JAVA
GUI JavaFX

Assoc. Prof. Eleni Mangina

eleni.mangina@ucd.ie
Room B2.05
School of Computer Science and Informatics
University College Dublin, Ireland

Topics

- GUI Applications With JavaFX
- GUI Components
- Labels
- Event Handling
- Text Fields
- Command Buttons, Radio Buttons, and Checkboxes
- Combo Boxes
- Layout Managers
- Lambda Expressions
- Animation
- Mouse and Touch Events
- Drawing Charts using ObservableLists
- CSS (Style Sheets)

Introduction

GUIs enable the user to:

- Select the next function to be performed
- Enter data
- Set program preferences, such as colors or fonts
- Display information

GUIs also make the program easier to use.

- A GUI is a familiar interface to users. Users can learn quickly to operate your program, in many cases without consulting documentation or extensive training.

JavaFX (1 of 6)

- One way to create a GUI is to use the Swing components.
- Recently, Oracle introduced JavaFX, a new approach to creating GUIs.
- JavaFX includes new features such as:
  - Separating the GUI from its functionality
  - Binding data to a GUI component
  - Chart creation
  - Animations

JavaFX (2 of 6)

The top-level structure in a JavaFX application is the stage, which corresponds to a window.

A stage can have one or more scenes, which are top-level containers for nodes that make up the window contents.

A node can be a user interface control, such as a button or a drop-down list; a layout; an image or other media; a graphical shape; a web browser; a chart; or a group.

JavaFX (3 of 6)

To create a JavaFX GUI, we add nodes to a scene.

These nodes are arranged in a hierarchy, called a scene graph, in which some nodes are children of other nodes.

The top node is called the root of the scene graph.

JavaFX (4 of 6)

JavaFX applications can be built in several ways:

- Using FXML, a scripting language based on XML (Extensible Markup Language). In this case, we create an FXML file where we specify the layout container and nodes and their properties.
- Programmatically (Java code)

JavaFX (5 of 6)

If the GUI is always the same and will not change when the user interacts with it, FXML is a good choice.

If the GUI is dynamic and its number or types of controls are determined at runtime, it is better to define the GUI programmatically. Sometimes, defining the GUI programmatically is more convenient.

JavaFX (6 of 6)

We start by using FXML and use Java code later.

By defining our GUI using FXML, we separate the GUI from the logic of the code.

Also, FXML allows nonprogrammers to contribute to a team by defining the GUI without knowledge of programming.

FXML (1 of 2)

Here is a Shell FXML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- import classes -->
<?import javafx.scene.layout.HBox ?>
<!-- use appropriate layout manager -->
<HBox xmlns:fx="http://javafx.com/fxml" >
<!-- define GUI components here -->
</HBox>
```

See Example 12.2.

FXML (2 of 2)

In FXML (and XML), a comment starts with <!- - and ends with -->. There are three of them in the previous file.

This Shell FXML file defines an *HBox* component. *HBox* is short for Horizontal Box. It is simply a rectangle in which we can place components and arrange them horizontally.

That *HBox* component will be the root of the scene in this Shell Application example.

FXML Elements

Elements begin with a start tag with the element's name enclosed in angle brackets (< >).

```
<HBox xmlns:fx="http://javafx.com/fxml" >
```

Some elements, such as the *HBox* definition here, have closing tags, which are simply the element's name preceded by a forward slash (/), also enclosed in angle brackets.

```
</HBox>
```

Some tags, not illustrated here, have an empty closing tag and end with />.

FXML Attributes

Elements can have attributes, which further define the element or set properties of the element.

Attribute definitions are inserted between the element's name and its closing tag.

The syntax for a JavaFX attribute is

```
attributeName = "value"
```

or

```
attributeName = 'value'
```

Example:

```
xmlns:fx="http://javafx.com/fxml"
```

Shell Application File

Here is a *FXShellApplication.java* file:

```
// appropriate import statements here
public class FXShellApplication extends Application
{
  public void start( Stage stage )
  {
    // define the scene, add it to the stage
  }
  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

See Example 12.1.

Shell Application File: Inside *start* Method

```
public void start( Stage stage )
{
  try
  {
    // some of the methods called here throw exceptions
  }
  catch( Exception e )
  {
    // debugging code here
  }
}
```

See Example 12.1.

Shell Application File: Inside *try* Block
(1 of 2)

```
public void start( Stage stage )
{
   // Locate the FXML resource
   // Load the FXML resource, instantiate root Node;
   // here we use HBox

   // create a scene associated with the root
   // and set its width and height

   // assign the scene to the stage object
   // optionally, set the title of the stage
   // show the stage
}
```

See Example 12.1.

Shell Application File: Inside *try* Block
(2 of 2)

```
public void start( Stage stage )
{
  URL url
   = getClass( ).getResource( "fxml_shell.xml" );
  HBox root = FXMLLoader.load( url );

  Scene scene = new Scene( root, 300, 275 );

  stage.setScene( scene );
  stage.setTitle( "JavaFX Shell" );
  stage.show( );
}
```

See Example 12.1.

Useful Classes and Methods (1 of 2)

Package	javafx.application
Return value	**Method name and argument list**
`void`	`init( )`
	called by the JavaFX runtime when the application is launched. Overriding this method is optional.
`void`	`launch( String ... args )`
	static method that launches the application. Passes any command-line arguments to the application.
`Application.Parameters`	`getParameters( )`
	returns any command-line arguments as an *Application.Parameters* object.
`void`	`start( Stage primaryStage )`
	the main entry point for JavaFX applications. The application places its scene onto the *primaryStage*. This *abstract* method needs to be implemented.
`void`	`stop( )`
	called when the application signals that it is ready to end. Overriding this method is optional.

Useful Classes and Methods (2 of 2)

Package	java.lang
Return value	**Method name and argument list**
URL	`getResource( String resource )` returns a URL object representing the location of the *resource*, or *null* if the *resource* is not found

Common Error Trap

Be sure to check that your FXML file uses proper syntax, elements, and attributes. If there are errors in the FXML file, the window will not open.

If you omit the call to the *show* method, the window will not open when the application begins.

GUI Components

A component performs at least one of these functions:

- Displays information
- Collects data from the user
- Allows the user to initiate program functions

The Java Class Library provides a number of component classes in the *javafx.scene.control* package.

JavaFX Components (1 of 2)

Package	javafx.fxml
Return value	**Method name and argument list**
`<T> T`	`load( URL location )` *static* method that instantiates the nodes defined in the FXML file specified by *location* and returns a reference to the root node

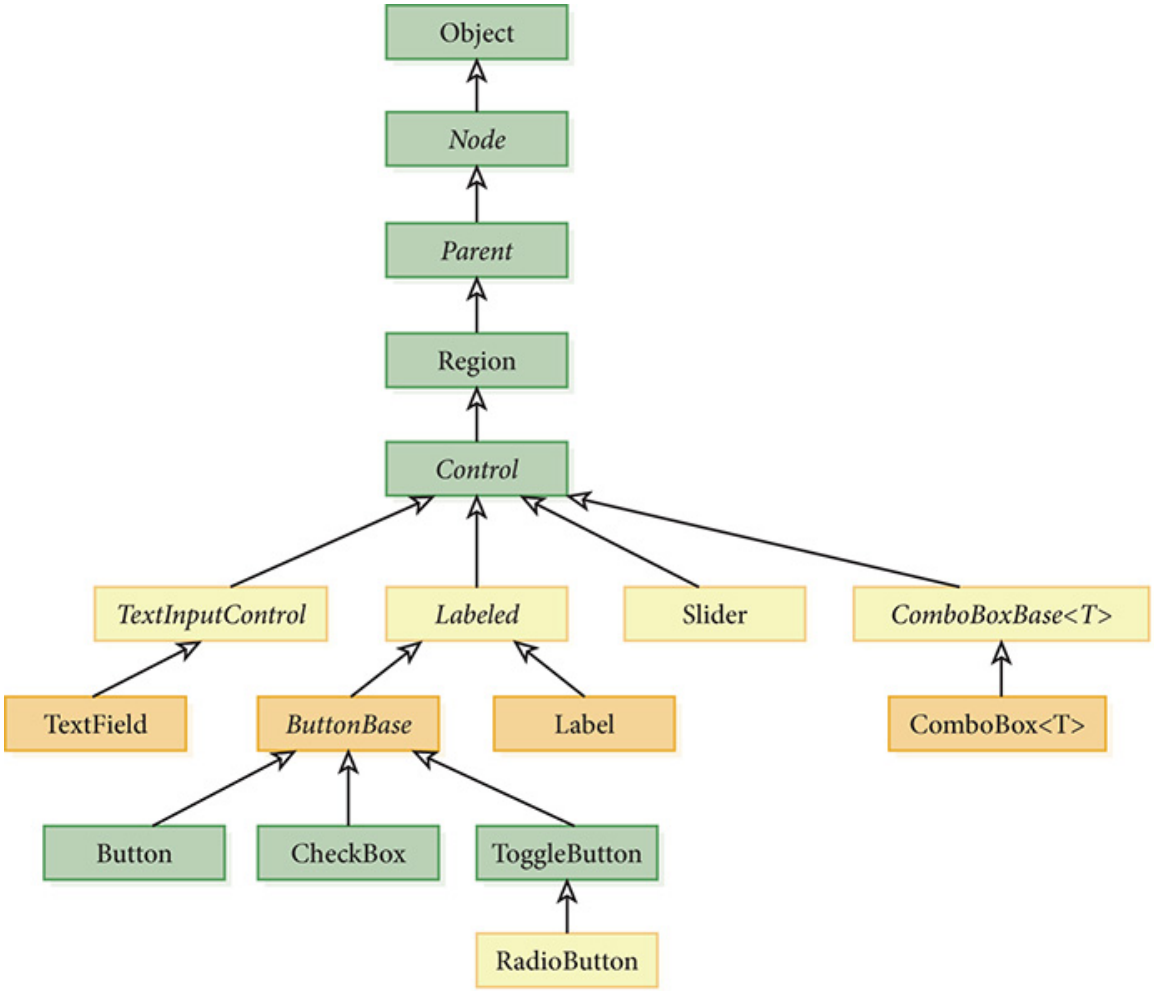
JavaFX Components (2 of 2)

Package	javafx.scene

```
Scene( Parent root, double width, double height )
```

instantiates the *Scene* with a top-level node of *root* and sets the width and height to the pixel values specified

