Lecture 6

Project

- By now you should be in a project group
- By the end of this week we will require you to confirm your group membership and you will not be able to change again

https://junit.org/junit5/

Junit: https://junit.org/junit5/

Fibonacci.java

```java
public class Fibonacci {
    public static int compute(int n) {
        int result = 0;

        if (n <= 1) {
                result = n;
        } else {
                result = compute(n - 1) + compute(n - 2);
        }

        return result;
    }
}
```

$T(n) = T(n-1) + T(n-1)$

0,1,1,2,3,5,8

FibonacciTest.java

```java
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class FibonacciTest {
    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
                { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 }
        });
    }

    private int fInput;

    private int fExpected;

    public FibonacciTest(int input, int expected) {
        this.fInput = input;
        this.fExpected = expected;
    }

    @Test
    public void test() {
        assertEquals(fExpected, Fibonacci.compute(fInput));
    }
}
```

T(n) = T(n-1) +T(n-1)

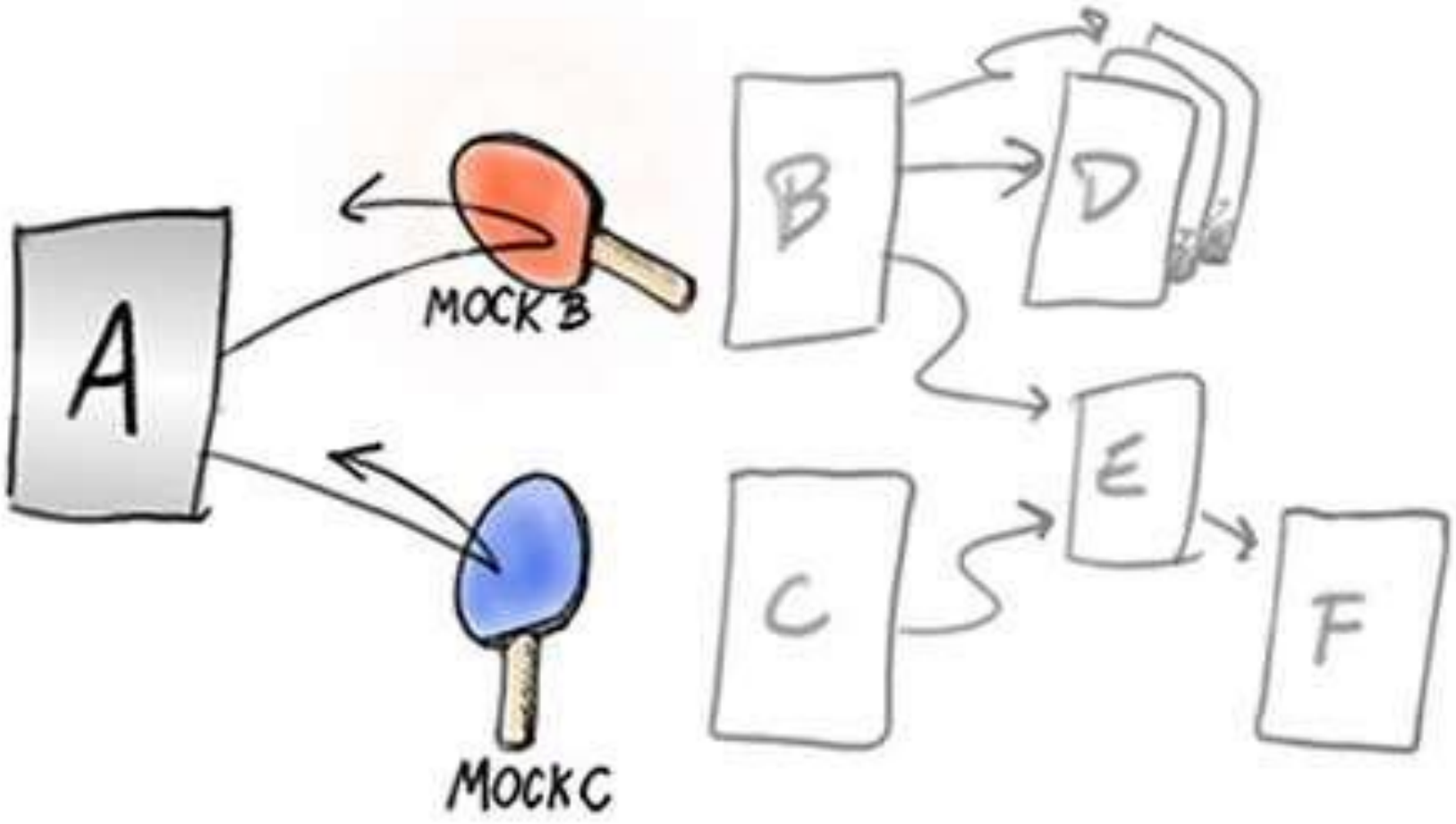
0,1,1,2,3,5,8

Test Driven Development (TDD)

- If proper testing is done **while developing** it saves time in the end
- “Unit testing” is ideally automated
- Unit testing is the most basic level of testing means testing individual units of functionality in software in isolation:
  - A “unit” is a method, a procedure, a section of code, or a class.
- In test driven development programmers code, test, and design closely together (just the detailed design/refactoring for simplicity and reducing redundancy)
  - Coding is progressed just enough to make the next test pass

Other more advanced considerations for unit testing

- Mocking



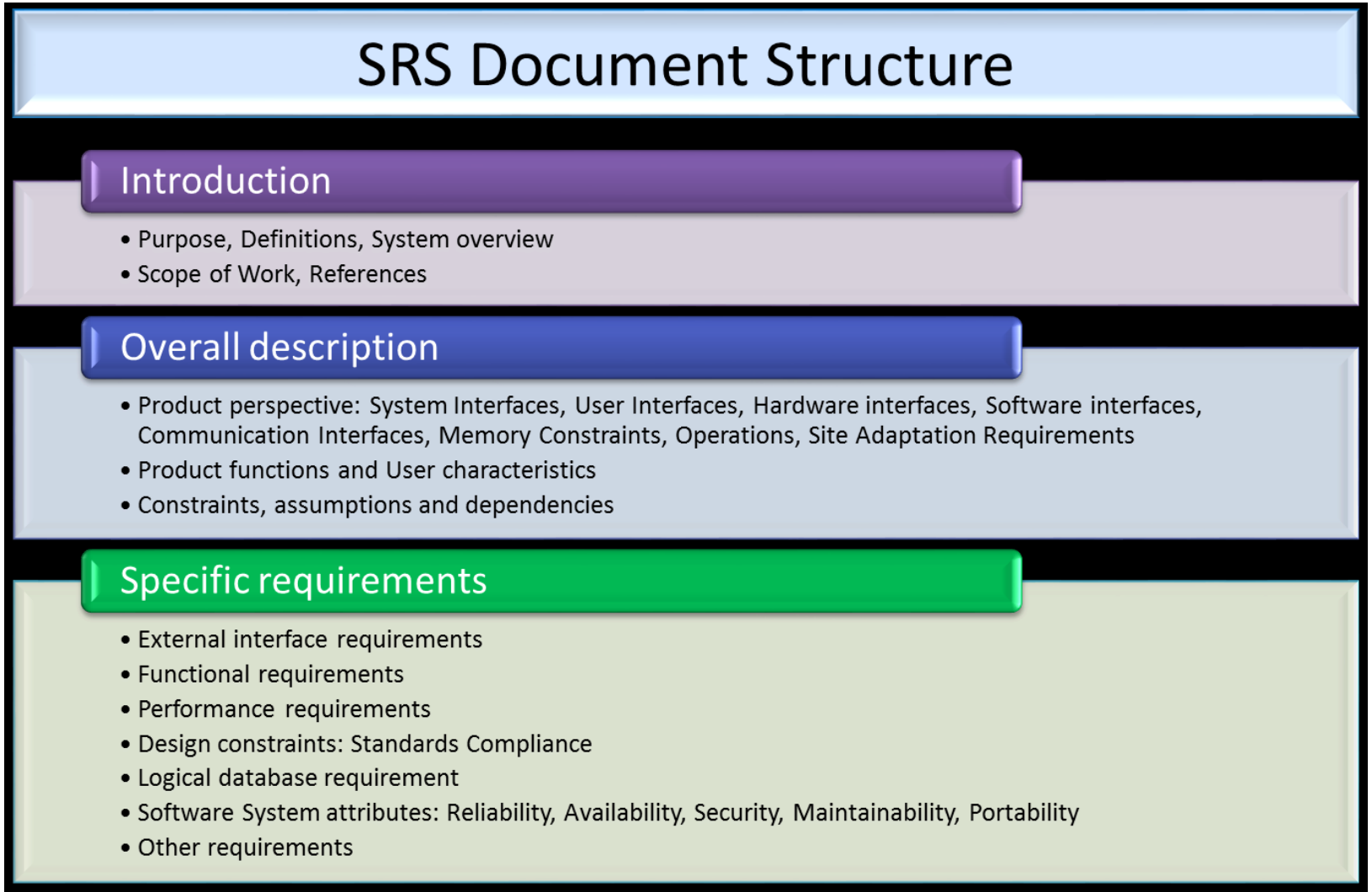
Other types of testing

- Acceptance testing based on requirements by the user

- System testing
- Integration testing
- **Unit testing**

- System, integration and unit testing can and should be largely automated

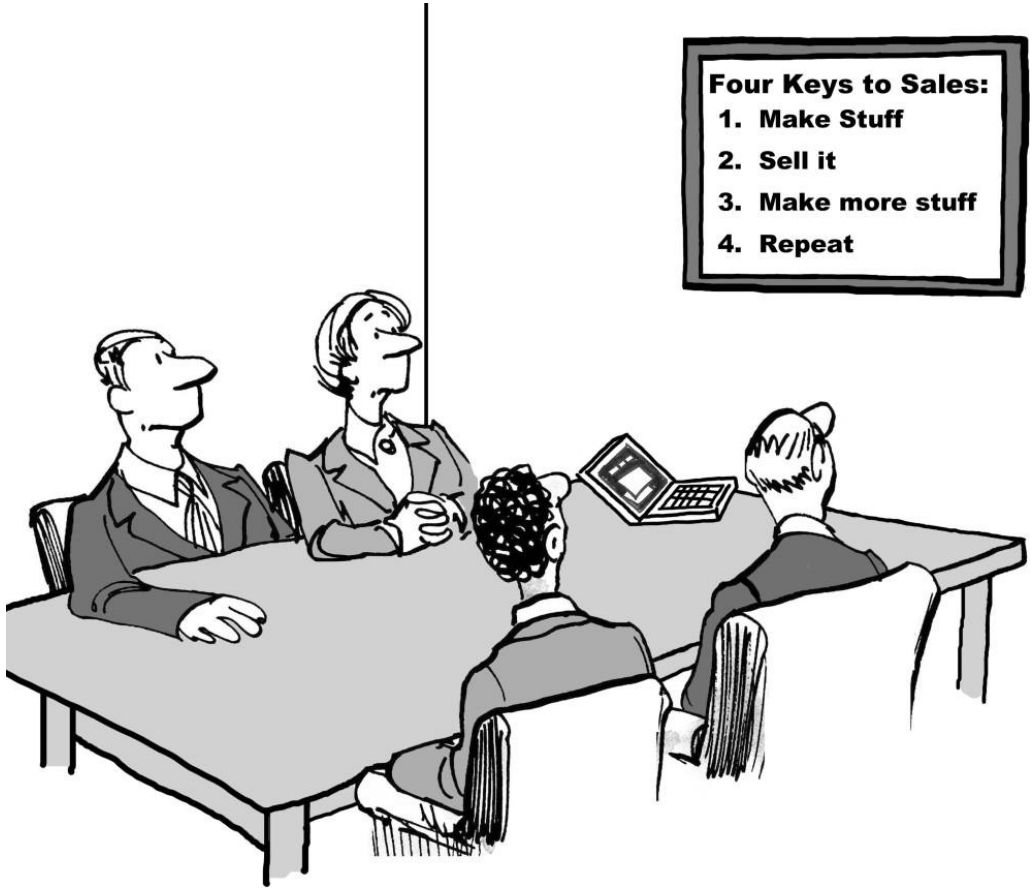
Acceptance testing should be based on formally agreed requirements that are well documented



Requirements

- Sales
- Scoping and feasibility studies
- Software requirements gathering
- Requirements specification
- Requirements validation

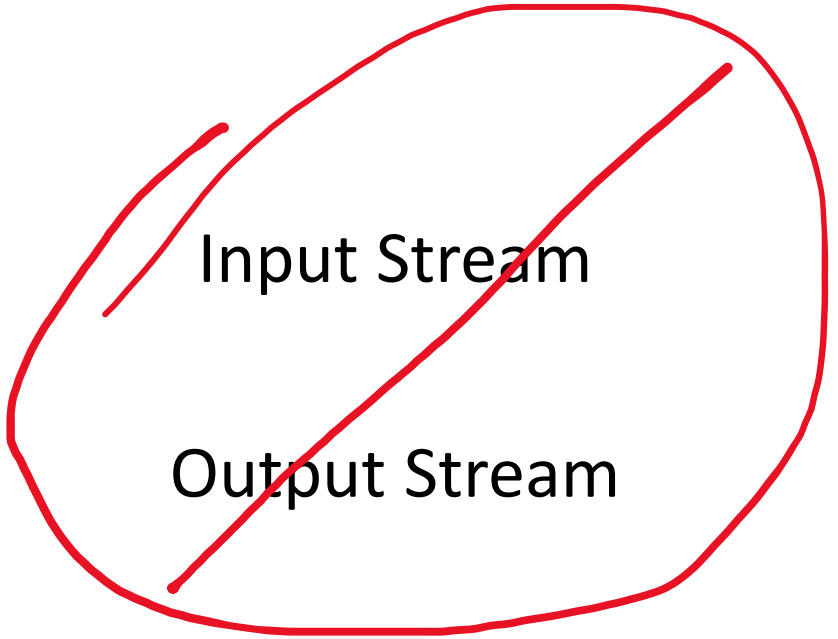
- The requirements should be prioritised and listed in order of importance and urgency
- They should be clarified and negotiated with the relevant stakeholders/customer



Software development processes

Streams

Streams



They are not input streams or output streams

Beware that in spite of the name, it's unrelated to files.
It's about chaining processing.

Streams

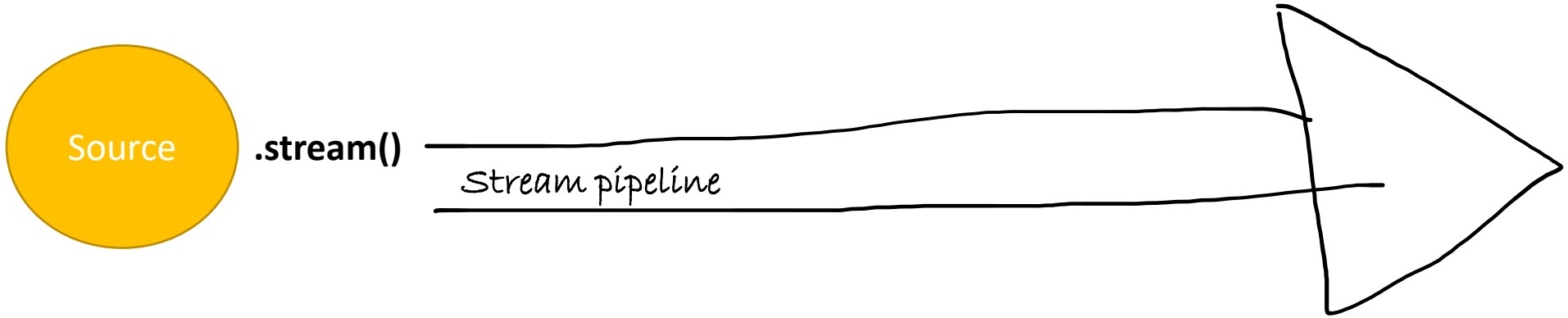
- Can demonstrate the concept with String operations
- When you apply a method to a string you can apply a new method to the result… and so on.

```
String str = “now let’s have some fun”;
Str.toUpperCase().replace(’S’, ‘D’).substring(15, 19).replace(‘M’, ‘N’);
```

```
Str                                                    "now let's have some fun"
        .toUpperCase()                                  "NOW LET'S HAVE SOME FUN"
                .replace(’S’, ‘D’)                      "NOW LET'D HAVE DOME FUN"
                        .substring(15, 19)              "DOME"
                                .replace(‘M’, ‘N’);     "DONE"
```

Streams

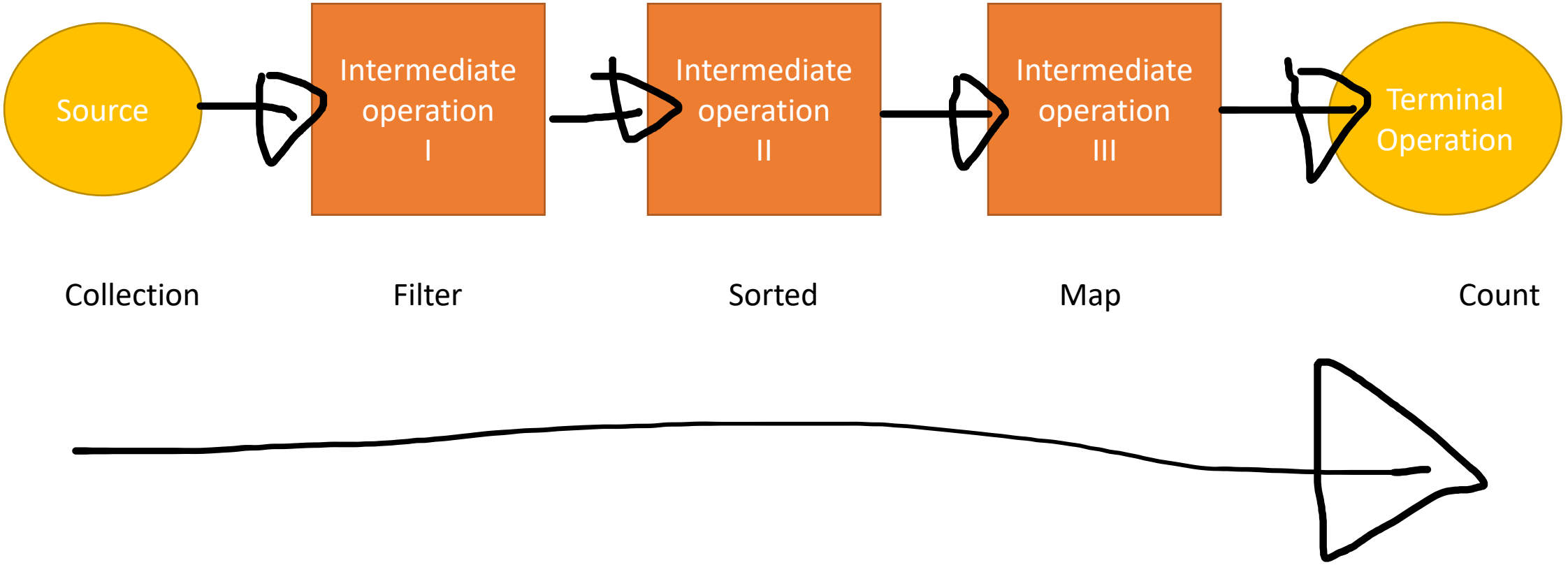
- The same idea may be applied to collections, arrays, functions, IO and others:



- Because serial processing is often applied to collections of data, the idea is to do something similar in Java
- If you know SQL this will not be new as it is also based on similar ideas

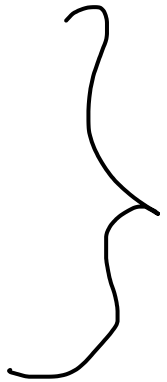
```
SELECT students.name, students.lastname
FROM students
WHERE students.lastname IS NOT NULL
UNION

SELECT students.name, 'N/A'
FROM students
WHERE students.lastname IS NULL;
```

Stream Pipeline



Streams

- Intermediate operations
  - Filter
  - Distinct
  - Sorted
  - Map



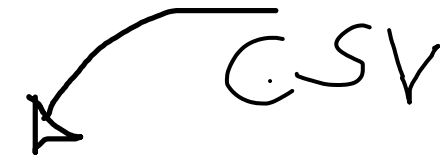
With stream operations you have methods that return a stream (which can be fed into something else), and those that don't and terminate the streaming process.

- Terminal Operations (return a value by combing stream contents)
  - count
  - min/max
  - average
  - reduce (you set your own accumulator function – can use anonymous functions!) **(https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html)**

Another Example

Suppose we have an array of Film objects read from a file:

```
class Film {
        private String      title;
        private String      countries;
        private int         year
        private double      billionRMB

public Film(String title, String countries, int year, double billionRMB){
        …
}
…
public toString() …
}
…
ArrayList<Film> films = new ArrayList<Film>();
```



We can build a collection read from a file, and then the problem is how to search this collection. We can search on many different criteria – film title, year of release, country, how much it made so far.

Retrieve information using different conditions…

Remember predicate – built in functional interface?

```
import java.util.function.Predicate;
static void filter (Predicate<Film> pred){
        Film f;
        ListIterator<Film> iter = films.listIterator();
        while(iter.hasNext()){
                f = iter.next();
                if (pred.test(f)){
                        System.out.println(f);
                }
        }
}
```

The predicate functional interface has a single method called test

Remember predicate – built in functional interface?

