GCIS-124
Software Development & Problem Solving

4: Graphical User Interfaces

RIT | Golisano College of Computing and Information Sciences

SUN	MON (2/5)	TUE	WED (2/7)	THU	FRI (2/9)	SAT
	Unit 3: Inheritance		Unit 4: Graphical User Interfaces			
	Assignment 3.1 Due (start of class)	You Are Here	Unit 3 Mini-Practicum Assignment 3.2 Due (start of class)			
SUN	MON (2/12)	TUE	WED (2/14)	THU	FRI (2/16)	SAT
	Unit 4: GUIs		Midterm		Unit 5: Data Structures I	
	Assignment 4.1 Due (start of class)		Midterm Exam 1 (Units 1-3) Written Practical		Unit 4 Mini-Practicum Assignment 4.2 Due (start of class)	

4.0 Accept the Assignment

You will create a new repository at the beginning of every new unit in this course. Accept the GitHub Classroom assignment for this unit and clone the new repository to your computer.

- Your instructor will provide you with a new GitHub classroom invitation for this unit.
- Upon accepting the invitation, you will be provided with the URL to your new repository, click it to verify that your repository has been created.
- You should create a `SoftDevII` directory if you have not done so already.
    - Navigate to your `SoftDevII` directory and clone the new repository there.
- Open your repository in VSCode and make sure that you have a terminal open with the **PROBLEMS** tab visible.

Asking for Help

Could be Improved

I don't understand part 4.c. on the homework.

You have all the resources that you need to solve the problems that have been given to you, but sometimes you may not be able to find the answers to the problems that you encounter.

Asking for help in this course is both ***expected*** and ***encouraged***.

Spending time trying to solve the problem yourself will be a more valuable learning experience in the long run.

If you ***do*** ask for help, try to be as detailed as possible to make it easier for others to help you by providing as much detail as you can.

MUCH Better

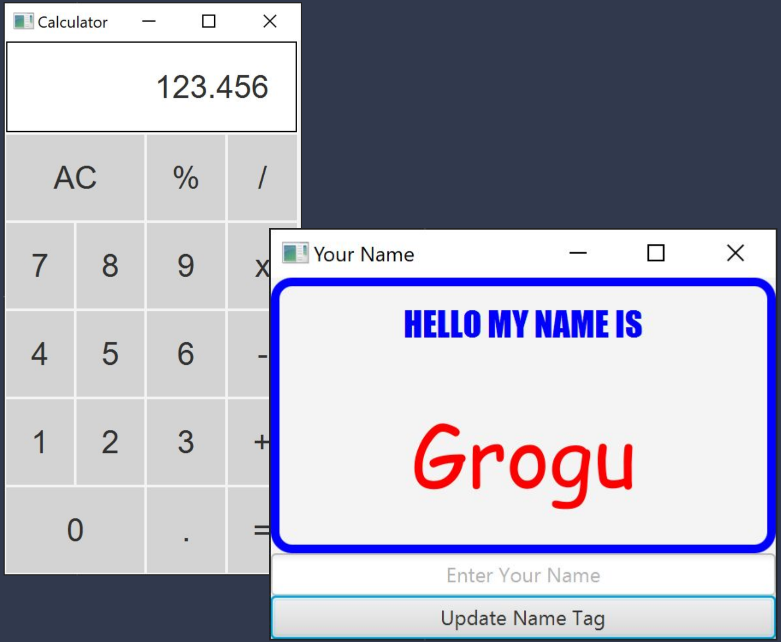
Part 4.c. on homework 2.2 asks us to write a function that prompts the user to enter two floating point values and to print the product.

I reviewed slides 17-18 in the lecture, and I finished the practice activity. That all works fine.

But when I try to use the values entered by the user, I am getting an "input mismatch" error. Does anyone have a suggestion?

1. Begin by reviewing the lecture slides related to the problem that you are trying to solve
2. Try to solve the related class activities without looking at the answer
3. When asking for help, try to be as specific as possible about the problem you are having

Overview of the Unit

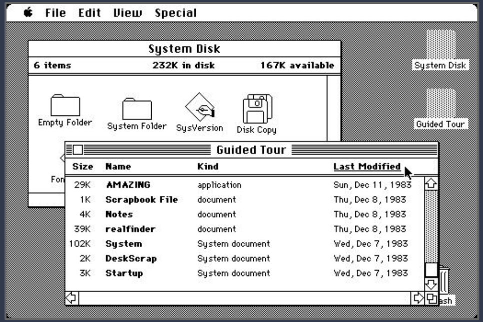


This week we will be constructing GUIs of varying levels of complexity using the JavaFX GUI toolkit.

- Up until this point in Software Development & Problem Solving, the only ***user interfaces*** that we have used have been ***command line interfaces***.
  - These types of interfaces are also called ***plain text user interfaces***(***PTUIs***).
- In this unit we will begin exploring ***Graphical User Interfaces*** (GUIs).
  - A GUI comprises ***graphical controls*** that can display a wide variety of highly customizable text, images, and buttons as well as play sounds and animations.
- Specifically, we will explore:
  - JavaFX Controls
  - Event-Driven Programming
  - Images & Audio
  - Model-View Controller
- We will begin by focusing on building and running basic ***JavaFX applications*** including several JavaFX controls such as ***labels*** and ***buttons***.

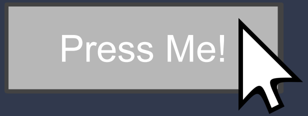
GUIs: What Even Are They ?

- A ***user interface*** is the mechanism that a program provides so that the user can ***interact*** with the program.
- So far in this course we have exclusively used ***text-based*** user interfaces.
  - ***Output*** is printed to a ***text display***.
  - ***Input*** is provided via the keyboard.
  - The interfaces are often called ***command-line interfaces*** (***CLIs***) or ***plain-text user interfaces*** (***PTUIs***).
- ***Graphical User Interfaces*** (***GUIs***, pronounced "***gooey***") are interfaces that allow the user to interact with the program using ***graphical controls***.
  - Such interfaces combine controls like buttons and labels with images, fonts, sounds, and animation.
- When building a GUI, you do not want to try and program everything from scratch.
- Instead, you will build a GUI by combining existing controls that have already been written as part of a ***GUI toolkit***, such as ***JavaFX***.
- During this unit we will focus on a few of the ***many*** controls that are available in JavaFX.
  - You can always search the online documentation to learn about the others.



Size	Name	Kind	Last Modified
29K	AMAZING	application	Sun, Dec 11, 1983
1K	Scrapbook File	document	Thu, Dec 8, 1983
4K	Notes	document	Thu, Dec 8, 1983
39K	realfinder	document	Thu, Dec 8, 1983
102K	System	System document	Wed, Dec 7, 1983
2K	DeskScrap	System document	Wed, Dec 7, 1983
3K	Startup	System document	Wed, Dec 7, 1983

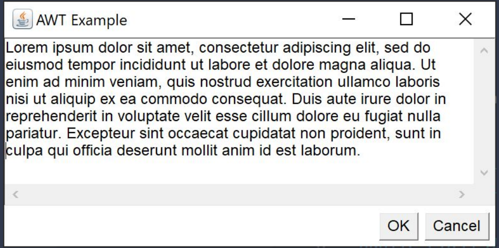
The Apple Macintosh, released in 1984, had the first widely used GUI operating system.



A GUI control like a button visually reacts when the mouse hovers over it, and animates when it is pressed.

You ***do not*** want to spend time trying to write controls like this on your own!

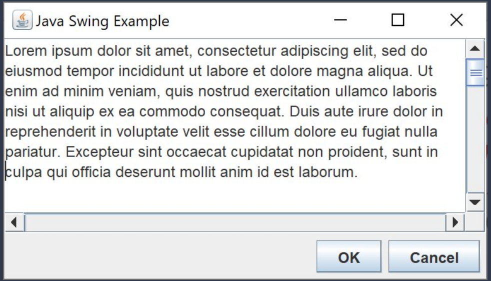
Java GUIs



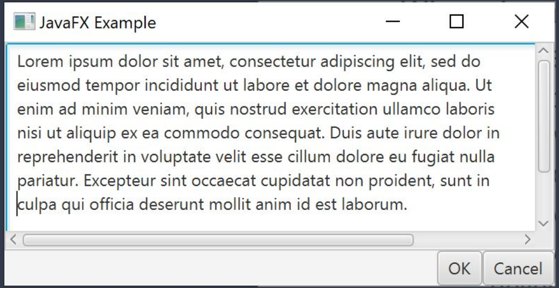
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

AWT

Java Swing



Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.



Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

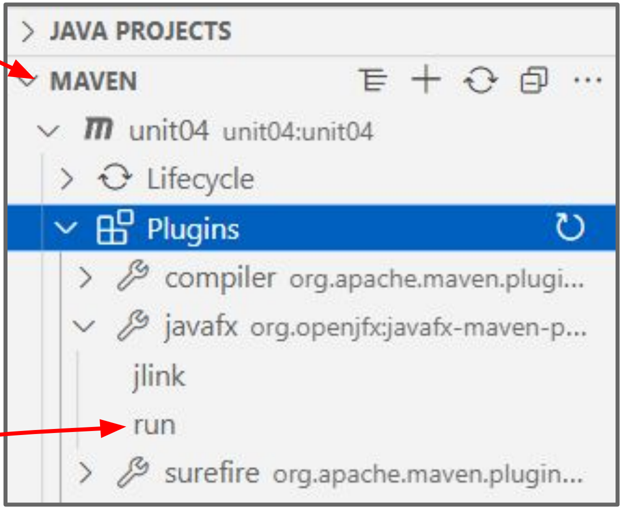
JavaFX

- When Java first launched in 1996, it included a GUI toolkit called the ***Abstract Window Toolkit*** (***AWT***).
  - This toolkit used ***heavyweight*** windows and controls that were ***operating system-specific***.
  - It made it difficult to write a GUI that would look, feel, and work the same on any operating system.
  - This was contradictory to Java's mission to allow developers to "***write once, run anywhere***."
- ***Java Swing*** is a ***lightweight***, ***platform-independent*** GUI toolkit that was first officially added to Java in version 1.2 (1998).
  - GUIs written in Swing should look, feel, and work the same on any operating system or version of Java.
  - Most Java GUIs have been written using Swing.
- ***JavaFX*** is a much more modern GUI toolkit that was first developed in 2005.
  - Like Swing, JavaFX GUIs are platform-independent.
  - JavaFX provides much more sophisticated and customizable controls with advanced features such as transforms and other animations.
  - It was officially included in Java 8, but removed again from Java 11.
  - Today it exists as an open source project.

4.1 Hello, JavaFX!

Your VSCode project has already been configured to use JavaFX. Let's test your configuration by running the "`HelloFX`" app that has been provided to you.

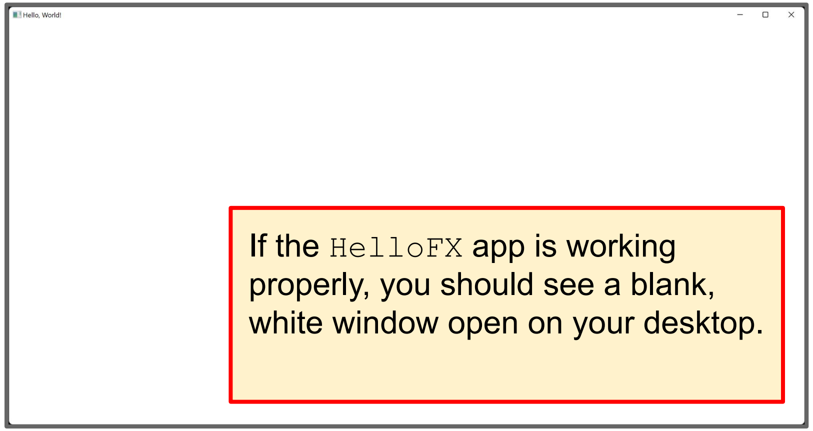
Expand the ***MAVEN*** explorer at the bottom left of the window…



JAVA PROJECTS
MAVEN
unit04 unit04:unit04
Lifecycle
Plugins
compiler org.apache.maven.plugi...
javafx org.openjfx:javafx-maven-p...
jlink
run
surefire org.apache.maven.plugin...

…then use ***Plugins* → *javafx* → *run*** to run the app.

- Open the `HelloFX` application.
  - You will find it in the `src/main/java/unit04` directory.
- Use the VSCode play button to run the application.



If the `HelloFX` app is working properly, you should see a blank, white window open on your desktop.

JavaFX Applications

- A GUI written using JavaFX usually begins by extending the `Application` class.
  - `Application` is an ***abstract class***.
  - Child classes are required to implement the `start(Stage)` method.
  - The `start` method is used to configure the application's `Stage`.
- The `Stage` is a ***window*** in which the JavaFX application is running.
  - The `Stage` is used to display a `Scene`.
  - There may be more than one `Stage` on the screen at a time, each of which will appear as a separate window.
  - The ***primary*** `Stage` is created for you and passed into the `Application`'s `start` method.
- A `Scene` contains a single JavaFX ***control***.
  - This control may be a GUI ***widget*** such as a button, label, or text field.
  - It may also be a ***pane*** which acts as a ***container*** to hold other controls.
- We will explore many different controls and panes in this unit.

All of the JavaFX applications that you write in this unit will start with a very similar ***boilerplate***.

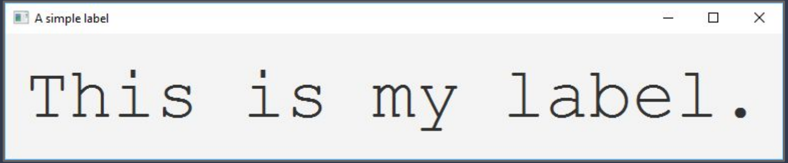
You will begin by ***extending*** the `Application` class, which in turn requires you to ***implement*** the `start(Stage)` method.

```java
public class Boilerplate extends Application {
    @Override
    public void start(Stage stage) {
        stage.setTitle("Boilerplate");
        stage.show();
    }

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

You will also need to include a `main` method so that VS Code can run your program.

JavaFX Controls

A basic JavaFX label displays a string of text.



This is my label.

All JavaFX controls are ***highly customizable***.

For example, once you have created your `Label`, you can change attributes like the ***font***, the ***padding*** (space between the text and the edges), ***background color***, and more. You can even use CSS to style controls!

We'll look some of the ***many*** customization options shortly...

- A ***control*** is a graphical element that is the building block of a GUI.
  - In JavaFX, controls are also called ***nodes***.
- Adding a control to the GUI is fairly simple.
  - ***Construct*** a new instance of the control by invoking its constructor.
  - ***Configure*** it, e.g. height, width, font size, colors, etc.
  - ***Add*** it to a `Scene`, e.g. `Scene scene = new Scene(control);`
- The `Label` is one of the most basic controls that JavaFX offers.
  - A ***label*** is used to display a ***string of text***.
  - The label's constructor accepts a `String` parameter that specifies the label's text.
  - The text can be changed after the label is created using the `setText(String)` method.
  - The `setFont(Font)` method changes the font.
  - The `setPadding(Insets)` method changes the amount of space between the text and the edges of the label.
- Because of all of the customization options, the code to build even simple GUIs will grow to dozens if not hundreds of lines very quickly.

A Closer Look at JavaFX Applications

```java
public class Example extends Application {

  @Override
  public void start(Stage stage) throws Exception {
    Label label = new Label("Little Bunny Foo Foo");

    Scene scene = new Scene(label);

    stage.setTitle("Watership Down");
    stage.setScene(scene);
    stage.show();
  }

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

A Closer Look at JavaFX Applications

JavaFX applications require ***lots*** of imports (not shown here). You should just let VS Code do all the work there.