Inheritance Hierarchy for Some GUI Classes



JavaFX Layout Classes

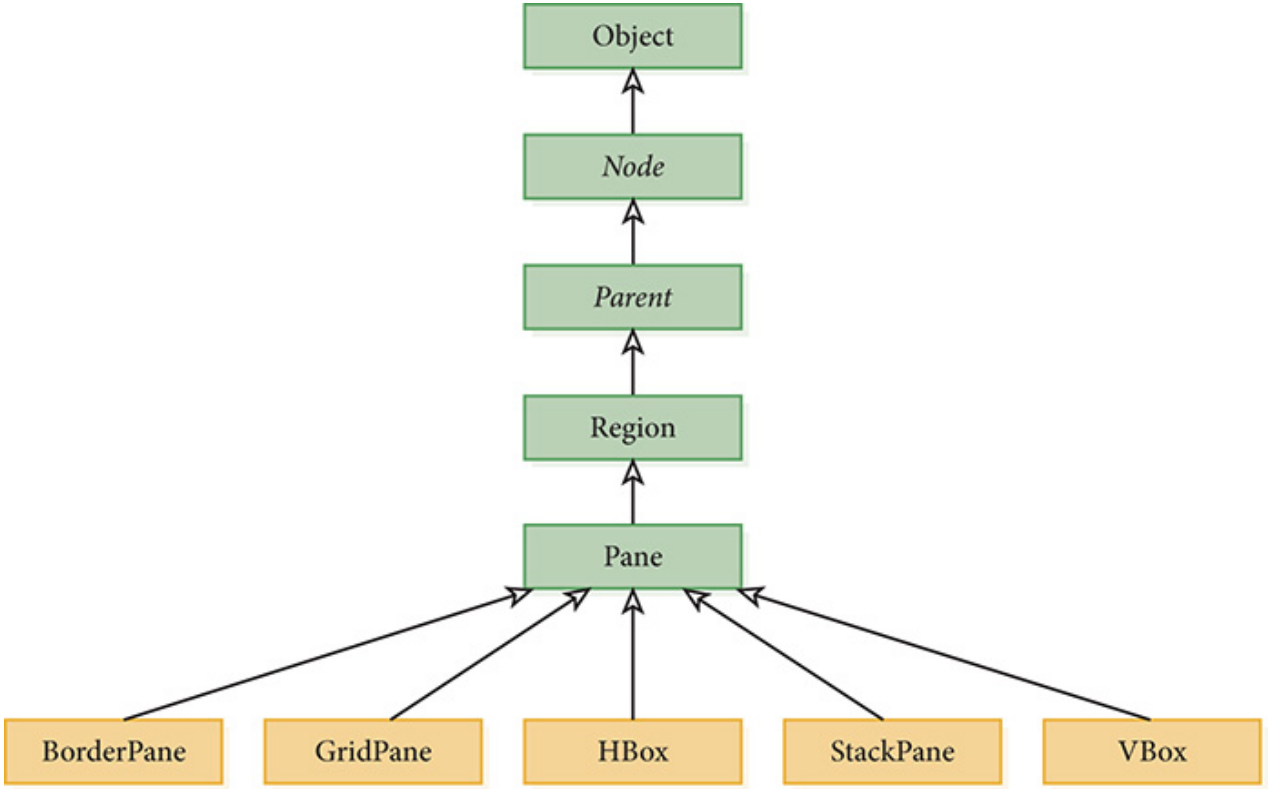
In our applications, we place the controls into the scene by using layout containers, which organize the controls according to each layout's rules.

These layout container classes are in the *javafx.scene.layout* package.

Useful JavaFX Layout Classes

Package	javafx.stage
Return value	**Method name and argument list**
`void`	`setScene( Scene scene )`
	specifies the scene to be hosted by the *Stage*.
`void`	`setTitle( String title )`
	sets the text to appear in the window title bar.
`void`	`show( )`
	makes the window visible. By default, windows are not visible.

The Layout Classes Hierarchy



The *Label* and *ImageView* Components (1 of 2)

- A *Label* component does not interact with a user.
- The *Label* displays some information, for example:
  - A title
  - An identifier for another component
- The *ImageView* displays an image.

The *Label* and *ImageView* Components (2 of 2)

In FXML, a *Label* includes many attributes, such as *text* and *textFill* (the color of the text).

We place an *Image* element inside the *ImageView* element and set its *url* attribute to the name of the file that contains the image. To indicate that the file name is relative to the current folder, we place @ before it.

FXML

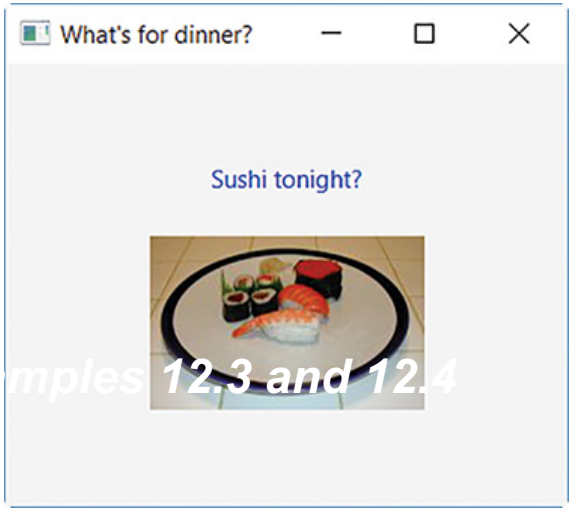
```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.*?>
<?import javafx.scene.image.*?>
<?import javafx.scene.layout.*?>

<VBox id="root" alignment="CENTER" spacing="25" >
<Label text="Sushi tonight?" textFill="BLUE" />
<ImageView>
  <Image url="@sushi.jpg" />
</ImageView>
</VBox>
```

See Example 12.4.

Using *Label* and *ImageView* Components

Because we specify a *VBox* in the FXML file, we assign a *VBox* to the element returned by the call to the *load* method inside the *start* method of the *Application* class.

```
VBox root = FXMLLoader.load( url );
```



Event Handling

- GUI programming uses an event-driven model of programming.
- The program responds to events caused by the user interacting with a GUI component.
  - For example, we might display some text fields, a few buttons, and a selectable list of items. Then our program will "sit back" and wait for the user to do something.
  - When the user enters text into a text field, presses a button, or selects an item from the list, our program will respond, performing the function that the user has requested, and then sit back again and wait for the user to do something else.

Event Handling—MVC

JavaFX supports and encourages the Model-View-Controller (MVC) architecture for writing GUI applications. In this architecture:

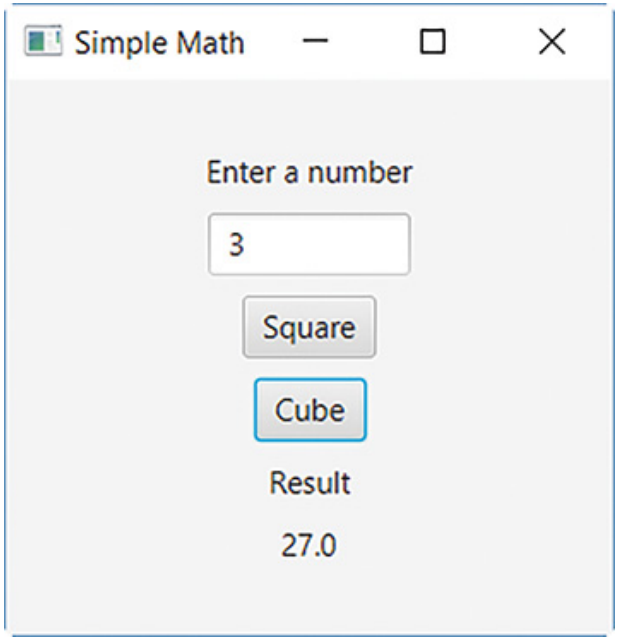
- The Model manages the data of the application and its state.
- The View presents the user interface.
- The Controller handles events generated by the user.

The Model-View-Controller Architecture

- The Controller is the middleman between the View and the Model. When the user interacts with the View, the Controller asks the Model to perform some calculations and then updates the View accordingly.
- These three components can be placed in the same file or in different files.
- Separating the Model, View, and Controller functionality makes the application easier to design, code, maintain, and reuse.

TextField and *Button* Example

See Examples 12.5 to 12.8



The user enters a number in the *TextField* and presses a *Button*. The result is displayed in a *Label*.

Our event handler uses the *getSource* method to determine which button was pressed.

Model

Our application enables the user to calculate the square and cube of a number.

We provide a Model, the *SimpleMath* class, that includes two *static* methods to perform these calculations.

See Example 12.5.

Event Handling
View, Controller, and FXML

Inside the FXML file defining the GUI, we can specify what class the Controller is for the application.

We can also set up event handling and specify what method (of the Controller) will be called for what event on what component.

FXML File

```
…
<VBox fx:controller="SimpleMathController"
      xmlns:fx="http://javafx.com/fxml"
      alignment="center" spacing="10" >
…
</VBox>
```

The value of the *fx:controller* attribute, *SimpleMathController*, is the class name of the controller for the application.

See Example 12.6.

Inside the FXML file

```
<Label text="Enter a number" />
<TextField fx:id="operand" maxWidth="100" />
<Button fx:id="square" text="Square"
        onAction="#calculate" />
<Button fx:id="cube" text="Cube"
        onAction="#calculate" />
<Label text="Result" />
<Label fx:id="result" />
```

The *calculate* method, the value of the *onAction* attribute of both *Buttons*, will be called when the user presses a *Button*.

See Example 12.6.

Event Handling
View, Controller, and FXML

Inside the Controller class, *SimpleMathController*, we define:

- Instance variables that have been given an id in the FXML file
- Methods that have been specified for some event in the FXML file
- Other methods (and instance variables) as needed

SimpleMathController.java

```
…
public class SimpleMathController
{
   @FXML private TextField operand;
   @FXML private Label result;
   @FXML private Button square;
   @FXML private Button cube;
   @FXML protected void calculate( ActionEvent event )
   {
        …
   }
   …
```

See Example 12.7.

Event Handling

- The *getSource* method of the *ActionEvent* class enables us to retrieve the GUI component at the origin of the event.
- If the same method is used to handle a click on several *Buttons* (here the *square* and *cube Buttons*), we can call *getSource* to identify which *Button* was clicked.

Package	javafx.scene.control
JavaFX Class	**Purpose**
Label	Displays an image or read-only text. Labels are often used to identify the contents of *TextFields*.
TextField	A single-line text box for accepting user input.
Button	Command button that the user clicks to signal that an operation should be performed.
RadioButton	Toggle button that the user clicks to select one option in a group.
CheckBox	Toggle button that the user clicks to select or deselect 0, 1, or more options in a group.
ComboBox	List of options from which the user selects one item.
Slider	Displays a set of continuous values along a horizontal or vertical line. The user can select a value by moving the knob, or thumb.

Inside the *calculate* Method

```
// square and cube are the two Buttons
// event is the ActionEvent parameter of calculate
// result is the Label

if ( event.getSource( ) == square )
 result.setText(
        String.valueOf( SimpleMath.square( op ) ) );
else if ( event.getSource( ) == cube )
 result.setText(
        String.valueOf( SimpleMath.cube( op ) ) );
```

See Example 12.7.

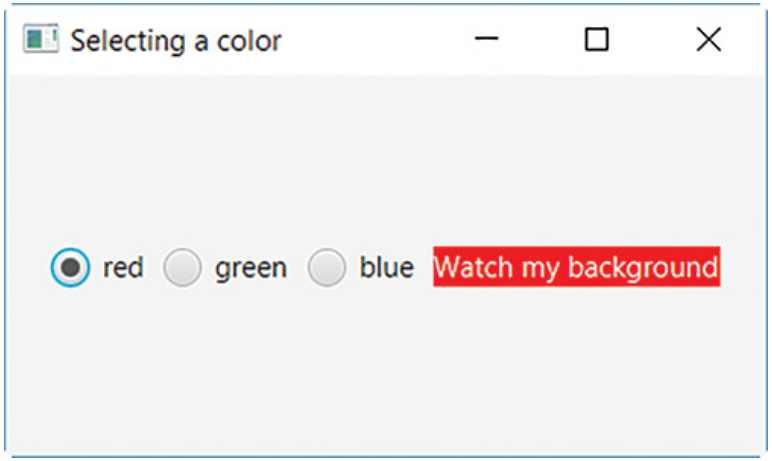
RadioButton and *CheckBox*

- Radio buttons are typically used to allow the user to select one option from a group.
  - Radio buttons are meant to be mutually exclusive, in that clicking on any radio button deselects any previously selected radio button.
- Checkboxes often are associated with the sentence “check all that apply”; that is, the user may select 0, 1, or more options.
  - A checkbox is a toggle button, in that successive clicks alternate between selecting and deselecting the option for that particular checkbox.