- Using this method we can very concisely apply different filters for country, year, revenue, etc:

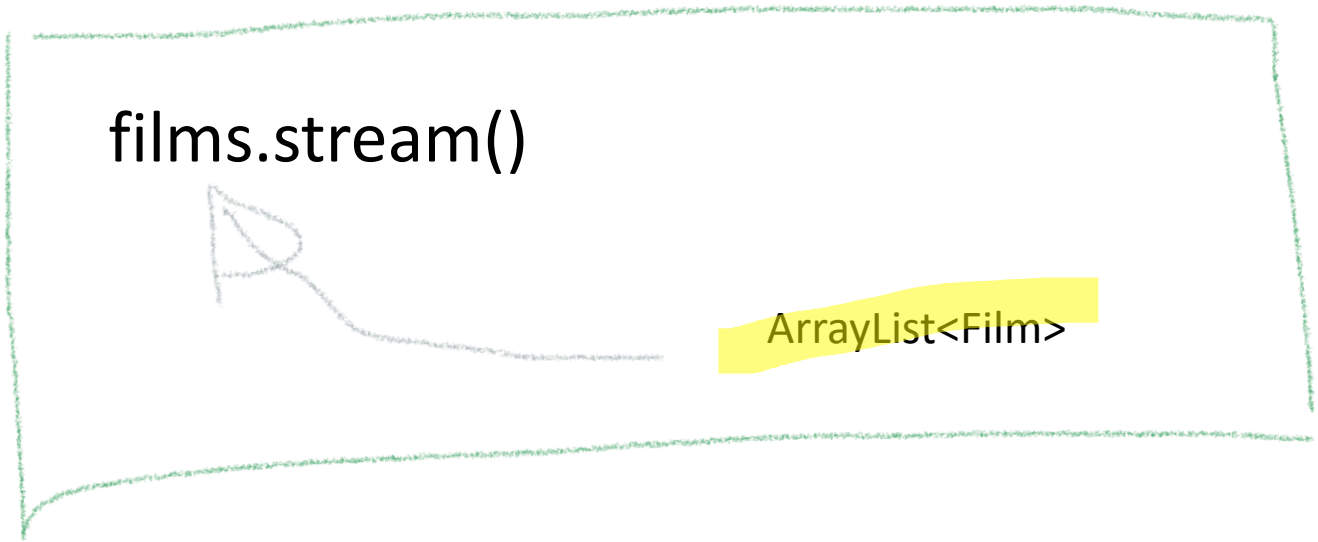
```
filter((film)->film.getYear() == 2014);
```

- To extract the films from a collection meeting the condition that the year is 2014

```
import java.util.function.Predicate;
static void filter (Predicate<Film> pred){
        Film f;
        ListIterator<Film> iter = films.listIterator();
        while(iter.hasNext()){
                f = iter.next();
                if (pred.test(f)) {
                        System.out.println(f);
                }
        }
}
```

Streams

- First we take the collection and turn it into a stream:



- Then apply serial operations ending in a termination.

```
films.stream()
        .filter((film)->film.getYear() == 2014)
        .forEach(System.out::println);
```

We can display any film that "gets through" with a forEach() call (a terminal operation) that applies println() to it.

Note the use of the double colon notation that is used here to specify a method applied to each element.

You can insert intermediate operations before the terminal one, for instance sort the output, if of course Java knows how to sort Film objects.

Note that it's usually FAR more efficient to sort AFTER filtering rather than BEFORE filtering because the set to sort is smaller, even if both are possible ...

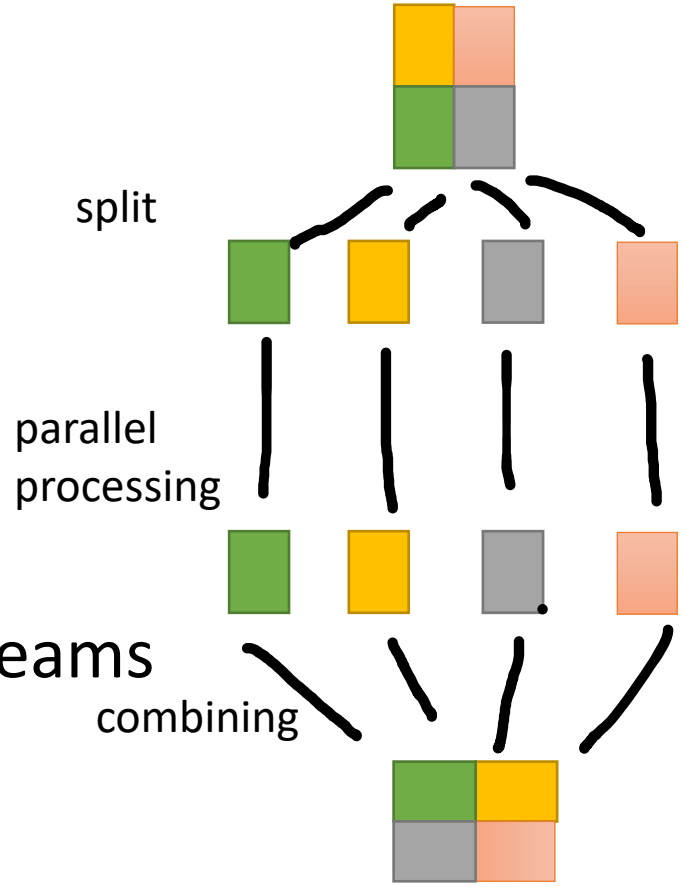
```
films.stream()
        .filter((film)->film.getYear() == 2014)
        .sorted()
        .forEach(System.out::println);
```

NOTE: needs to implement Comparable

Parallel Streams

.parallelStream()

Like a river that reaches the sea with a delta, streams can be split into multiple parallel streams for faster processing. This is important in applications of big data.



https://docs.oracle.com/javase/tutorial/collections/streams/index.html

further examples and discussions of streams can be found in the docs here

Anonymous Classes for Event Handlers

Nested Classes

```
class OuterClass {
        …
        class NestedClass {
            …
        }
}

class OuterClass {
        …
        public void doSomething() {
                class LocalClass {
                    …
                }
        }
}
```

Nested classes are classes declared inside other classes….

In Java you can also have local classes which are classes declared inside a method.

Why?

- Encapsulation
- Grouping related functionality together

- These nested or local objects are not needed outside the context in which they are declared (within another class or in a method)

It is useful for code clarity, but it also means that the objects are kind of "technical objects" that aren't expected to be used anywhere else and you won't have a reference to them generally available elsewhere.

Anonymous Classes

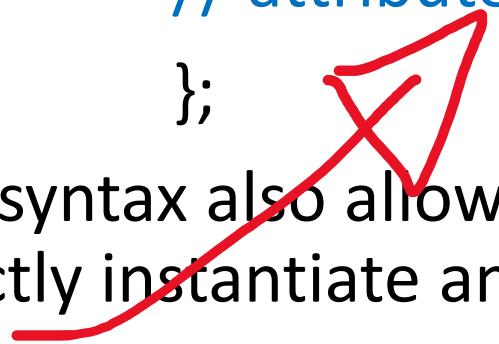
```
Class NamedClass implements Interface{
    …
}

Interface anObject = new NamedClass(…)
```

We saw a similar case with anonymous classes where some functionality requires an interface. For example an object that implements the “comparable” interface is often not really needed by a caller or other program parts generally – all that is needed here often is to provide a sorting utility method with a comparator. Other interfaces may have additional attributes and methods that are also not needed by the caller so would be unnecessary work to implement the object completely.

Of course one can however define everything and create an object however…

Anonymous Classes

```
Interface anObject = new Interface() {
                        // attribute and method implementation
                        };
```



And the Java syntax also allows creating an interface on the fly, and you can also directly instantiate an object by implementing the interface methods here…

This is very convenient for passing as a method parameter

Anonymous Classes

Also works for inheritance

```
class NamedClass extends ParentClass{
   …
}
NamedClass anObject = new NamedClass(…);
```

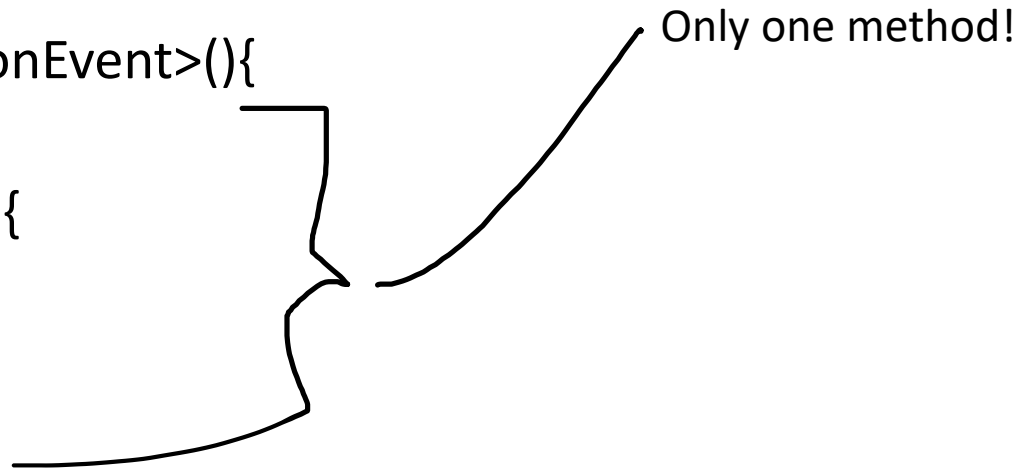
or

```
ParentClass anObject = new ParentClass(){
                        // attribute and method defitions
                                    };
```

GUI: Event Handlers

This is common for graphical user interface programming for instance event listers. Here is an example of listening for a button press…

```
…
Button but = new Button();
but.setText(”Say ‘Hello’”);
But.setOnAction(new EventHandler<ActionEvent>(){
        @Override
        public void handle(ActionEvent e){
                System.out.println(“Hi”);
        }
});
…
```



Only one method!

- If there is only one method then we can use Lambda expressions instead

In the very common case where your interface requires a single method, you can use lambda expressions. Lambda expressions come from functional programming, where you try not to store any state (which is completely opposed to attributes that store the state of an object ...)

- Cool!

Lambda Expressions

- Lambda expression

(parameter list) -> {statements}

- Recall that Lambda expressions are assigned to functional interfaces

```
@FunctionInterface
```

Functional interfaces have only one abstract method

- If there is only one method to define for an interface you don't have to give it a name

Using Lambda Expression in Event Listener

```
…
Button but = new Button();
but.setText(”Say ‘Hello’”);
But.setOnAction(new EventHandler<ActionEvent>(){
        @Override
        public void handle(ActionEvent e){
                System.out.println(“Hi”);
        }
});
…
```

Anonymous Class

Using Lambda Expression in Event Listener

Using Lambda Expression in Event Listener

```
…
Button but = new Button();
but.setText("Say 'Hello'");
But.setOnAction(new EventHandler<ActionEvent>(){
        @Override
        public void handle(ActionEvent e){
                System.out.println("Hi");
        }
});
…
```

Anonymous Class

As "handle()" is the only method of an event handler, it can also be written like this:

```
Button btn = new Button();
btn.setText("Say 'Hi'");
btn.setOnAction((e)->{
      System.out.println("Hi!"); });
```

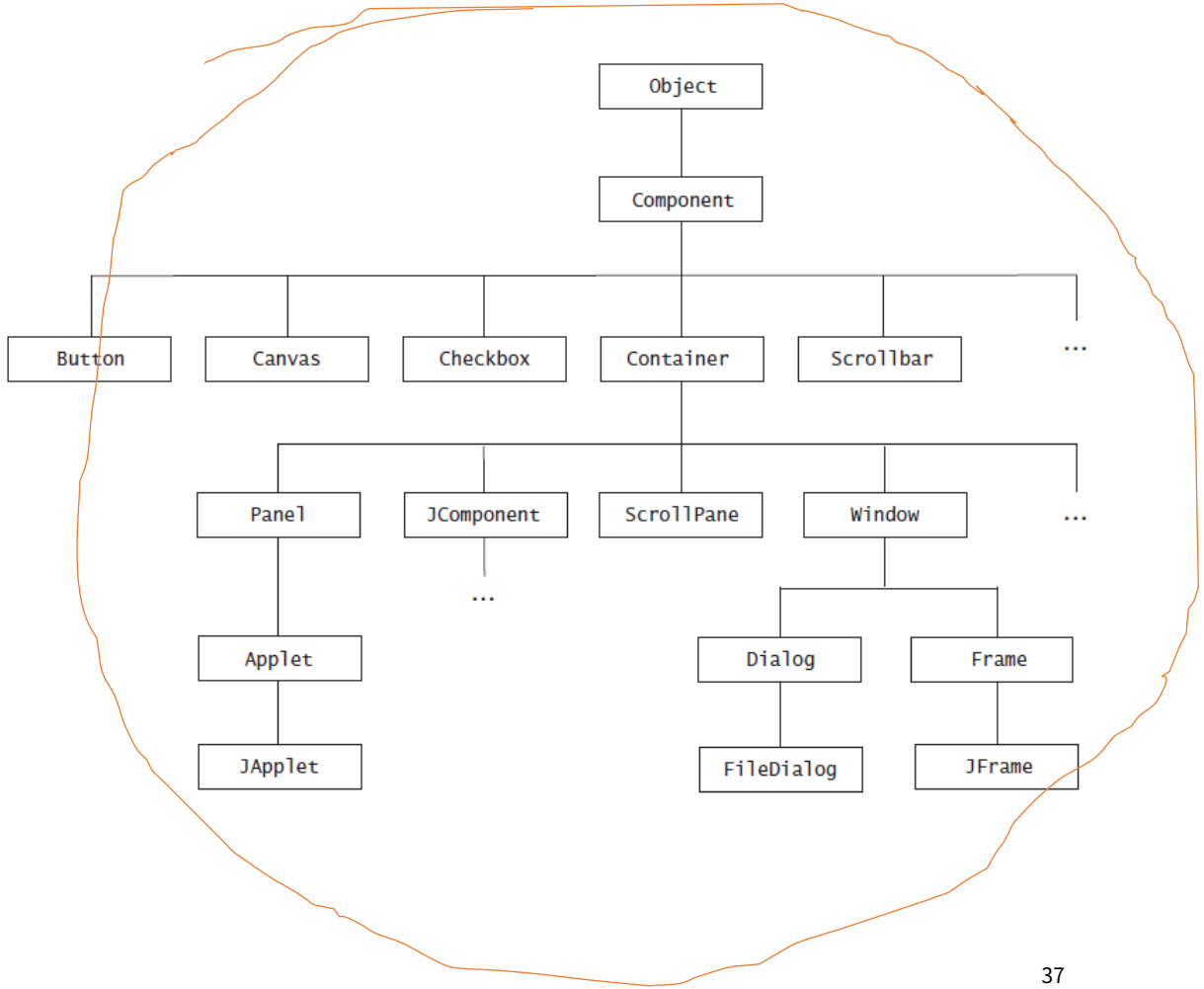
Much shorter

Other common uses are in searching and sorting collections where if you recall the comparable interface has only one method.


Graphical User Interfaces I

Event Based Programming and UI

- Event based programming is often used in GUIs
- In Java an interface is used to specify what methods are called when a user does something (e.g. mouse click). This is a “callback”: a call from a method in one class to another class through an interface
- The code for different UI components do callbacks via inheritance, if desired some methods can be overidden

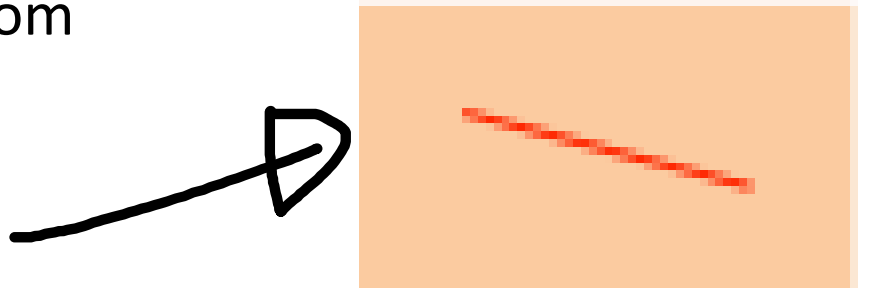


Graphical User Interface Programming

- Event driven programming is a different paradigm for creating a sequence of instructions and possible interactions with a user
- Event handlers / ”callbacks” are functions associated with certain events
- Different from other types of programming you are familiar with
- You rely on packages which you must import and use in your own program (there are many options)
- Need to interact closely with an environment (operating system, windows, display, etc)

Many Different Packages

- First of all you don't code everything by yourself, and instead use functions from packages that you must import when writing your program.
- There are **low-leve**l packages with functions (called "primitives") for performing tasks such as drawing a rectangle, a line or a curve or 3d rendering.
- You also have **high-level** packages that use the previous ones to draw for instance buttons, and automatically change them when they are clicked.



Historical Overview of Java GUI Packages

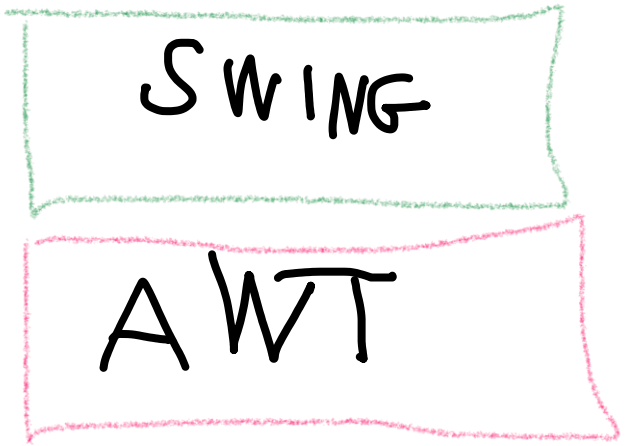
- 1995 Abstract Windows Toolkit (‘**AWT**’)
  Looks like other applications on the system

- Dec 1996 Java Foundation Classes (‘**Swing**’)
  Looks the same on all systems

In Java, several packages allow you to code a GUI. The first one was AWT, followed by "Java Foundation Classes" quiclky renamed "Swing".

Historical Overview of Java GUI Packages

```
import java.awt.*;
import javax.swing.*;
import javax.imageio.*;
```

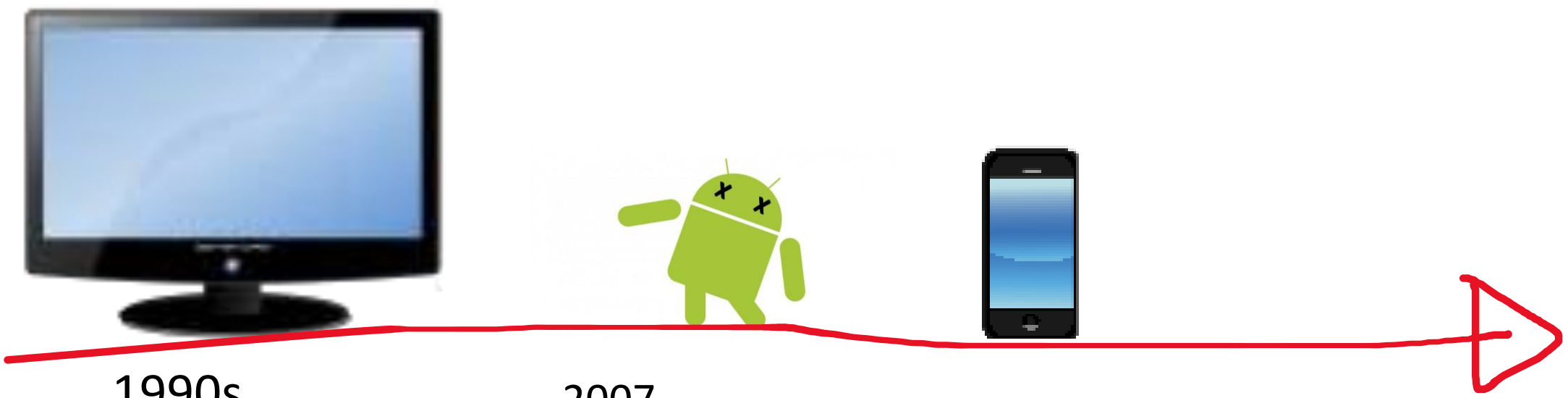


Swing relies on AWT, and whenever you code a Swing application you also need to import classes from AWT, as well as from other packages for images.

Historical Overview of Java GUI Packages

- A new package, JavaFX, was introduced in 2008.

2008 JavaFX `import javafx.*;`



1990s

2007
Android officially adopted by many Smartphones

JavaFX

JavaFX, with which you import classes from a single package (but many subpackages) supports other devices than computer screens for which AWT and Swing were written – mobile phones in particular. It also allows to define the looks of applications in external files called "style sheets" or "CSS" files (CSS means "Cascading Style Sheet" – 'cascade' is French for 'Waterfall'), a technique borrowed from web programming. However, because software has a long life, there is a lot of Swing around, Swing is still much in use and will probably stay around for quite a while. It's good to know both Swing and JavaFX (they aren't VERY different, class names change, basic ideas are the same).

Swing and AWT are replaced by the JavaFX platform for developing rich GUI applications.

The Basic Structure of a JavaFX Program

The `javafx.application.Application` class defines the essential framework for writing JavaFX programs.

Historical Overview of Java GUI Packages

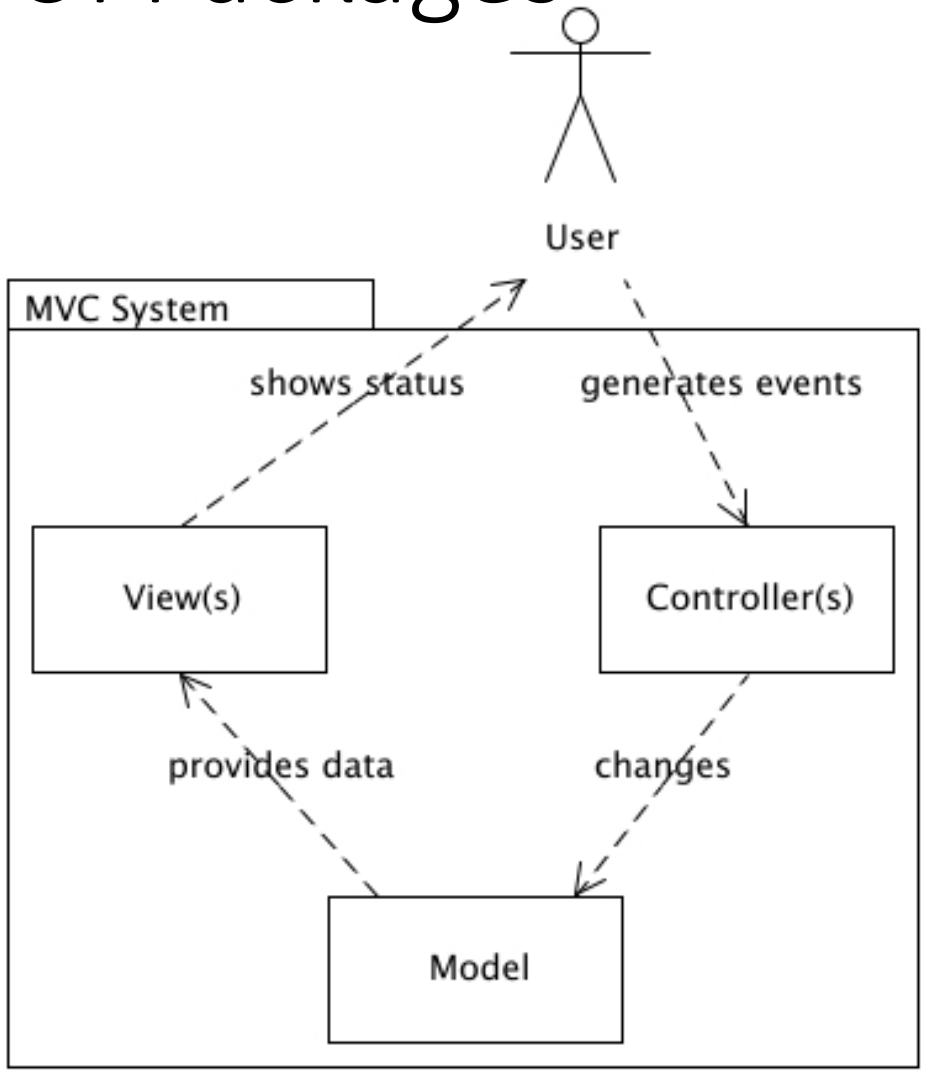
2008 JavaFX `import javafx.*;`

Model = data management

view = user interface (visual elements/looks)

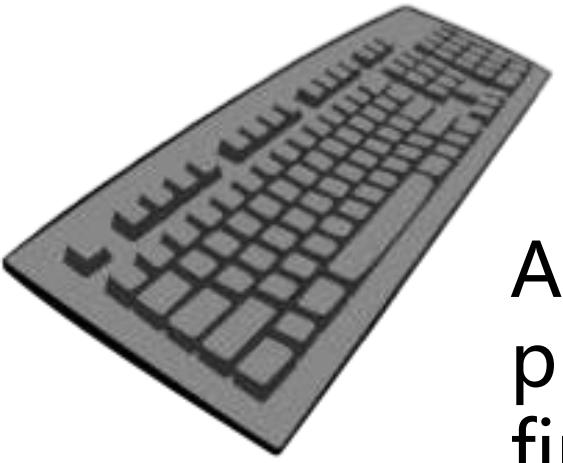
Controller = Logic

JavaFx applications often follow a popular structure known as "Model/View/Controller" (or MVC) in which data management, user interface and logic are clearly separated.