You will start by extending the `Application` class.

`Application` is an abstract class, and you will need to implement the abstract `start` method to make the Java compiler happy. Again, let VS Code do the work here.

You will need to add a `main` method that calls `launch` to start your GUI.

```java
public class Example extends Application {

  @Override
  public void start(Stage stage) throws Exception {
    Label label = new Label("Little Bunny Foo Foo");

    Scene scene = new Scene(label);

    stage.setTitle("Watership Down");
    stage.setScene(scene);
    stage.show();
  }

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

The ***main*** `Stage` will be created for you and passed into the `start` method.

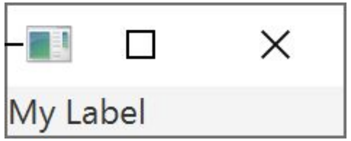
You will need to create and configure your controls, e.g. labels, buttons, etc.

Every `Stage` has a `Scene` that holds the control(s) in the GUI. You'll need to make that, too.

The last thing you need to do is show the `Stage`, which will display it on your computer.

4.2 ~~Taking~~ Making Control

A JavaFX GUI is built by creating controls and adding them to a scene. Usually *lots* of controls. Let's dip our toes into GUI development by creating our first basic GUI with a single JavaFX control: a label.

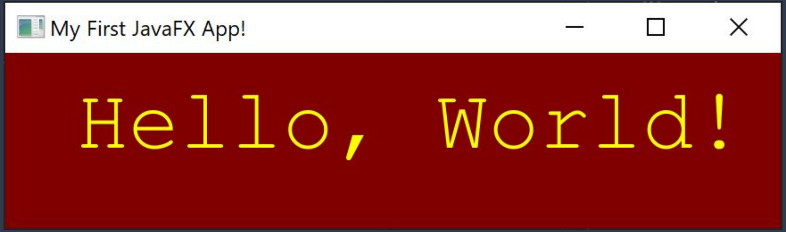


```
public class Example extends Application {

  @Override
  public void start(Stage stage)
                      throws Exception {
    Label label = new Label("My Label");

    Scene scene = new Scene(label);

    stage.setTitle("My First JavaFX GUI");
    stage.setScene(scene);
    stage.show();
  }

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

- Create a new JavaFX application in a file called "`LabelActivity.java`".
  - Create a new `Label` with the ***text*** of your choice.
    - Remember to let VS Code import the classes for you, but you may have more than one option! Always choose the JavaFX option!
  - Create a new `Scene` for your label.
  - Set the `Scene` on the primary `Stage`.
  - Set the ***title*** on the `Stage`.
  - Don't forget to ***show*** the stage!
- Define a `main` method with the appropriate signature.
  - Use the `launch(args)` method to start your application.
  - ***Run*** your application using the VSCode run button.

Customization

- JavaFX controls are ***highly customizable***. For example, you can customize:
  - Height & Width
  - Padding
  - Text Alignment
  - Font (family, size, color, style)
  - Background Fill
  - etc.
- Most customization options are available ***programmatically*** via methods such as `setHeight(double)` and `setFont(Font)`.
- JavaFX also supports the use of ***cascading style sheets*** (***CSS***) to customize controls.
  - The `setStyle(String)` method can be used to specify a ***CSS string***.
  - Just about any customization option that can be done programmatically can also be done with CSS.
  - JavaFX CSS also supports many customization options that are difficult or impossible to set programmatically.
- You can also apply ***transforms*** to animate your controls, e.g. fade in/out, blink, shrink/grow, and so on.
  - We'll talk more about these in the next class.



The label above includes a custom background fill color, font, text fill, and additional padding.

There are far too many ways to customize JavaFX controls for us to cover all of them in a single unit, especially as you start to explore CSS.

We will look at a few today, and a few more throughout the week.

If you are curious, there are lots of resources online that you can use to find new ways to customize.

4.3 Customize Your Label

JavaFX GUIs rarely use the default appearance of controls. Instead, the look and feel of the controls is customized to make the GUI more usable and attractive. Practice customizing your label now.

The `setFont(Font)` method is used to set a ***custom font*** and ***font size***. Experiment with different fonts on your computer.

The `setTextFill(Paint)` method is used to change the font color.

```
label.setFont(new Font("Courier New", 48));
label.setTextFill(Color.YELLOW);
label.setPadding(new Insets(40));
label.setBackground(new Background(
    new BackgroundFill(Color.MAROON,
    CornerRadii.EMPTY, Insets.EMPTY)));
label.setAlignment(Pos.CENTER);
```

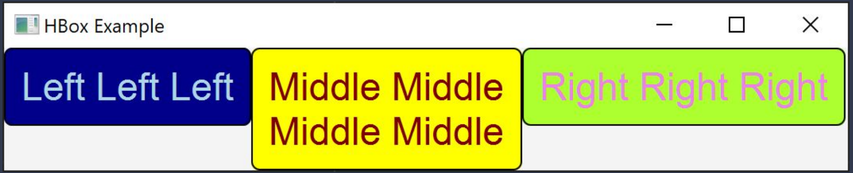
The `setBackground()` method is used to set the background fill (color). Yes, it is a lot of code for that...

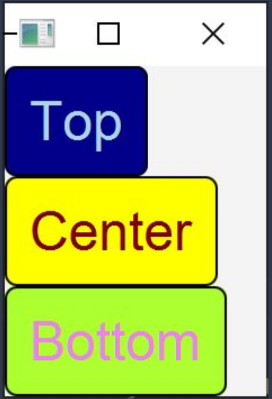
The `setPadding(Insets)` method sets the amount of space between the text and the outside edges.

The `setAlignment(Pos)` method is used to change alignment (left, right, center, etc.).

There are ***lots*** of other customization options...

Layouts





As you can see the labels in the `HBox` and `VBox` layouts ***do not*** fill any extra space by default.

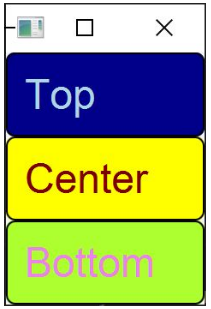
This is because controls want to fit their content snugly by default, and so they won't ***stretch***.

You can change this by using `setMaxWidth(double)` or `setMaxHeight(double)` to change the maximum size to which a widget can be stretched to a larger value, e.g. `Double.POSITIVE_INFINITY`.

- From time to time, you may actually want to create a GUI that has ***more than one control***.
  - Crazy!
- This means that it is necessary to write the code that tells JavaFX about the GUI's ***layout***.
- A layout is like a ***container*** into which ***nodes*** can be added.
- The layout then determines ***where*** the nodes are ***within*** the its own borders. This includes:
  - The ***position*** of each node.
  - The ***size*** (width and height) of each node.
- A JavaFX scene is constructed with a node. A layout ***is a node***.
  - In fact, it is a ***node of nodes***.
  - Layouts may also be added to other layouts!
  - Even simple GUIs will often combine multiple layouts.
- The simplest layouts are `HBox` and `VBox`.
  - `HBox` arranges its children ***horizontally*** from ***left to right***.
  - `VBox` arranges its children ***vertically*** from ***top to bottom***.
- The `getChildren()` method on either layout will return a collection of its children.
  - The `box.getChildren().add(Node)` can be used to add a child to the collection.
- The ***order*** in which each child is added determines its ***position***.

4.4 Adding a Layout

A JavaFX scene can only contain a single control, which doesn't make for very interesting user interfaces! Unless that control is a *layout* that contains lots of other controls working together to create the UI. Let's practice using the most basic JavaFX layouts: `HBox` and `VBox`.



```
Label label = new Label(text);
label.setMaxHeight(
    Double.POSITIVE_INFINITY);
label.setMaxWidth(
    Double.POSITIVE_INFINITY);
HBox hbox = new HBox();
hbox.getChildren().addAll(
    label1, label2, label3
);
```

Don't forget to set the max width and height on your controls so that they expand to fill in the available empty space!

- Open the `LabelActivity` class.
  - Create a new `HBox` or `VBox` named "`box`".
  - Make a total of ***at least three*** different labels and add them to the layout.
    - ***Customize*** each label differently so that they are ***visually distinct***.
    - Remember, `getChildren().add(Node)` can be used to add each label to the layout.
- ***Run*** your application.
- Try changing from one kind of layout to the other and running your app again.
  - You should only need to modify one line of code.

- You have probably already noticed that writing a GUI quickly results in ***lots*** of code.
- This is especially true when writing lots of code to customize the look and feel of the controls in the GUI.
  - You will usually write code so that all of the controls use the same visual ***theme***, e.g. common font, colors, etc.
  - This means ***lots*** of ***very similar*** code to set fonts, colors, padding, etc.
- In fact, you probably wrote three big blocks of code to make your labels in the last activity that only differed in a few, small ways.
  - You may have even created each new label by ***copying and pasting*** code and then making small changes.
- A better alternative is to write a ***factory method***.
  - A factory method contains the code to build some ***product***, e.g. a label for your GUI.
  - You add ***parameters*** for the ***attributes that change*** from one label to the next, e.g. the text displayed on the label.
  - Attributes that might be more consistent like padding or font are ***hard-coded***.
  - The method ***returns*** the new control at the end.
- Whenever you need a new control, you can then just call the corresponding factory method instead of duplicating a big block of code.
  - Remember: ***Don't Repeat Yourself!***

Factory Methods

A ***factory method*** can be used to reduce the number of lines of code by ***eliminating duplicate code*** when building a GUI.

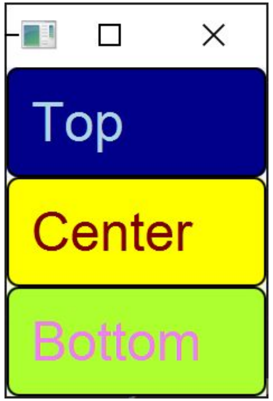
```java
private static Label makeLabel(String text,
                               Color foreground,
                               Color background) {
    Label label = new Label(text);
    label.setFont(new Font("Arial", 24));
    label.setMaxWidth(Double.MAX_VALUE);
    label.setPadding(new Insets(10));
    label.setTextFill(foreground);
    label.setBackground(..., background, ...);
    return label;
}
```

Make sure to add ***parameters*** for anything that ***might change*** from one control to the next...

...and to ***return*** the new control once it has been created and fully customized.

4.5 A Factory Method

It is generally the case that the same customizations are applied to all of the controls of a specific type within a GUI. For example, the same font, padding, and colors will be used on all of the labels in a JavaFX application. This can lead to a *lot* of duplicated (copy/pasted) code! Let's adhere to the DRY principle by writing a factory method to make labels with the same customizations!



Factory methods not only cut down on the lines of code, they also make it much easier to make new controls.

- Open the `LabelActivity` class.
  - Define a ***factory method*** for creating labels.
    - ***Stub out*** the method. Don't worry about parameters yet, but it should return a `Label`.
    - ***Copy & paste*** the code to create one of your labels into the function.
    - Add ***parameters*** for each of the attributes that ***change*** from one label to the next, e.g. text, background color, text fill, etc.
    - ***Return*** the customized Label.
  - ***Replace*** the blocks of code that make each of your labels with a call to your factory method.
    - Are there any parameters that you forgot?
    - Add at least another ***two*** labels to the GUI using your factory method.
- ***Run*** your application.
- Try changing from `VBox` to `HBox` (or vice versa) and running your app again.

Insets, Corner Radii, & Borders

Insets are used to add ***padding*** around the content in a control.

```java
label.setPadding(new Insets(10));
label.setBorder(new Border(
   new BorderStroke(Color.BLACK,
      BorderStrokeStyle.SOLID,
      new CornerRadii(5),
      BorderStroke.THIN)));
```

A Border is created with a BorderStroke, which is highly customizable. This example creates a ***thin***, ***solid***, ***black*** border with ***slightly rounded corners***.

Borders can be customized with ***colors***, ***images***, and ***dotted*** or ***dashed*** lines among many other options.

- Insets are often used to define extra space on the ***top***, ***bottom***, ***left***, and ***right*** of something.
  - For example, Insets are used to specify the amount of padding between the text on a label and the label's outside edges.
  - Insets can be created with a ***single value*** that is used for all four sides or with ***separate values*** for each.
  - `Insets.EMPTY` is a constant used for empty insets.
- CornerRadii are used to round the corners on an otherwise rectangular shape.
  - For example, a background fill color may have rounded corners.
  - Corner radii can be created with a ***single value*** that is used for all four corners, or with ***separate values*** for the ***top left***, ***top right***, ***bottom left***, and ***bottom right***.
  - `CornerRadii.EMPTY` is a constant used to indicate that corners should not be rounded.
- A Border is drawn around the outside edges of a component.
  - A `Border` is created with at least one BorderStroke.
  - A `BorderStroke` is created with a ***fill color***, ***stroke style***, ***corner radii***, and ***border width***.
- Borders and backgrounds can be configured to use the same corner radii so that the background does not ***bleed*** outside of the border.

An Illustrated Example

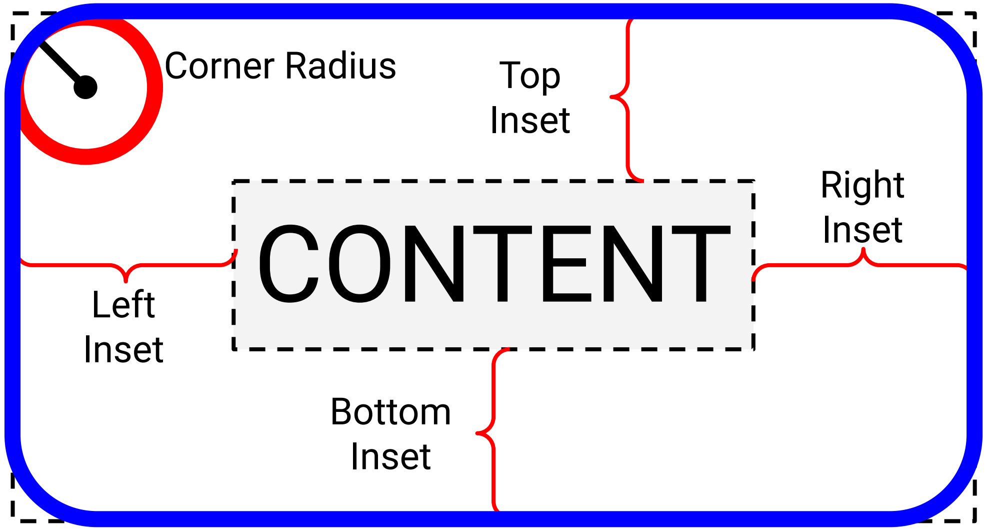
Insets are used to define how much space is between the ***content*** of a control and its edges.