Using Radio Buttons

See Examples 12.9–12.12

- The *red, green,* and *blue* *RadioButtons* are added to the same *ToggleGroup*.



- We use the *getSource* method to determine which radio button was selected and set the background of a *Label* to the corresponding color.

MVC Organization

The application includes four files:

- *ColorSelector.java*, the Model
- *fxml_changing_colors.fxml*, the View
- *ChangingColorsController.java*, the Controller
- *ChangingColors.java*, the application launcher, where the *main* method is located

The Model: *ColorSelector.java*

- Our application enables the user to change a color based on user input.
- We provide a Model, the *ColorSelector* class, that includes one *static* method. It accepts an *int* parameter and returns a *String*, a representation of a hex color. That *String* is used in the Controller to set the background color of a *Label* in the View.

See Example 12.9.

The View: *fxml_changing_colors.fxml* File (1 of 3)

- We place all the elements inside an *HBox*.
- We need to specify that the three radio buttons belong to the same group so that they are mutually exclusive.
- The *fx:define* FXML element is used to create objects not in the scene graph that need to be referenced later. We use it to define a *ToggleGroup* to make the radio buttons mutually exclusive.

See Example 12.10.

The View: *fxml_changing_colors.fxml* File (2 of 3)

```
…
<HBox fx:controller="ChangingColorsController"
      … >
<fx:define>
  <ToggleGroup fx:id="colorGroup" />
</fx:define>
…
</HBox>
```

We define a *ToggleGroup* and give it an id, *colorGroup*.

See Example 12.10.

The View: *fxml_changing_colors.fxml* File (3 of 3)

```
<RadioButton fx:id="red" text="red"
  selected="true" toggleGroup="$colorGroup"
  onAction="#colorChosen" />
<RadioButton fx:id="green" text="green"
  toggleGroup="$colorGroup" onAction="#colorChosen" />
…
<Label fx:id="label" text="Watch my background"
  textFill="WHITE" style="-fx-background-color:#FF0000" />
```

The three radio buttons belong to the same *ToggleGroup*. The red radio button is selected. When the user clicks on any radio button, the *colorChosen* method is called.

See Example 12.10.

Inside the Controller: *ChangingColorsController.java*

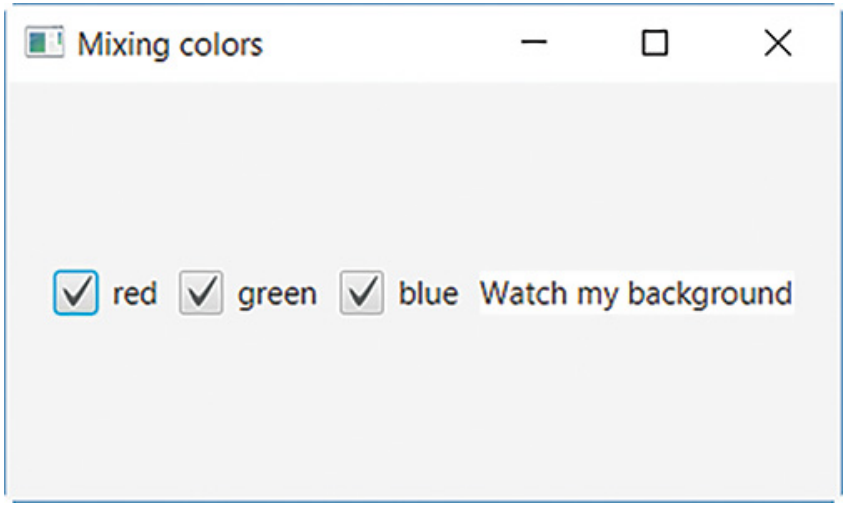
```
@FXML protected void colorChosen( ActionEvent event )
{
  String style = "-fx-background-color: ";
  if ( event.getSource( ) == red )
    style
     += ColorSelector.colorToHexString( ColorSelector.RED );
  else if ( event.getSource( ) == green )
    style
     += ColorSelector.colorToHexString( ColorSelector.GREEN );
 else if ( event.getSource( ) == blue )
   style
     += ColorSelector.colorToHexString( ColorSelector.BLUE );
 label.setStyle( style );
}
```

See Example 12.11.

Using *CheckBoxes*

See Examples 12.13–12.16

- Using the *getSource* method, the Controller toggles the red, green, and blue color intensities in the Model depending on which checkbox was selected or deselected.
- The Model mixes the red, green, and blue intensities to make up a color that becomes the background color of the *Label*.



MVC Organization

This application includes four files:

- *ColorMixer.java*, the Model
- *fxml_mixing_colors.fxml*, the View
- *MixingColorsController.java*, the Controller
- *MixingColors.java*, the application launcher, where the *main* method is located

The Model: *ColorMixer.java*

- Our application enables the user to update a color based on user input.
- We provide a Model, the *ColorMixer* class. It includes a *boolean* array of size 3. Whenever the user selects a checkbox, a method toggles the corresponding value in the array.
- A method converts the array values to a hex *String*, which is used in the Controller to set the background color of the *Label* in the View.

See Example 12.13.

The View: *fxml_mixing_colors.fxml* File

```
…
<CheckBox fx:id="red" selected="true"
  text="red" onAction="#mix" />
<CheckBox fx:id="green" selected="true"
  text="green" onAction="#mix" />
…
```

We place all the elements in an *HBox*. When the user clicks on any of the checkboxes, the *mix* method is called.

See Example 12.14.

Inside the Controller: *MixingColorsController.java*

```
@FXML protected void mix( ActionEvent event )
{
  if ( event.getSource( ) == red )
     mixer.toggleColor( ColorMixer.RED );
  else if ( event.getSource( ) == green )
    mixer.toggleColor( ColorMixer.GREEN );
  …

  String style = "-fx-background-color: ";
  style += mixer.hexStringColor( );
  label.setStyle( style );
}
```

See Example 12.15.

The *ComboBox* Component

The *ComboBox* implements a drop-down list.

- When the combo box appears, one item is displayed, along with a down-arrow button.
  - When the user presses the button, the combo box "drops" open and displays a list of items, with a scroll bar for viewing more items.
  - The user can select only one item from the list.
  - When the user selects an item, the list closes and the selected item is the one item displayed.

Using a *ComboBox*

See Examples 12.19–12.22

- The *ComboBox* items are an array of *Strings.*
- We also define a parallel array of *Images*. Each image shows food from a certain country.
- Initially, we programmatically select the first item using the *setSelectedIndex* method and initialize the food image label to display the first image.

MVC Organization

This application includes four files:

- *FoodSampler.java*, the Model
- *fxml_food_samplings.fxml*, the View
- *FoodSamplingsController.java*, the Controller
- *FoodSamplings.java*, the application launcher, where the *main* method is located

The Model: *FoodSampler.java*

Our application enables the user to select an image based on user input.

We provide a Model, the *FoodSampler* class. It enables us to select an *Image* within an array based on its index. That index value comes from the View; it is the index of the item selected.

See Example 12.19.

The View: *fxml_food_samplings.fxml* File

```
…
<ComboBox fx:id="countries" visibleRowCount="3"
    onAction="#itemSelected" >
</ComboBox>
<ImageView fx:id="foodImage">
</ImageView>
…
```

We place the above elements in an *HBox*. When the user selects an item from the combo box, the *itemSelected* method is called.

See Example 12.20.

The Controller: The *initialize* Method

Inside the Controller class, the *initialize* method is called after the scene graph has been created.

It can be used to add items to the scene graph that could not be fully defined in the FXML file.

By adding items defined in the Model to the *ComboBox* inside the Controller class, we can use the same View and the same Controller with a Model that has a different set of items and images.

It makes our View and our Controller reusable.

ObservableList Interface

The data inside a *ComboBox* is bound to an *ObservableList*. The *ObservableList* interface allows us to keep track of changes in the list when they occur. The data inside the *ComboBox* updates automatically when the list changes. To add elements to an *ObservableList*, we can use the *addAll* method, which accepts *varargs*.

Package	javafx.scene.layout
Layout Container	**Lays out its children nodes . . .**
HBox	In a single horizontal row.
VBox	In a single vertical column.
BorderPane	With at most one child in its top, left, right, bottom, and center positions.
GridPane	In a grid of rows and columns. A child can span more than one row or column.
StackPane	In a front-to-back stack.

ComboBox Methods

To get the *ObservableList* for a *ComboBox*, we use the *getItems* method. To get its selection model, we use the *getSelectionModel* method.

Package	javafx.scene.control
Return value	**Method name and argument list**
`String`	`getText( )`
	returns the text typed into the *TextField*
`void`	`setText( String newText )`
	sets the text in the *TextField* to *newText*

SingleSelectionModel Method

The *getSelectedIndex* method of the *SingleSelectionModel* class enables us to retrieve the index of the item that is selected.

Package	javafx.event
Return value	**Method name and argument list**
`Object`	`getSource( )`
	returns the object on which the event was triggered

Inside the Controller: *FoodSamplingsController.java* (1 of 2)

```
private FoodSampler sampler;

@FXML private ComboBox<String> countries;
private SingleSelectionModel<String> selectionModel;

public void initialize( )
{
    sampler = new FoodSampler( );
    // populate combobox with data from the Model
    countries.getItems( ).addAll( sampler.getCountryList( ) );
    // get a reference to the SingleSelectionModel
    selectionModel = countries.getSelectionModel( );
    …
```

See Example 12.21.

Inside the Controller: *FoodSamplingsController.java* (2 of 2)

```
@FXML protected void itemSelected( ActionEvent event )
{
  // retrieve index of country selected
  int index = selectionModel.getSelectedIndex( );
  // update the Model
  sampler.updateSelection( index );
  // update the View with Image from the Model
  foodImage.setImage( sampler.getImageSelected( ) );
}
```

See Example 12.21.

SOFTWARE ENGINEERING TIP

Arrange items in lists in a logical order so that the user can find the desired item quickly.

For example, list items alphabetically or in numeric order.

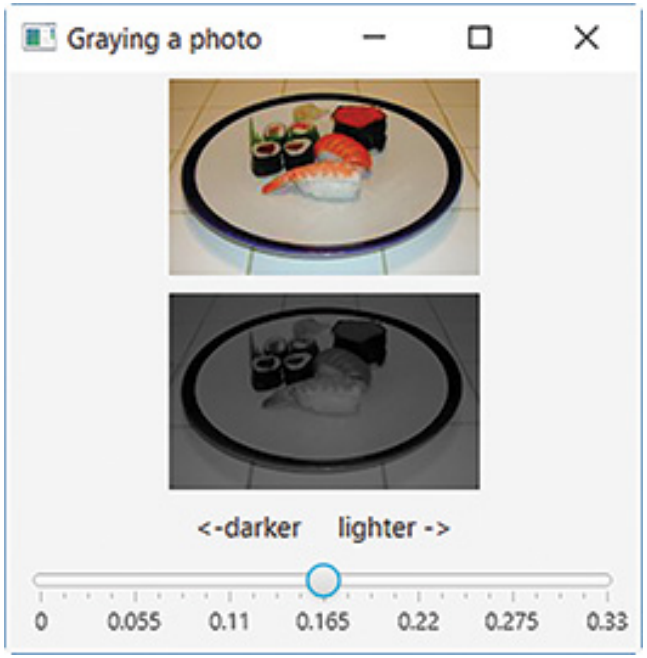
Also consider placing the most commonly chosen items at the top of the list.