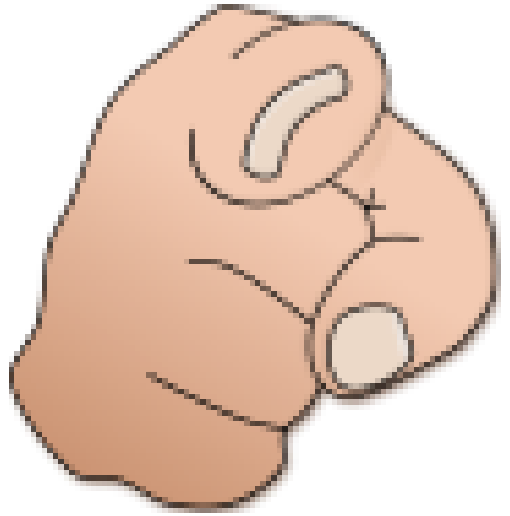


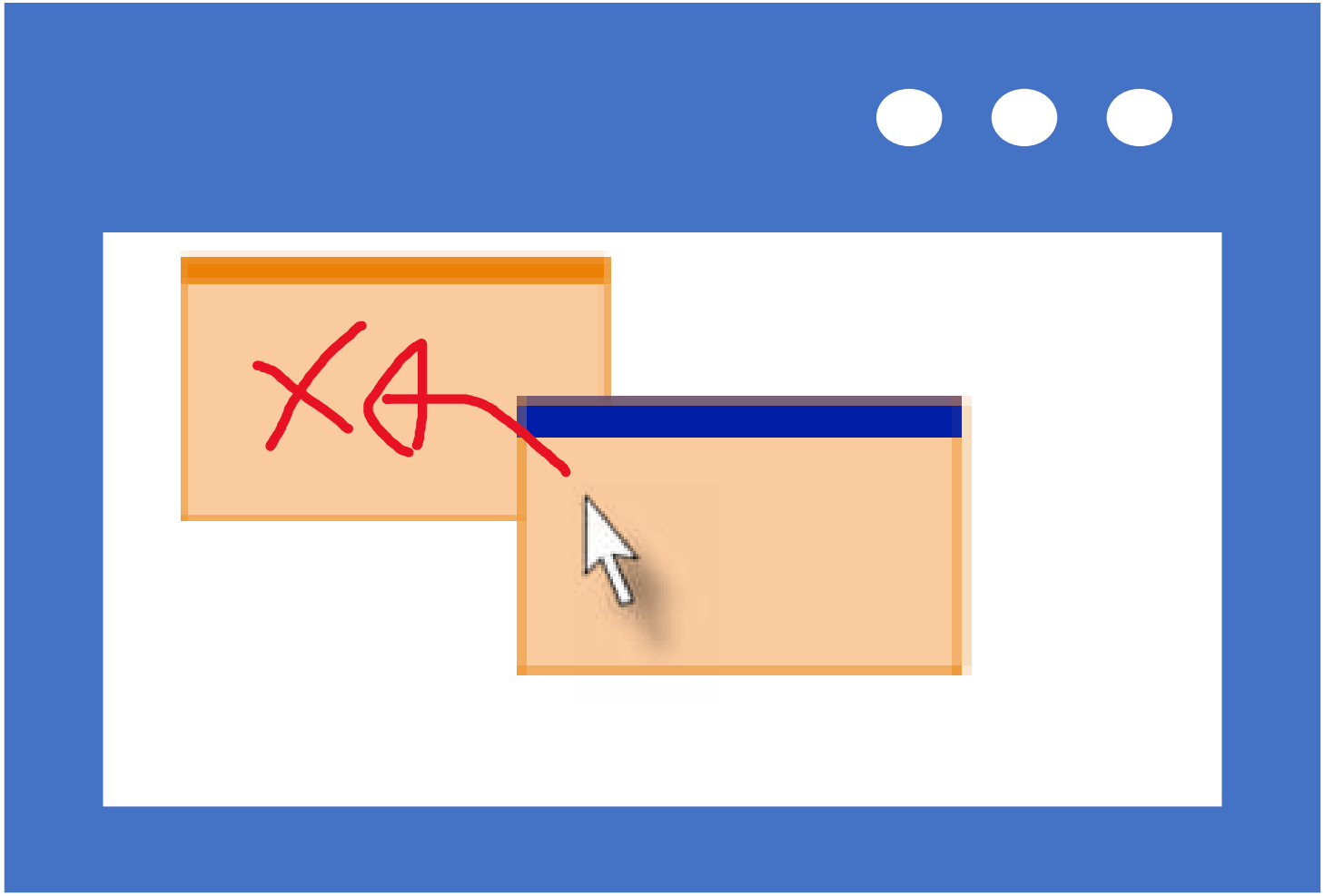
Event Driven Progamming

A Graphical Application is just a big loop



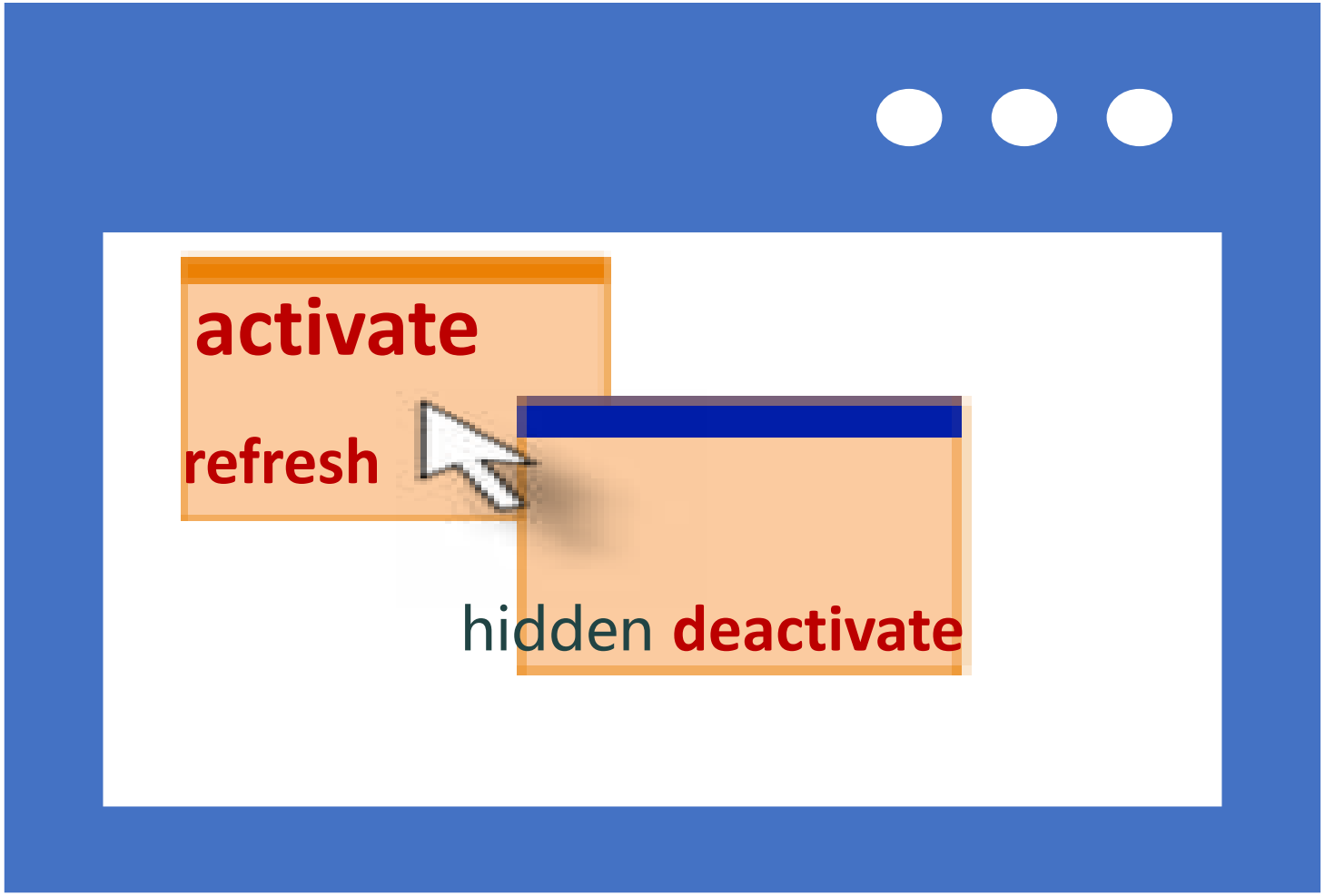
An event can be anything. A key pressed, a move of the mouse, a finger swiping a touch screen, somebody jumping in front of a webcam ... Anything that can be translated into an electrical signal reaching the computer.





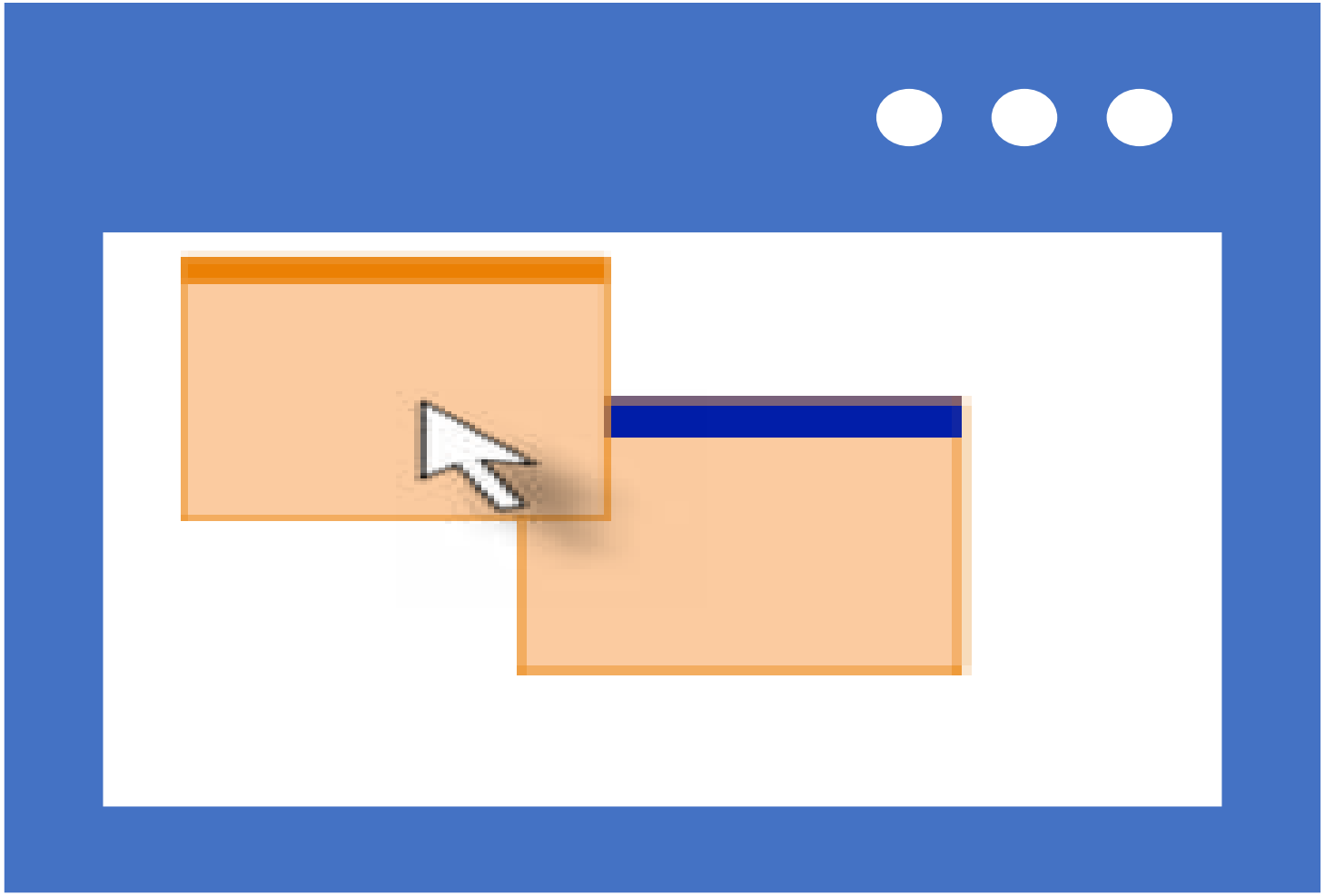
The user may move the mouse and click it outside the active window. The windows management system will get the coordinates and discover that another window is at this location.

This is an event



This means the inactive window has to be brought forwards. Note: there is a stack of objects to represent this internally.

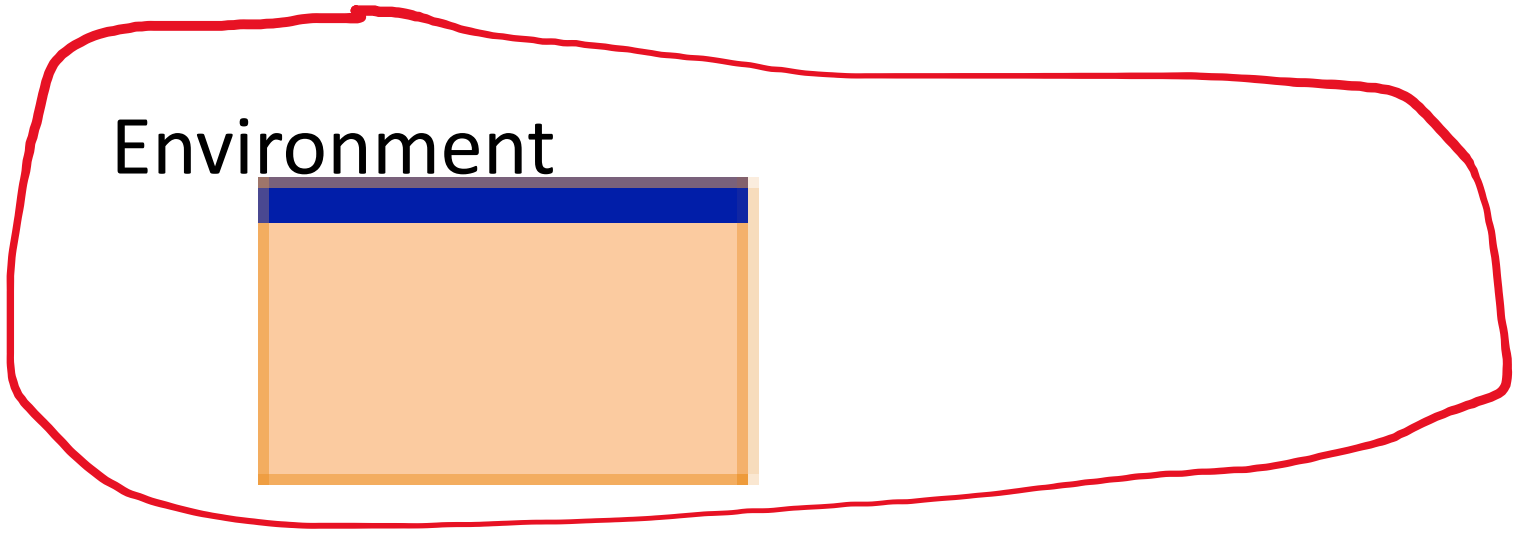
The previously active window must be redrawn to show it is inactive, the newly active window must be redrawn too. Including what was previously hidden.



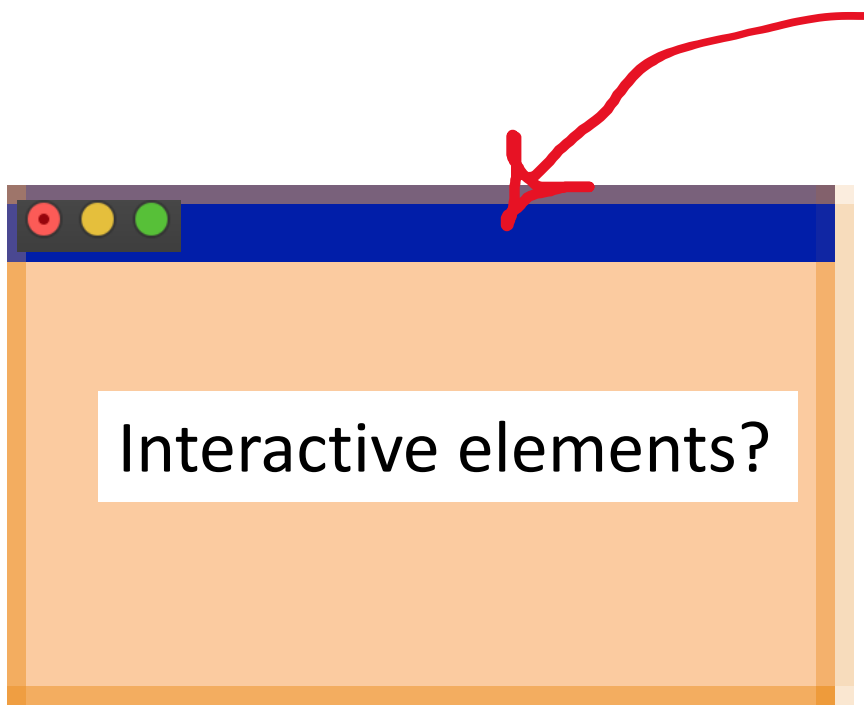
Then your screen will look like this. All this stuff is performed by the window manager and doesn’ t require you to write any code.

Environment

Observe that the GUI is running within an environment that provides services for managing windows (resizing, destruction, etc).

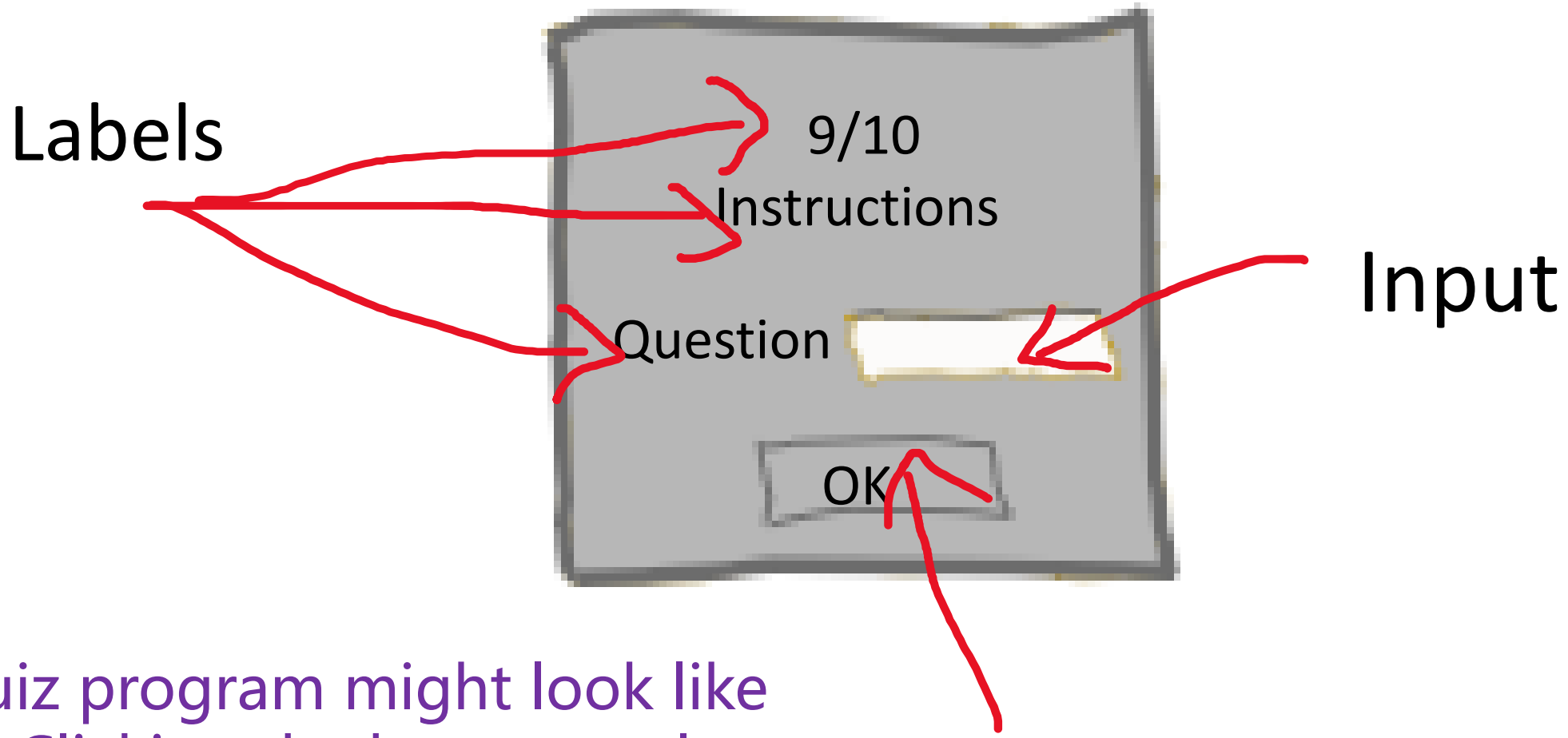


When you design your application you must decide on what the user will see: will your window have a title, will it be resizable, which elements will the user interact with in the window?