Usually the same value is used for all four insets, but the ***top***, ***bottom***, ***left***, and ***right*** insets can be configured separately.

A ***border*** is drawn around the outside edges of a control.

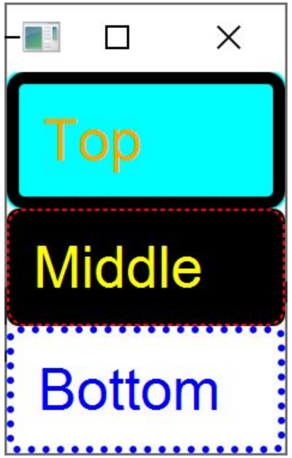
Borders can be customized in a number of ways including ***color***, ***stroke style***, ***corner radii***, and ***thickness***.

The ***corner radii*** specify the ***radius*** of the ***circle*** that is used to draw ***rounded edges***, e.g on a ***border*** or a ***background fill***. As with insets, all four corners can be configured separately.



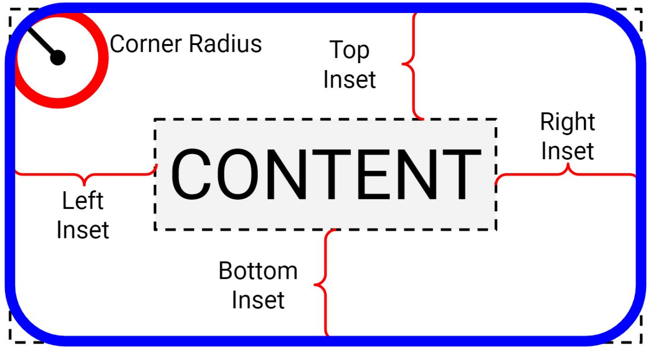
4.6 Border Practice

Borders can really change the appearance of the controls in your user interfaces. Practice adding borders to the labels in your application now. Experiment with different styles to see how they look.



```
label.setPadding(new Insets(10));
label.setBorder(new Border(
    new BorderStroke(Color.BLACK,
        BorderStrokeStyle.SOLID,
        new CornerRadii(5),
        BorderStroke.THIN)));
```

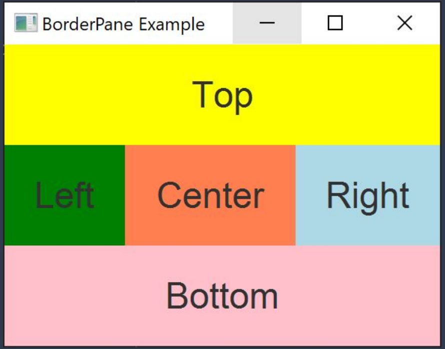
- Open the `LabelActivity` class.
  - Add a `Border` to each of your labels.
  - Use the example code for inspiration, but play around with the different customization options for ***color***, ***stroke style***, ***corner radii***, and ***width***.
  - ***Do not*** worry if your background fill doesn't match your border shape exactly.
- ***Run*** your application.



BorderPane

A `BorderPane` positions ***up to five*** nodes into regions: ***top***, ***bottom***, ***left***, ***right***, and ***center***.

```java
BorderPane pane = new BorderPane();
pane.setTop(makeLabel("Top", Color.YELLOW));
pane.setBottom(makeLabel("Bottom", Color.PINK));
pane.setLeft(makeLabel("Left", Color.GREEN));
pane.setRight(makeLabel("Right", Color.LIGHTBLUE));
pane.setCenter(makeLabel("Center", Color.CORAL));
```

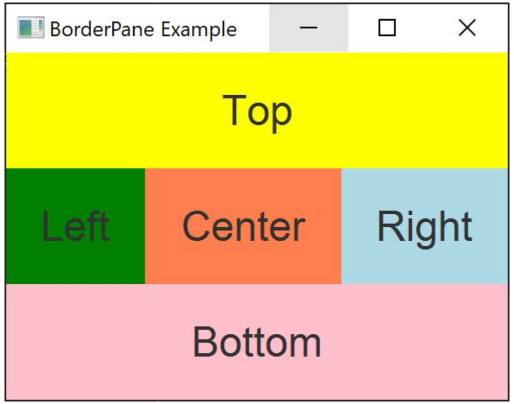


If present, the ***top*** and ***bottom*** ranges expand horizontally to the edges of the layout.

- `HBox` and `VBox` are the simplest layouts.
  - Children are arranged ***sequentially*** in the order that they are added.
  - The default width and height of each child is determined by the child.
  - A virtually ***unlimited*** number of children can be added to the layout.
- A `BorderPane` is a little more complex; it divides itself into ***five*** regions: ***top***, ***bottom***, ***left***, ***right***, and ***center***.
- ***One*** child node can be added to each of the five regions.
  - If ***two or more*** nodes are added to the same region, only the ***last*** one is kept.
  - Remember, a ***layout*** is also a node!
- `BorderPane` provides a different setter for each region, e.g. `setTop(Node)`, `setLeft(Node)`, etc.
- If a region is ***empty***, it will not take up any space at all (it will be ***invisible***).
  - If present, the top and bottom regions ***extend horizontally*** to the edges of the layout.

4.7 Make a BorderPane

The JavaFX BorderPane can be used to quickly create a layout with 5 controls neatly arranged in the window. Practice using it by writing a new Application uses a `BorderPane` to display five labels, each with a different background color.



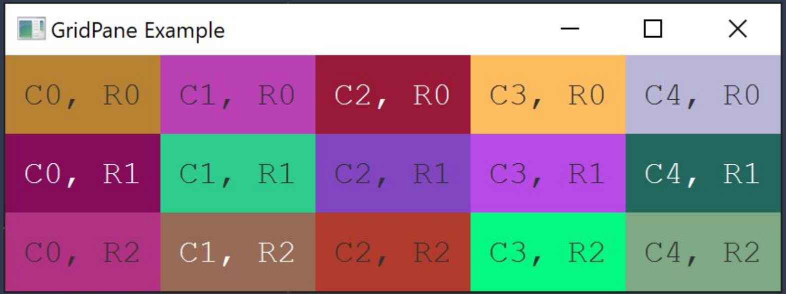
```
BorderPane pane = new BorderPane();
pane.setTop(makeLabel("Top", Color.YELLOW));
pane.setBottom(makeLabel("Bottom", Color.PINK));
pane.setLeft(makeLabel("Left", Color.GREEN));
pane.setRight(makeLabel("Right", Color.LIGHTBLUE));
pane.setCenter(makeLabel("Center", Color.CORAL));
```

- Create a new JavaFX application in a file called "`BorderPaneActivity.java`".
  - Extend `Application` and override the `start(Stage)` method.
- In the `start(Stage)` method, create a `BorderPane` that contains ***five labels***.
  - The ***text*** and ***background color*** of each label should be different.
  - ***Hint***: copy and paste the ***factory method*** from the label activity.
- Define a `main` method with the appropriate signature and use it to launch your app.

GridPane

A `GridPane` displays child nodes in a grid arranged into columns and rows.

```java
GridPane gridPane = new GridPane();
for(int col=0; col<5; col++) {
   for(int row=0; row<3; row++) {
      Label label = makeLabel(col, row);
      gridPane.add(label, col, row);
   }
}
```



```
C0, R0  C1, R0  C2, R0  C3, R0  C4, R0
C0, R1  C1, R1  C2, R1  C3, R1  C4, R1
C0, R2  C1, R2  C2, R2  C3, R2  C4, R2
```

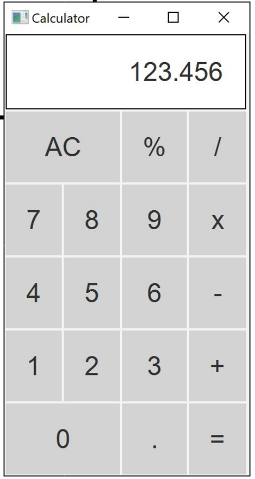
Like other ***parent nodes***, you may need to set the ***max width*** and ***max height*** to stretch children.

- A `GridPane` organizes child nodes into ***columns*** and ***rows***.
  - Make sure to read that twice: whenever they are used together, ***columns*** come before ***rows***.
- The number of columns and rows does not need to be specified when the `GridPane` is created.
  - It is inferred from the children that are added.
- Unlike some of the other ***parent*** nodes, children are added directly to a `GridPane` using its `add(Node, int col, int row)` method.
  - e.g. `gridPane.add(label, 3, 4)` will add the label to column 3, row 4.
- Individual child nodes can be configured to ***span*** multiple columns or rows.
  - This is done by providing two additional arguments when adding a new node to the layout
  - to indicate the ***column span*** and the ***row span*** of the node.
  - e.g. `gp.add(label, 3, 4, 1, 2)` will add a label that spans ***1 column*** and ***2 rows***.

4.8 A GridPane

The GridPane is an incredibly powerful and flexible layout. Practice using it by writing the code necessary to build a Calculator GUI. Do not worry about getting it *exactly* the same as the example.

```
GridPane gridPane = new GridPane();
for(int col=0; col<5; col++) {
   for(int row=0; row<3; row++) {
      Label label = makeLabel(col, row);
      gridPane.add(label, col, row);
   }
}
int colSpan = 2;
int rowSpan = 3;
gridPane.add(makeLabel(6, 4), 6, 4,
    colSpan, rowSpan);
```

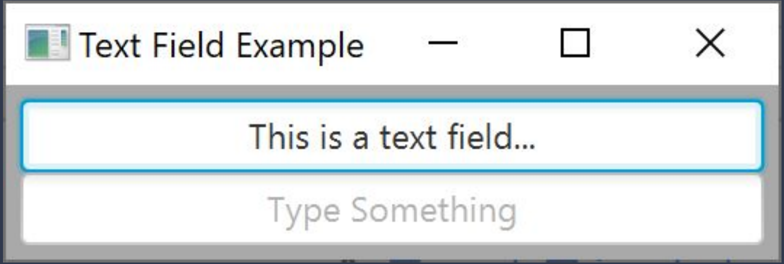


- Create a new JavaFX application in a file named "`GridPaneActivity.java`".
  - Try to make a GUI that looks like the example to the left using ***only*** a `GridPane` of labels.
  - Remember, `gridPane.add(child, col, row)` will add the child to the specified column and row and `gridPane.add(child, col, row, colSpan, rowSpan)` will add a child that spans multiple columns and/or rows.
  - ***Hint***: most of the labels have the same settings (font, alignment, background color, min/max size, etc.); consider using a factory method to make them.
  - ***Hint***: Instead of using `Insets.EMPTY` when setting a background fill, ***set the insets to 1 pixels*** (i.e. `new Insets(1)`) to create the white border around the buttons.
- ***Run*** your application.

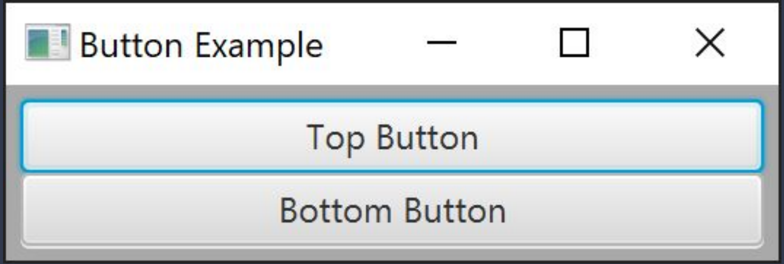
TextField & Button

- A `TextField` is a control into which the user can type a single line of text.
  - A TextField can be created ***empty*** or with default text by passing a String into its constructor.
- A `TextField` provides lots of useful methods including:
  - `setText(String)` changes the text in the field.
  - `getText()` returns the current text.
  - `setPromptText(String)` changes the prompt in the text field.
- A `Button` is a control that the user can interact with using the mouse or keyboard.
  - Buttons are created with a ***text*** and/or image label.
  - Buttons respond to ***mouse events*** like ***enter***, ***exit***, ***hover***, and ***click***.
- In JavaFX, buttons are really just fancy ***labels***, and so have a lot of the same methods including `setText()` and `getText()`.
- Trying to set the ***background*** on a button will remove its default highlights and shading.

A `TextField` may or may not include prompt text. The user can type a single line of text into it.

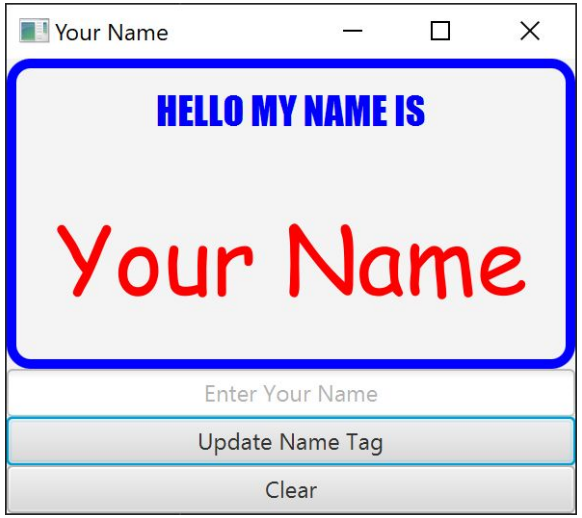


A `Button` reacts to the presence of the mouse. It changes color and animates when the mouse is clicked or the ***enter*** key is pressed on the keyboard.



4.9 Say Hello

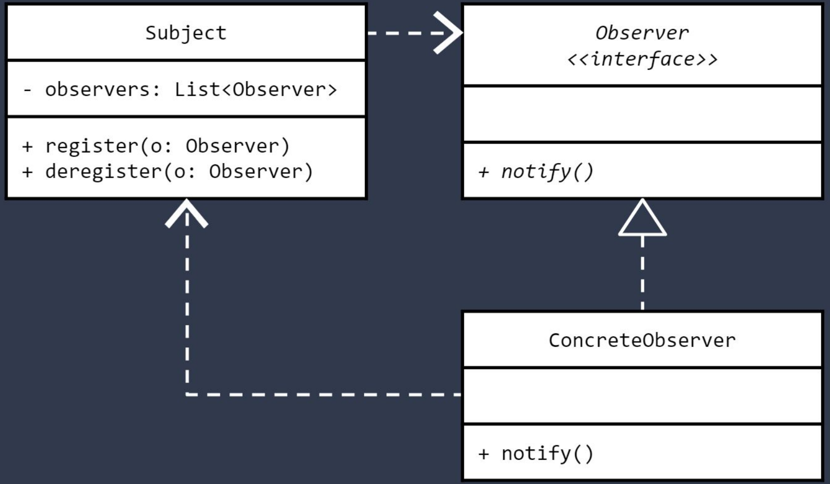
Let's put everything that we have learned together to create a JavaFX GUI that uses buttons, labels, text fields, layouts, and customization to display a name tag!