Using a *Slider*

See Examples 12.23–12.26

A *Slider* displays a range of continuous values and allows the user to select a value by sliding a knob, called the thumb, along a track.

Depending on that value, we compute a gray intensity and display a grayed image.



MVC Organization

This application includes four files:

- *PhotoGrayer.java*, the Model
- *fxml_photo_grayer.fxml*, the View
- *PhotoGrayerController.java*, the Controller
- *PhotoGraying.java*, the application launcher, where the *main* method is located

The Model: *PhotoGrayer.java*

Our application enables the user to change an image to a gray image based on user input.

We provide a Model, the *PhotoGrayer* class. It enables us to create a gray *Image* based on an existing *Image* and a double value that is used as a coefficient to compute a gray shade for each pixel in the original *Image*.

See Example 12.24.

The View: *fxml_photo_grayer.fxml* File

```
…
<ImageView fx:id="originalImageView" >
  <Image url = "Sushi.jpg" />
</ImageView>
<ImageView fx:id="grayImageView" />
<Label text="&lt;-darker lighter -&gt;" />
<Slider fx:id="slider" min="0" max=".33"
  value=".165" showTickMarks="true"
  showTickLabels="true" majorTickUnit="0.055" />
…
```

We place the above elements in a *VBox*. This View does not set up event handling.

See Example 12.23.

The Controller: *PhotoGrayerController.java*

We set up event handling (and handle the event) inside the Controller.

See Example 12.25.

Methods of the *Slider* Class

FXML Element	Meaning
fx:define	Used to create objects not in the scene graph that need to be referenced later
FXML Attributes	**Meaning**
fx:id	Defines a name that can be referenced across the FXML application
fx:controller	Defines a class that contains event handling code for one or more controls
onAction	Defines a method name (preceded by "#") in the controller that should be executed when the user interacts with the control
FXML Annotation	**Meaning**
@FXML	Allows FXML to access a *private* or *protected* class, method, or data
FXML Prefixes	**Meaning**
-fx-	Used to distinguish a JavaFX style attribute from a CSS attribute
$	Used as a prefix to a variable name when the variable is used as a property value
@	Used as a prefix for a URI to specify that the path of the URI starts with the current folder

Handling Events

When the user interacts with a GUI component, the component fires an event.

Up to this point, we have set up event handling in the FXML file.

We use this *Slider* example to show how we can set up event handling (and handle the event) programmatically.

Handling Events Programmatically

To allow a user to interact with our application through a GUI component, we need to perform the following functions:

1. Write an event handler class (called a listener)
2. Instantiate an object of that listener
3. Register the listener on one or more components

Note that an application can instantiate more than one listener.

The *ObervableValue* Interface

We use the *addListener method* to set up event handling for the *Slider*. Thus, we need to implement the *ChangeListener* interface so we can call *addListener* with a *ChangeListener* reference.

Package	javafx.scene.control
Return value	**Method name and argument list**
`int`	`getSelectedIndex( )`
	returns the index of the currently selected item.
`T`	`getSelectedItem( )`
	returns the currently selected item.
`void`	`select( int index )`
	selects the item at *index*. Any previously selected item is deselected.

ChangeListener Method

The *ChangeListener interface* includes one method, *changed,* that we must override when implementing *ChangeListener*.

Package	javafx.scene.control
Return value	**Method name and argument list**
`ObservableList`	`getItems( )`
	returns the values of the items in the *ComboBox* as an *ObservableList*

Inside the Controller: *PhotoGrayerController.java* (1 of 2)

```
private class SliderHandler implements
              ChangeListener<Number>
{
  public void changed( ObservableValue<? extends
   Number> o, Number oldValue, Number newValue )
  {
    // update grayImageView
    grayImageView.setImage(
      photoGrayer.gray( newValue.doubleValue( ) ) );
  }
}
```

See Example 12.25.

Inside the Controller: *PhotoGrayerController.java* (2 of 2)

```
// slider is the Slider reference
public void initialize( )
{
  …
  // set up event handling for slider
  SliderHandler sh = new SliderHandler( );
  slider.valueProperty( ).addListener( sh );
}
```

See Example 12.25.

Building a GUI Programmatically

- We have been defining GUIs using FXML.
- Sometimes, however, the GUI must be dynamic because we may only know how many components should be included at runtime.
- In those cases, we want to build the GUI programmatically.
- Other times, it is just more convenient to build the GUI programmatically.

Layout Containers

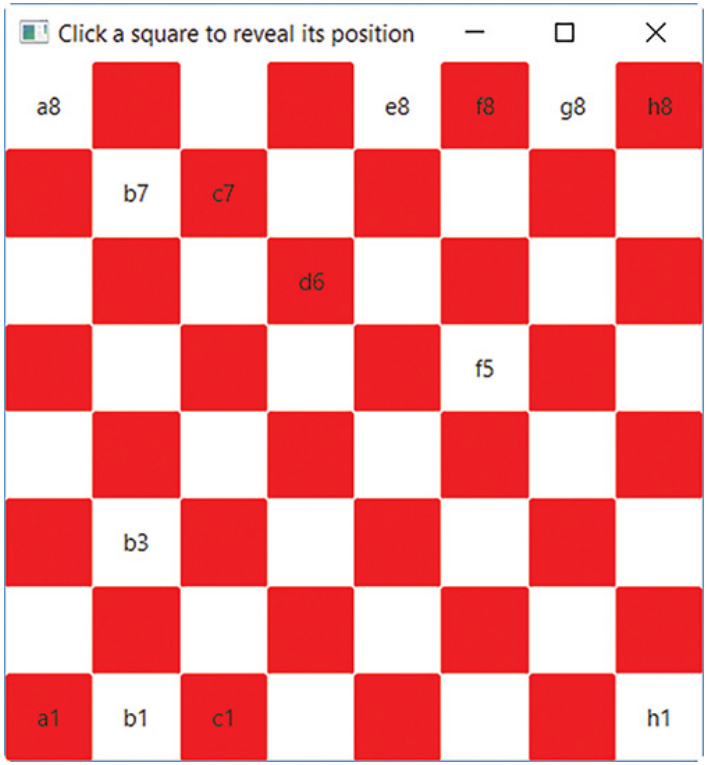
Layout containers determine how the components are organized in the window.

- Some layout containers are:
  - *HBox*, *VBox*
    - Adds components horizontally or vertically
  - *GridPane*
    - Adds components to a table-like grid with equally sized cells
  - *BorderPane*
    - Adds a component to any of five predefined areas

Using *GridPane*

See Examples 12.27–12.29.

We implement a chessboard as a two-dimensional array of *Buttons*. When a button is pressed, we reveal its position on the board.



GridPane

- The *GridPane* organizes the layout as a grid.
  - We can visualize the layout as a table made up of equally sized cells in rows and columns.
  - Each cell can contain one component.
- A component is added to the grid by specifying the row index and column index of the cell in the grid where this component is placed.

MVC Organization

In this application, we put the View and the Controller together.

This application includes three files:

- *BoardGame.java*, the Model
- *BoardView.java*, the View and the Controller
- *ChessBoard.java*, the application launcher, where the *main* method is located

The Model: *BoardGame.java*

The *BoardGame* class provides methods to:

- Set and retrieve the number or rows and columns
- Color the cells
- Retrieve the text and color of a cell

See Example 12.27.

Inside the View/Controller: *BoardView.java*

Our View "is a" *GridPane* and thus inherits all the functionality of a *GridPane.*

```
public class BoardView extends GridPane
```

See Example 12.28.

The View/Controller: *BoardView.java*

View part 1: Create the buttons.

View part 2: Set the rows, columns, row constraints, and column constraints of the *GridPane.*

View part 3: Add the buttons to the *GridPane.*

Controller: Set up event handling and handle the events

See Example 12.28.

Methods Inherited by *Button*

Package	javafx.collections
Return value	**Method name and argument list**
`boolean`	`addAll( E... elements )` adds the elements to the list

Inside the View/Controller: Adding the *Buttons*

The instance variable *squares* is a 2-dimensional array of *Buttons.*

```
// inside the BoardView constructor
// i and j are the row and column indexes
squares[i][j] = new Button( );
// color the button
squares[i][j].setStyle( "-fx-background-color:"
             + game.getSquareColor( i, j ) );
// add the button
add( squares[i][j], j, i );
```

See Example 12.28.

Inside the View/Controller: Handling Events

```
private class ButtonHandler implements
  EventHandler<ActionEvent>
{
  public void handle( ActionEvent event )
  {
    // handle the event here
    // we can call getSource to identify
    // which button triggered the event
  }
}
```

See Example 12.28.

Inside the View/Controller: Setting Up Event Handling

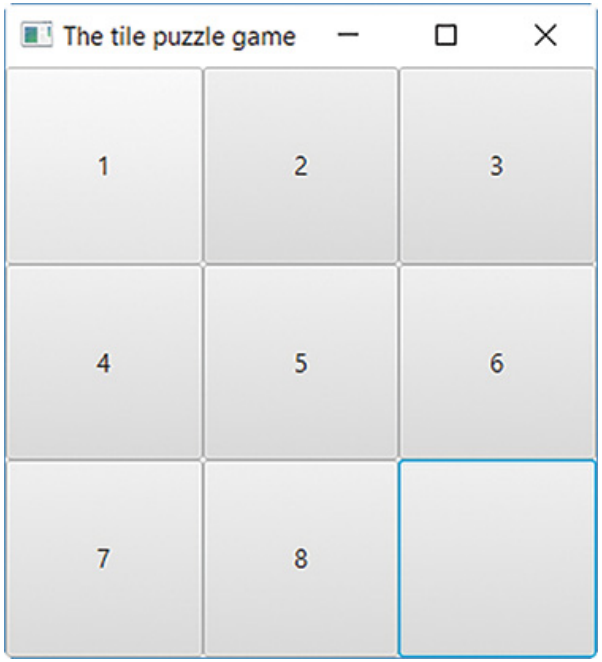
```
ButtonHandler bh = new ButtonHandler( );
…
// i and j are the row and column indexes
// squares is an array of Buttons
// register listener on button
squares[i][j].setOnAction( bh );
```

See Example 12.28.

Dynamic Layouts

- Layout containers can be instantiated dynamically based on runtime parameters or user input.
- Layouts also can be changed at runtime.

We write a Tile Puzzle Game where we randomly generate the grid size before each new game.