What does the user see?

Components of a window...

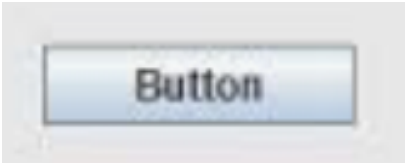


A quiz program might look like this. Clicking the button evaluates the answer.

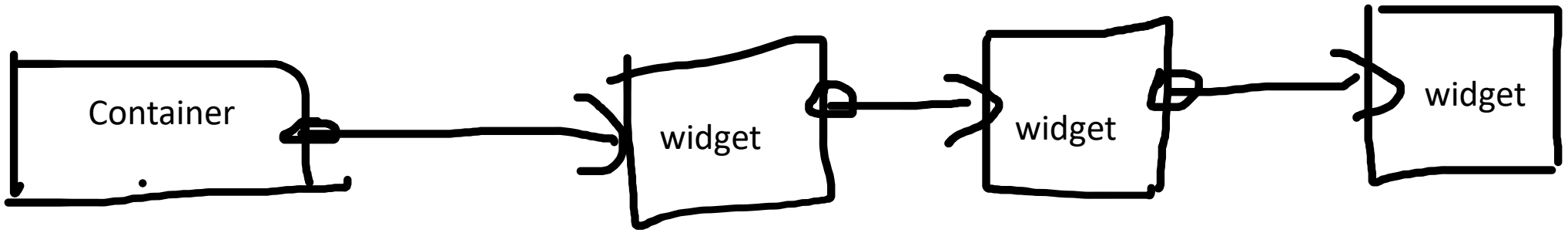
Your program must prepare everything in advance and, like a conjuror, only reveal elements at the right time when they should be visible.

Widget = Window Gadget

- Visual elements are called “widgets” - short for window gadgets
- Some widgets include:
  - labels
  - entry fields
  - drop down lists
  - check boxes
  - and buttons

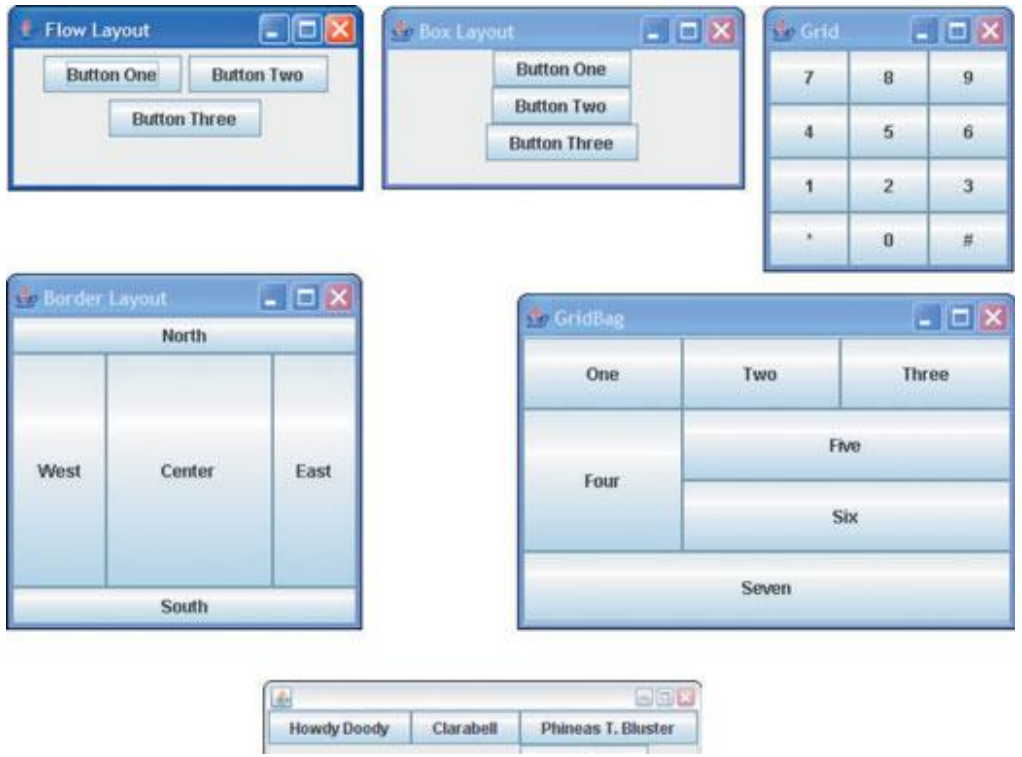


Containers



Something that you are probably not familiar with is the notion of "container". If you see widgets, you don't see containers that are nothing more than linked lists of widgets and (more about this soon) other containers.

- The purpose of containers is to make creating a layout easier.
- A layout means how the various widgets are displayed on the screen in relation to each other.



If you have a fixed-size window, things are easy. You can say "I want this widget to appear at these coordinates relative to the upper-left corner of the window".

Unfortunately the easy case isn’t the most common…

Some containers are fixed

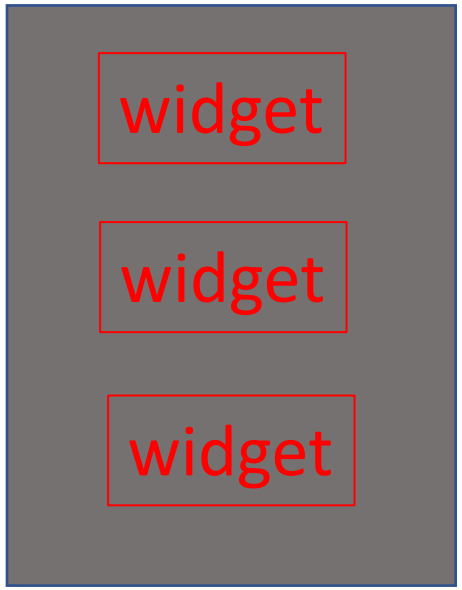
Usually people can and do resize windows and you want a “fluid layout”. If you gave absolute coordinates assuming a given window size it will soon be a big mess.

Most are not

Boxes

Containers are for solving these issues. Boxes come in two flavours and display widgets next to each other (with some padding in between) in only one direction.

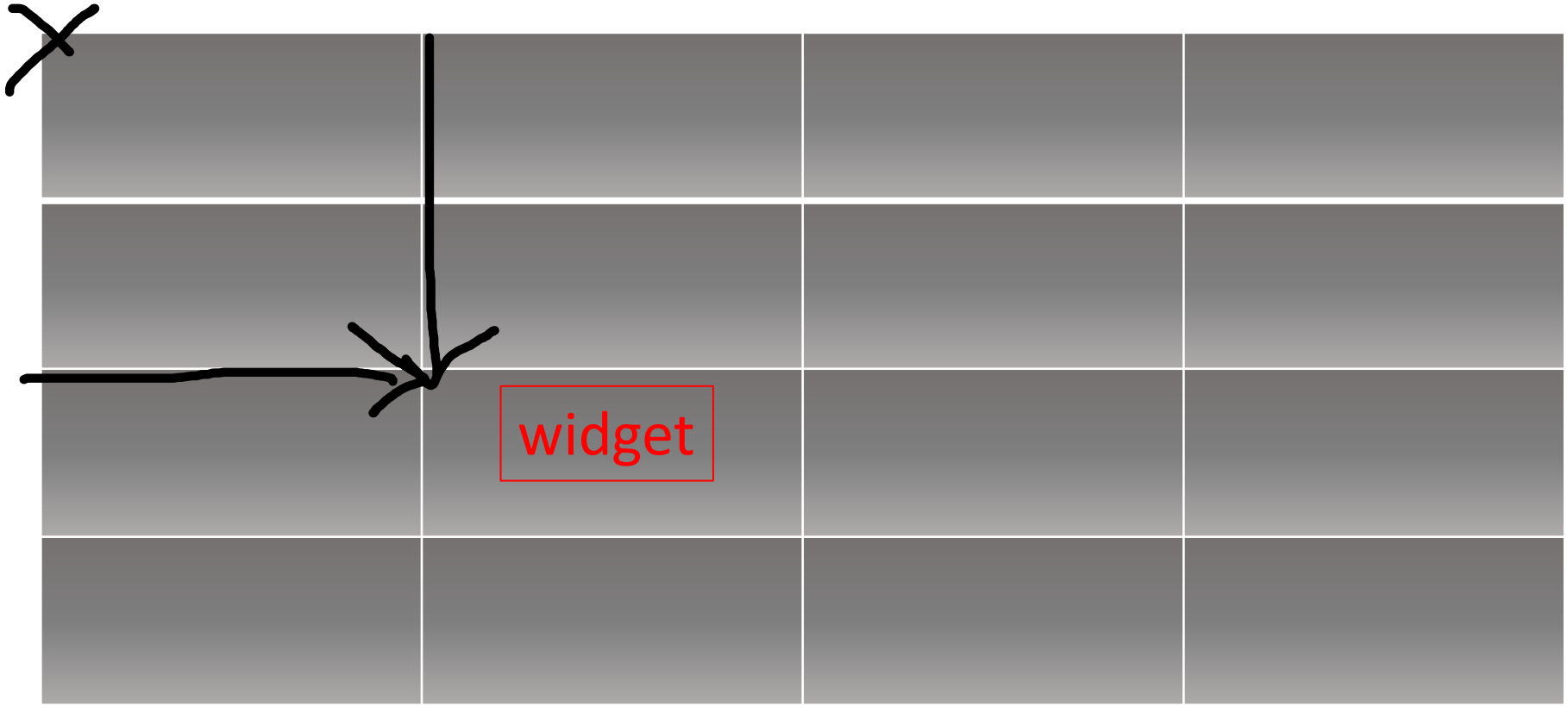
Horizontal Boxes



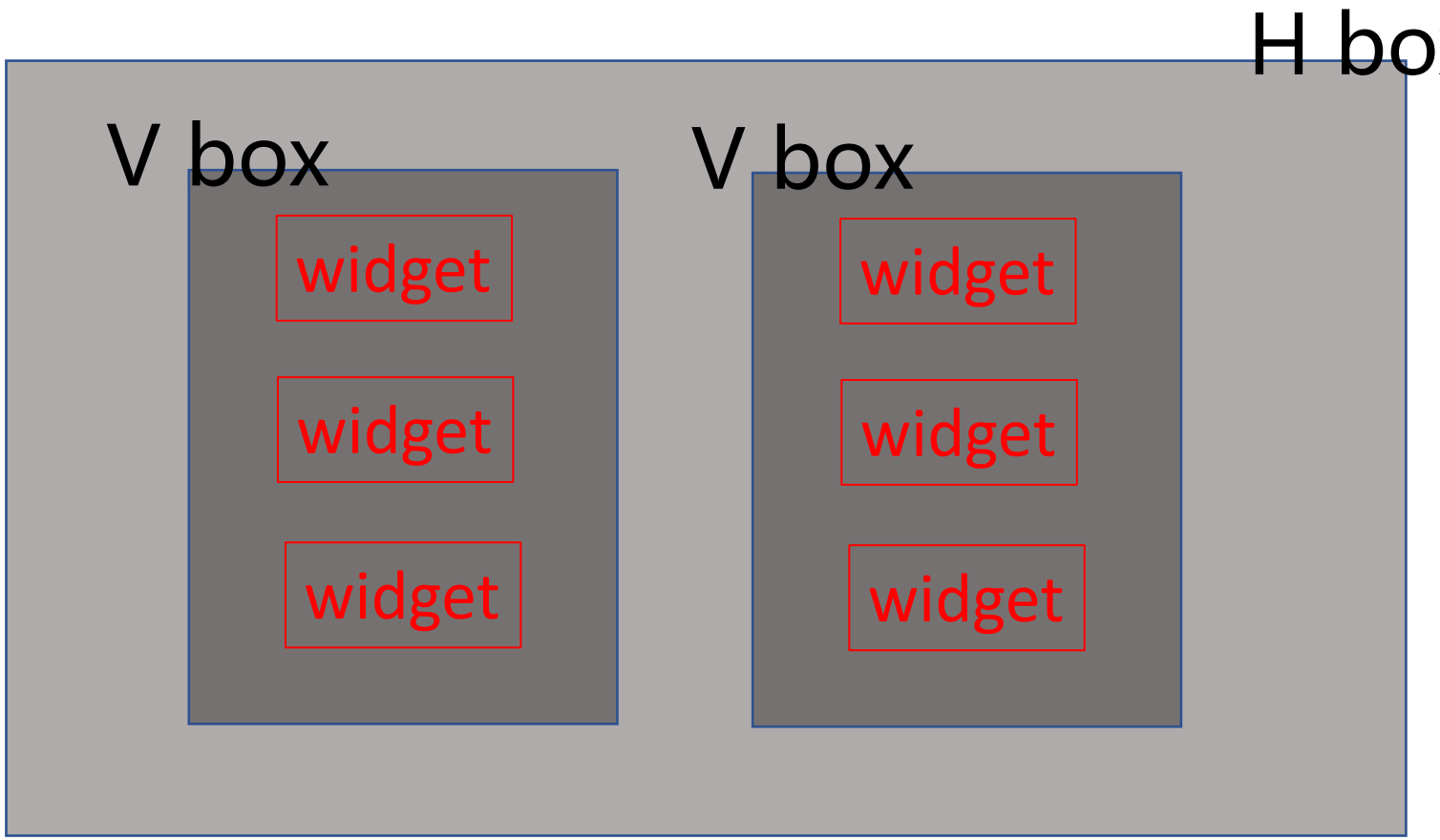
Vertical Boxes

Grids

Other frequently used containers are grids which allow you to place widgets at **relative** row column coordinates instead of absolute distance coordinates



For instance you can have two vertical boxes (each one showing widgets vertically) and add them to a horizontal box (side by side). When the window is resized, the global layout is respected and it still looks (more or less) as intended



Callback

- The last important idea to understand with graphical user interfaces is the one of "callbacks", often called "handlers" in Java, which is the name given to a function associated with an event. For instance, clicking a button might trigger a search inside a database. This is a function that you write, and associate with the button.

- A “callback” is a function associated with an event

Predefined Events

There is an extensive set of predefined events. You only handle those that matter to you. You must often perform a number of checks when the window is destroyed. For instance a text editor will ask you whether you want to save your changes.

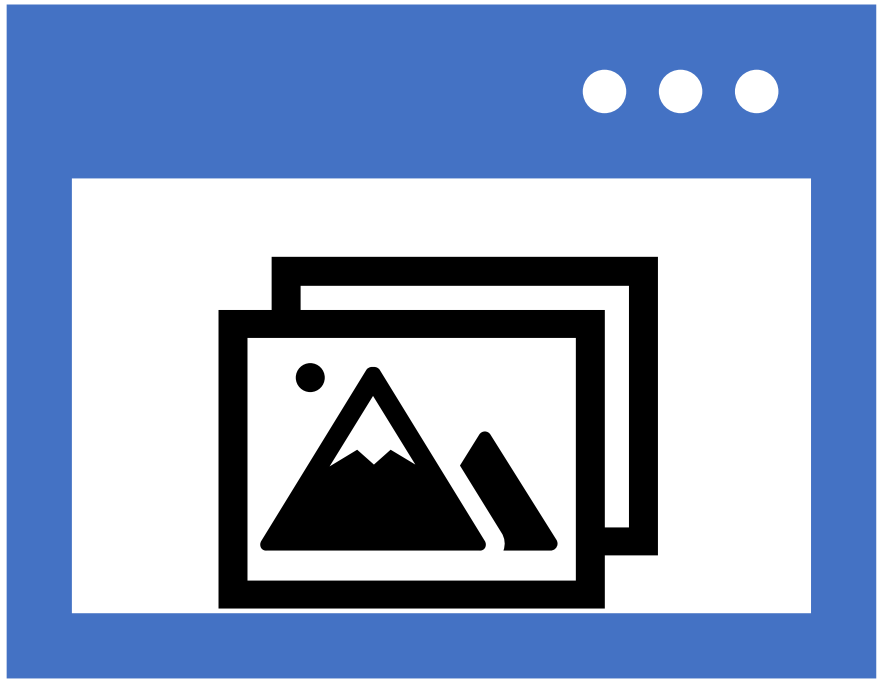
Destroy window

Button press / release

Key press / release

Focus in / out

Move in / out

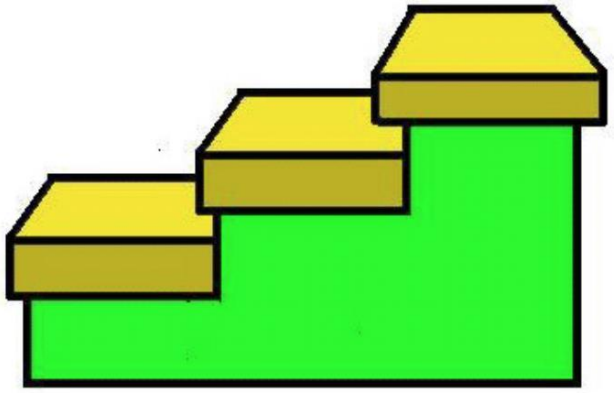


Your graphical application will run in a Window Manager that also reacts to events. For instance you may have an active window and an inactive window on the screen.

Graphical user Interfaces II
First Application

The previous presentation on GUIs (“gooey”) summarized the way Event Driven Programming is applied to Graphical User Interface Development

This presentation starts to describe how to implement all this in practice with JavaFX



Swing and AWT are replaced by the JavaFX platform for developing rich GUI applications.

The Basic Structure of a JavaFX Program

The `javafx.application.Application` *class defines the essential framework for writing JavaFX programs.*

A simple JavaFX displays a button in the window.

MyJavaFX.java

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MyJavaFX extends Application {                    extend Application
7    @Override // Override the start method in the Application class
8    public void start(Stage primaryStage) {                      override start
9      // Create a scene and place a button in the scene
10     Button btOK = new Button("OK");                            create a button
11     Scene scene = new Scene(btOK, 200, 250);                   create a scene
12     primaryStage.setTitle("MyJavaFX"); // Set the stage title  set stage title
13     primaryStage.setScene(scene); // Place the scene in the stage  set a scene
14     primaryStage.show(); // Display the stage                  display stage
15   }
16
17   /**
18    * The main method is only needed for the IDE with limited
19    * JavaFX support. Not needed for running from the command line.
20    */
21   public static void main(String[] args) {                     main method
22     Application.launch(args);                                  launch application
23   }
24 }
```

Life of a javafx application

Step 1 - Create an instance of the Application class

The program class must extend Application

A JavaFX application derives from the Application class in the JavaFx package. It means that it automatically inherits standard attributes and methods.

A simple JavaFX displays a button in the window.

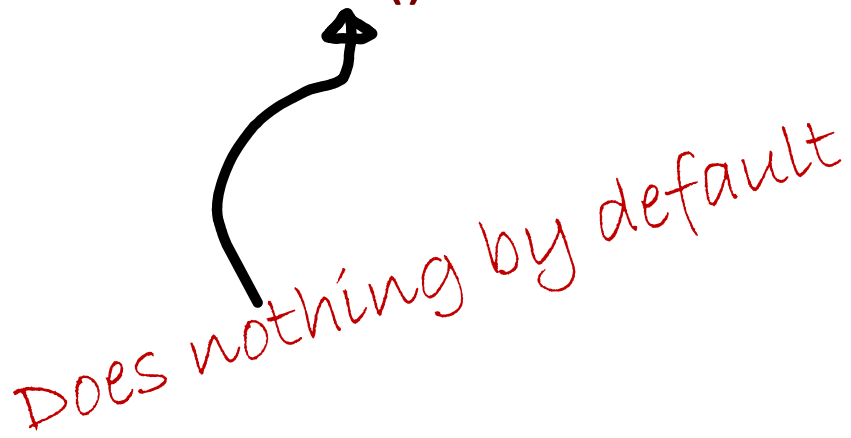
MyJavaFX.java

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MyJavaFX extends Application {                    extend Application
7    @Override // Override the start method in the Application class
8    public void start(Stage primaryStage) {                      override start
9      // Create a scene and place a button in the scene
10     Button btOK = new Button("OK");                            create a button
11     Scene scene = new Scene(btOK, 200, 250);                   create a scene
12     primaryStage.setTitle("MyJavaFX"); // Set the stage title  set stage title
13     primaryStage.setScene(scene); // Place the scene in the stage  set a scene
14     primaryStage.show(); // Display the stage                  display stage
15   }
16
17   /**
18    * The main method is only needed for the IDE with limited
19    * JavaFX support. Not needed for running from the command line.
20    */
21   public static void main(String[] args) {                     main method
22     Application.launch(args);                                  launch application
23   }
24 }
```

Life of a javafx application

Step 1 - Create an instance of the Application class

Step 2 - Call the init() method



JavaFx will automatically call a function called init(). By default, this function does nothing. You can write your own version, and connect to a network or a database, or read a parameter file.

A simple JavaFX displays a button in the window.

