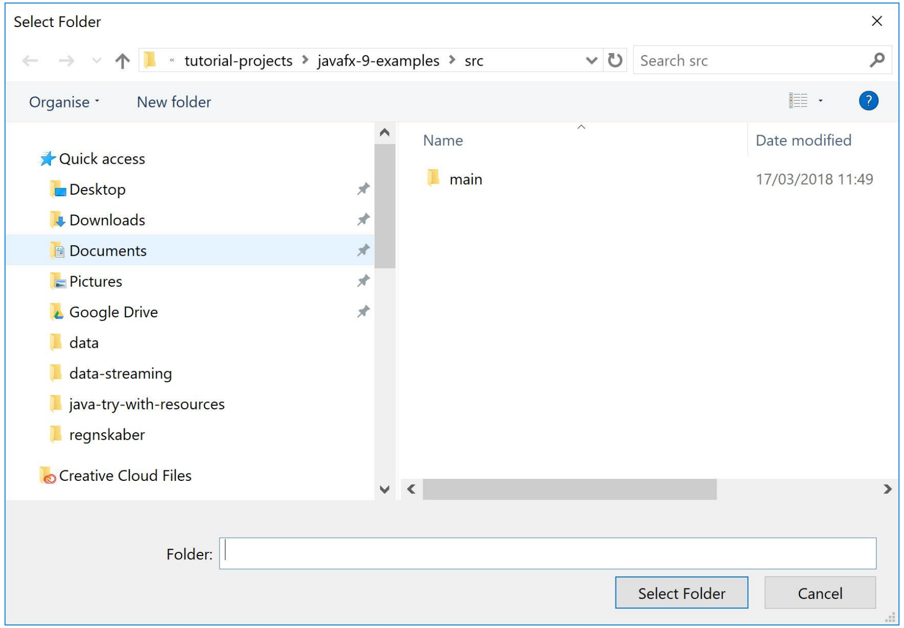
Here is an example of adding file name filters:

```java
FileChooser fileChooser = new FileChooser();

fileChooser.getExtensionFilters().addAll(
   new FileChooser.ExtensionFilter("Text Files", "*.txt")
   ,new FileChooser.ExtensionFilter("HTML Files", "*.htm")
);
```

This examples adds two file name filters to the FileChooser. The user can choose between these file name filters inside the FileChooser dialog.

DirectoryChooser → Introduction

- A JavaFX *DirectoryChooser* is a dialog that enables the user to select a directory via a file explorer from the user's local computer.
- The JavaFX DirectoryChooser is implemented in the class javafx.stage.DirectoryChooser.
- In this JavaFX DirectoryChooser tutorial I will show you how to use the DirectoryChooser dialog.
- Here is an example screenshot of how a JavaFX DirectoryChooser looks:



DirectoryChooser → Creating a DirectoryChooser

In order to use the DirectoryChooser you must first create a DirectoryChooser instance.

Here is an example of creating a JavaFX DirectoryChooser:

```
DirectoryChooser directoryChooser = new DirectoryChooser();
```

DirectoryChooser → Showing the DirectoryChooser Dialog

In order to make the DirectoryChooser visible you must call its showDialog() method.

Here is an example of showing a JavaFX DirectoryChooser:

```
File selectedDirectory = directoryChooser.showDialog(primaryStage);
```

The File returned by the showDialog() method represents the directory the user selected in the DirectoryChooser.

The stage parameter is the JavaFX Stage that should "own" the DirectoryChooser dialog. By "owning" is meant what Stage from which the DirectoryChooser dialog is shown. This will typically be the Stage in which the button sits that initiates the showing of the DirectoryChooser.

Showing a DirectoryChooser is typically done as a result of a click on a button or menu item.

DirectoryChooser → Showing the DirectoryChooser Dialog

Here is a full JavaFX example that shows a button that opens a DirectoryChooser when it is clicked:

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.control.Button;
import javafx.stage.DirectoryChooser;
import java.io.File;

public class DirectoryChooserExample extends Application {
   public static void main(String[] args) {
       launch(args);
   }
   @Override
   public void start(Stage primaryStage) {
       primaryStage.setTitle("JavaFX App");
       DirectoryChooser directoryChooser = new DirectoryChooser();
       directoryChooser.setInitialDirectory(new File("src"));
       Button button = new Button("Select Directory");
       button.setOnAction(e -> {
           File selectedDirectory = directoryChooser.showDialog(primaryStage);
           System.out.println(selectedDirectory.getAbsolutePath());
       });
       VBox vBox = new VBox(button);
       Scene scene = new Scene(vBox, 960, 600);
       primaryStage.setScene(scene);
       primaryStage.show();
   }
}
```

DirectoryChooser → Setting Initial Directory

You can set the initial directory of the JavaFX DirectoryChooser, meaning the root directory the DirectoryChooser will be located at when opened. This is also shown in the example above. You set the initial directory via the method setInitialDirectory().

Here is an example of setting the initial directory of a JavaFX DirectoryChooser:

```
directoryChooser.setInitialDirectory(new File("data/json/invoices"));
```

This example will set the initial directory of the given DirectoryChooser to data/json/invoices .

END