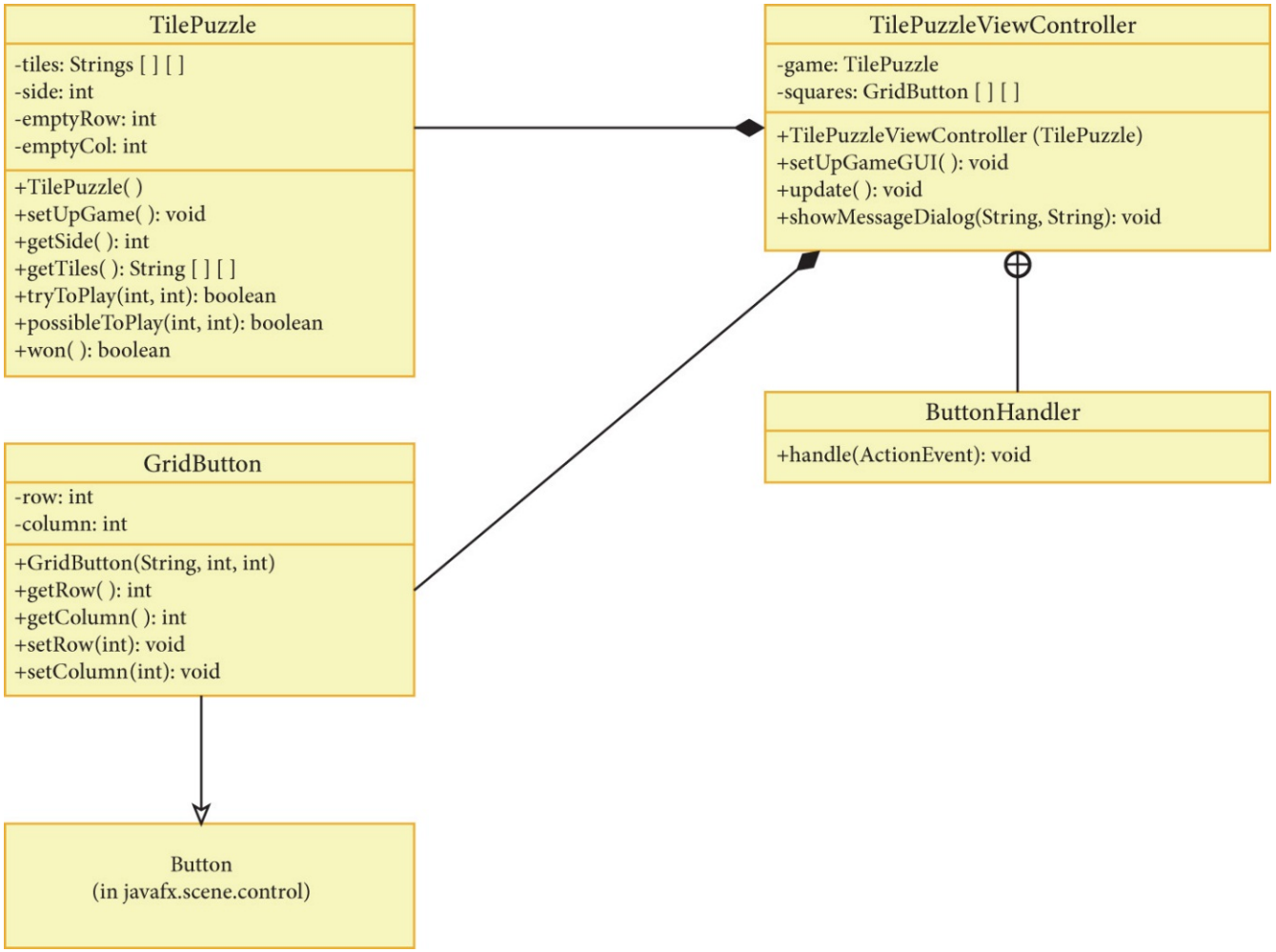
MVC Organization

This application also combines the View and the Controller in one file.

It includes four files:

- *TilePuzzle.java*, the Model
- *GridButton.java*: a *GridButton* "is a" *Button* that is on a grid and knows its row and column indexes on that grid
- *TilePuzzleViewController.java*, the View and the Controller
- *PlayTilePuzzle.java*, the application launcher, where the *main* method is located

The Tile Puzzle Game UML



The Model: *TilePuzzle.java*

The *TilePuzzle* class provides methods to enable play and enforce the rules of the tile puzzle game.

See Example 12.30.

Extending a GUI Component Class: *GridButton.java*

We are interested in knowing where each *Button* is located on the grid.

Rather than using the *getSource* method when a button is clicked, we create *GridButton*, a class that extends *Button*. A *GridButton* knows its row and column index on the grid.

See Example 12.31.

The View/Controller: *TilePuzzleViewController.java*

Like *BoardView*, *TilePuzzleViewController* extends *GridPane*.

We create and add the buttons, set up event handling, and handle the events; this is very similar to the board example.

See Example 12.32.

The Tile Puzzle Game

- The reusable *GridButton* class simplifies the processing of the event:

```
private class ButtonHandler implements
                EventHandler<ActionEvent>
{
  public void handle( ActionEvent event )
  {
    GridButton button = ( GridButton )
                          event.getSource( );
    if ( game.tryToPlay( button.getRow( ),
                           button.getColumn( ) ) )
      update( );
  }
}
```

Inside the View/Controller: *TilePuzzleViewController.java*

If we want to change the layout, we can clear the existing parameters (# of rows and columns) of the layout and clear all the existing GUI components (the "children" of this *GridPane*) in it.

```
getChildren( ).clear( );
getRowConstraints( ).clear( );
getColumnConstraints( ).clear( );
```

See Example 12.32.

The View/Controller: *TilePuzzleViewController.java*

After the user wins, we want to congratulate the user before we set up a new game. We use an *Alert* box for this.

Methods of the *Alert* Class

Character	XML Encoding Sequence
<	<
>	>
&	&
"	"
'	'
Character	**FXML Encoding Sequence**
@	\@
$	\$
%	\%

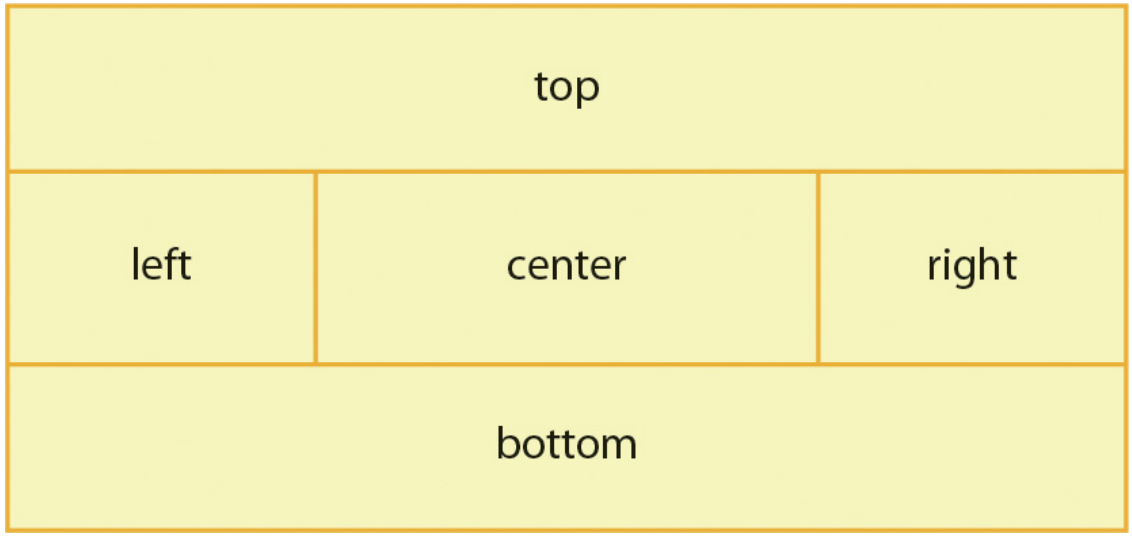
Inside the View/Controller: *TilePuzzleViewController.java*

```
// title and message are Strings
Alert alert
 = new Alert( AlertType.INFORMATION );
alert.setTitle( title );
alert.setHeaderText( "" );
alert.setContentText( message );
alert.showAndWait( );
```

See Example 12.32.

BorderPane

A *BorderPane* organizes a container into five areas:



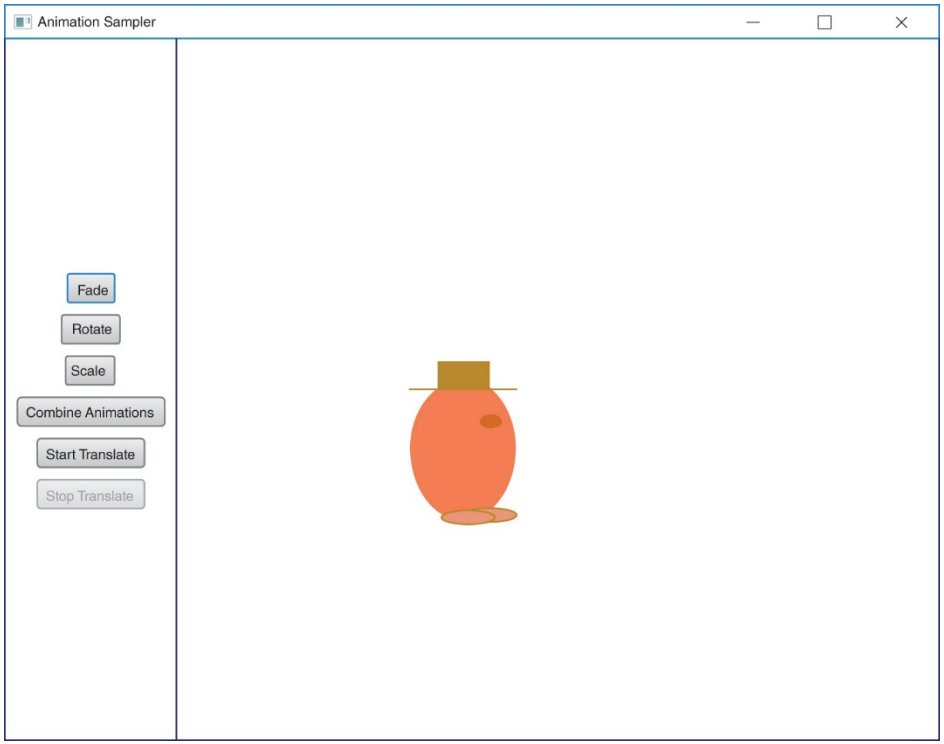
BorderPane Areas

- Each area can hold at most one component.
- The size of each area expands or contracts depending on:
  - The size of the component in that area
  - The sizes of the components in the other areas
  - Whether the other areas contain a component

Using a *BorderPane*

An Animation Application

We place a VBox that contains several Buttons in the left area, and another VBox in the center area; inside that VBox, we place a Canvas (in which the animations take place).



Animation Application (1 of 2)

This application demonstrates three things:

- How to use *BorderPane*
- How to use lambda expressions
- How to create animations

Animation Application (2 of 2)

To keep the application simple and demonstrate the animations better, the Controller file includes some functionality that could have been placed in the Model.

This application includes four files:

- *Sprite.java*, part of the Model
- *fxml_animation.fxml*, the basic View
- *AnimationController.java*, the "Controller"
- *AnimationSampler.java*, the application launcher, where the *main* method is located

Sprite.java

The *Sprite* class encapsulates a sprite, including its position, scale, and the ability to draw itself.

It extends the *Canvas* class so that it can be animated.

```
public class Sprite extends Canvas
{
   // instance variables
   // draw method
   public void draw( GraphicsContext gc )
   …
```

See Example 12.34.

The *fxml_animation.fxml* File

```
…
<BorderPane xmlns:fx="http://javafx.com/fxml"
        fx:controller="AnimationController">
    <center>
      <VBox fx:id="canvasContainer" … />
   </center>
   <left>
     <VBox spacing="10" alignment="center" …>
       …
     </VBox>
   </left>
</BorderPane>
```

We only use the left and center areas of the *BorderPane*. We will place the *Sprite Canvas* inside the *VBox* in the center area.