MyJavaFX.java

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MyJavaFX extends Application {                    extend Application
7    @Override // Override the start method in the Application class
8    public void start(Stage primaryStage) {                      override start
9      // Create a scene and place a button in the scene
10     Button btOK = new Button("OK");                            create a button
11     Scene scene = new Scene(btOK, 200, 250);                   create a scene
12     primaryStage.setTitle("MyJavaFX"); // Set the stage title  set stage title
13     primaryStage.setScene(scene); // Place the scene in the stage  set a scene
14     primaryStage.show(); // Display the stage                  display stage
15   }
16
17   /**
18    * The main method is only needed for the IDE with limited
19    * JavaFX support. Not needed for running from the command line.
20    */
21   public static void main(String[] args) {                     main method
22     Application.launch(args);                                  launch application
23   }
24 }
```

Life of a javafx application

Step 1 - Create an instance of the Application class

Step 2 - Call the init() method

Must be rewritten!

Step 3 - Call the start(javafx.stage.Start) method

What you must write (and override) is a function called "start()" that takes a "Stage" (the name given to windows in JavaFx) as parameter. The function adds the widgets to the window and defines how it looks, and how widgets will react.

A simple JavaFX displays a button in the window.

MyJavaFX.java

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MyJavaFX extends Application {                    extend Application
7    @Override // Override the start method in the Application class
8    public void start(Stage primaryStage) {                      override start
9      // Create a scene and place a button in the scene
10     Button btOK = new Button("OK");                            create a button
11     Scene scene = new Scene(btOK, 200, 250);                   create a scene
12     primaryStage.setTitle("MyJavaFX"); // Set the stage title  set stage title
13     primaryStage.setScene(scene); // Place the scene in the stage  set a scene
14     primaryStage.show(); // Display the stage                  display stage
15   }
16
17   /**
18    * The main method is only needed for the IDE with limited
19    * JavaFX support. Not needed for running from the command line.
20    */
21   public static void main(String[] args) {                     main method
22     Application.launch(args);                                  launch application
23   }
24 }
```

Life of a javafx application

Step 1 - Create an instance of the Application class

Step 2 - Call the init() method

Step 3 - Call the start(javafx.stage.Stage) method

Step 4 - Wait for the application to finish:
the application calls Platform.exit()
or window closed

You must write the event handlers you need – *and nothing else* – JavaFx will run the application until it calls an exit routine (perhaps associated with a "Quit" button) or it receives the event "Window destroyed".

A simple JavaFX displays a button in the window.

MyJavaFX.java

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MyJavaFX extends Application {                    extend Application
7    @Override // Override the start method in the Application class
8    public void start(Stage primaryStage) {                      override start
9      // Create a scene and place a button in the scene
10     Button btOK = new Button("OK");                            create a button
11     Scene scene = new Scene(btOK, 200, 250);                   create a scene
12     primaryStage.setTitle("MyJavaFX"); // Set the stage title  set stage title
13     primaryStage.setScene(scene); // Place the scene in the stage  set a scene
14     primaryStage.show(); // Display the stage                  display stage
15   }
16
17   /**
18    * The main method is only needed for the IDE with limited
19    * JavaFX support. Not needed for running from the command line.
20    */
21   public static void main(String[] args) {                     main method
22     Application.launch(args);                                  launch application
23   }
24 }
```

Life of a javafx application

Step 1 - Create an instance of the Application class

Step 2 - Call the init() method

Step 3 - Call the start(javafx.stage.Stage) method

Step 4 - Wait for the application to finish:
the application calls Platform.exit()
or window closed

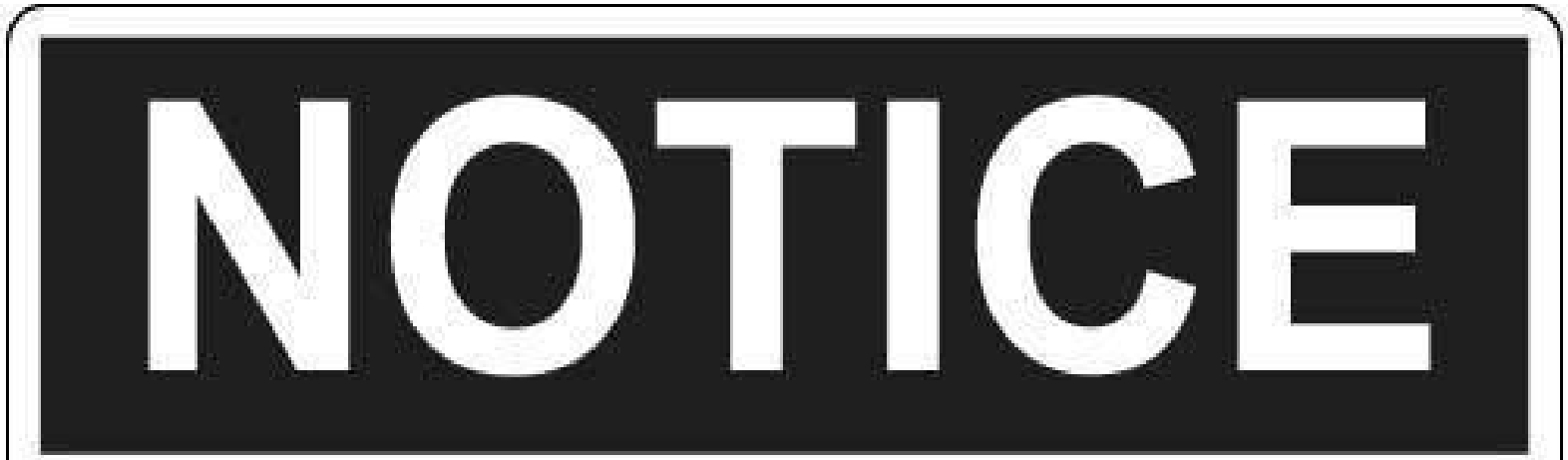
Step 5 - Call the stop() method

It will then call a stop() method where you can undo what you have done in init() – for example disconnect from a database or network. Like with init(), and exit() rewriting stop() is only done if you *need* to.

A simple JavaFX displays a button in the window.

MyJavaFX.java

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MyJavaFX extends Application {                    extend Application
7    @Override // Override the start method in the Application class
8    public void start(Stage primaryStage) {                      override start
9      // Create a scene and place a button in the scene
10     Button btOK = new Button("OK");                            create a button
11     Scene scene = new Scene(btOK, 200, 250);                   create a scene
12     primaryStage.setTitle("MyJavaFX"); // Set the stage title  set stage title
13     primaryStage.setScene(scene); // Place the scene in the stage  set a scene
14     primaryStage.show(); // Display the stage                  display stage
15   }
16
17   /**
18    * The main method is only needed for the IDE with limited
19    * JavaFX support. Not needed for running from the command line.
20    */
21   public static void main(String[] args) {                     main method
22     Application.launch(args);                                  launch application
23   }
24 }
```



No explicit test

No explicit loop

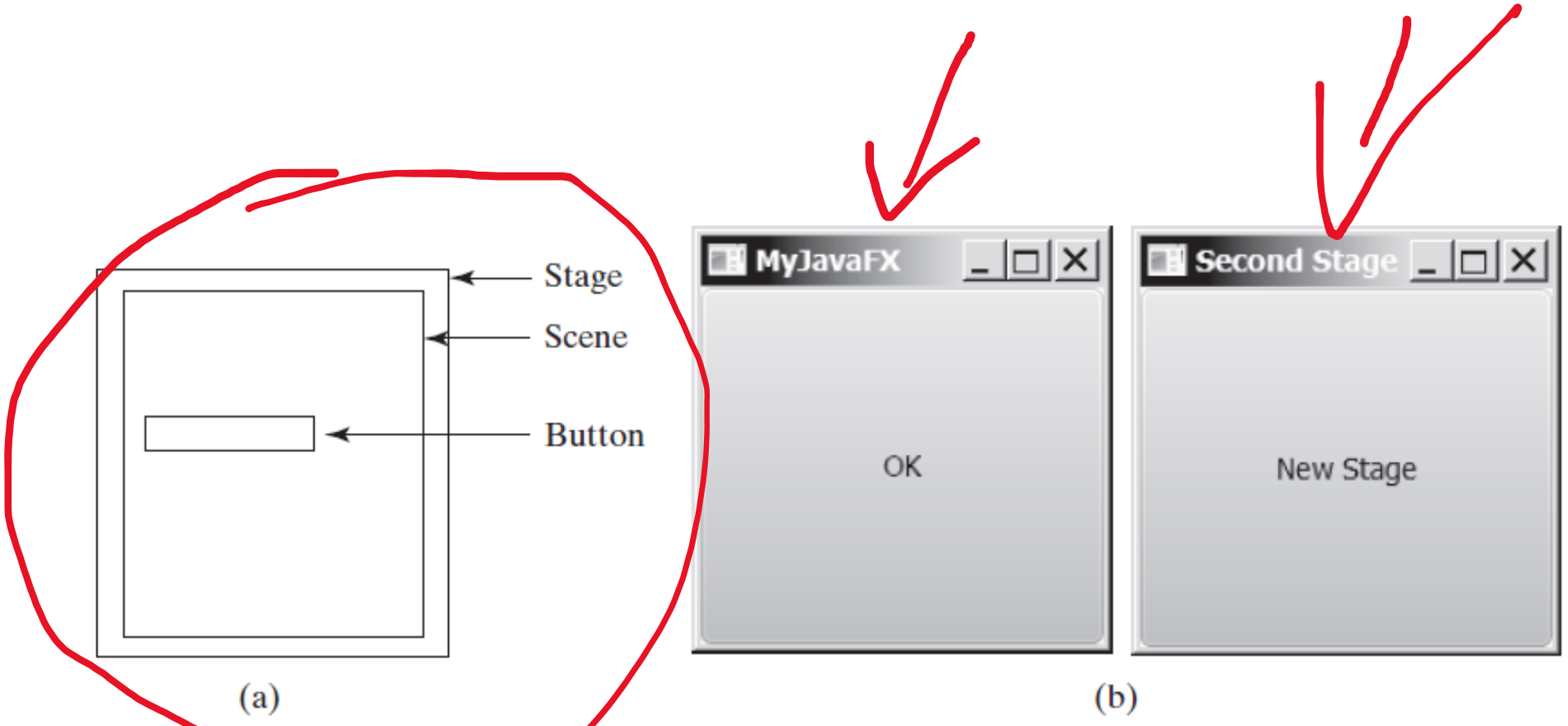
Just events

In a GUI application is that you just declare everything, and there is no procedural logic (if ... and loops) outside event handlers.

What's next??

- In GUI parts I and II, concepts that are important for designing GUIs and using JavaFX were introduced

- Let us extends these concepts to demonstrate the use of "stages" and "containers" and other elements of JavaFX more generally



(a) Stage is a window for displaying a scene that contains nodes.
(b) Multiple stages can be displayed in a JavaFX program.

MultipleStageDemo.java

primary stage in start

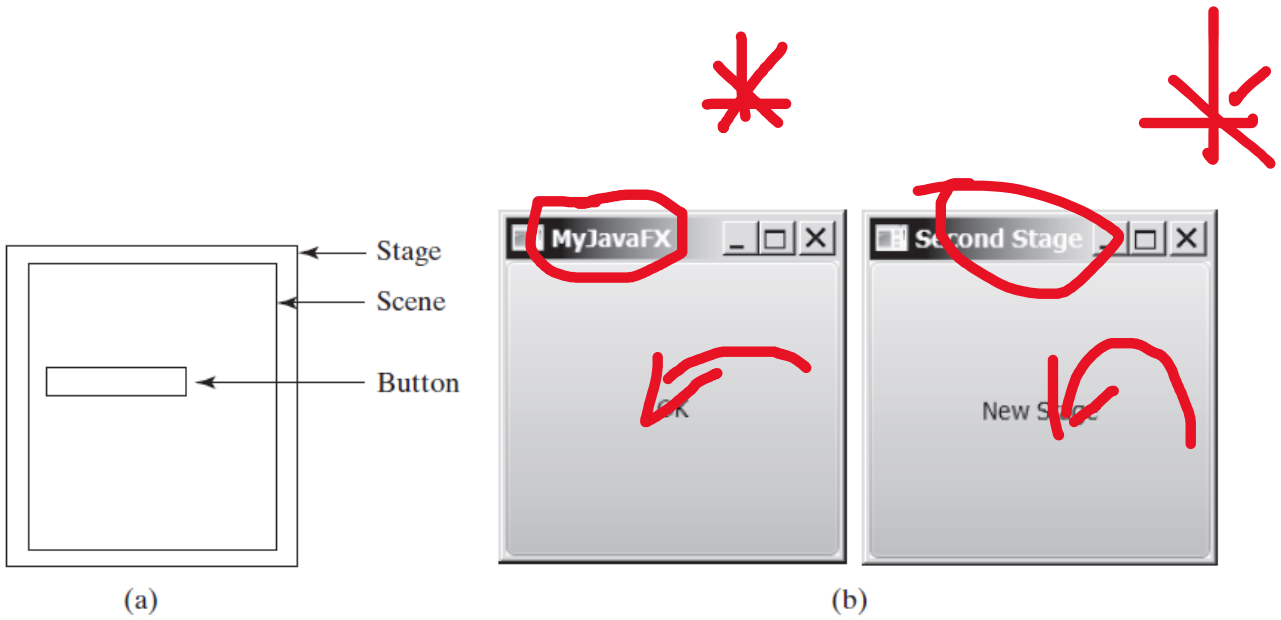
display primary stage

create second stage

display second stage

main method omitted

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MultipleStageDemo extends Application {
7    @Override // Override the start method in the Application class
8    public void start(Stage primaryStage) {
9      // Create a scene and place a button in the scene
10     Scene scene = new Scene(new Button("OK"), 200, 250);
11     primaryStage.setTitle("MyJavaFX"); // Set the stage title
12     primaryStage.setScene(scene); // Place the scene in the stage
13     primaryStage.show(); // Display the stage
14
15     Stage stage = new Stage(); // Create a new stage
16     stage.setTitle("Second Stage"); // Set the stage title
17     // Set a scene with a button in the stage
18     stage.setScene(new Scene(new Button("New Stage"), 200, 250));
19     stage.show(); // Display the stage
20   }
21 }
```



(a) Stage is a window for displaying a scene that contains nodes.
(b) Multiple stages can be displayed in a JavaFX program.

MultipleStageDemo.java

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MultipleStageDemo extends Application {
7    @Override // Override the start method in the Application class
8    public void start(Stage primaryStage) {
9      // Create a scene and place a button in the scene
10     Scene scene = new Scene(new Button("OK"), 200, 250);
11     primaryStage.setTitle("MyJavaFX"); // Set the stage title
12     primaryStage.setScene(scene); // Place the scene in the stage
13     primaryStage.show(); // Display the stage
14
15     Stage stage = new Stage(); // Create a new stage
16     stage.setTitle("Second Stage"); // Set the stage title
17     // Set a scene with a button in the stage
18     stage.setScene(new Scene(new Button("New Stage"), 200, 250));
19     stage.show(); // Display the stage
20   }
21 }
```

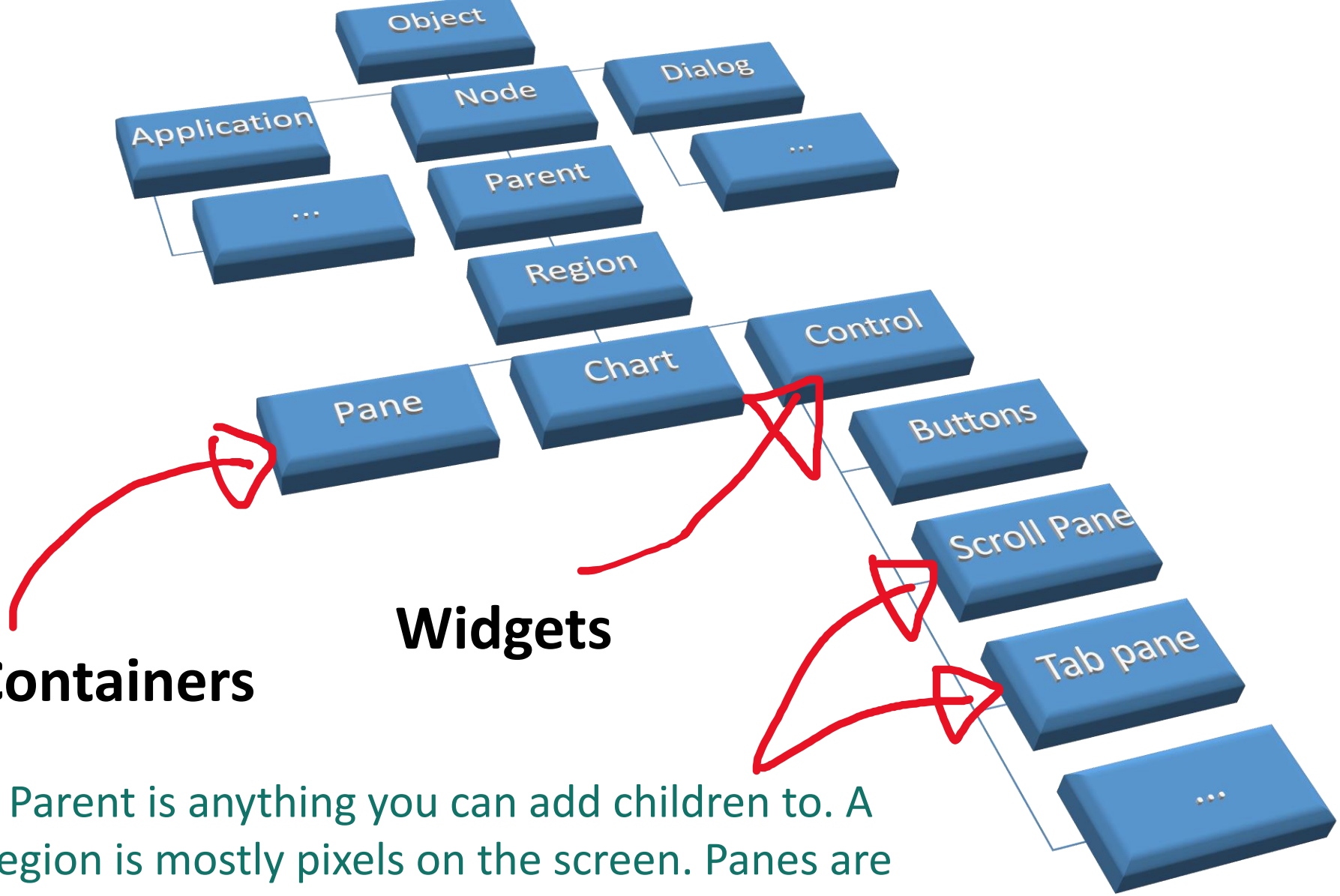
primary stage in start

display primary stage

create second stage

display second stage

main method omitted



Containers

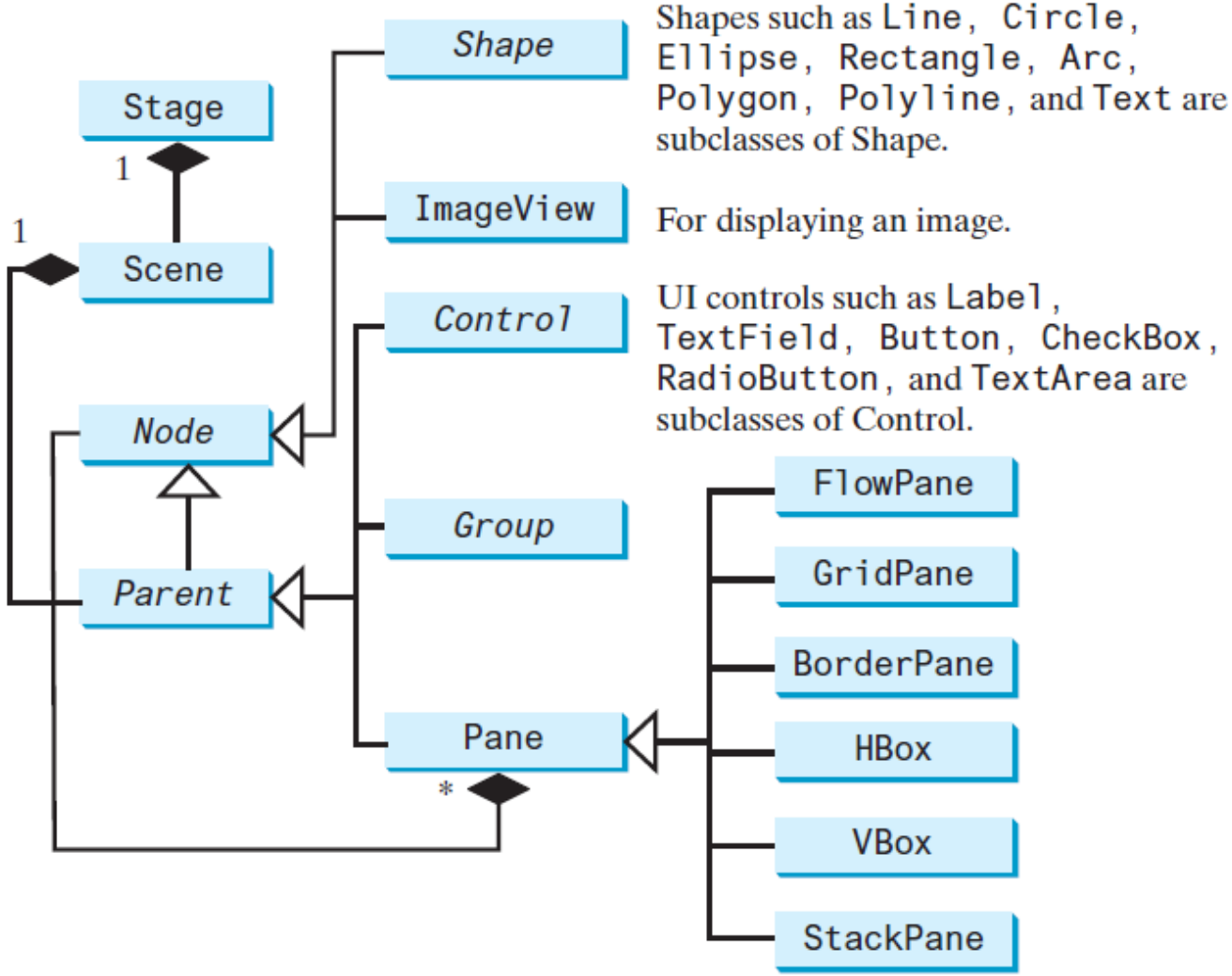
Widgets

A Parent is anything you can add children to. A Region is mostly pixels on the screen. Panes are containers, but beware that some widgets that you might regard as containers are in fact "Controls".

In the JavaFx class hierarchy we have at the top, three classes that directly extend Object:

- **Application** (we have talked about it already),
- **Node** (basically anything on screen, visible or not) and
- **Dialog**. A Dialog is a kind of minimal application performing a specialized task (when you open a window to choose a file to open, it's a dialog).

Panes, Groups, UI controls, and shapes are subtypes of **Node**.