The above layout uses **"Impact"** and "Comic Sans MS" fonts as well as ***red*** and ***blue*** coloring.

- Create a new JavaFX application named "`NameTag`".
  - Create a GUI that includes ***at least one label***, ***text field***, and two ***buttons***.
  - Try to match the provided example. You ***do not*** have to match it ***exactly***.
- ***Run*** your application.
  - What happens when you ***click*** the button?

The Observer Design Pattern



In the ***Observer Pattern***, the ***Subject*** maintains a list of ***Observers*** that should be ***notified*** whenever something interesting happens.

Classes that ***implement*** the ***Observer*** interface can ***register*** to be notified.

- Right now, when a button is clicked in the `NameTag` application, ***nothing happens***.
    - This makes sense! We haven't actually written any code to execute when the button is pressed.
- But we'd *like* something to happen when each of the buttons is pressed.
    - The first button should update the name tag label with whatever name the user typed into the text field.
    - The second button should clear the name tag label.
- But *how* do we know when the user has pressed the button?
- Let's consider ***The Observer Design Pattern*** in which there is some ***subject*** of interest.
    - We'd like to know when something interesting happens to the ***subject***.
- The subject maintains a list of interested ***observers***.
    - Each ***concrete observer*** implements the same interface.
    - The interface defines a method that can be used to ***notify*** the observer when something interesting happens.
- Wouldn't it be nice if we could ***observe*** the Button so that it would ***notify*** us when it is pressed?

Event Handlers

- EventHandler<T> is an ***observer interface*** that must be implemented by classes that want to be ***notified*** when an interesting event occurs on a JavaFX control.
    - `EventHandler` is a ***generic interface*** that defines a type parameter `T`.
    - We will go into more detail about ***generics*** in Java in the coming weeks - for now, just follow the pattern.
    - The real type that is used in place of `T` ***must extend*** the Event class.
    - The `handle(T)` method is called when the event of interest occurs.
- There are many different classes that extend `Event`, including ActionEvent.
    - An `ActionEvent` is created when the user interacts with a button by pressing it with the mouse.
- A `Button` is a ***subject***.
    - Interested ***observers*** may use the `button.setOnAction(EventHandler)` method to ***register*** to be ***notified*** when an `ActionEvent` occurs.
    - The observer must implement the `EventHandler<ActionEvent>` interface.
    - When the button is ***pressed***, the `handle(ActionEvent)` method is called on the registered observer.
    - Only ***one*** observer may register on each button.

The EventHandler interface is implemented by ***observers*** interested in being notified when events occur. The type of event is specified in `<>`.

```
public class PrintOnPress
        implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        System.out.println(event);
    }
}
```

The event handler must be ***registered*** with a specific `Button` ***subject*** using the `setOnAction(EventHandler)` method.

```
Button button = new Button("Press Me!");
button.setOnAction(new PrintOnPress());
```

The `handle(ActionEvent)` method will be called whenever the ***subject*** Button is pressed.

4.11 Event Handlers

Event handlers can be used to make your GUI applications respond when the user interacts with the controls. Let's check it out by making an `EventHandler` that will update the name tag label when the update button is pressed.

```
public class PrintOnPress
        implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        System.out.println(event);
    }
}
```

```
Button button = new Button("Press Me!");
button.setOnAction(new PrintOnPress());
```

Pressing the ***Update Name Tag*** button should now change the text in the name tag label.

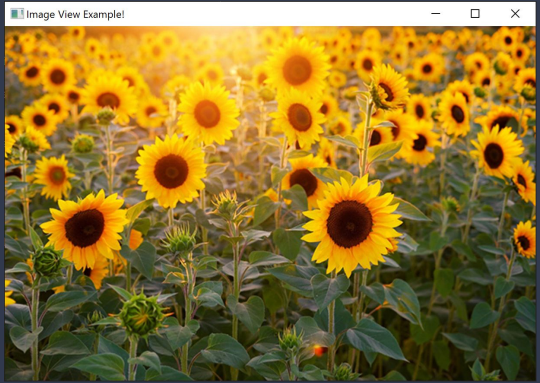
- ***Create*** a new Java class in a file named "`Updater.java`" that implements the `EventHandler<ActionEvent>`.
  - You will need to declare fields for a `TextField` and a `Label` and write an ***initializing constructor***.
  - The `handle(ActionEvent)` method should get the text from the `TextField` and use it to update the `Label`.
- ***Open*** the `NameTag` class.
  - Create an instance of your `Updater` class with the correct `TextField` and `Label`.
  - ***Register*** the `Updater` to be notified when the update `Button` is pressed.
- ***Run*** your application.
  - What happens when you press the update button ***now***?

Image & ImageView

An `Image` can be loaded from a String URL...

```java
Image image =
    new Image("file:media/images/sunflowers.jpg");
ImageView view = new ImageView(image);
HBox box = new HBox();
box.getChildren().add(view);
```

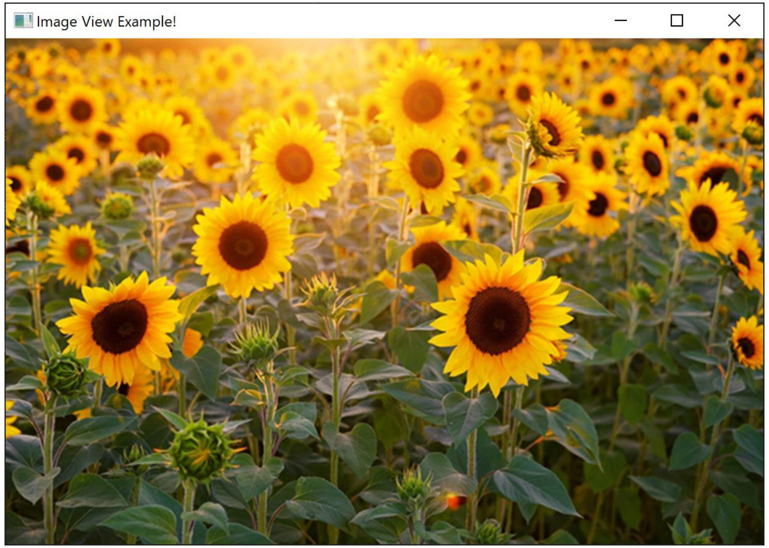
...and an `ImageView` can be used to display it. But the ImageView must be added to a ***parent*** like an `HBox` before it can be added to a Scene.



- The JavaFX [`Image`]() class can be used to load and display images.
  - It supports PNG, JPG, GIF, and BMP formats.
- There are two basic ways to load an image.
  - A `String` ***URL*** that indicates the location of the image, e.g. `"file:media/images/sunflowers.jpg"` refers to the image file in the specified directory in your project.
  - An `InputStream` that can be used to read the image data.
- A JavaFX `Image` is not a control in the same way that a `Label` is, and so in order to use it, it must be added to some other control, e.g. as an icon on a button or even a background image in a layout.
- An [`ImageView`]() is a control just for displaying images.
  - It can be created ***blank***, with an `Image`, or with a string URL used to load an `Image`.
  - It provides `getImage()` and `setImage()` methods that can be used to get or change the image displayed.
- An `ImageView` ***cannot*** be added directly to a `Scene`.
  - It must first be added to a ***parent*** like an `HBox`.

4.12 An Image Viewer

JavaFX applications can display more than just labels and buttons. They can display images, too! Let's write a JavaFX application that displays a single image.

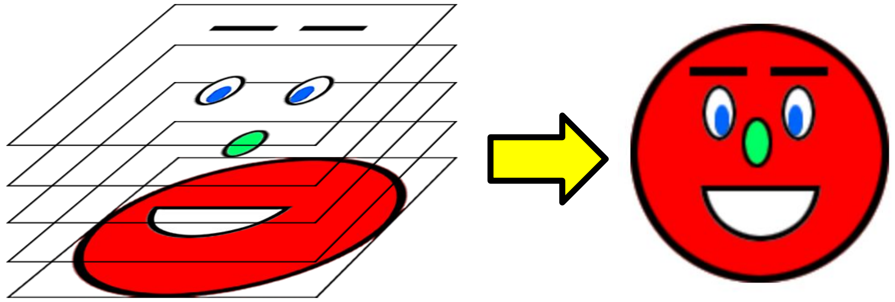


```
Image image =
   new Image("file:media/images/sunflowers.jpg");
ImageView view = new ImageView(image);
HBox box = new HBox();
box.getChildren().add(view);
```

- Create a new JavaFX application named "`ImageViewer`".
  - ***Extend*** `Application` and ***override*** `start(Stage)`.
  - Create an `Image` using the URL "`file:media/images/smb.gif`".
  - Use the `Image` to create a new `ImageView`.
  - Add the `ImageView` to a ***parent*** such as an `HBox`, `VBox`, or `BorderPane`.
  - Complete your app by making a `Scene`, setting the title, and showing the stage.
- Define a `main` method with the appropriate signature and use it to launch your app.

StackPane

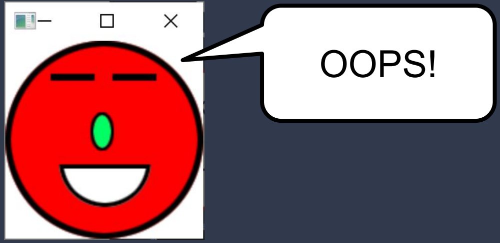
- A `StackPane` stacks its children ***one on top of the other*** in the order that they are added.
  - The ***first*** child is on the ***bottom***.
  - The ***last*** child is on the ***top***.
- Like `HBox` and `VBox`, the `getChildren()` method returns a collection of child ***nodes***.
  - And `pane.getChildren().add(Node)` can be used to add a new child to the top of the stack.
- One possible application of a `StackPane` is to combine multiple images with transparent regions to create a ***composite image***.



Like most ***parent*** nodes in JavaFX, it is possible to add multiple children to a `StackPane` at the same time using the `addAll(Node…)` method.

```java
StackPane pane = new StackPane();
pane.getChildren().addAll(
    new ImageView("file:eyesblue.png"),
    new ImageView("file:headred.png"),
    new ImageView("file:mouthbasic.png"),
    new ImageView("file:nosegreen.png"),
    new ImageView("file:browsbasic.png")
);
```

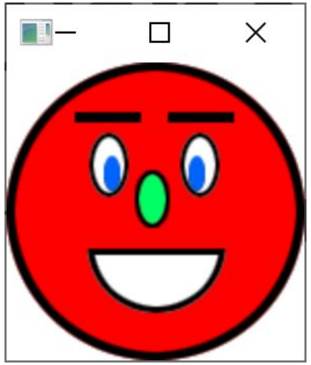
Care must be taken to add the children ***from the bottom up*** to avoid unintentionally occluding children closer to the bottom of the stack.



4.13 A StackPane

The Stack Pane allows you to stack controls one on top of the other, with the first on the bottom and the last on the top. You can use it to combine images with transparent backgrounds to create a composite image. Do it now!

There are 1024 unique combinations of emoji parts.



```
StackPane pane = new StackPane();
pane.getChildren().addAll(
    new ImageView("file:eyesblue.png"),
    new ImageView("file:headred.png"),
    new ImageView("file:mouthbasic.png"),
    new ImageView("file:nosegreen.png"),
    new ImageView("file:browsbasic.png")
);
```

- Explore the "`media/images/emojis`" folder in your repository.
    - You will see that many PNG images have been provided to you, each of which includes a variation of some component part of an emoji: ***head***, ***eyes***, ***nose***, ***mouth***, or ***eyebrows***.
    - You can preview the images by opening them in VS Code (though some will be hard to see if you are using a dark theme).
    - Each of these images can be referenced using a URL in the format `"file:media/images/emojis/<filename>"` .
- Create a new JavaFX application in a file named "`StackPaneActivity.java`".
    - Use a `StackPane` to display an emoji made from component parts of your choice.
        - You must include a head, eyes, nose, mouth, and eyebrows.
- ***Run*** your application.

Media Player

A Media is created using a ***URI String***. One way to do this is to make a `java.io.File` from the path to the media, and then convert it into a URI string.

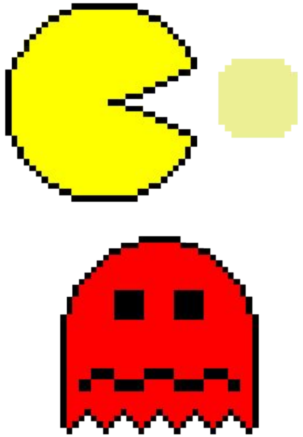
```
String path = "media/sounds/buzzer.wav";
String uri = new File(path).toURI().toString();
Media media = new Media(uri);
MediaPlayer player = new MediaPlayer(media);
player.play();
```

One the `Media` has been created, a `MediaPlayer` is needed to control playback. You will need a different `MediaPlayer` for each `Media`.

- `Media` is a class that represents some, well, ***media***, e.g. an audio or video file.
    - An instance of the `Media` class is created with the URI the points to the ***source*** of the media. This may be an ***Internet address*** or a ***file*** in the local file system.
- A `MediaPlayer` can be used to play `Media`.
    - A `MediaPlayer` is created with an instance of the `Media` class.
- Once created, a `MediaPlayer` provides many methods to control playback of the `Media`.
    - `play()` - begins playing the media.
    - `pause()` - pauses playback.
    - `stop()` - stops playback and resets to the beginning.
    - `seek(Duration)` - skips to the specified time in the media.
    - `setRate(double)` - sets the playback rate (up to 8X).
    - And many more.
- A `MediaPlayer` does not have a ***visual component***, meaning that, by itself, it will not ***display*** video.
    - A `MediaView` can be used to display video. We will not be using `MediaView` in this unit, but you should always feel free to explore on your own.

4.14 Waka Waka

JavaFX GUIs can be used to play sound and video. Let's implement a soundboard that plays Pac-Man sound effects.



```
String path = "media/sounds/buzzer.wav";
String uri = new File(path).toURI().toString();
Media media = new Media(uri);
MediaPlayer player = new MediaPlayer(media);
player.play();
```

- Examine the contents of the "`media/sounds`" folder in your project.
    - It contains several `.WAV` sound files including "`start.wav`", "`chomp.wav`", "`eat.wav`", and "`end.wav`".
- Create a new JavaFX application in a file named "`PacMan.java`"
    - Create a GUI that includes one `Button` for each of the 4 provided Pac-Man sounds.
    - Add an event handler to each `Button` that plays the corresponding sound. You will need to make a `Media` for each sound file, and a `MediaPlayer` to play it.
    - ***Hint***: The relative path to each file is "`media/sounds/<file.wav>` ". Use this to create a file and convert it to a URI.
- ***Run*** your application and test the buttons.
    - What happens when you click one of buttons several times quickly?