See Example 12.35.

The *fxml_animation.fxml* File

```
…
<left>
   <VBox spacing="10" alignment="center"
     style="-fx-border-color:navy;-fx-border-width:1;-fx-padding:8">
     <Button text="Fade" fx:id="fadeButton" />
     <Button text="Rotate" fx:id="rotateButton" />
     <Button text="Scale" fx:id="scaleButton" />
     <Button text="Combine Animations" fx:id="combineButton" />
     <Button text="Start Translate" fx:id="startTranslateButton" />
     <Button text="Stop Translate" fx:id="stopTranslateButton" />
   </VBox>
</left>
…
```

The *VBox* in the *left* area contains *Buttons* for the various animations of the sprite.

See Example 12.35.

Transition Class

The *Animation abstract* class is the superclass of many specialized classes that provide animation functionality:

- *FadeTransition*: fades a *Node* in or out
- *RotateTransition*: rotates a *Node* a specified number of degrees
- *ScaleTransition*: varies the size of a *Node*
- *TranslateTransition*: moves a *Node* by varying its x or y coordinate or both
- *SequentialTransition*: plays a series of transitions one after the other
- *ParallelTransition*: plays multiple transitions simultaneously

Methods of the *Animation* Class

Package	javafx.scene.image
Constructor	
`Image( String URL )`	constructs an *Image* from the file named in *URL*
Return value	**Method name and argument list**
`PixelReader`	`getPixelReader( )` — returns a *PixelReader* object for accessing the pixels in the *Image*
`double`	`getHeight( )` — returns the height of the image in pixels
`double`	`getWidth( )` — returns the width of the image in pixels

Specialized *Animation* Classes Have Specialized Methods

Package	javafx.scene.image
Constructor	
`WritableImage( int width, int height )`	constructs an empty WritableImage of the specified width and height
Return value	**Method name and argument list**
`PixelWriter`	`getPixelWriter( )` returns a *PixelWriter* object for writing the pixels in the *Image*

Coding an *Animation*

```
FadeTransition ftFade;

// fade for 3 seconds, then reverse
ftFade = new FadeTransition(
             Duration.seconds( 3 ), canvas );
ftFade.setFromValue( 1.0 );
ftFade.setToValue( 0.1 );
ftFade.setCycleCount( 2 );
ftFade.setAutoReverse( true );

// to run it
ftFade.play( );
```

Event Handling

We can use the *setOnAction* method of the *Node* class to set up event handling. In earlier examples, we implemented the *EventHandler* interface, instantiated an object of that class, and passed it to *setOnAction*.

Instead, we could use an anonymous class, as shown below (in bold is an object reference of an anonymous class implementing *EventHandler*):

```
fadeButton.setOnAction( new EventHandler<ActionEvent>( )
                        {
                           public void handle( ActionEvent event )
                           {
                             ftFade.play( );
                           }
                         }
                    );
```

Lambda Expressions (1 of 2)

A lambda expression contains the following elements:

- A comma-separated list of parameters enclosed in parentheses. The data types of the parameters may be omitted. The parentheses may also be omitted if there is only one parameter.
- The arrow token, ->
- A method body, which can be a single expression or a block enclosed in curly braces.
  - If the body of the method consists of a single expression, then the JVM evaluates the expression and returns its value. As an alternative, you can use a *return* statement, but that requires curly braces.

Lambda Expressions (2 of 2)

Lambda expressions can be used only with functional interfaces, which are interfaces that require the class implementing that interface to define only one method.

The *EventHandler* interface is a functional interface (it only contains one *abstract* method, *handle*).

Thus, we can use a lambda expression with it.

Event Handling Using Lambda Expressions

We can replace this code:

```
fadeButton.setOnAction( new EventHandler<ActionEvent>( )
                        {
                           public void handle( ActionEvent event )
                           {
                             ftFade.play( );
                           }
                        }
                    );
```

With this one, using a lambda expression:

```
fadeButton.setOnAction( event -> ftFade.play( ) );
```

Nesting Components

Components can be nested.

Because the layout containers are subclasses of *Parent*, itself a subclass of *Node*, they are both containers and components. As such, they can contain other layout containers, which in turn can contain components.

We can use this feature to nest components to achieve more precise layouts.

Using Layout Containers to Nest Components

We manage the overall window with a *BorderPane*.

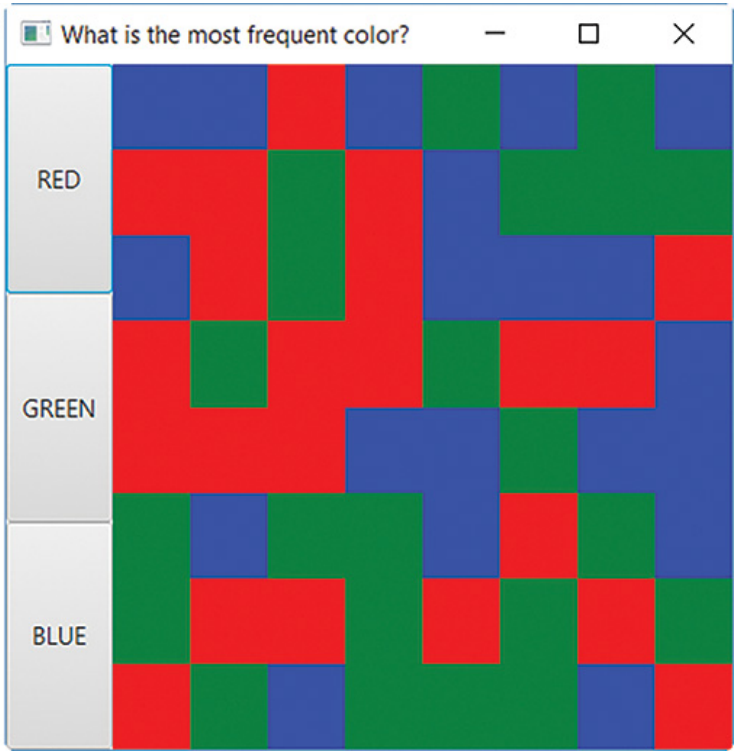
Only two positions are used:

- *left* contains a *VBox* that contains three *Buttons*.
- *center* contains a *GridPane* that contains a two-dimensional array of *Labels*.

Guess the Most Frequent Color Game

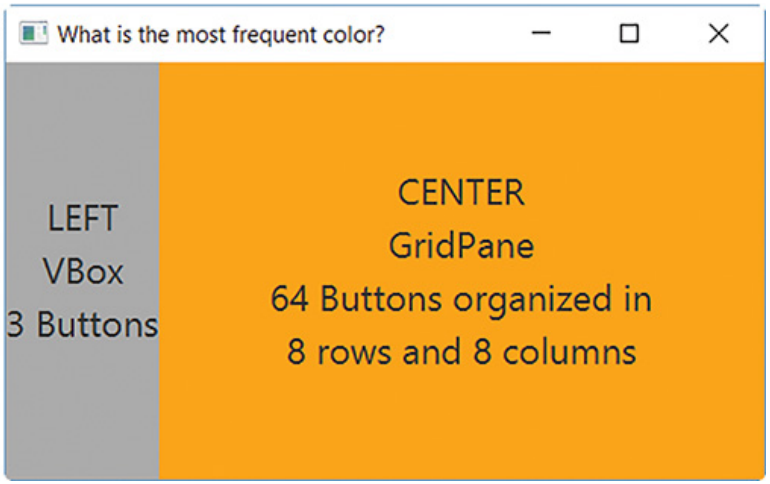
The GUI

(A)



The

(B)



A Reusable Controller (1 of 3)

We have been using this pattern for our Controller:

```
public class Controller
{
  private Model model;
  …
```

It ties the Controller to the Model class. In this example, we show how to make the Controller more reusable.

A Reusable Controller (2 of 3)

Instead of using a class for the Model, the Controller uses an interface.

```
public class Controller
{
  private ModelInterface model;
  …
```

The *ModelInterface* specifies methods that the Controller will call to manage the game. The *Model* class uses this pattern:

```
public class Model implements ModelInterface
{
    // ModelInterface methods here
}
```

A Reusable Controller (3 of 3)

```
public class Controller
{
  private ModelInterface model;
  …
```

Now the Controller is reusable with any Model class that implements the *ModelInterface* interface. We have improved the reusability of our Controller.

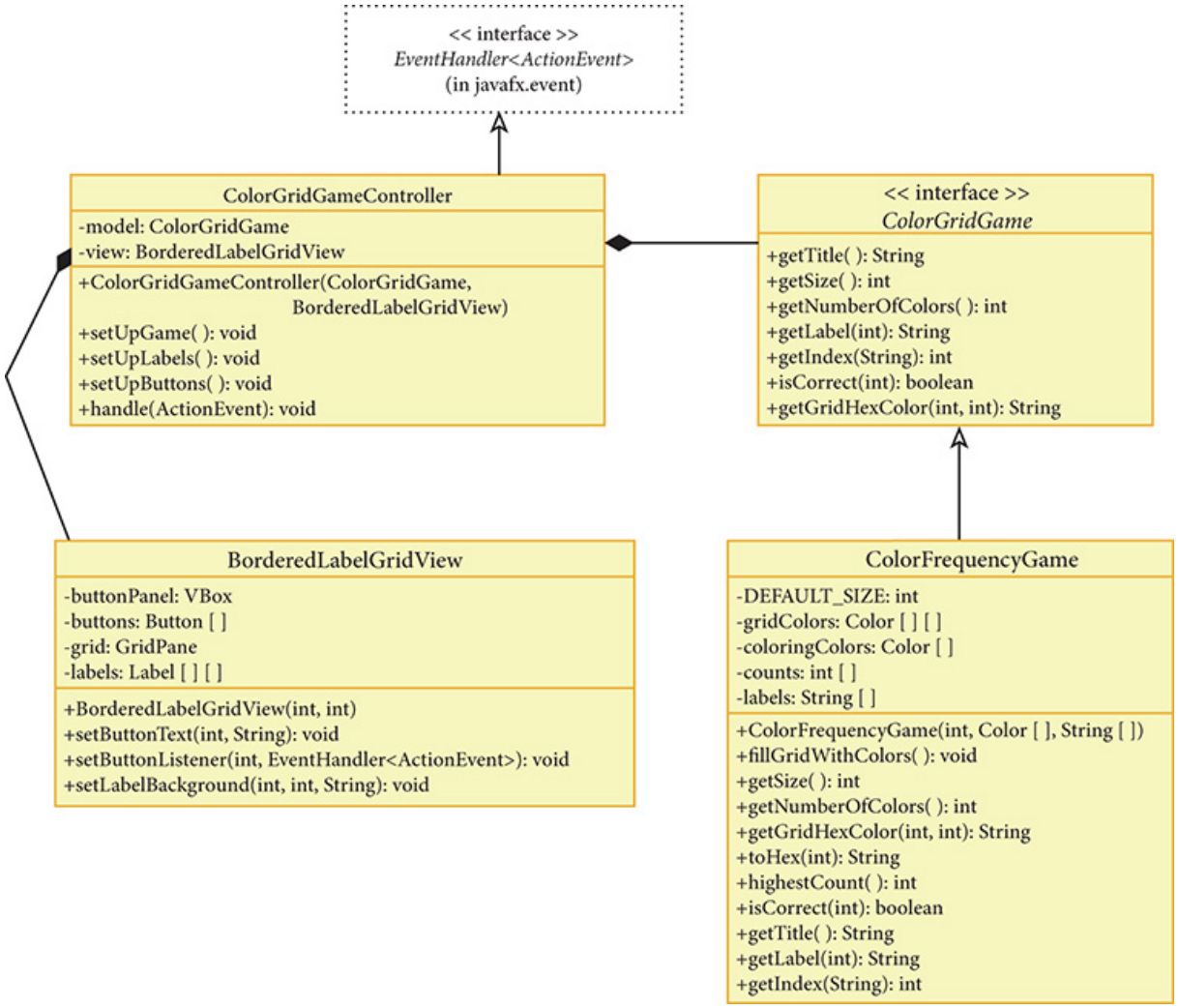
MVC Organization

In this application, we separate the View and the Controller together.