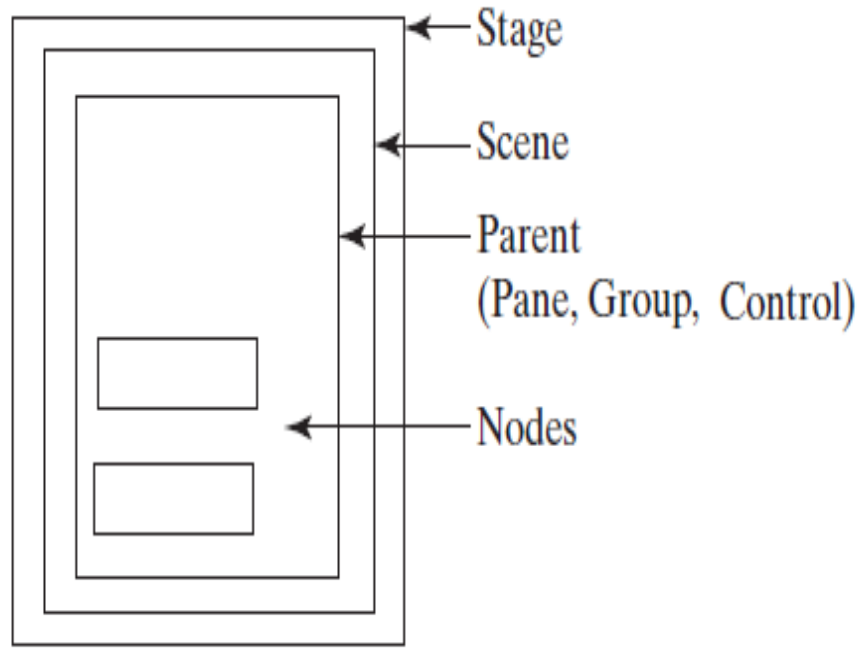


Shapes such as `Line`, `Circle`, `Ellipse`, `Rectangle`, `Arc`, `Polygon`, `Polyline`, and `Text` are subclasses of Shape.

For displaying an image.

UI controls such as `Label`, `TextField`, `Button`, `CheckBox`, `RadioButton`, and `TextArea` are subclasses of Control.

Nodes can be shapes, image views, UI controls, groups, and panes.



Panes and groups are used to hold nodes.

ButtonInPane.java

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5  import javafx.scene.layout.StackPane;
6
7  public class ButtonInPane extends Application {
8    @Override // Override the start method in the Application class
9    public void start(Stage primaryStage) {
10     // Create a scene and place a button in the scene
11     StackPane pane = new StackPane();
12     pane.getChildren().add(new Button("OK"));
13     Scene scene = new Scene(pane, 200, 50);
14     primaryStage.setTitle("Button in a pane"); // Set the stage title
15     primaryStage.setScene(scene); // Place the scene in the stage
16     primaryStage.show(); // Display the stage
17   }
18 }
```

create a pane

add a button

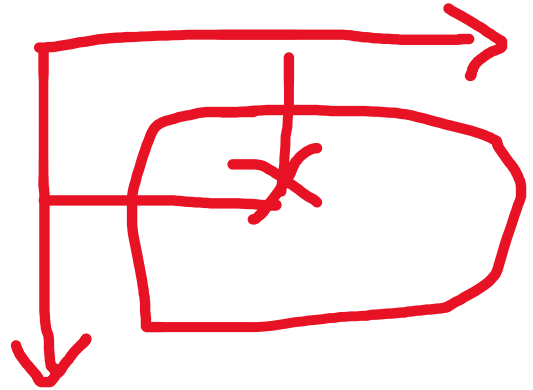
add pane to scene

display stage

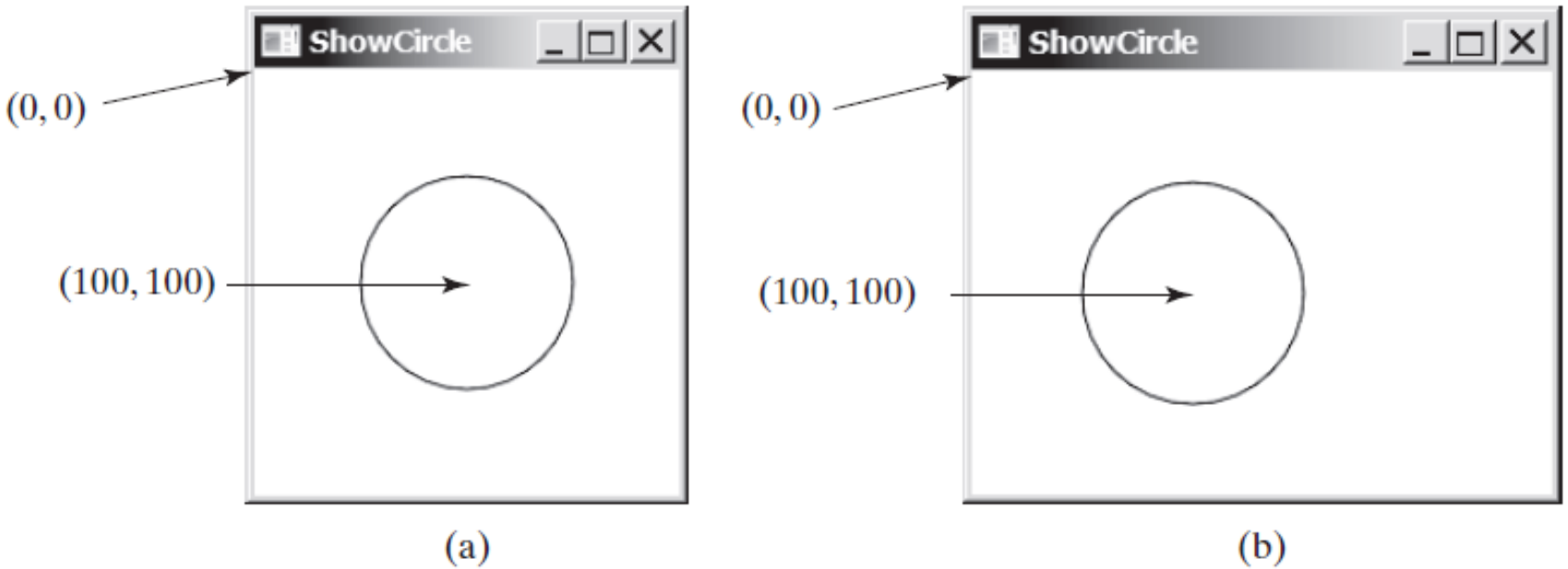
main method omitted

A button is placed in the center of the pane.

ShowCircle.java

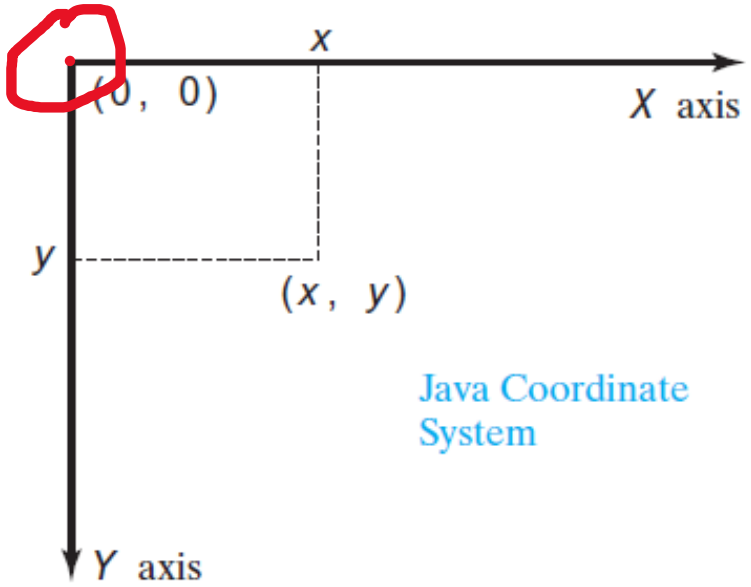


```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.layout.Pane;
4  import javafx.scene.paint.Color;
5  import javafx.scene.shape.Circle;
6  import javafx.stage.Stage;
7
8  public class ShowCircle extends Application {
9    @Override // Override the start method in the Application class
10   public void start(Stage primaryStage) {
11     // Create a circle and set its properties
12     Circle circle = new Circle();                          create a circle
13     circle.setCenterX(100);                                set circle properties
14     circle.setCenterY(100);
15     circle.setRadius(50);
16     circle.setStroke(Color.BLACK);
17     circle.setFill(Color.WHITE);
18
19     // Create a pane to hold the circle
20     Pane pane = new Pane();                                create a pane
21     pane.getChildren().add(circle);                        add circle to pane
22
23     // Create a scene and place it in the stage
24     Scene scene = new Scene(pane, 200, 200);               add pane to scene
25     primaryStage.setTitle("ShowCircle"); // Set the stage title
26     primaryStage.setScene(scene); // Place the scene in the stage
27     primaryStage.show(); // Display the stage              display stage
28   }
29 }                                                          main method omitted
```

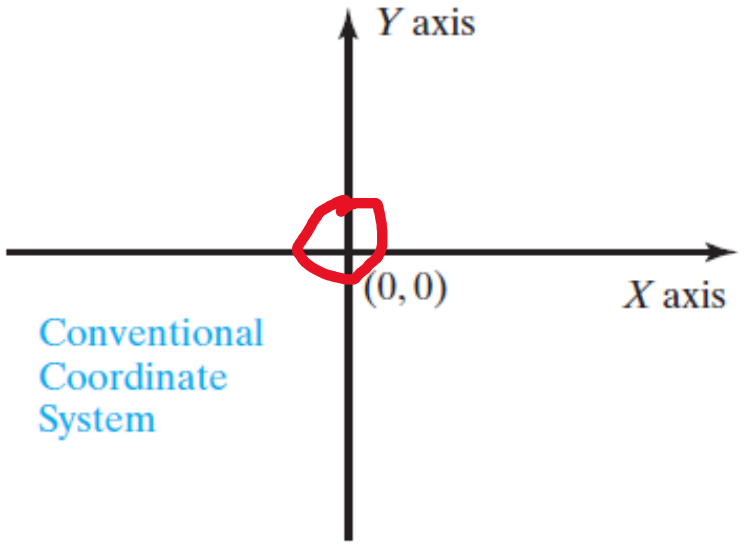


(a) A circle is displayed in the center of the scene.

(b) The circle is not centered after the window is resized.



(a)



(b)

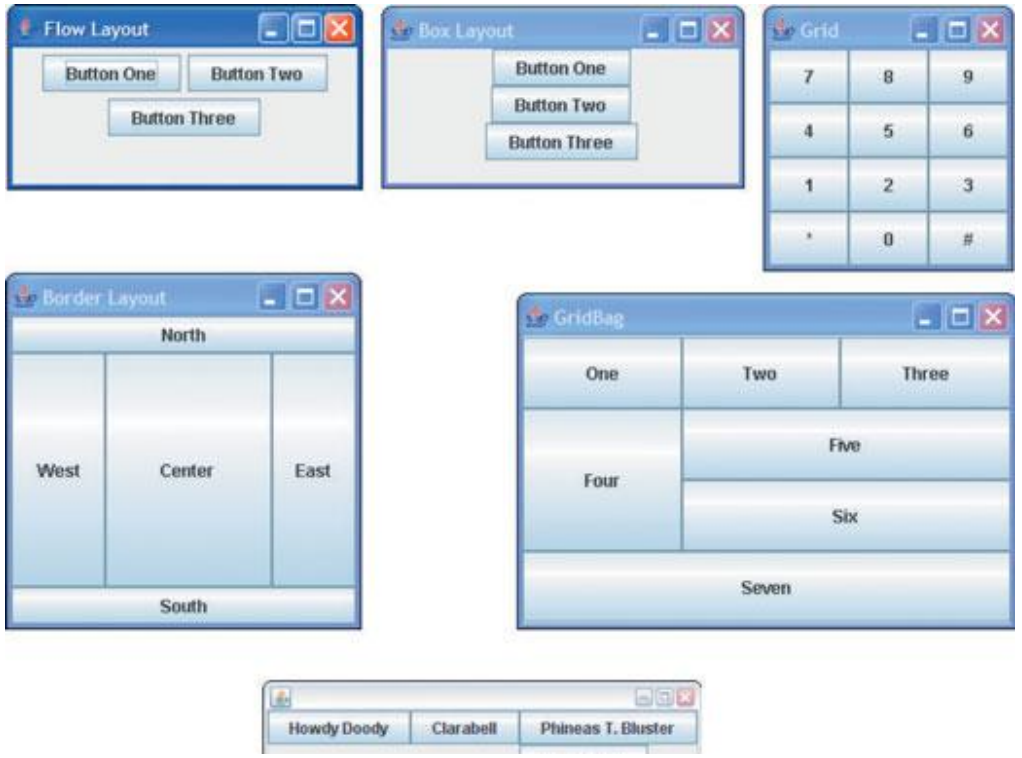
The Java coordinate system is measured in pixels, with `(0, 0)` at its upper-left corner.

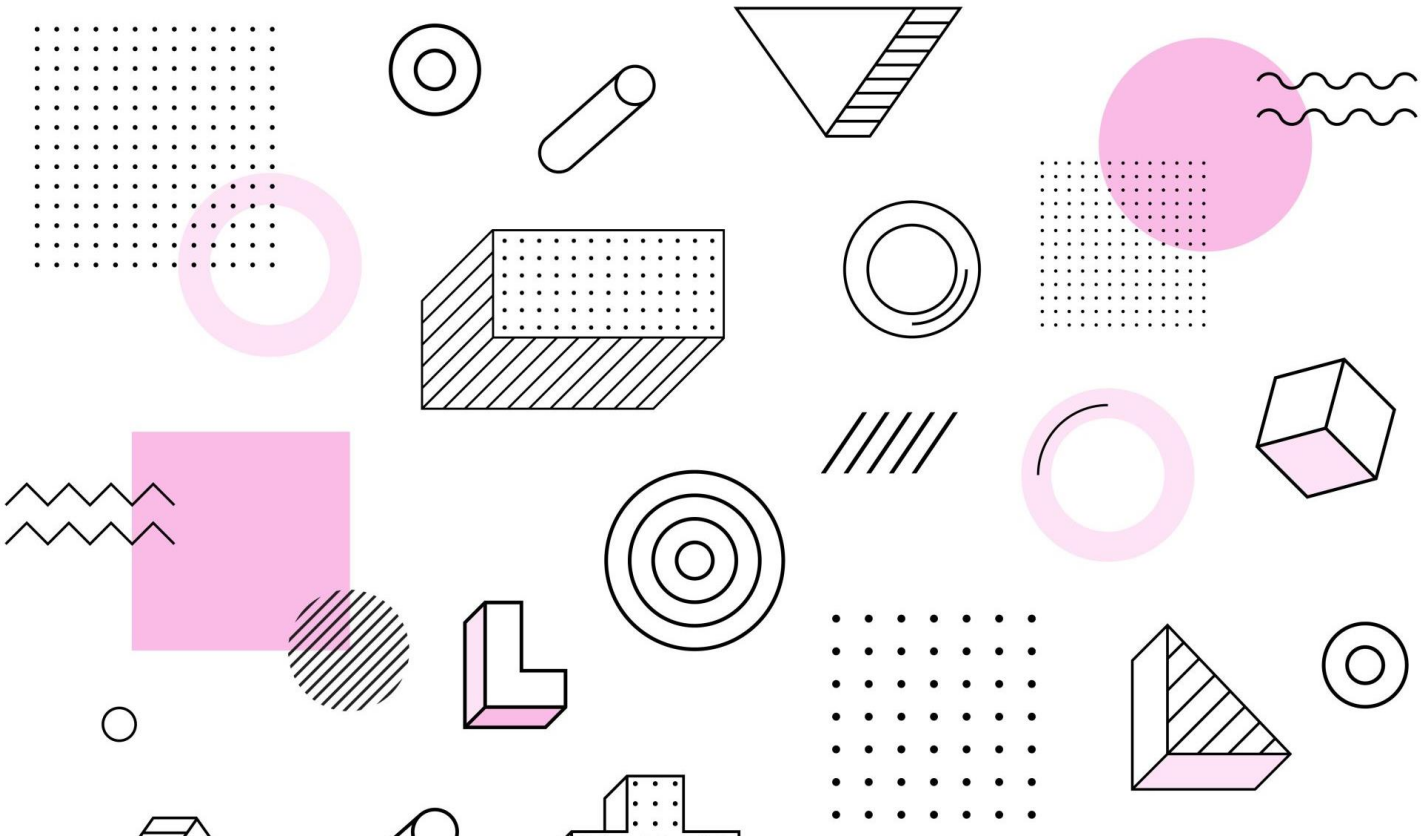
If you have a fixed-size window, things are easy. You can say "I want this widget to appear at these coordinates relative to the upper-left corner of the window".

Unfortunately the easy case isn’t the most common…

Some containers are fixed

In JavaFX Panes are used for Laying Out Containers – lets look at how they work…



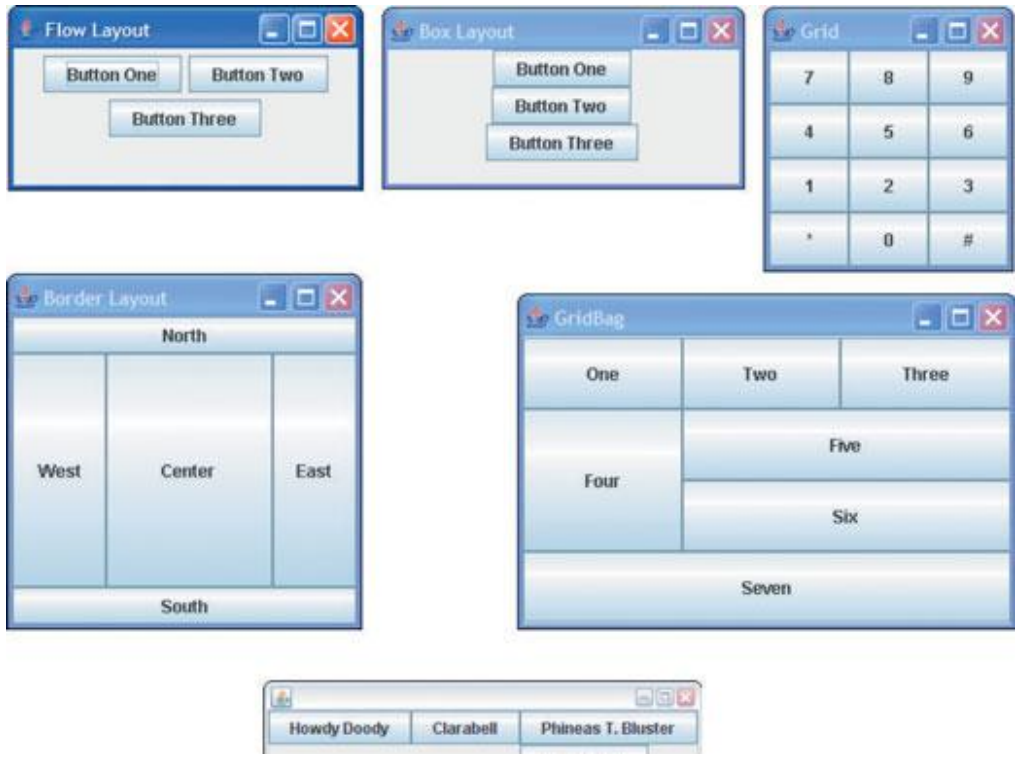


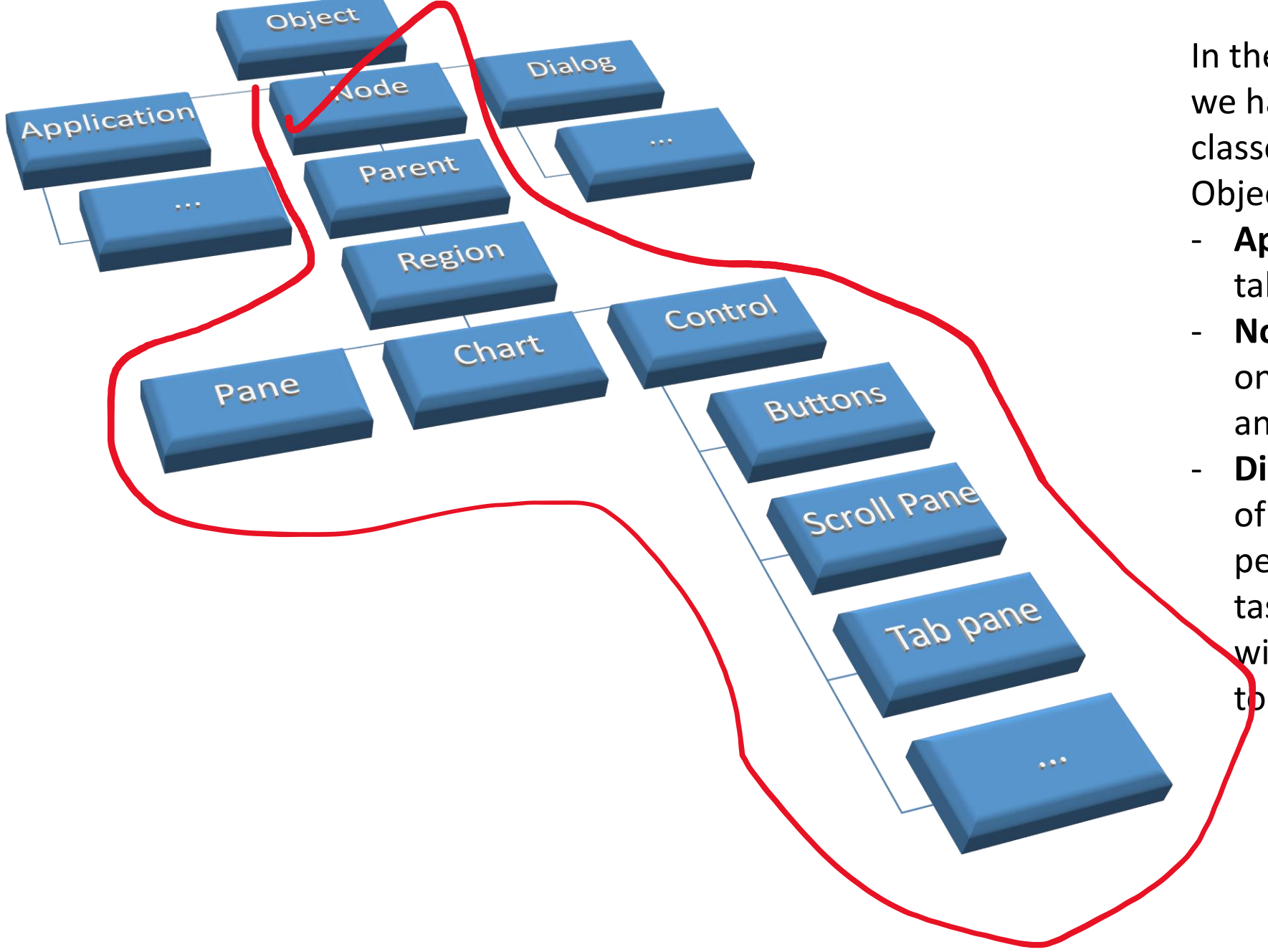
Graphicical User Interface IV

Group Layouts

In JavaFX Panes are used for Laying Out Containers

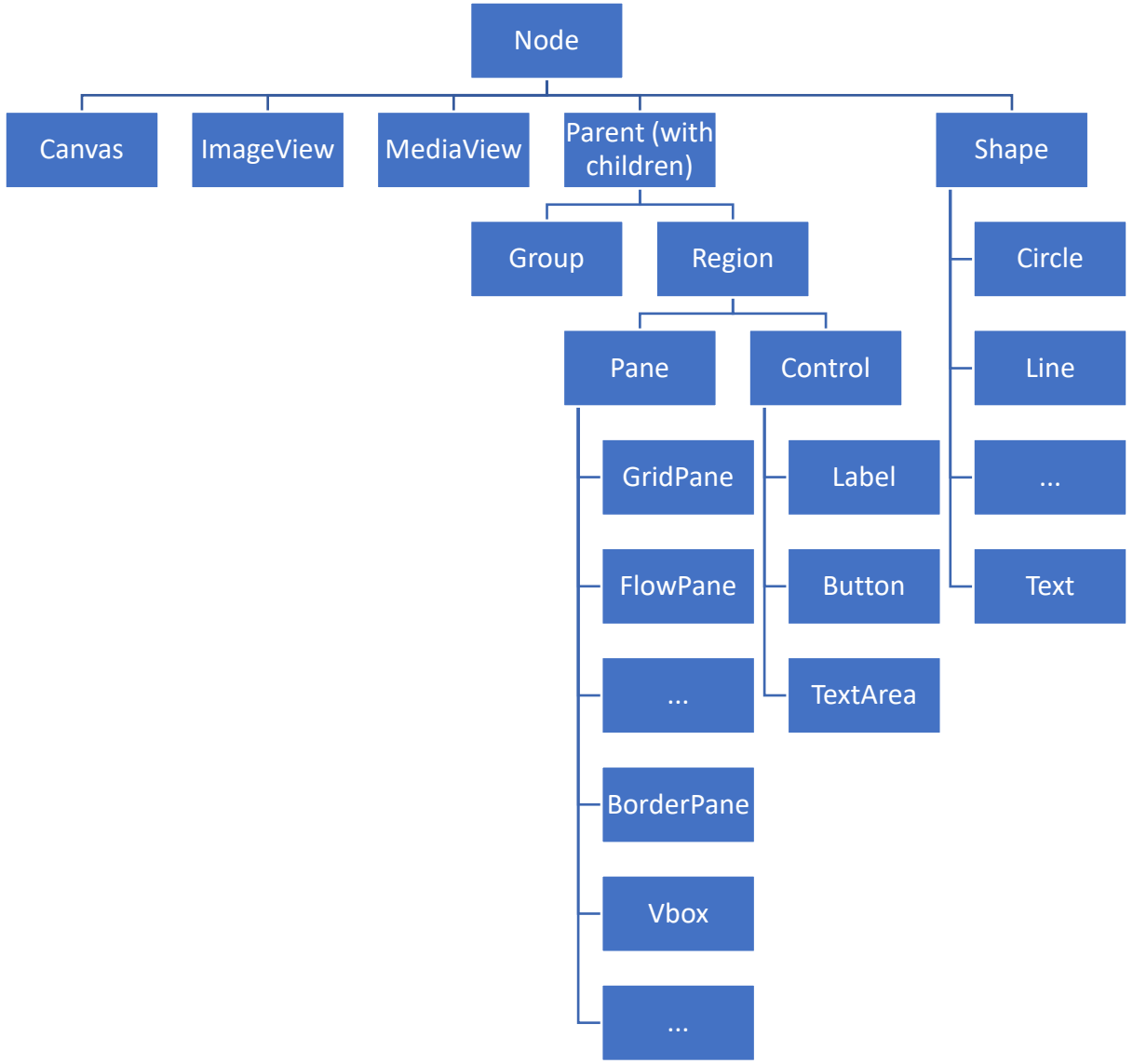
- The purpose of containers is to make creating a layout easier.
- A layout means how the various widgets are displayed on the screen in relation to each other.





In the JavaFx class hierarchy we have at the top, three classes that directly extend Object:

- **Application** (we have talked about it already),
- **Node** (basically anything on screen, visible or not) and
- **Dialog**. A Dialog is a kind of minimal application performing a specialized task (when you open a window to choose a file to open, it's a dialog).

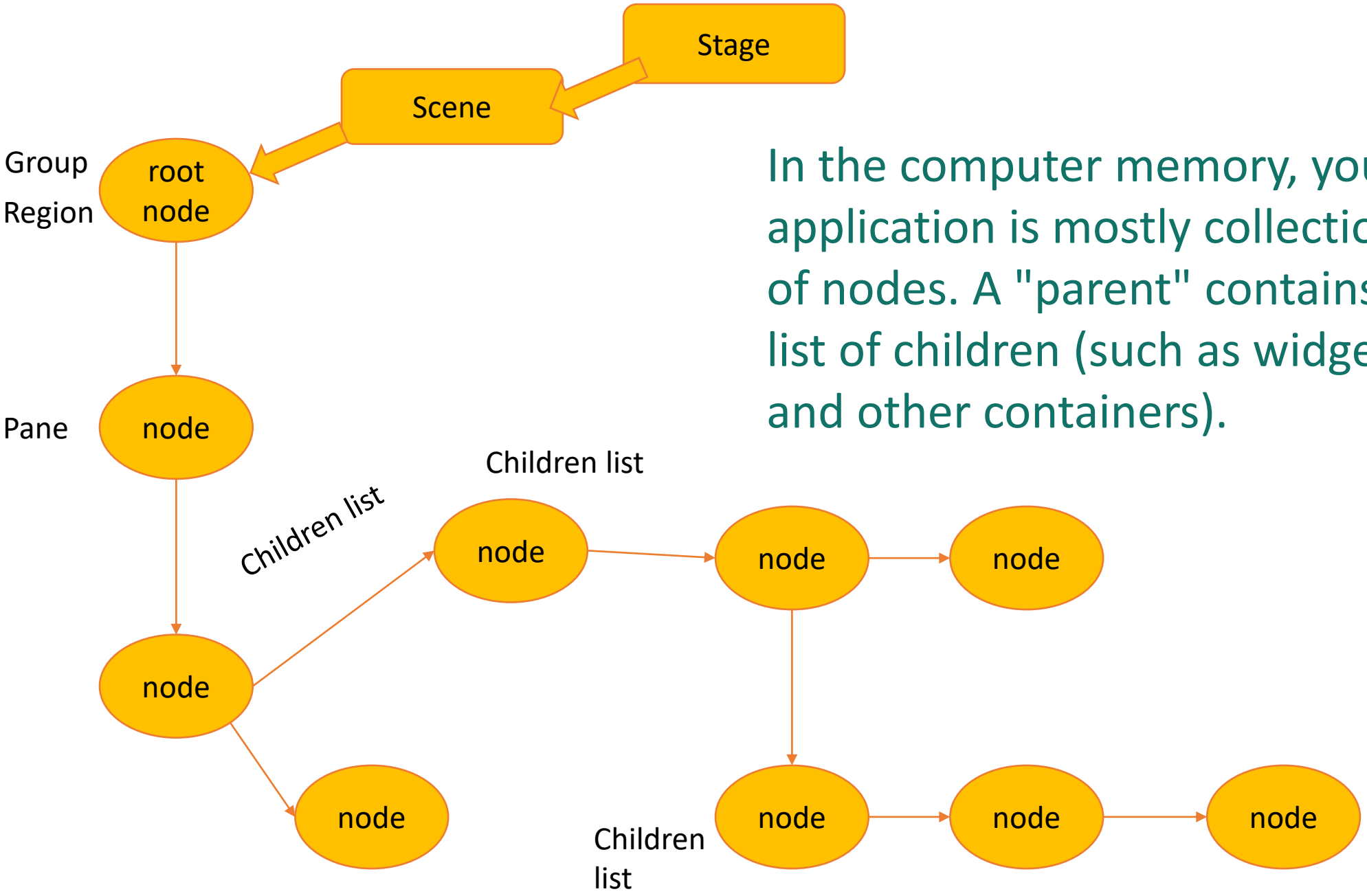


More classes that inherit from node…

JavaFX Package Hierarchy

- javafx.application
- javafx.scene
- javafx.scene.layout
- javafx.scene.control
- javafx.scene.input
- javafx.event
- javafx.geometry
- javafx.util

You also have a package hierarchy but beware that the package grouping isn't the same as the object hierarchy – grouping here is more by function than inherited methods or attributes.



In the computer memory, your application is mostly collections of nodes. A "parent" contains a list of children (such as widgets, and other containers).

Layout Panes and Groups

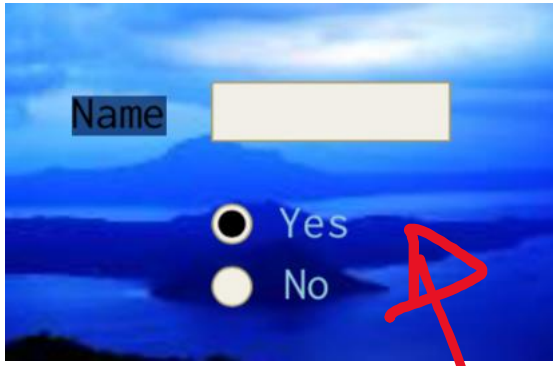
JavaFX provides many types of panes for automatically laying out nodes in a desired location and size.

Panes and groups are the containers for holding nodes. The `Group` class is often used to group nodes and to perform transformation and scale as a group. Panes and UI control objects are resizable, but group, shape, and text objects are not resizable. JavaFX provides many types of panes for organizing nodes in a container, as shown in Table .

Panes for Containing and Organizing Nodes

Class	*Description*
`Pane`	Base class for layout panes. It contains the `getChildren()` method for returning a list of nodes in the pane.
`StackPane`	Places the nodes on top of each other in the center of the pane.
`FlowPane`	Places the nodes row-by-row horizontally or column-by-column vertically.
`GridPane`	Places the nodes in the cells in a two-dimensional grid.
`BorderPane`	Places the nodes in the top, right, bottom, left, and center regions.
`HBox`	Places the nodes in a single row.
`VBox`	Places the nodes in a single column.

More advanced types of panes can also be added later



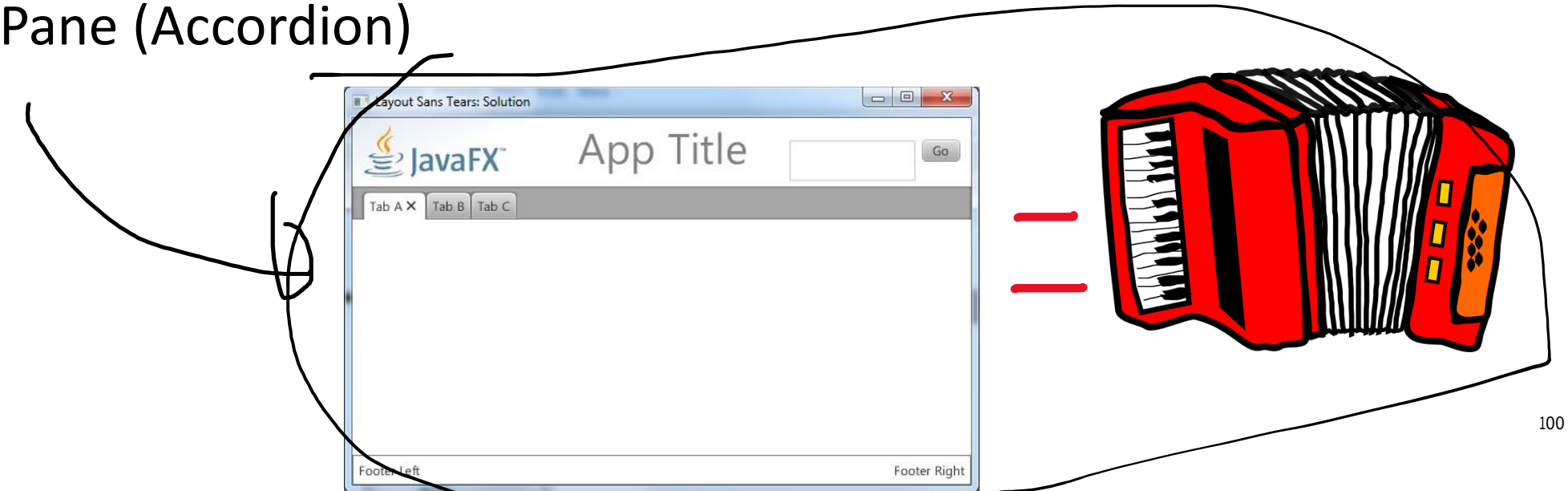
Most often your main Window will be one of these.

The StackPane allows to have elements on top of each other, which is mostly interesting for background images.

More Sophisticated Types of Panes Can be Added Afterwards

- AnchorPane
- ScrollPane
- SplitPane
- TabPane
- TitledPane (Accordion)

Controls



Typical Design