- The current version of the Pac-Man application creates a new `Media` and `MediaPlayer` each time a button is clicked.
    - This is not only inefficient, it means that the same sound can be played concurrently causing an overlapping, echo effect.
- Instead, we can create ***one*** `MediaPlayer` to play each sound.
    - This allows us to reuse the same player over and over.
    - Calling `play()` on a `MediaPlayer` that is already playing won't start the media over. No more echo!
- There is another problem, though: once the media plays to the end, calling `play()` again will not restart it.
    - The simplest fix for this is to call the `stop()` method to reset the player to the ***beginning***.
    - Calling `stop()` ***right before*** `play()` will restart the media every time the button is pushed.
- The `getStatus()` method can be used to get the current status of the `MediaPlayer`. It returns a `MediaPlayer.Status` (an enum), including:
    - `Status.READY`
    - `Status.PAUSED`
    - `Status.PLAYING`
    - `Status.STOPPED`

Advanced Controls

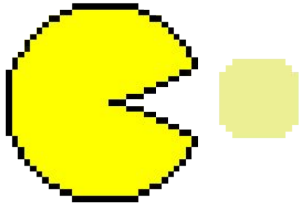
Rather than create a new `MediaPlayer` each time the button is pressed, it can be created ***once***… and saved as a field in your event handler…

```java
public class SoundPlayer
          implements EventHandler<ActionEvent> {
   private final MediaPlayer player;

   public SoundPlayer(MediaPlayer player) {
      this.player = player;
   }

   @Override
   public void handle(ActionEvent event) {
      player.stop();
      player.play();
   }
}
```

The event handler can make sure that the sound restarts every time the button is pressed by calling `stop()` immediately before `play()` each time the button is pressed.

4.15 Play It Once, Sam.

Right now the Pac-Man soundboard will play the same sound each time the button is pressed, causing a strident echo effect. Fix the Pac-Man application so that it creates one `MediaPlayer` per audio clip, and stops the media before trying to play it each time the button is pressed.



```
player.stop();
player.play();
```

- Open the `PacMan` application.
  - Modify your logic to create ***one*** `MediaPlayer` for each audio clip ***before*** the `Button` to play the clip is created. Pass the clip into the constructor of your event handler!
  - Each time the button is pressed:
    - Use `getStatus()` to print the status of the player ***before*** doing anything else.
    - Call `stop()` and then `play()` to restart the clip.
    - ***Do not*** create a new `MediaPlayer`!
- ***Run*** your application and verify that pressing each button multiple times restarts the corresponding audio clip.

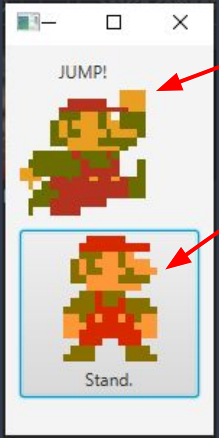
Graphics

- An `ImageView` displays an `Image` and doesn't do much of anything else.
- So what if you need an image to be displayed with accompanying text, or visibly respond to mouse clicks like a button?
- JavaFX labels and buttons will display ***graphics*** as well as text.
    - A graphic may be any other `Node`, e.g an `ImageView` that is displayed on the control.
- For example, a `Label` can also be created with a graphic, e.g. `new Label(text, graphic)`
    - The graphic will be displayed alongside the text.
- Similarly, `new Button(text, graphic)` will create a new `Button` that uses the specified graphic.
    - The button will still display with beveled edges and react visible when the mouse hovers or clicks.
    - If ***only*** the graphic is needed, use an ***empty string*** (`""`) for the text.
- By default the text displays to the ***right*** of the image.
    - The `setContentDisplay(ContentDisplay)` method can be used to display the image in a different location, e.g. `ContentDisplay.TOP` .

A `Label` may be created with a ***graphic*** using another JavaFX node, e.g. an `ImageView`.

```java
ImageView jumpImage = new ImageView("file:jump.png");
Label label = new Label("JUMP!", jumpImage);
label.setContentDisplay(ContentDisplay.BOTTOM);
```

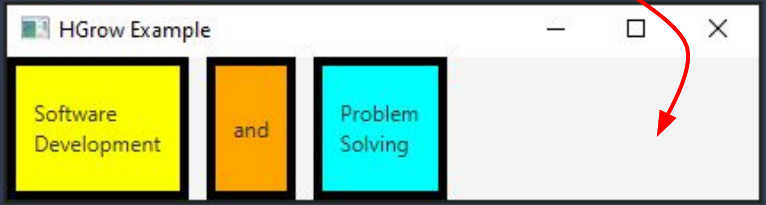
By default, text is displayed on the ***right***. That can be changed by setting the content display, e.g. to display the image on the ***bottom***.



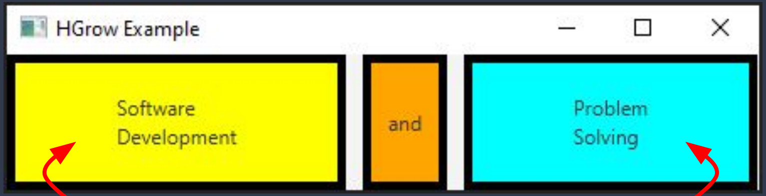
Buttons have all of the same options as labels. In this case the content display is set to display the image at the ***top***.

HGrow and VGrow

By default, an `HBox` will ***not*** stretch its child nodes for fill any extra horizontal space, even if the ***max width*** has been set.



The `HBox.setHGrow(node, Priority)` method can be used to signal that it is ***OK*** to stretch a child node horizontally.



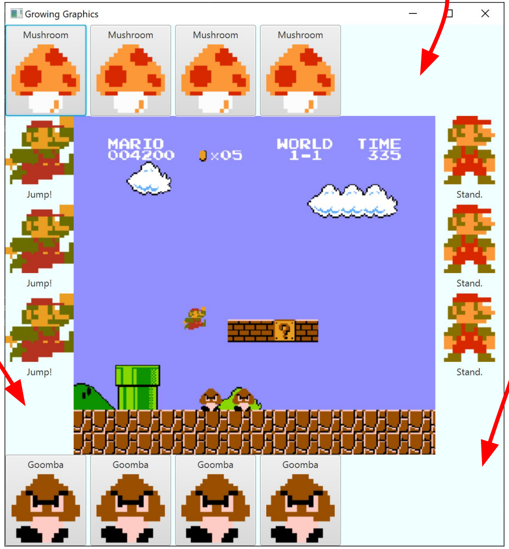
In this example, the left and right labels have been set to `Priority.ALWAYS`.

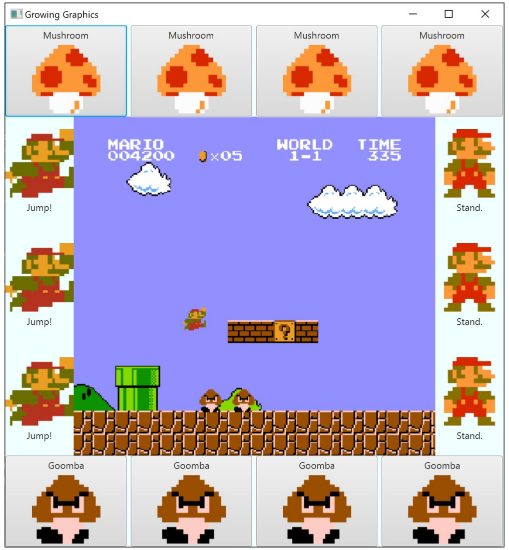
- Setting the ***max height*** on the child nodes in an `HBox` will stretch them to fill any extra ***vertical*** space, but not any extra ***horizontal*** space.
  - This is because the HBox doesn't know which children should be stretched and which should be left alone.
- There is a ***static*** method on the HBox class that can be used to set the ***horizontal grow priority*** for any JavaFX node.
  - `HBox.setHgrow(node, Priority)`
- The priority options are:
  - `Priority.ALWAYS` - the node should always be stretched to fill in extra space.
  - `Priority.NEVER` - the node should not be stretched at all.
  - `Priority.SOMETIMES` - the node should only be stretched if no other node in the same layout is set to `Priority.ALWAYS`.
- If two or more child nodes have the ***same priority***, any extra space will be ***divided equally*** between them.
- The `VBox` class has a static method that works similarly for ***vertical grow priority***.
  - `VBox.setVgrow(node, Priority)`

An Illustrated Example

If the ***horizontal*** and ***vertical grow priorities*** of the child nodes in an `HBox` or a `VBox` are not set, there may be extra space in the layout...

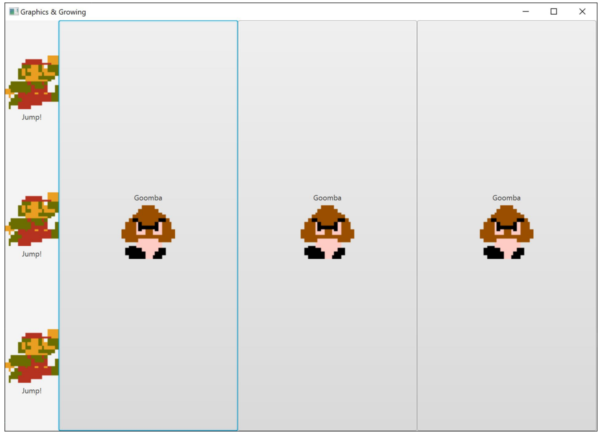
Using the `HBox.setHgrow(node, Priority)` and `VBox.setVgrow(node, Priority)` methods with the buttons and labels can fix that.





4.16 Graphics & Growing

We can add "graphics" to buttons and labels in JavaFX applications. We can also tell controls to "grow" horizontally or vertically to fill in extra space. Try it now!



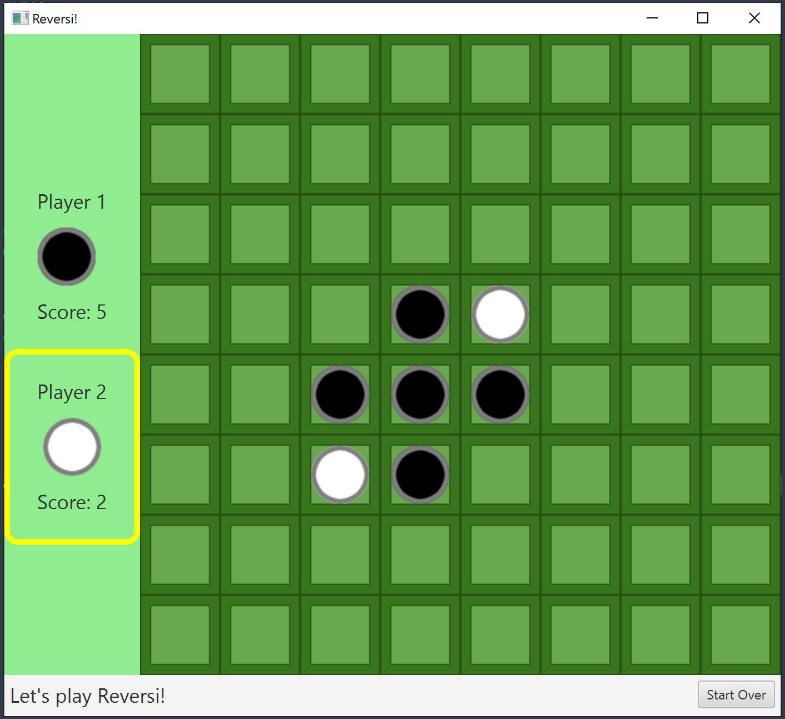
Buttons with graphics may exhibit ***odd behavior***, e.g. changing size when clicked.

Try setting the ***padding*** to at least one pixel to correct this problem.

- Create a new JavaFX application in a file named "`GrowingGraphics.java`". Your application must combine the following:
  - At least ***one*** label that uses a graphic in a `VBox`.
  - At least ***one*** Button that uses a graphic in an `HBox`.
  - The `HBox.setHGrow(Node, Priority)` and `VBox.setVgrow(Node, Priority)` methods must be used to insure that the controls stretch to fill in any extra space.
- ***Run*** you application to verify that if works as expected.
  - ***Hint***: test your application by maximizing the window.

```
ImageView jumpImage = new ImageView("file:jump.png");
Label label = new Label("JUMP!", jumpImage);
label.setContentDisplay(ContentDisplay.BOTTOM);
HBox.setHgrow(label, Priority.ALWAYS);
```

Reversi



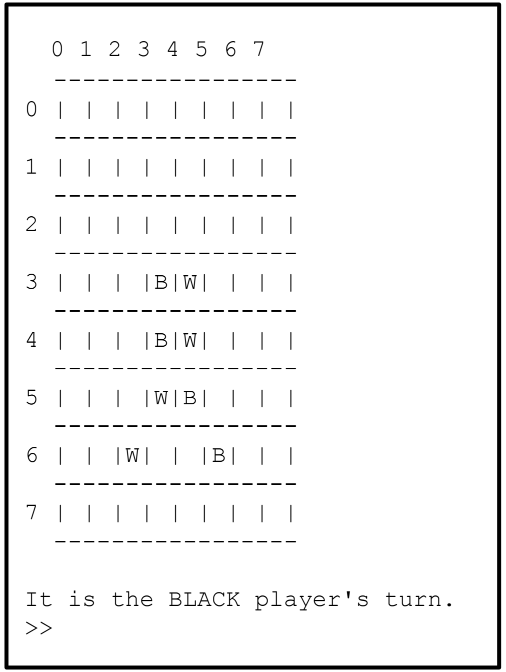
Let's play Reversi!

We will be building a GUI that two players can use to play a game of Reversi!

- ***Reversi*** is a classic board game that uses circular pieces that are ***black*** on one side and ***white*** on the other.
  - Reversi is also called ***Othello***.
- One player chooses ***black*** and the other player chooses ***white***.
- At the start of the game, four pieces are played in the center of the board.
  - Two with the black side facing up, and two with the white side facing up.
  - The ***black*** player goes first.
- During their turn the player will play ***one*** piece on an empty square with their color facing up.
  - A move is only ***valid*** if two of the player's pieces form a line with at least one of the opponent's pieces in the middle.
  - The opponent's pieces are then ***flipped*** to the player's color.
  - If it is not possible to make a valid move, the player must ***pass***.
- The game is over when ***neither*** player can make a move.
  - The player with the most pieces on the board ***wins***.

4.17 Play a Game of Reversi

Before we can implement a GUI to play Reversi, we'll need to understand how the game works. A Command-Line Interface (CLI) has been provided to you so that you can play. Take a moment and play a few rounds until you understand how the game works.