This application includes five files:

- *ColorGridGame.java*, the interface part of the Model
- *ColorFrequencyGame.java*, the class part of the Model
- *BorderedLabelGridView.java*, the View
- *ColorGridGameController.java*, the Controller
- *PlayColorCount.java*, the application launcher, where the *main* method is located

Color Frequency UML



The View: *BorderedLabelGridView.java*

BorderedLabelGridView extends *BorderPane*.

It includes a *VBox* and a *GridPane* instance variables.

The *VBox* contains three *Buttons*. To add the *Button button* to a *VBox* and place the *VBox* in the left area of this *BorderPane*:

```
buttonPanel = new VBox( );
buttonPanel.getChildren( ).add( button );
setLeft( buttonPanel );
```

See Example 12.40.

Mouse and Touch Events

- Any mouse activity (clicking, moving, or dragging) by the user generates a *MouseEvent*.
- Any touch activity by the user generates a *TouchEvent*.
- When any mouse or touch activity occurs, we will be interested in determining where it happened on the window.

Methods of the *MouseEvent* Class

To determine the (x, y) coordinate of the mouse event, we call these *MouseEvent* methods:

Package	javafx.scene.image
Return value	**Method name and argument list**
`Color`	`getColor( int x, int y )` returns the *Color* of the pixel at coordinate (*x*, *y*) in the image

Method of the *TouchEvent* Class

To determine the point of a touch event, we call this *TouchEvent* method:

Package	javafx.scene.image
Return value	**Method name and argument list**
`void`	`setColor( int x, int y, Color c )` sets the pixel at coordinate (*x*, *y*) to the color *c*

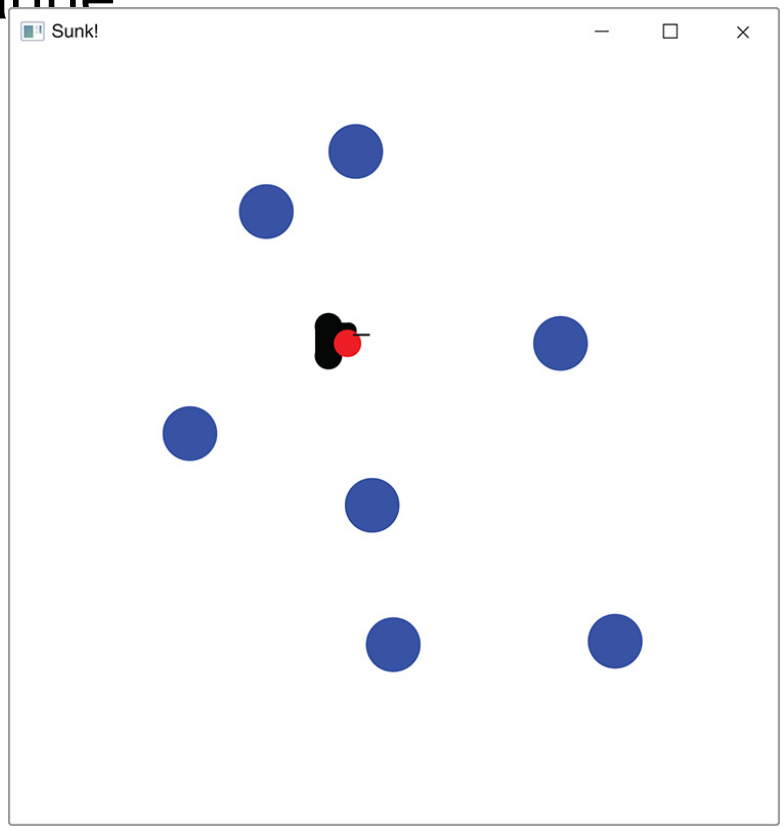
Methods of the *TouchPoint* Class

To determine the (x, y) coordinate of a touch point, we call these *TouchPoint* methods:

Package	javafx.scene.paint
Return value	**Method name and argument list**
`double`	`getRed( )`
	returns the red component of the color in the range 0.0–1.0
`double`	`getGreen( )`
	returns the green component of the color in the range 0.0–1.0
`double`	`getBlue( )`
	returns the blue component of the color in the range 0.0–1.0
`Color`	`gray( double value )`
	returns a *Color* object where the red, green, and blue components are set to *value*, which can range from 0.0 to 1.0

The Sub Hunt Game

The user either touches or clicks inside the window to try to sink a submarine. We provide feedback to help the user locate the submarine



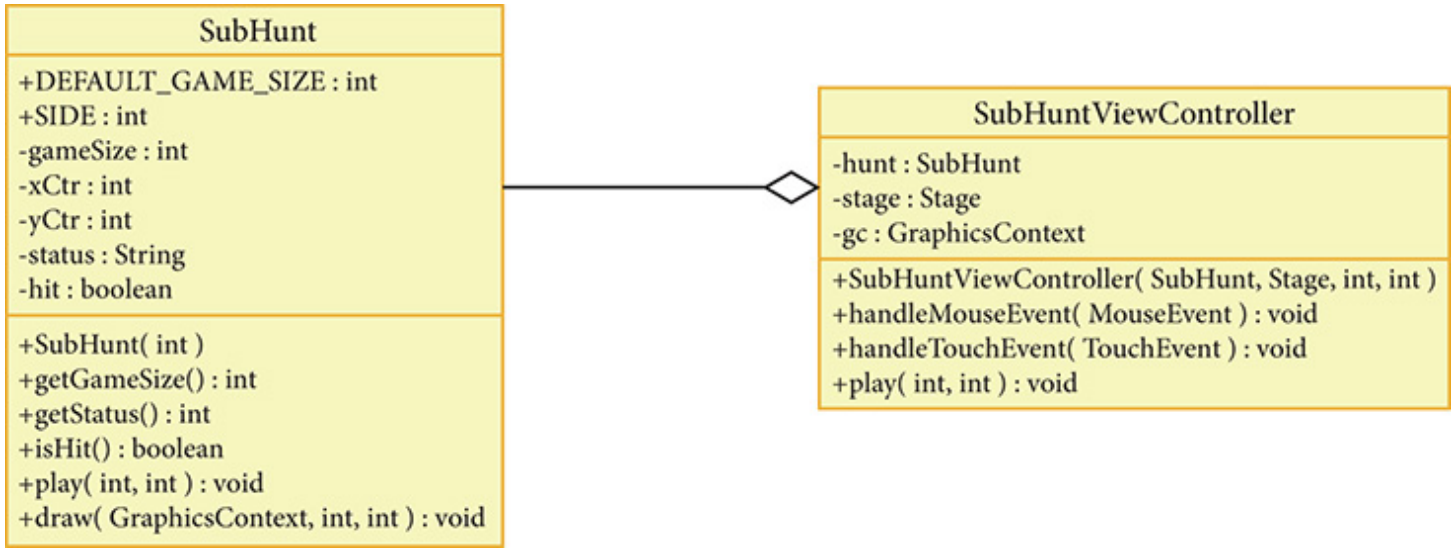
MVC Organization

This application also combines the View and the Controller in one file.

It includes three files:

- *SubHunt.java*, the Model
- *SubHuntViewController.java*, the View and the Controller
- *PlaySubHunt.java*, the application launcher, where the *main* method is located

The Sub Hunt Game UML



The Model: *SubHunt.java*

The *SubHunt* class provides methods to enable play and draw the state of the game.

```
public void play( int x, int y )

public void draw( GraphicsContext gc,
                  int x, int y )
```

See Example 12.43.

The View/Controller: *SubHuntViewController.java*

SubHuntViewController extends *VBox*. Since all we do is draw inside the window, we do not really care what type of layout container we have.

It handles both mouse and touch events in the same manner.

See Example 12.44.

Node Methods to Set Up Event Handling

These methods are inherited from *Node* by *VBox*:

Package	javafx.scene.control
Return value	**Method name and argument list**
`double`	`getValue( )`
	returns the current value of the *Slider* as a *double*
`DoubleProperty`	`valueProperty( )`
	returns the current value of the *Slider* as a *DoubleProperty*, which is a *Property* wrapper class for a *double*, and implements the *ObservableValue<Number>* interface

Inside the View/Controller: *SubHuntViewController.java* (1 of 3)

We use lambda expressions to set up event handling to handle both mouse and touch events.

```
this.setOnMouseClicked( event -> handleMouseEvent( event ) );
this.setOnTouchPressed( event -> handleTouchEvent( event ) );
```

The *handleMouseEvent* and *handleTouchEvent* methods handle the event.

See Example 12.44.

Inside the View/Controller: *SubHuntViewController.java* (2 of 3)

```
public void handleMouseEvent( MouseEvent event )
{
    int x = ( int ) event.getSceneX( );
    int y = ( int ) event.getSceneY( );
    play( x, y );
}
```

The above *play* method calls the *play* and *draw* methods of the Model.

See Example 12.44.

Inside the View/Controller: *SubHuntViewController.java* (3 of 3)

```
public void handleTouchEvent( MouseEvent event )
{
    int x = ( int ) event.getTouchPoint( )
                         .getSceneX( );
    int y = ( int ) event.getTouchPoint( )
                         .getSceneY( );
    play( x, y );
}
```

The above *play* method calls the *play* and *draw* methods of the Model.

See Example 12.44.

Removing a Listener

In the mouse and touch handler, if the mouse click or touch has hit the submarine, we "unregister" the mouse and touch listeners. To do so, we simply set the listeners to *null*.

```
this.setOnMouseClicked( null );
this.setOnTouchPressed( null );
```

After doing so, further mouse clicks and touches will no longer cause the handler to be called.

Inside the View/Controller: *SubHuntViewController.java*

```
public void play( int x, int y )
{
  sub.play( x, y );
  sub.draw( gc, x, y );
  stage.setTitle( sub.getStatus( ) );
  if ( sub.isHit( ) )
  {
    this.setOnMouseClicked( null );
    this.setOnTouchPressed( null );
  }
}
```

If the sub has been hit, we disable the listeners.