```
public static void start(Stage stage) {
        stage.setTitle("Window Title");
        Group root = new Group();
        Scene scene = new Scene(root);
        BorderPane pane = new BorderPane();
        root.getChildren().add(pane);

        // Add containers and widgets to pane

        stage.setScene(scene);
        stage.show();
```

Here is a basic start method for a javaFX program

Panes

ShowFlowPane.java

```
1  import javafx.application.Application;
2  import javafx.geometry.Insets;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Label;
5  import javafx.scene.control.TextField;
6  import javafx.scene.layout.FlowPane;
7  import javafx.stage.Stage;
8
9  public class ShowFlowPane extends Application {
10   @Override // Override the start method in the Application class
11   public void start(Stage primaryStage) {
12     // Create a pane and set its properties
13     FlowPane pane = new FlowPane();
14     pane.setPadding(new Insets(11, 12, 13, 14));
15     pane.setHgap(5);
16     pane.setVgap(5);
17
18     // Place nodes in the pane
19     pane.getChildren().addAll(new Label("First Name:"),
20       new TextField(), new Label("MI:"));
21     TextField tfMi = new TextField();
22     tfMi.setPrefColumnCount(1);
23     pane.getChildren().addAll(tfMi, new Label("Last Name:"),
24       new TextField());
25
26     // Create a scene and place it in the stage
27     Scene scene = new Scene(pane, 200, 250);
28     primaryStage.setTitle("ShowFlowPane"); // Set the stage title
29     primaryStage.setScene(scene); // Place the scene in the stage
30     primaryStage.show(); // Display the stage
31   }
32 }
```

extend Application

create FlowPane

add UI controls to pane

add pane to scene

place scene to stage

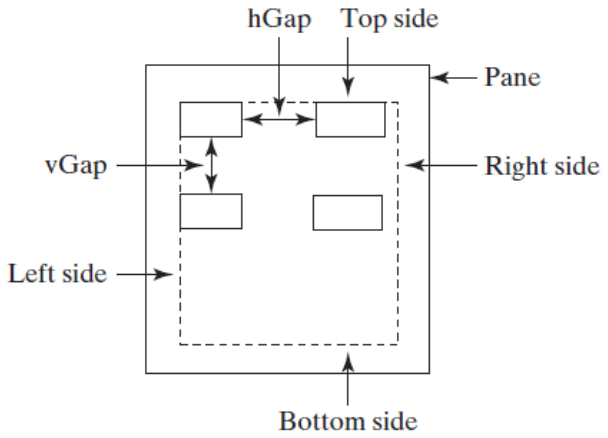
display stage

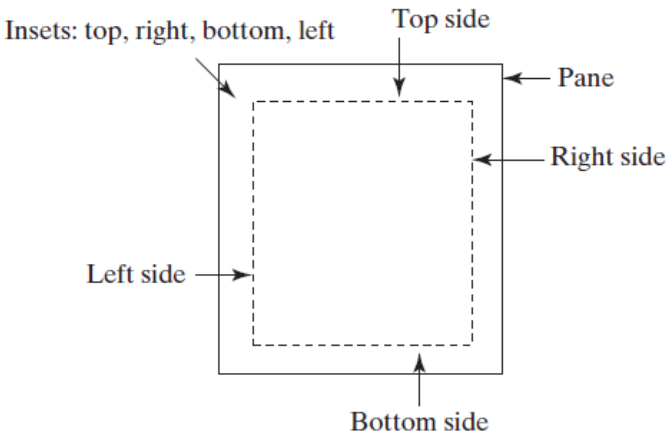
Panes

```
howFlowPane.java

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;

public class ShowFlowPane extends Applicat
  @Override // Override the start method
  public void start(Stage primaryStage) {
    // Create a pane and set its propertie
    FlowPane pane = new FlowPane();
    pane.setPadding(new Insets(11, 12, 13, 14));
    pane.setHgap(5);
    pane.setVgap(5);

    // Place nodes in the pane
    pane.getChildren().addAll(new Label("First Name:"),
      new TextField(), new Label("MI:"));
    TextField tfMi = new TextField();
    tfMi.setPrefColumnCount(1);
    pane.getChildren().addAll(tfMi, new Label("Last Name
      new TextField());

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane, 200, 250);
    primaryStage.setTitle("ShowFlowPane"); // Set the st
    primaryStage.setScene(scene); // Place the scene in
    primaryStage.show(); // Display the stage
  }
}
```

(a) (b)

The nodes fill in the rows in the `FlowPane` one after another.





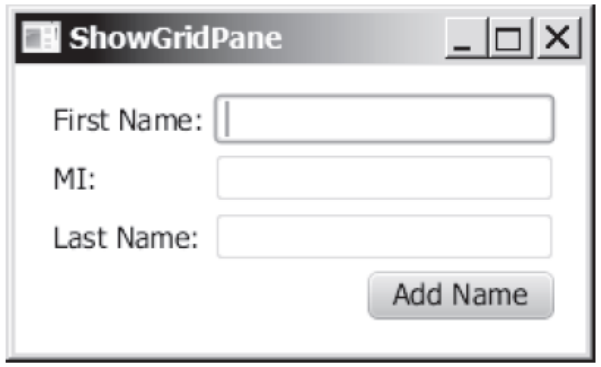
You can specify `hGap` and `vGap` between the nodes in a `FlowLPane`.

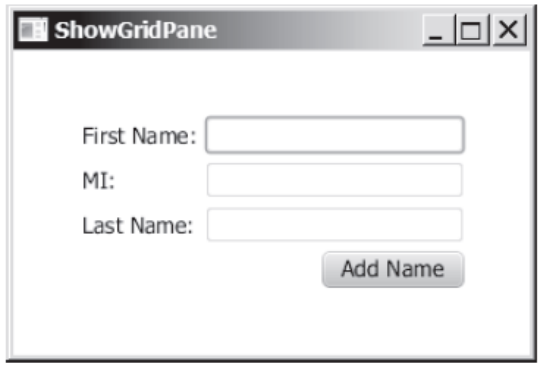
Panes

ShowGridPane.java