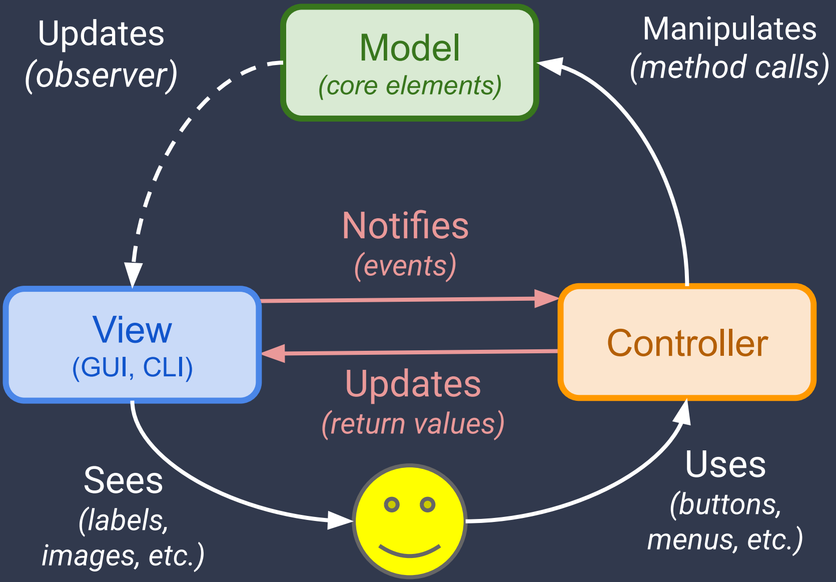


```
   0 1 2 3 4 5 6 7
  -----------------
0 | | | | | | | | |
  -----------------
1 | | | | | | | | |
  -----------------
2 | | | | | | | | |
  -----------------
3 | | | |B|W| | | |
  -----------------
4 | | | |B|W| | | |
  -----------------
5 | | | |W|B| | | |
  -----------------
6 | | |W| | |B| | |
  -----------------
7 | | | | | | | | |
  -----------------

It is the BLACK player's turn.
>>
```

- The `unit04.reversi.view.ReversiCLI` class provides a simple ***command-line interface*** for playing a game of Reversi.
  - ***Run*** the class.
  - Use the "`help`" command to display the possible commands.
  - Make a few ***moves***.
  - ***Ask*** questions if there is something that you don't understand!

- In software development, an ***architecture*** is a high level design that determines how the classes in a system are organized.
  - In Java this is often manifested as a set of packages into which related classes are placed.
- ***Model View Controller*** (***MVC***) is an architectural pattern that guides software developers when creating applications that have a user interface.
- Every class in the system is placed into one of three categories.
  - The ***model*** contains the basic building blocks of the application. This includes the core elements of the application as well as any logic needed to maintain or update the state of the application.
  - The ***view*** is the part of the application that the user can see. It typically represents the current state of the model, and is updated as the model changes.
  - The ***controller*** contains the core application logic. When input is provided to the application, the controller determines how to translate this into actions in the model. It also may update the user interface.
- In GUIs some or all of the controller logic is often ***embedded*** in the GUI.
  - For example, the buttons & event handlers that accept user input and make calls on the model.

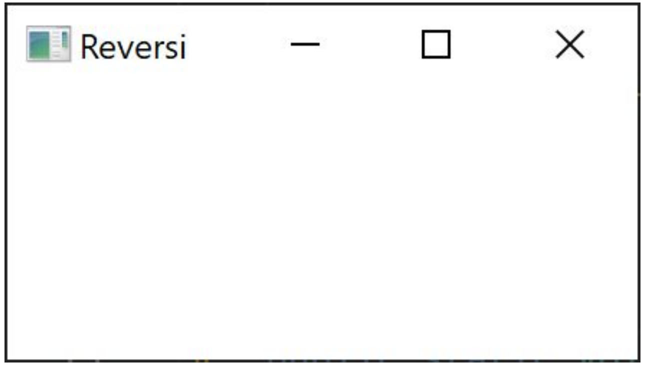
Model View Controller



There are many possible implementations on the ***MVC*** pattern. The variation generally centers on how the three parts of the pattern communicate.

4.18 Bootstrapping

The Reversi model has been provided to you. You will build a GUI on top of that model that allows two people to play games of Reversi! Begin setting up the GUI that will be used as the new view.



Implementing a GUI is often an iterative process during which features are added slowly and tested.

Let's get started by writing the boilerplate code needed to show a JavaFX window.

- Create a new JavaFX application in a class named "`ReversiGUI.java`" in the `unit04.reversi.view`package.
  - Loading the same `Image` over and over again ***wastes memory***. Instead, create a constant `Image` for each of the required images that can be reused:
    - `"file:media/images/reversi/square.png"`
    - `"file:media/images/reversi/blank.png"`
    - `"file:media/images/reversi/blackpiece.png"`
    - `"file:media/images/reversi/whitepiece.png"`
  - Add the following ***private fields***:
    - A `Reversi` game (the model).
    - ***Labels*** for a status message, Player 1 score, and Player 2 score.
  - Use the `start(Stage)` method to set the title and show the stage.
- ***Run*** your application to make sure that everything is working.

Background Images

- We have customized many different JavaFX controls with a solid background color using a `BackgroundFill`.
- It is also possible to set the background to an ***image*** using a `BackgroundImage`, which is created with the following attributes:
  - The `Image` to use for the background.
  - The `BackgroundRepeat` for both the X and Y directions.
  - The `BackgroundPosition`, an enum that may be set to `DEFAULT` or `CENTER`.
  - The `BackgroundSize` which determines the width and height of the image. This can be set to `BackgroundSize.DEFAULT` .
- If the image is smaller than the control, the `BackgroundRepeat` attribute determines whether or not the image is repeated.
  - `BackgroundRepeat.REPEAT` repeats the image as often as needed to fill the area.
  - `BackgroundRepeat.NO_REPEAT` does not repeat at all.

Creating a background that uses an image requires specifying ***five*** (***5!***) different attributes.

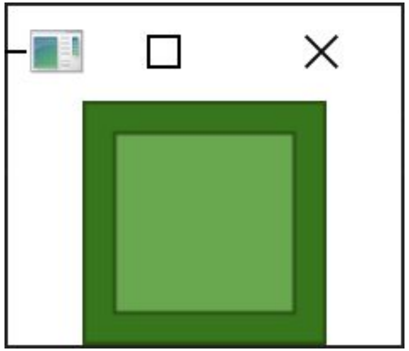
```
control.setBackground(new Background(
    new BackgroundImage(
        new Image("file:background.png"),
        BackgroundRepeat.NO_REPEAT, // X
        BackgroundRepeat.NO_REPEAT, // Y
        BackgroundPosition.CENTER,
        BackgroundSize.DEFAULT)));
```

The example above will create a background image that is ***centered***, ***does not repeat*** and uses the ***default size*** for the image.

If a `Button` is configured with a background image, it will lose its default appearance including border, shading, and beveled edges.

4.19 A Button factory

Every square on the Reversi board will be represented by a Button. That's a *lot* of buttons all using the same graphics and customization options. Sounds like a job for a *factory method*!

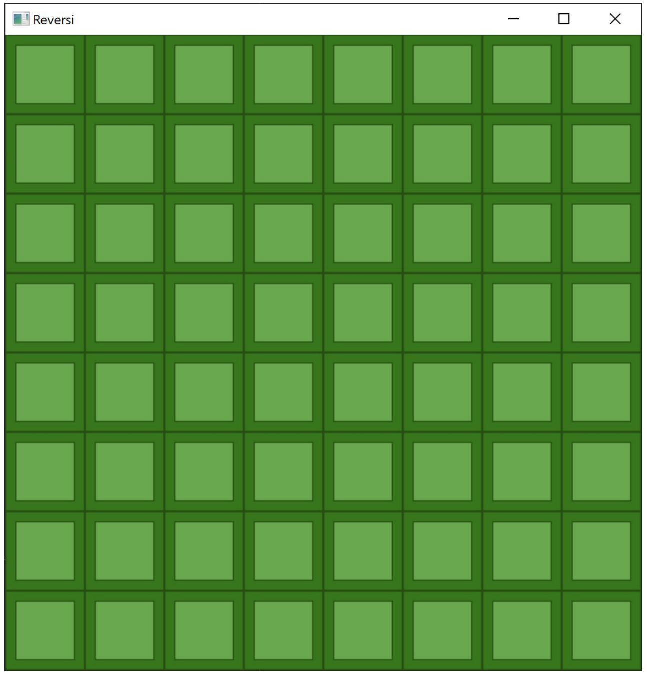


```
control.setBackground(new Background(
    new BackgroundImage(
        new Image("file:background.png"),
        BackgroundRepeat.NO_REPEAT, // X
        BackgroundRepeat.NO_REPEAT, // Y
        BackgroundPosition.CENTER,
        BackgroundSize.DEFAULT)));
```

- Open the `ReversiGUI` application and write a factory method named `makeReversiButton()`. This method will make a `Button` that can be used to represent a single square on the board.
    - For now, do not specify any parameters.
    - Create the new `Button` with an ***empty string*** (`""`).
    - Set the ***background image*** on the button to use the `Image` ***constant*** that you created for the "`square.png`" image.
    - Set the ***padding*** to ***0***.
    - An image with a background but ***no content*** will not size itself correctly. ***For now*** use the `setPrefSize(72, 72)` method to force the button to be 72x72 pixels.
    - Update your `start(Stage)` method to add ***one*** of your buttons to a `Scene`.
- ***Run*** your application.

4.20 Building the Board

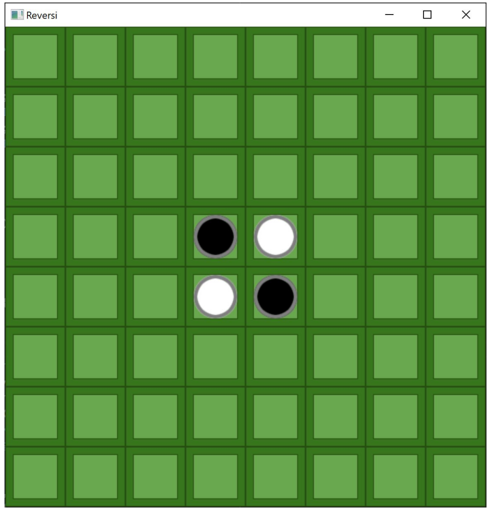
Now that we've got a factory method for creating squares on the board, we'll need to add 64 of them to the user interface. A `GridPane` would be perfect for laying the buttons out in 8 rows and 8 columns!



- Open the `ReversiGUI` application and navigate to the `start(Stage)` method.
    - Create a new `GridPane`.
    - Use the `makeReversiButton()` factory method to add a `Button` to the `GridPane` for every square on the `Reversi` board.
        - Use `add(child, column, row)` to add a button.
        - ***Hint***: Use the `Reversi.COLS` and `Reversi.ROWS` constants.
    - Add your `GridPane` to the `Scene`.
- ***Run*** your application.

4.21 Adding Graphics

We will want to change the graphics on the buttons to display images for white and black pieces are they are played on the board. Update your factory method to add an `ImageView` to each `Button` so that we can change it from blank to white and/or black.

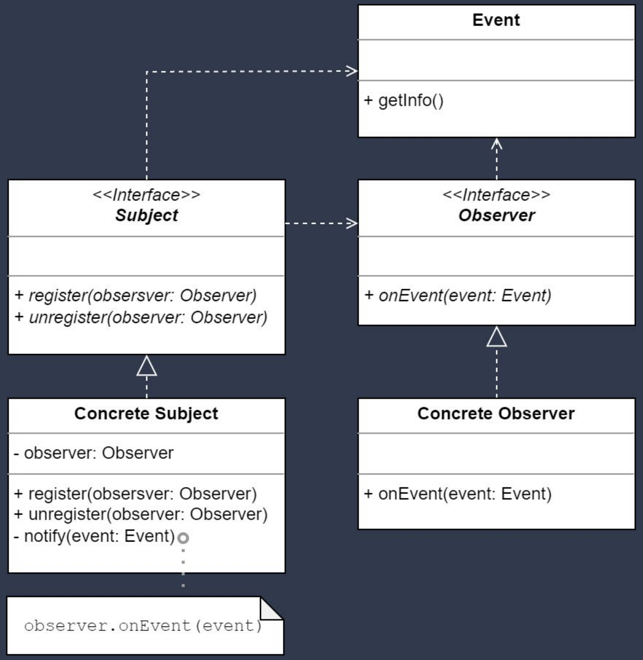


```
ImageView jumpImage = new ImageView("file:jump.png");
Label label = new Label("JUMP!", jumpImage);
label.setContentDisplay(ContentDisplay.BOTTOM);
```

- Open the `ReversiGUI` class and navigate to the `makeReversiButton()` factory method.
  - Use the `getSquare(row, col)` method on the `Reversi` game to get the `Square` that corresponds with the new `Button`'s location on the board. Add any necessary parameter(s) to the method.
  - Create an `ImageView` using the appropriate `Image` ***constant*** to indicate whether the square is currently empty, or occupied by a piece.
  - Set the `ImageView` as the ***graphic*** on the new `Button`.
- Navigate to the `start(Stage)` method.
  - Make sure to ***initialize*** the `Reversi` game.
  - Modify your code to pass the correct parameter(s) into the factory method.
- ***Run*** your application.