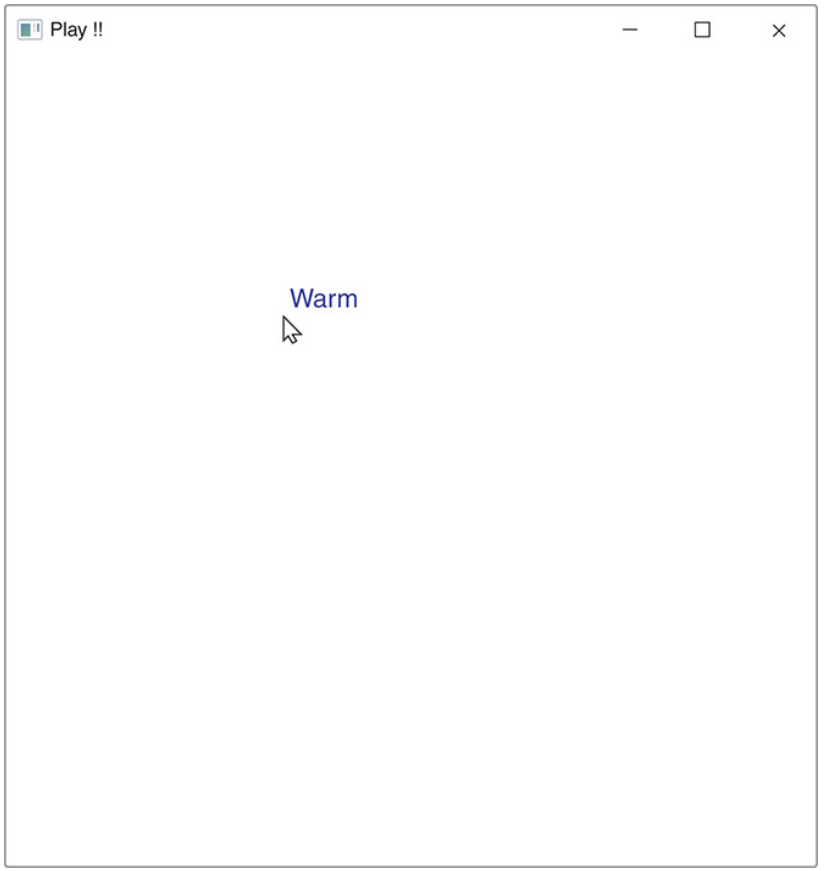
See Example 12.44.

A Treasure Hunt Game (1 of 2)

- To illustrate mouse and touch movements, we implement a treasure hunt game:
  - We hide a treasure in the window.
  - The user attempts to find the treasure by moving the mouse around the window or by moving his/her finger on the screen.
  - We indicate how close the user is to the treasure by displaying a message at the user's location.
  - When the user moves the mouse or his/her finger over the treasure, we draw the treasure and remove the listener to end the game.

A Treasure Hunt Game (2 of 2)

The user moves the mouse or his/her finger on the screen to try to locate a treasure. We provide feedback to help the user.



See Examples 12.46–12.48.

MVC Organization

This application also combines the View and the Controller in one file.

It includes three files:

- *TreasureHunt.java*, the Model
- *TreasureHuntViewController.java*, the View and the Controller
- *PlayTreasureHunt.java*, the application launcher, where the *main* method is located

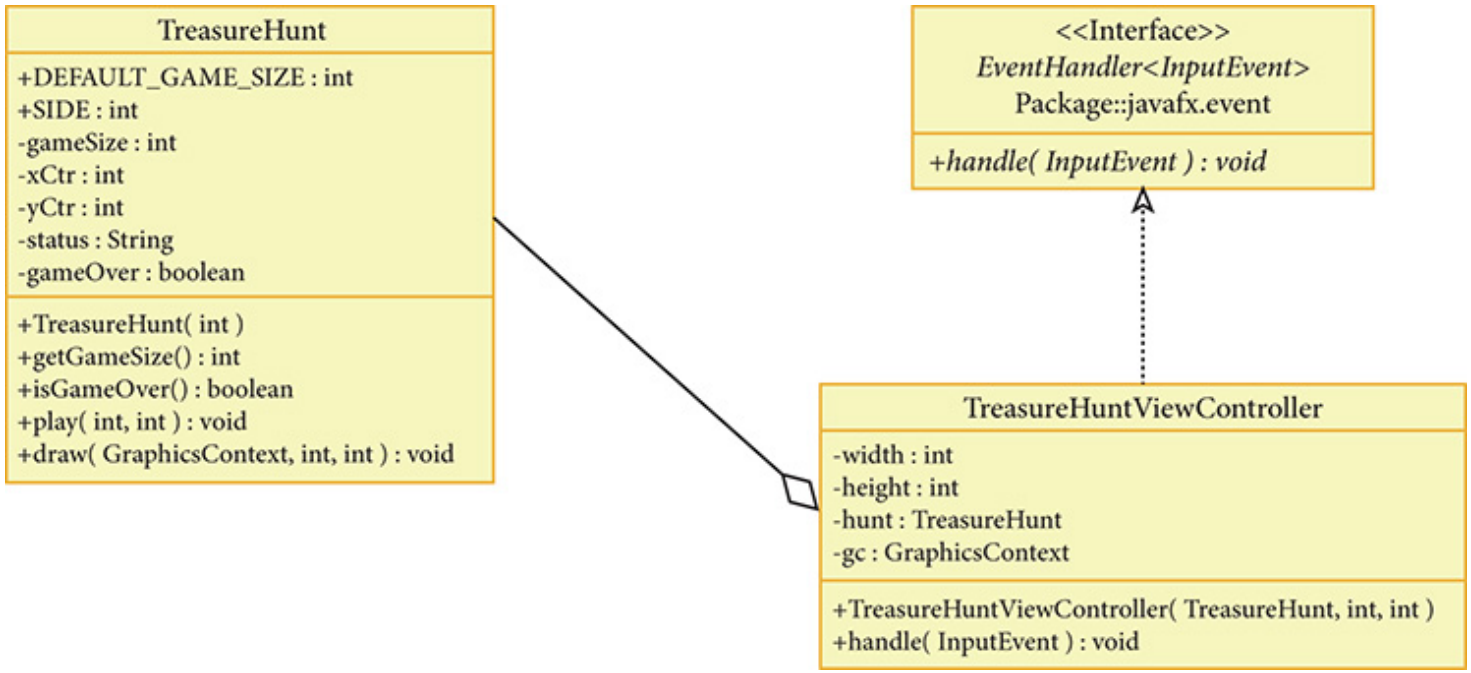
The Model: *TreasureHunt.java*

Like the *SubHunt* class, the *TreasureHunt* class provides methods to enable play and draw the state of the game.

```
public void play( int x, int y )

public void draw( GraphicsContext gc,
                  int x, int y )
```

See Example 12.46.

The Treasure Hunt UML



Node Methods to Set Up Event Handling

These methods are inherited from *Node* by *VBox*:

Package	javafx.beans.value
Return value	**Method name and argument list**
`void`	`changed( ObservableValue<? extends T> observable, T oldValue, T newValue )` The *ChangeListener* must implement this method to handle changes to the *ObservableValue*

The View/Controller: *TreasureHuntViewController.java* (1 of 4)

Like in the sub hunt game, *TreasureHuntViewController* extends *VBox*.

This time, it implements the *EventHandler* interface. As such, it is a listener and can listen to itself.

```
this.setOnMouseMoved( this );
this.setOnTouchMoved( this );
```

See Example 12.47.

The View/Controller: *TreasureHuntViewController.java* (2 of 4)

Because *TreasureHuntViewController* implements the *EventHandler* interface, we must implement the *handle* method to handle the mouse and touch events.

```
public void handle( InputEvent event )
```

See Example 12.47.

The View/Controller: *TreasureHuntViewController.java* (3 of 4)

Because the *handle* method is called for both mouse and touch move events, we need to identify what event took place (so we can retrieve the x and y coordinate of the event correctly).

The *MouseEvent* and *TouchEvent* classes provide constants for the purpose of comparing an event (in this case, the current event) to a particular event.

See Example 12.47.

The View/Controller: *TreasureHuntViewController.java* (4 of 4)

```
if ( event.getEventType( ) ==
                MouseEvent.MOUSE_MOVED )
{
  // event is a MouseEvent, handle it here
}
else if ( event.getEventType( ) ==
                TouchEvent.TOUCH_MOVED )
{
  // event is a TouchEvent, handle it here
}
```

See Example 12.47.

Charts and *ObservableLists* (1 of 2)

JavaFX includes classes that make it easy to display various types of charts, including line charts, pie charts, and bar charts.

To display a chart, we first create a list that stores the data we want to display. We then assign that data to the appropriate chart object.

After we include the chart object in a scene, the chart draws itself automatically.

Charts and *ObservableLists* (2 of 2)

The *ObservableList* interface, from the *javafx.collections* package, is a list that can be linked to a GUI component, such as a *ComboBox*.

Whenever the values in the list change, the GUI component is automatically updated to reflect those changes.

Several chart classes, such as a *PieChart* and *BarChart*, can be constructed by linking their data to an *ObservableList*.

PieCharts and *ObservableLists*

- We use the *PieChart* to display a pie chart.
  - *PieChart.Data* is an inner class of *PieChart* and represents one slice of the pie chart.
- The *PieChart* constructor accepts an *ObservableList* of *PieChart.Data* as its only parameter.
- A slice is defined by its label and its value. The *PieChart.Data* constructor accepts a *String* parameter for the label and a *double* parameter for the value.

PieChart

Package	java.lang
Return value	**Method name and argument list**
`double`	`doubleValue( )` *abstract* method in the *Number* class overridden by the *Double* class; it returns the value of a *Double* object as a *double*.

ObservableList

We can create an *ObservableList* from a *List*. *ArrayList* is a subclass of *List*, so we can use an *ArrayList* to create an *ObservableList*.

Package	javafx.beans.value
Return value	**Method name and argument list**
`void`	`addListener( ChangeListener<? super T> listener )`
	adds a *ChangeListener* whose code will execute whenever the value of the *ObservableValue* changes
`T`	`getValue( )`
	returns the current value of the *ObservableValue* object
`void`	`removeListener( ChangeListener<? super T> listener )`
	removes the listener so that it is no longer notified of changes to the value of the *ObservableValue*

Adding the PieChart

```
// list is an ArrayList of PieChart.Data items
ObservableList<PieChart.Data> pieChartData
    = FXCollections.observableList( list );

PieChart chart = new PieChart( pieChartData );
chart.setTitle( "Monthly Budget" );

// We can use chart, a Node, to define the scene
Scene scene = new Scene( chart, 500, 400 );
```

See Example 12.49.

BarCharts and *ObservableLists*

A *BarChart* displays a bar chart.

Like *PieChart*, it is also a subclass of *Node*.

Thus, we can place a *BarChart* in our *Node* graph.

It can also be bound to an *ObservableList*.

See Examples 12.50–12.55.

CSS (1 of 5)

JavaFX supports Cascading Style Sheets (CSS), a language that describes how a document should be styled.

CSS is often used with web pages.

The JavaFX version of CSS is similar to the CSS used with web pages.

A CSS style sheet is defined in a separate file and imported into the application.

CSS (2 of 5)

The general syntax to style a GUI component is:

```
selector
{
   attribute1: value1;
   attribute2: value2;
   …
}
```

CSS (3 of 5)

The *selector* can be a GUI component, such as a *Label*, but it can also be something identifying a group of components or a specific component.

The attributes are prefixed with *-fx-*

```
VBox
{
    -fx-background-color: skyblue;
}
```

CSS (4 of 5)

We can style several attributes for a selector.

To style a single GUI element identified by its id, we precede the id with #

```
Label
{
    -fx-background-color: deepskyblue;
    -fx-text-fill: blue;
}
#result
{
    -fx-background-color: aqua;
    -fx-font-weight: bold;
}
```

CSS (5 of 5)

We need to import the style sheet into the application.

Assume *simple_math.css* is the name of the file for the style sheet.

```
scene.getStylesheets( )
     .add( "simple_math.css" );
```