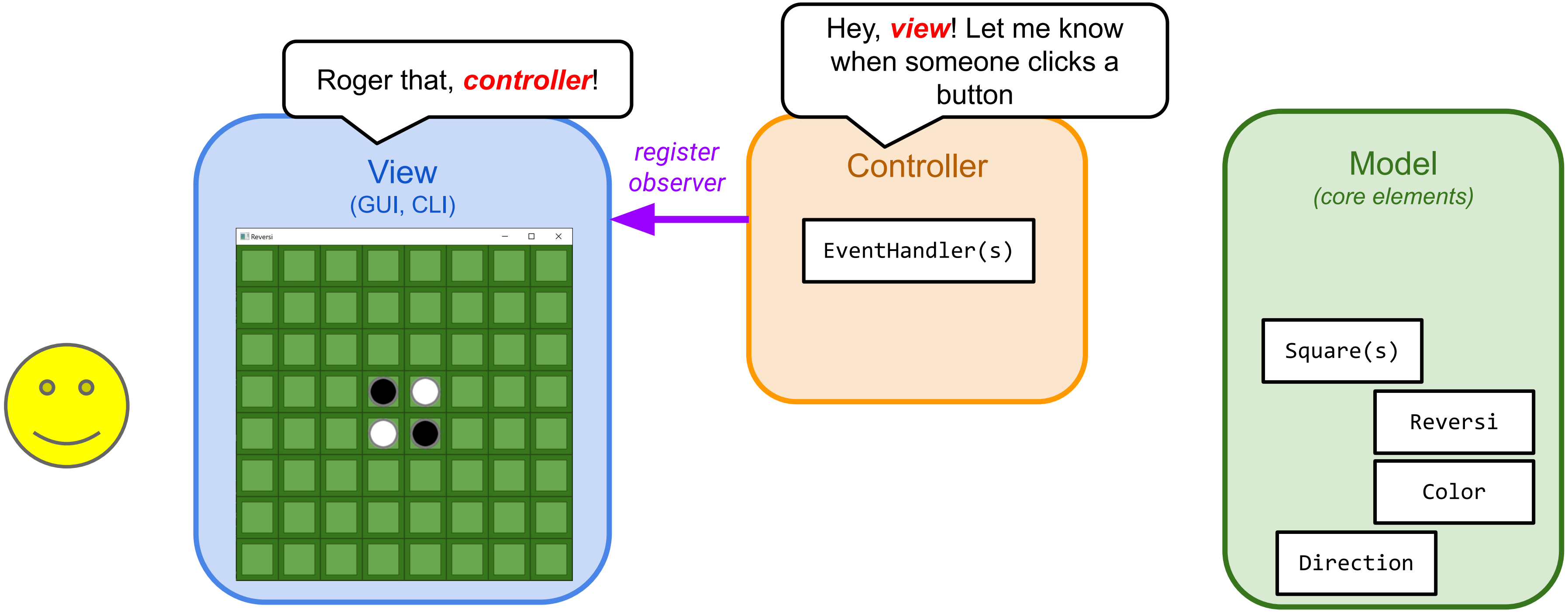
Review: Observer Design Pattern

- ***The Observer Design Pattern*** includes several different ***participants***.
  - The ***subject*** is an object of interest in the system.
  - The ***observer*** is an interface that should be implemented by an object that wants to be notified when something interesting happens to the subject.
  - A ***concrete observer*** is an object that implements the observer interface.
- The subject has a reference to at least one ***registered*** observer.
  - When an event of interest occurs, the subject notifies its registered observers.
- An example of a practical implementation of the Observer Pattern is the JavaFX `Button` control.
  - The `Button` is the subject of interest; other parts of the program might be interested in knowing when the button is pressed.
  - `EventHandler<ActionEvent>` defines the observer interface that must be implemented by any object that wants to be notified when a button is pressed. It is a ***functional interface***.
  - Typically, a small class is written that implements the interface and is registered using the `button.setOnAction(e)` method.
- The practice of writing code that is executed in response to events like button presses is referred to as ***Event-Driven Programming***.

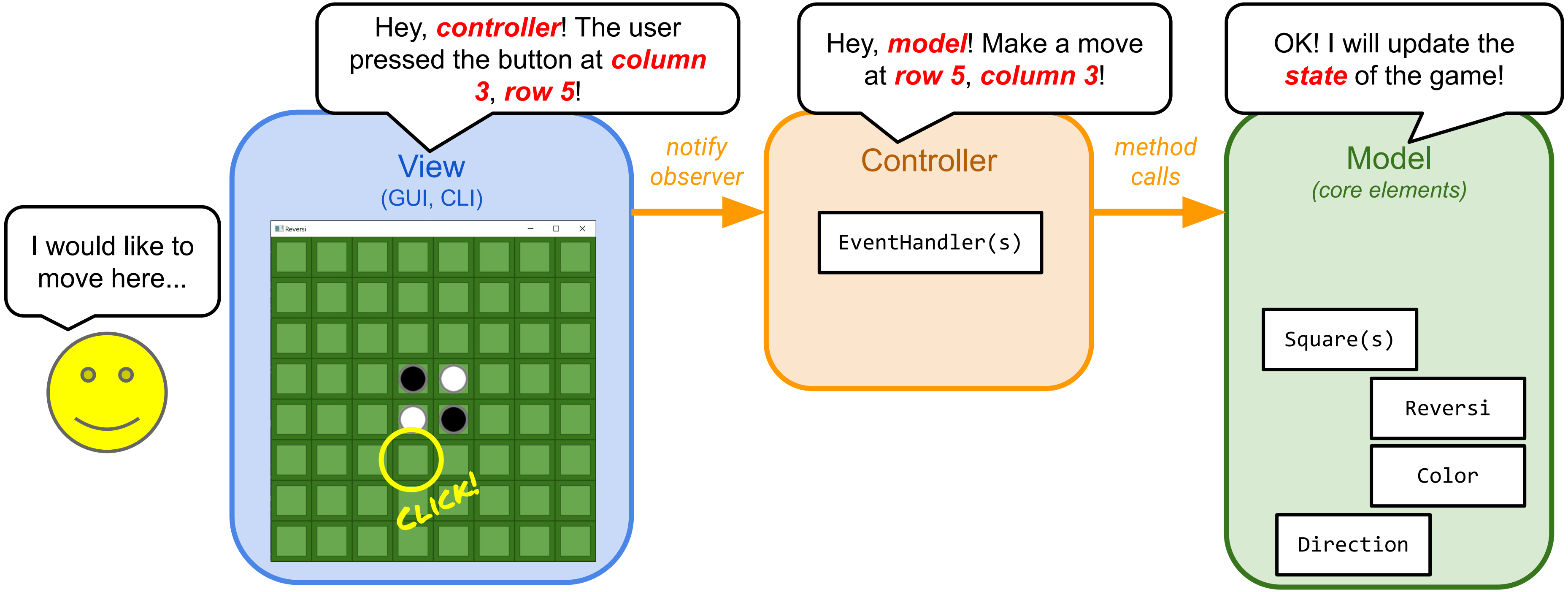


In the ***Observer Pattern***, the ***Subject*** maintains a collection of ***Observers*** that should be ***notified*** whenever something interesting happens.

An Illustrated Example: Registering Observers

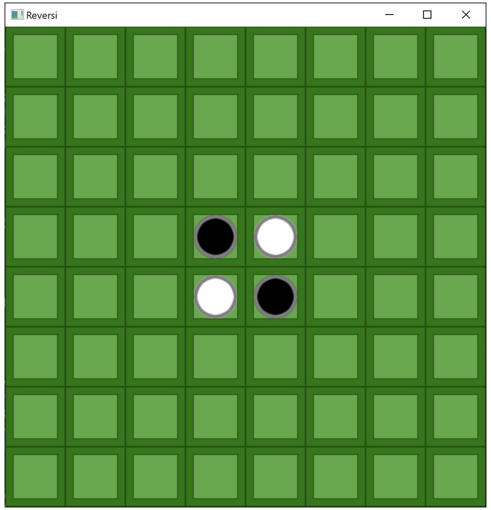


An Illustrated Example: Event Handling


I would like to move here...
Hey, controller! The user pressed the button at column 3, row 5!
View
(GUI, CLI)
Reversi
CLICK!
notify observer
Hey, model! Make a move at row 5, column 3!
Controller
EventHandler(s)
method calls
OK! I will update the state of the game!
Model
(core elements)
Square(s)
Reversi
Color
Direction

4.22 Making Moves

We'll be implementing event handlers that will try to make moves in a Reversi game whenever the user presses one of the square buttons. The first step is to add a method to the `ReversiGUI` that the event handlers can use to try to make the move.

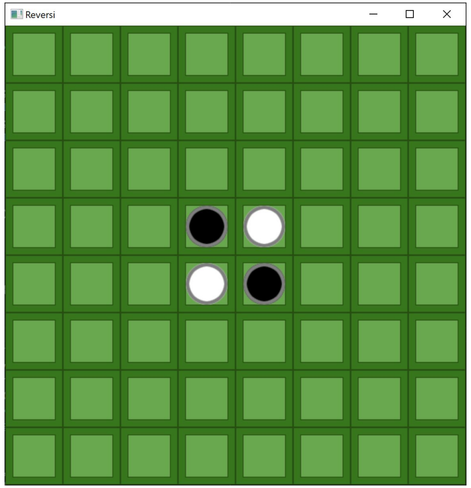


Ultimately we will want to make moves on the board *and* update the board to display pieces as they are played and flipped.

- Open the `ReversiGUI` and define a new method called "`makeMove`" that declares parameters for the `row` and `col` location of the move.
  - Try to make the move at the specified location on the `Reversi` board.
  - If a `ReversiException` occurs, print a message to standard error, e.g. `System.err.println("Bad move!");`
- Test your method by temporarily adding code to the end of your `start` method that makes a move at ***row 5, column 3***.

4.23 Using Buttons to Make Moves

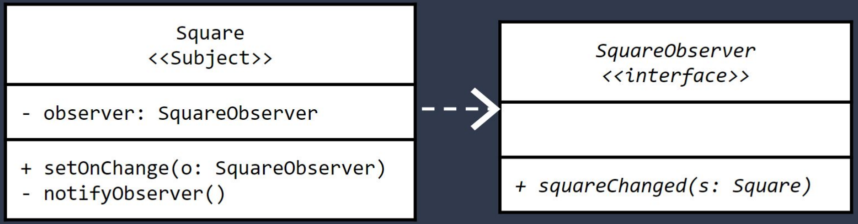
In order to be notified when a button is pressed, we must implement an `EventHandler` for `ActionEvent`s (***observer***) and use `button.setOnAction()` (to ***register*** it with the ***subject***). Let's make a new event handler that will try to move when a square button is pressed.



Ultimately we will want to make moves on the board *and* update the board to display pieces as they are played and flipped.

- Define a new Java class named "`MoveMaker`" and implement the `EventHandler<ActionEvent>`interface.
  - Add fields for a specific `row`, `col`, and the `ReversiGUI`.
  - Add a constructor.
  - You will need to override the `handle(ActionEvent)` method so that it calls `makeMove` on the `ReversiGUI`.
- Modify the code that creates all of your square buttons so that it creates a `MoveMaker` and sets it as the action handler on each button.
  - Each `MoveMaker` should use the same `row` and `col` as the button!
- ***Run*** your application to test out your event handlers.
  - Don't forget to comment out `makeMove(5, 3)` in your `start` method!

Closing the Loop


Square
<<Subject>>
- observer: SquareObserver
+ setOnChange(o: SquareObserver)
- notifyObserver()
SquareObserver
<<interface>>
+ squareChanged(s: Square)

The first step is to create the ***observer interface*** that will be implemented by any object in the system that wishes to be notified when the state of a `Square` changes.

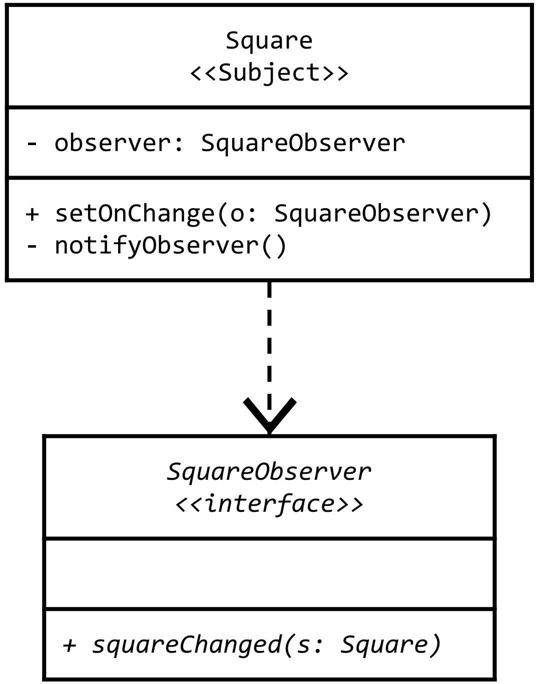
The second step is to update the Square class to maintain a ***collection of registered observers***...

...and to ***notify*** those observers whenever the ***state changes***.

- If all has gone well, then valid moves are now being made on the `Reversi` game board.
- The only problem is that the GUI is ***not updating*** to show that the state of the game has changed.
- Currently, the only way to update the GUI is to ***scan the entire game board*** and manually update the image on every button.
    - Not only is this ***inefficient***, we would need an additional data structure to keep track of all of the buttons.
- A better alternative would be if each ***square*** on the `Reversi` board was the ***subject*** in an Observer Pattern.
    - Each square would keep track of an observer using a field.
    - If the state of the square changed, it would notify its observer.
    - We could then register a ***concrete observer*** for each square that updates the image on the corresponding button whenever the square changes!
- We'll need to modify the `Reversi` ***model*** to support the Observer Pattern.
    - Add an ***observer interface***.
    - Add a ***registered observer*** in each square.
    - The square will ***notify*** its observer when it changes.

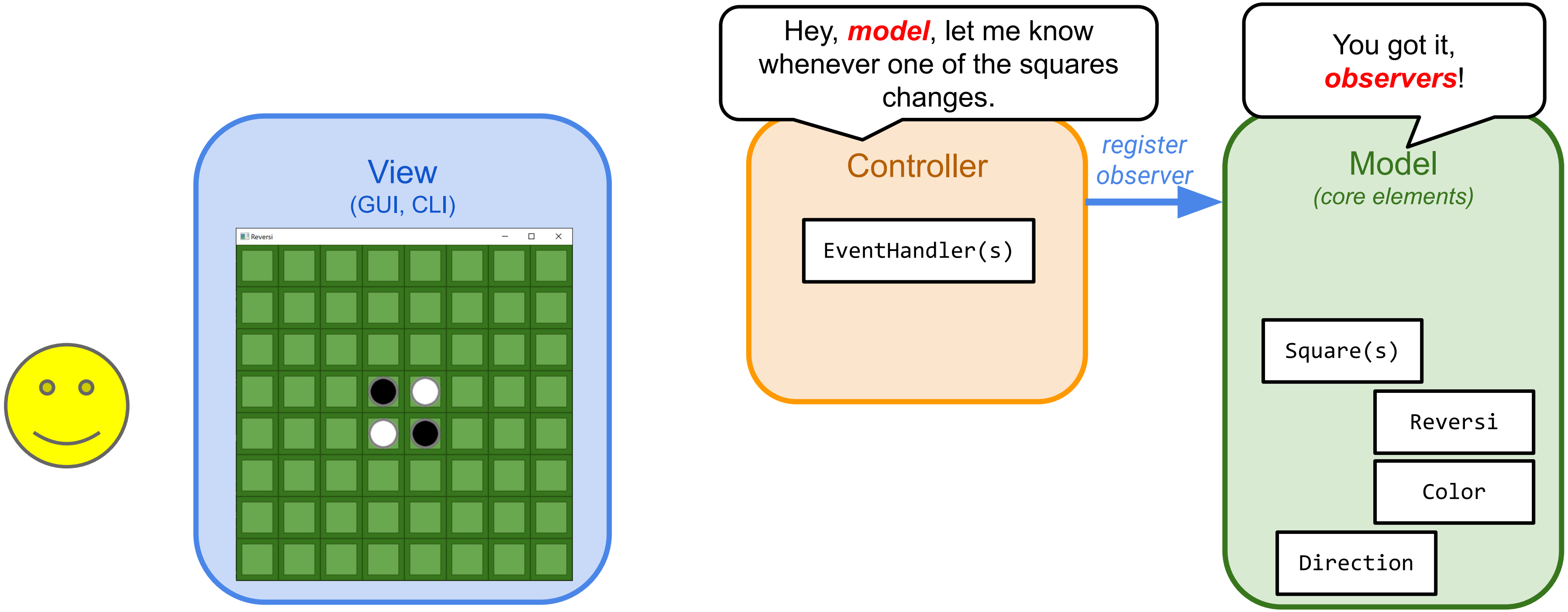
4.24 An Observable Square

We register event handlers (observers) with buttons (subjects) so that we know when they are clicked and can make changes to the model. Now we need to close the loop by observing the model so that we can update the GUI if/when the model changes!

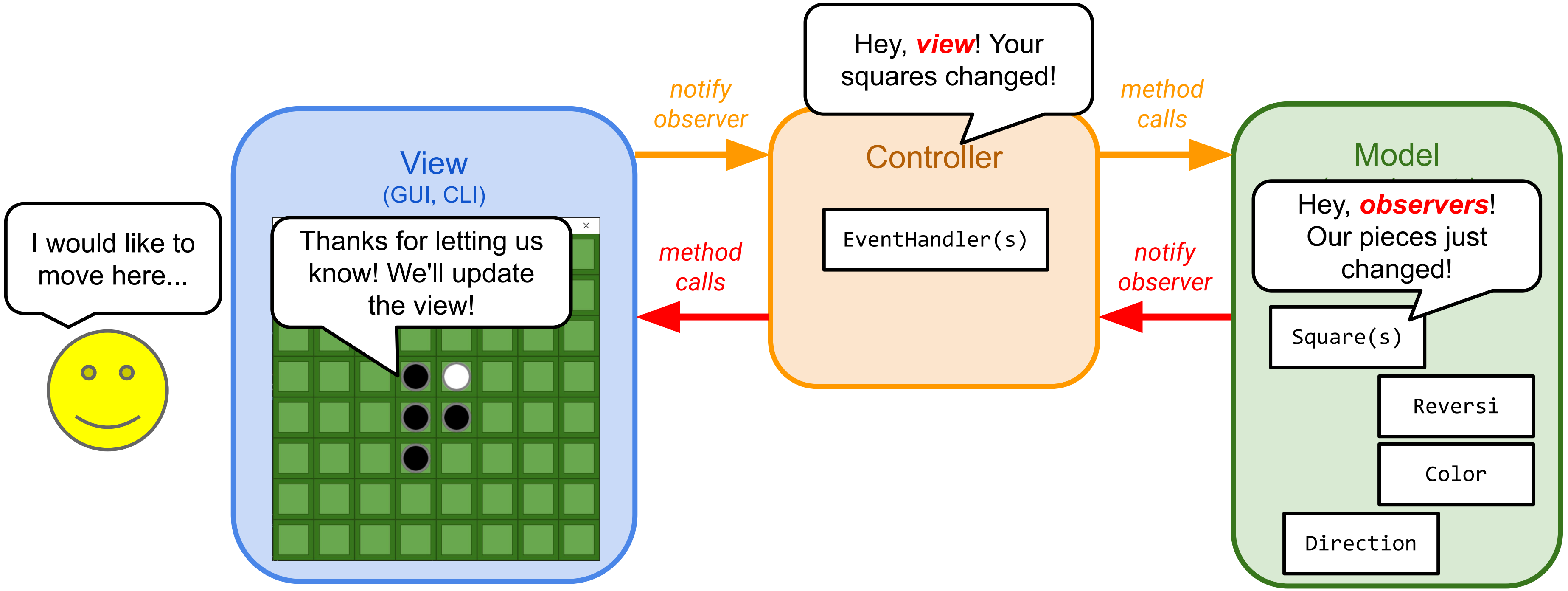

Square
<<Subject>>
- observer: SquareObserver
+ setOnChange(o: SquareObserver)
- notifyObserver()
SquareObserver
<<interface>>
+ squareChanged(s: Square)

- Use the UML to the left as a guide to create a new Java interface in a file named "`SquareObserver.java`" in the `unit04.reversi.model` package.
    - This will be the ***observer interface*** that is implemented by objects that wish to be notified when the state of the square changes, e.g. a new piece is played on the square.
- Open the `Square` class and add the methods and fields indicated in the UML to the left.
    - Scroll through the code and make sure to ***notify registered observers*** whenever the piece on the square changes (including when it is set to `EMPTY`).

An Illustrated Example: Registering Observers

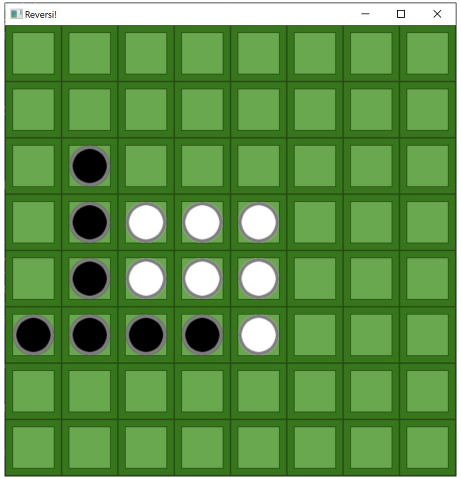


An Illustrated Example: Notifying Observers


I would like to move here...
View
(GUI, CLI)
Thanks for letting us know! We'll update the view!
notify observer
method calls
Hey, view! Your squares changed!
Controller
EventHandler(s)
method calls
notify observer
Model
Hey, observers! Our pieces just changed!
Square(s)
Reversi
Color
Direction

4.25 Implementing a `SquareObserver`

Right now the GUI doesn't update when moves are made on the Reversi board. That's because the GUI doesn't know that the pieces on the board have been changed! The next step is to make a ***concrete observer*** by implementing the `SquareObserver` interface inside the `view` package. When the observer is notified that the board has changed, it will update the GUI.

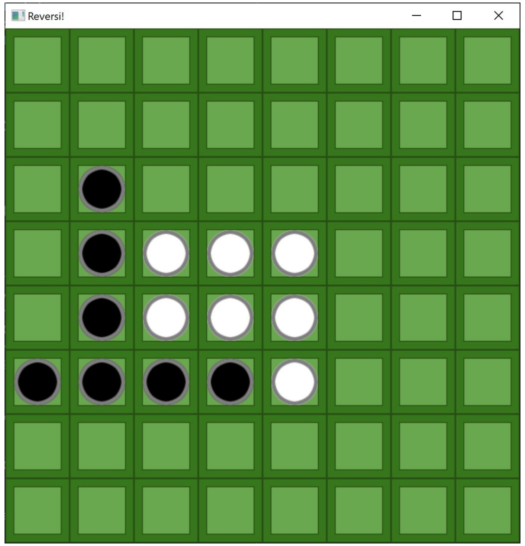


As the squares on the Reversi board are changed in the model, the GUI isn't updating. Let's fix that!

- You will need to make a `SquareChanger` class that implements the `SquareObserver` interface. It will update the image on its button whenever the square that it is observing changes.
  - Declare a field for the `ImageView` that is displayed on the button.
  - Implement an initializing constructor.
  - When the `squareChanged` method is called, change the image on the button to the appropriate image, e.g. if a white piece was just played, change the image to the `"whitepiece.png"` image.

4.26 Updating the GUI

Finish closing the loop by adding a `SquareObserver` to each `Square` on the board that will update the `ImageView` on the corresponding `Button` whenever the square changes.



As the squares on the Reversi board are changed in the model, the GUI isn't updating. Let's fix that!

- Open the `ReversiGUI` application and navigate to the `makeReversiButton()` factory method.
    - Add a `SquareObserver` to the new `Button`'s `Square` that will ***change the image*** on the button's `ImageView` whenever the square changes.
    - ***Hint***: you already wrote code to change the image on the `ImageView`. Maybe you should put that in a method?
- ***Run*** the application and try making a few moves!

Model View Controller (MVC) in a Nutshell

View
(GUI, CLI)

Presents the state of the model to the user.

Provides mechanisms for user input.

Notifies the controller when the user acts.

May observe the model.

Controller

Translates user input into calls on the model.

May observe the view for events indicating user interaction.

May observe the model and update the view.

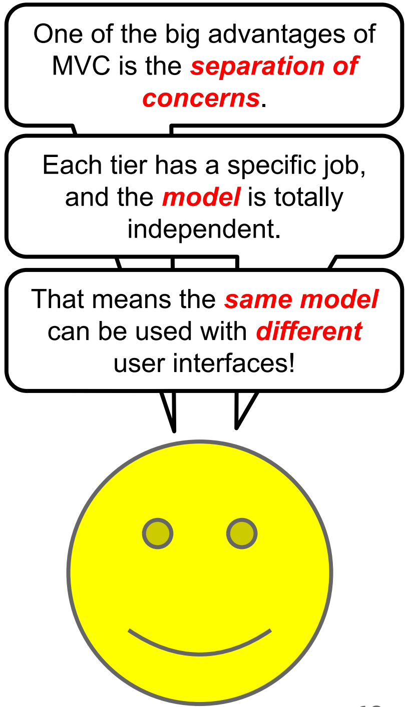
Model
(core elements)

Basic building blocks of the application. Often based on ***real world*** things.

Enforces the rules.

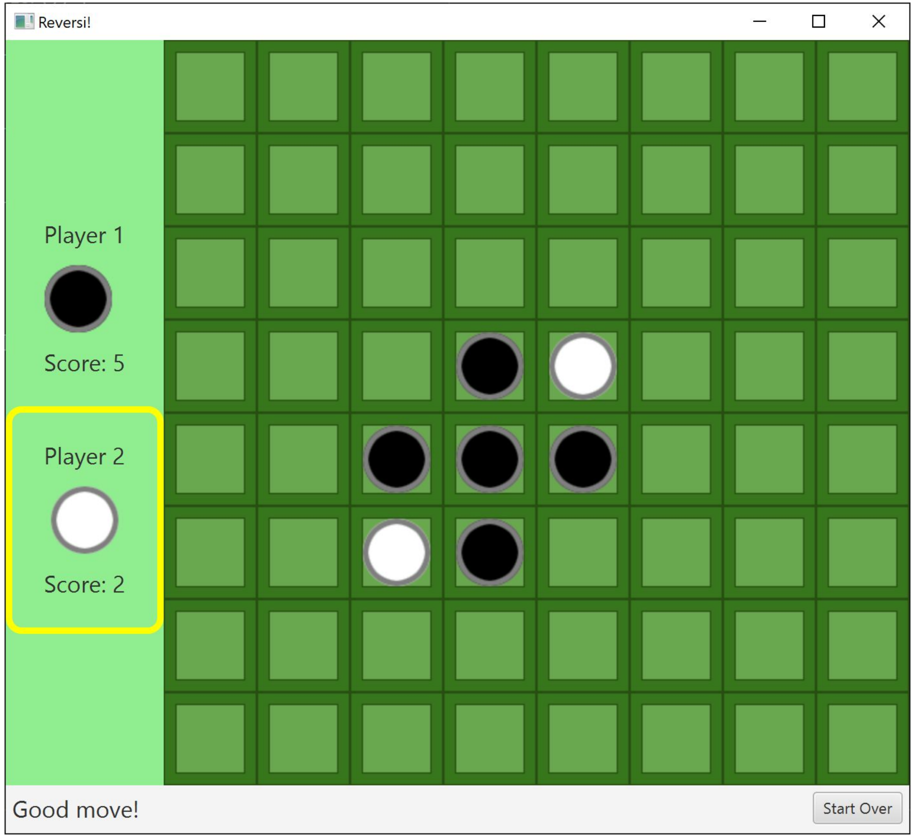
Maintains and updates the state.

Never depends on the controller or the view.


One of the big advantages of MVC is the ***separation of concerns***.
Each tier has a specific job, and the ***model*** is totally independent.
That means the ***same model*** can be used with ***different*** user interfaces!

4.27 Lots more to do!

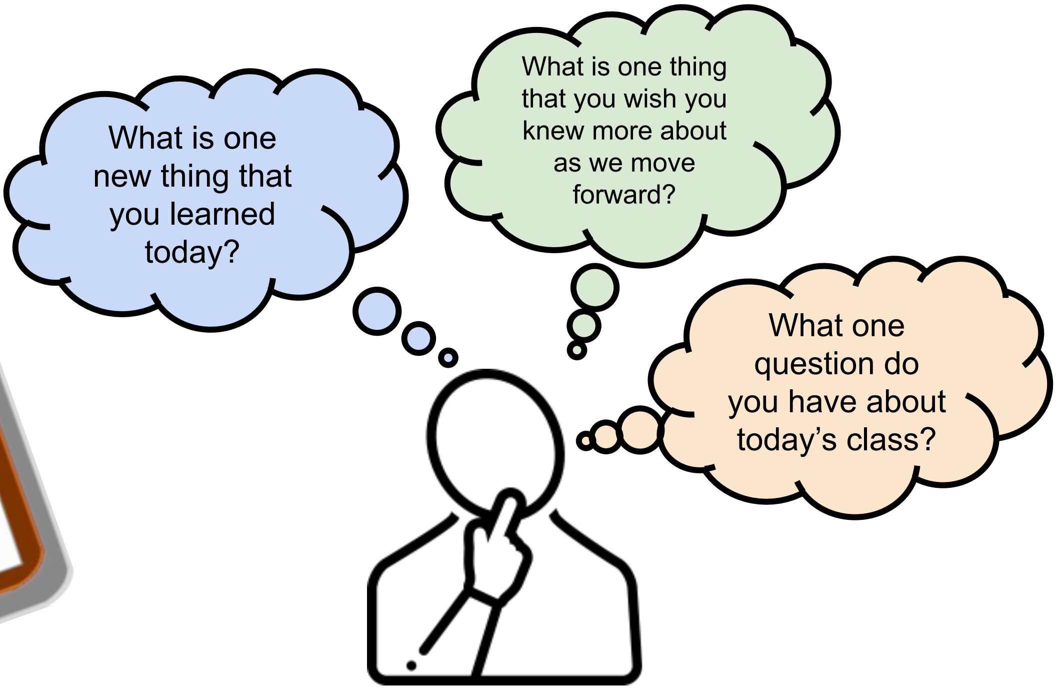
If you have extra time (or just want to), make additional improvements to enhance the usability of the game. You may need to change the model to enable certain features.



Good move!

Summary & Reflection

- JavaFX
  - Controls
  - Layouts
  - Images & Media
- Event-Driven Programming
  - EventHandler
  - ActionEvent
  - setOnAction()
- MVC (Model/View/Controller)
- The Observer Pattern


What is one new thing that you learned today?
What is one thing that you wish you knew more about as we move forward?
What one question do you have about today's class?

Please answer the questions above in your notes for today.