```
1  import javafx.application.Application;
2  import javafx.geometry.HPos;
3  import javafx.geometry.Insets;
4  import javafx.geometry.Pos;
5  import javafx.scene.Scene;
6  import javafx.scene.control.Button;
7  import javafx.scene.control.Label;
8  import javafx.scene.control.TextField;
9  import javafx.scene.layout.GridPane;
10 import javafx.stage.Stage;
11
12 public class ShowGridPane extends Application {
13   @Override // Override the start method in the Application class
14   public void start(Stage primaryStage) {
15     // Create a pane and set its properties
16     GridPane pane = new GridPane();
17     pane.setAlignment(Pos.CENTER);
18     pane.setPadding(new Insets(11.5, 12.5, 13.5, 14.5));
19     pane.setHgap(5.5);
20     pane.setVgap(5.5);
21
22     // Place nodes in the pane
23     pane.add(new Label("First Name:"), 0, 0);
24     pane.add(new TextField(), 1, 0);
25     pane.add(new Label("MI:"), 0, 1);
26     pane.add(new TextField(), 1, 1);
27     pane.add(new Label("Last Name:"), 0, 2);
28     pane.add(new TextField(), 1, 2);
29     Button btAdd = new Button("Add Name");
30     pane.add(btAdd, 1, 3);
31     GridPane.setHalignment(btAdd, HPos.RIGHT);
32
33     // Create a scene and place it in the stage
34     Scene scene = new Scene(pane);
35     primaryStage.setTitle("ShowGridPane"); // Set the stage title
36     primaryStage.setScene(scene); // Place the scene in the stage
37     primaryStage.show(); // Display the stage
38   }
39 }
```

create a grid pane
set properties
add label
add text field
add button
align button right
create a scene
display stage





The `GridPane` places the nodes in a grid with a specified column and row indices.

Panes

ShowBorderPane.java

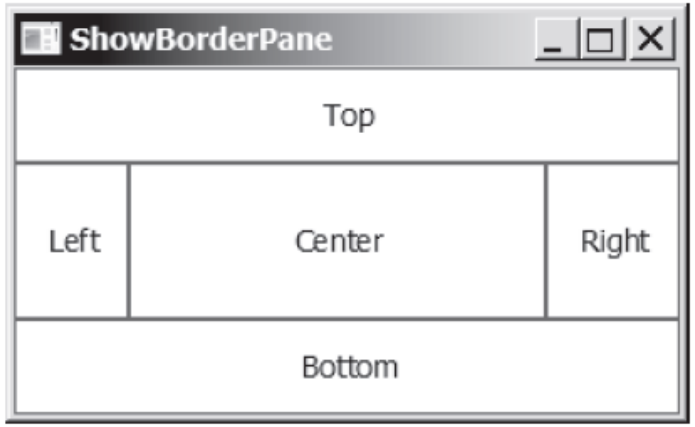
```
1  import javafx.application.Application;
2  import javafx.geometry.Insets;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Label;
5  import javafx.scene.layout.BorderPane;
6  import javafx.scene.layout.StackPane;
7  import javafx.stage.Stage;
8
9  public class ShowBorderPane extends Appli
10   @Override // Override the start method
11   public void start(Stage primaryStage) {
12     // Create a border pane
13     BorderPane pane = new BorderPane();
14
15     // Place nodes in the pane
16     pane.setTop(new CustomPane("Top"));
17     pane.setRight(new CustomPane("Right"));
18     pane.setBottom(new CustomPane("Bottom"));
19     pane.setLeft(new CustomPane("Left"));
20     pane.setCenter(new CustomPane("Center"));
21
22     // Create a scene and place it in the stage
23     Scene scene = new Scene(pane);
24     primaryStage.setTitle("ShowBorderPane"); // Set the stage title
25     primaryStage.setScene(scene); // Place the scene in the stage
26     primaryStage.show(); // Display the stage
27   }
28 }
29
30 // Define a custom pane to hold a label in the center of the pane
31 class CustomPane extends StackPane {
32   public CustomPane(String title) {
33     getChildren().add(new Label(title));
34     setStyle("-fx-border-color: red");
35     setPadding(new Insets(11.5, 12.5, 13.5, 14.5));
36   }
37 }
```

create a border pane

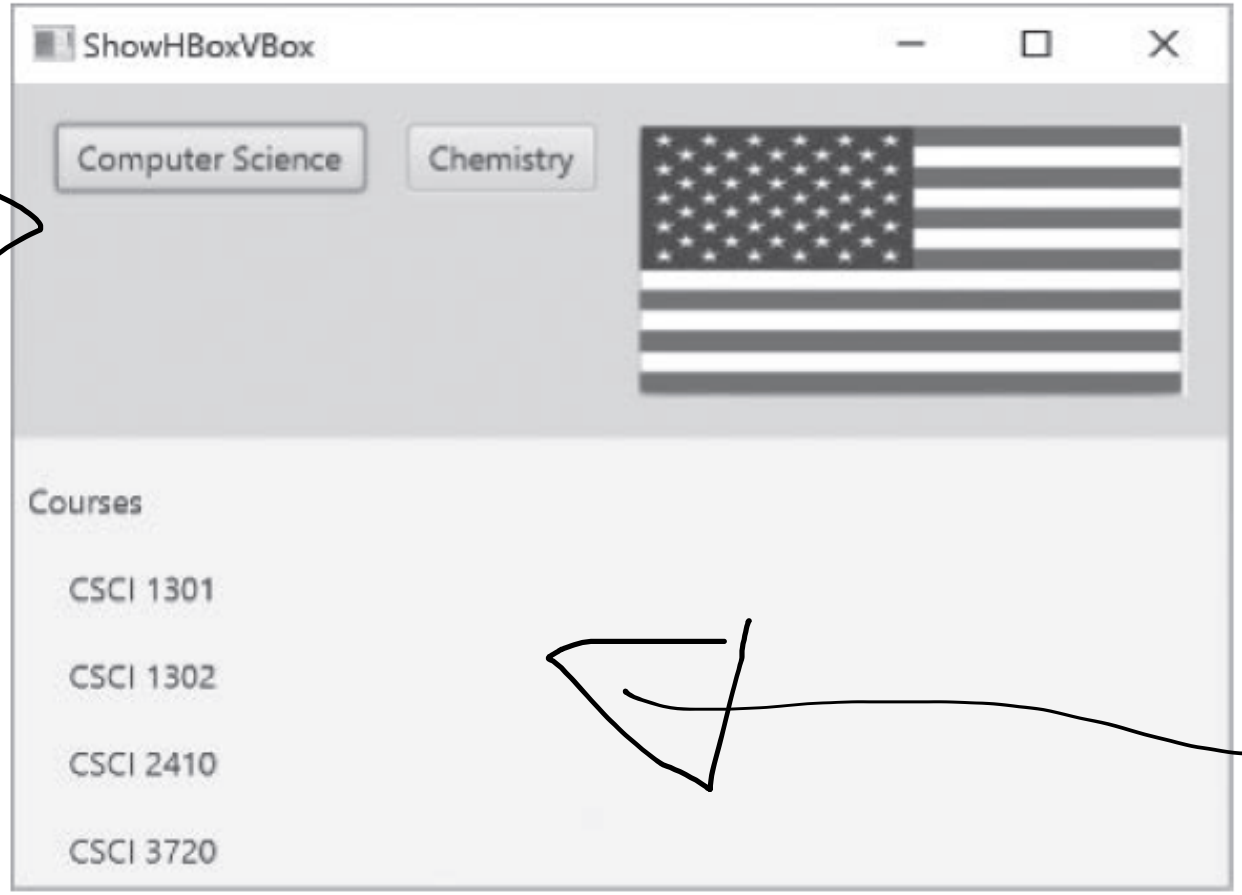
add to top
add to right
add to bottom
add to left
add to center

define a custom pane

add a label to pane
set style
set padding



The BorderPane places the nodes in five regions of the pane.



The **HBox** places the nodes in one row, and the **VBox** places the nodes in one column.

ShowHBoxVBox.java

```
1  import javafx.application.Application;
2  import javafx.geometry.Insets;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Button;
5  import javafx.scene.control.Label;
6  import javafx.scene.layout.BorderPane;
7  import javafx.scene.layout.HBox;
8  import javafx.scene.layout.VBox;
9  import javafx.stage.Stage;
10 import javafx.scene.image.Image;
11 import javafx.scene.image.ImageView;
12
13 public class ShowHBoxVBox extends Application {
14   @Override // Override the start method in the Application class
15   public void start(Stage primaryStage) {
16     // Create a border pane
17     BorderPane pane = new BorderPane();
18
19     // Place nodes in the pane
20     pane.setTop(getHBox());
21     pane.setLeft(getVBox());
22
23     // Create a scene and place it in the stage
24     Scene scene = new Scene(pane);
25     primaryStage.setTitle("ShowHBoxVBox"); // Set the stage title
26     primaryStage.setScene(scene); // Place the scene in the stage
27     primaryStage.show(); // Display the stage
28   }
```

create a border pane

add an HBox to top

add a VBox to left

create a scene

display stage

```
                                29
getHBox                         30    private HBox getHBox() {
                                31      HBox hBox = new HBox(15);
                                32      hBox.setPadding(new Insets(15, 15, 15, 15));
                                33      hBox.setStyle("-fx-background-color: gold");
add buttons to HBox             34      hBox.getChildren().add(new Button("Computer Science"));
                                35      hBox.getChildren().add(new Button("Chemistry"));
                                36      ImageView imageView = new ImageView(new Image("image/us.gif"));
                                37      hBox.getChildren().add(imageView);
return an HBox                  38      return hBox;
                                39    }
                                40
getVBox                         41    private VBox getVBox() {
                                42      VBox vBox = new VBox(15);
                                43      vBox.setPadding(new Insets(15, 5, 5, 5));
add a label                     44      vBox.getChildren().add(new Label("Courses"));
                                45
                                46      Label[] courses = {new Label("CSCI 1301"), new Label("CSCI 1302"),
                                47          new Label("CSCI 2410"), new Label("CSCI 3720")};
                                48
                                49      for (Label course: courses) {
set margin                      50        VBox.setMargin(course, new Insets(0, 0, 0, 15));
add a label                     51        vBox.getChildren().add(course);
                                52      }
                                53
return vBox                     54      return vBox;
                                55    }
                                56  }
```

The `Color` Class

The `Color` class can be used to create colors.

JavaFX defines the abstract `Paint` class for painting a node. The `javafx.scene.paint.Color` is a concrete subclass of `Paint`.

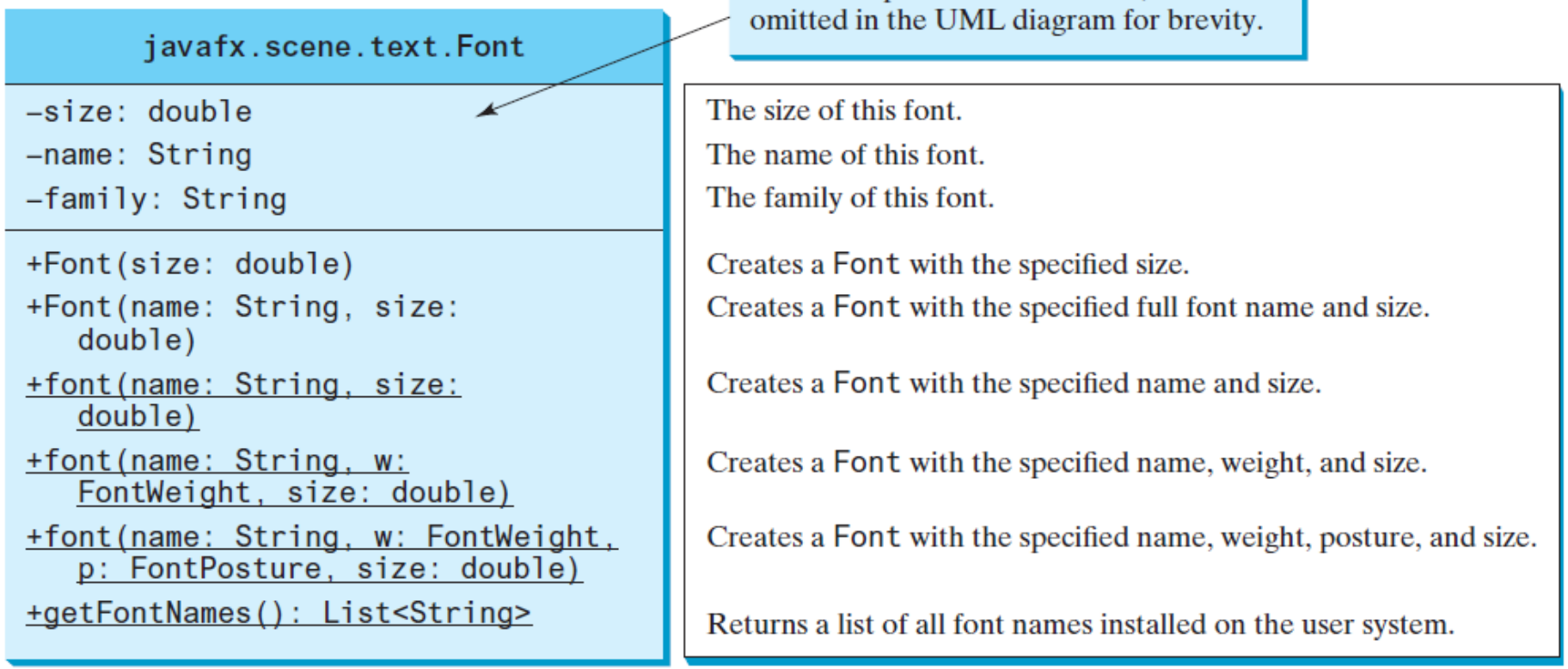
The `getter` methods for property values are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.paint.Color	
-red: double	The red value of this `color` (between 0.0 and 1.0).
-green: double	The green value of this `color` (between 0.0 and 1.0).
-blue: double	The blue value of this `color` (between 0.0 and 1.0).
-opacity: double	The opacity of this `color` (between 0.0 and 1.0).
+Color(r: double, g: double, b: double, opacity: double)	Creates a `Color` with the specified red, green, blue, and opacity values.
+brighter(): Color	Creates a `Color` that is a brighter version of this Color.
+darker(): Color	Creates a `Color` that is a darker version of this Color.
+color(r: double, g: double, b: double): Color	Creates an opaque `Color` with the specified red, green, and blue values.
+color(r: double, g: double, b: double, opacity: double): Color	Creates a `Color` with the specified red, green, blue, and opacity values.
+rgb(r: int, g: int, b: int): Color	Creates a `Color` with the specified red, green, and blue values in the range from 0 to 255.
+rgb(r: int, g: int, b: int, opacity: double): Color	Creates a `Color` with the specified red, green, and blue values in the range from 0 to 255 and a given opacity.

The Font Class

A `Font` *describes font name, weight, and size.*

```
Font font1 = new Font("SansSerif", 16);
Font font2 = Font.font("Times New Roman", FontWeight.BOLD,
  FontPosture.ITALIC, 12);
```

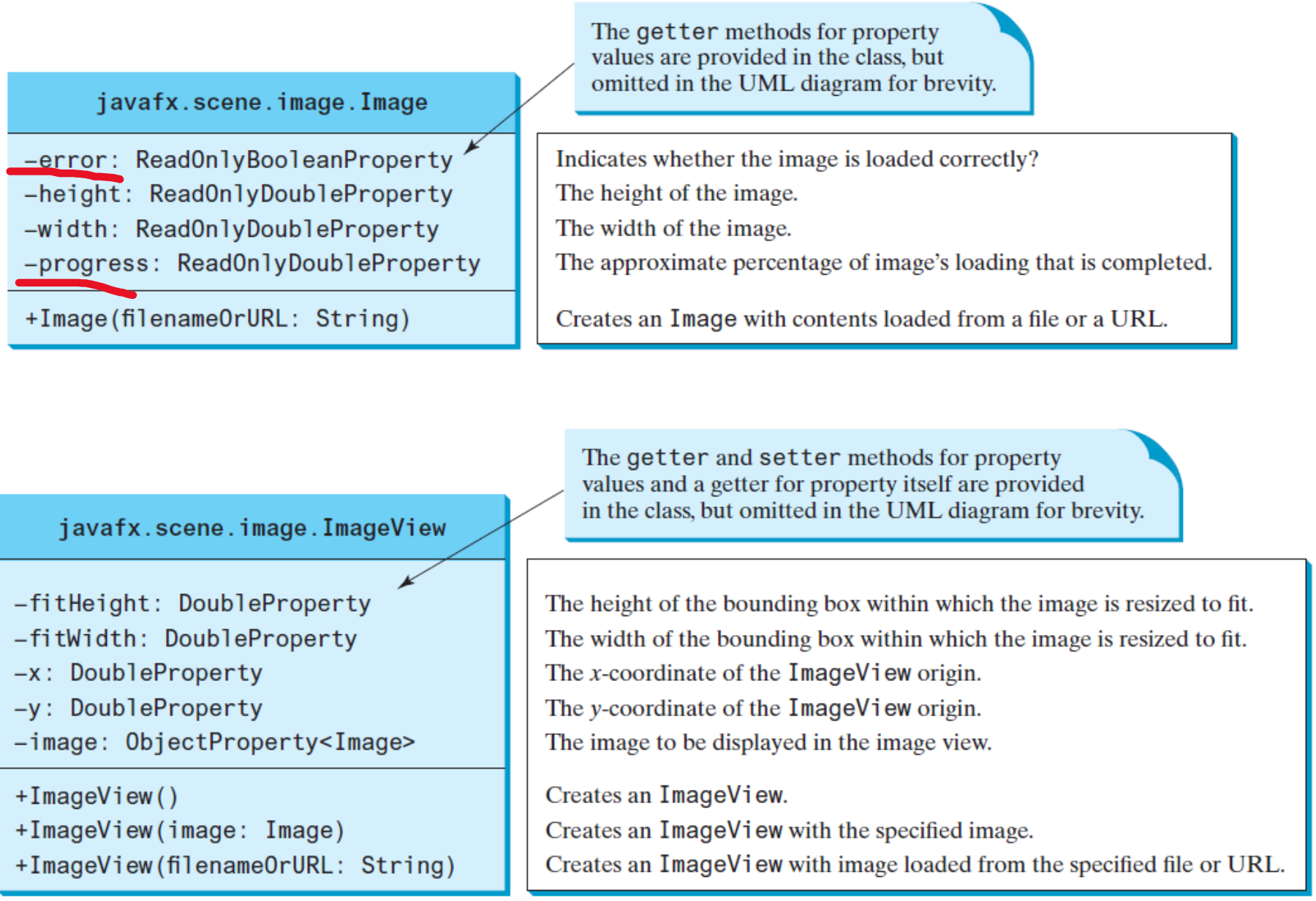


The `Image` and `ImageView` Classes

The `Image` *class represents a graphical image, and the* `ImageView` *class can be used to display an image.*

```
Image image = new Image("image/us.gif");
ImageView imageView = new ImageView(image);
```

```
ImageView imageView = new ImageView("image/us.gif");
```

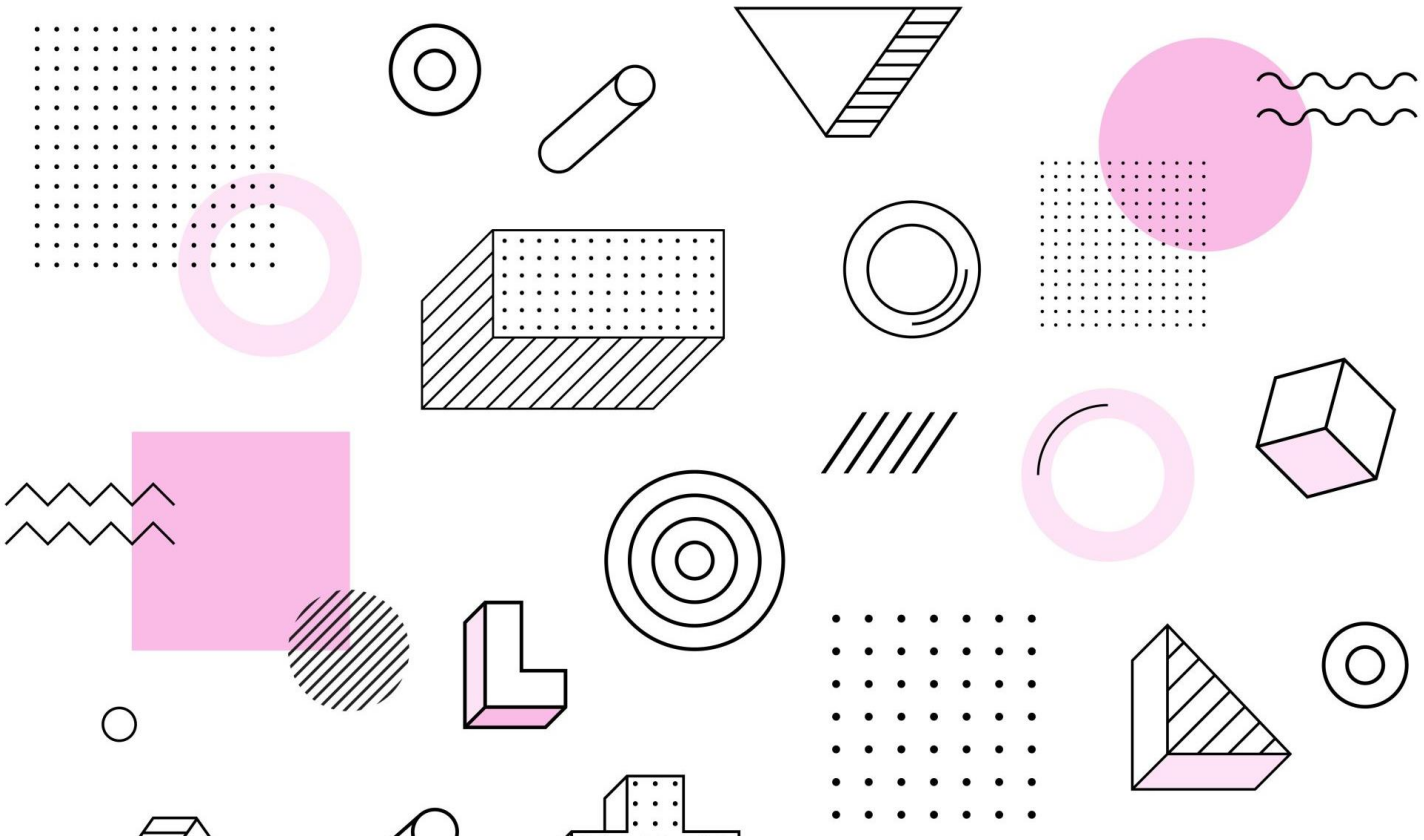

javafx.scene.image.Image
-error: ReadOnlyBooleanProperty
-height: ReadOnlyDoubleProperty
-width: ReadOnlyDoubleProperty
-progress: ReadOnlyDoubleProperty
+Image(filenameOrURL: String)
The getter methods for property values are provided in the class, but omitted in the UML diagram for brevity.
Indicates whether the image is loaded correctly?
The height of the image.
The width of the image.
The approximate percentage of image's loading that is completed.
Creates an Image with contents loaded from a file or a URL.
javafx.scene.image.ImageView
-fitHeight: DoubleProperty
-fitWidth: DoubleProperty
-x: DoubleProperty
-y: DoubleProperty
-image: ObjectProperty<Image>
+ImageView()
+ImageView(image: Image)
+ImageView(filenameOrURL: String)
The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
The height of the bounding box within which the image is resized to fit.
The width of the bounding box within which the image is resized to fit.
The x-coordinate of the ImageView origin.
The y-coordinate of the ImageView origin.
The image to be displayed in the image view.
Creates an ImageView.
Creates an ImageView with the specified image.
Creates an ImageView with image loaded from the specified file or URL.

ShowImage.java

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.layout.HBox;
4  import javafx.scene.layout.Pane;
5  import javafx.geometry.Insets;
6  import javafx.stage.Stage;
7  import javafx.scene.image.Image;
8  import javafx.scene.image.ImageView;
9
10 public class ShowImage extends Application {
11   @Override // Override the start method in the Application class
12   public void start(Stage primaryStage) {
13     // Create a pane to hold the image views
14     Pane pane = new HBox(10);                          create an HBox
15     pane.setPadding(new Insets(5, 5, 5, 5));
16     Image image = new Image("image/us.gif");           create an image
17     pane.getChildren().add(new ImageView(image));      add an image view to pane
18
19     ImageView imageView2 = new ImageView(image);       create an image view
20     imageView2.setFitHeight(100);                      set image view properties
21     imageView2.setFitWidth(100);
22     pane.getChildren().add(imageView2);                add an image to pane
23
24     ImageView imageView3 = new ImageView(image);       create an image view
25     imageView3.setRotate(90);                          rotate an image view
26     pane.getChildren().add(imageView3);                add an image to pane
27
28     // Create a scene and place it in the stage
29     Scene scene = new Scene(pane);
30     primaryStage.setTitle("ShowImage"); // Set the stage title
31     primaryStage.setScene(scene); // Place the scene in the stage
32     primaryStage.show(); // Display the stage
33   }
34 }
```

Next

- More widgets and other components for controlling interactions…

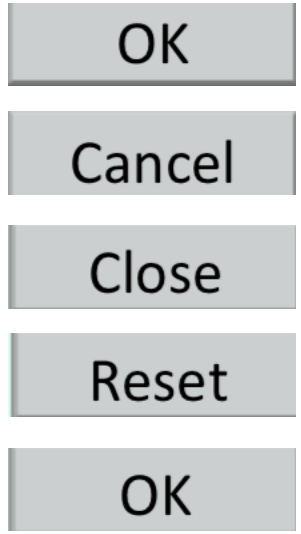


Graphical User Interface V

Various Widgets

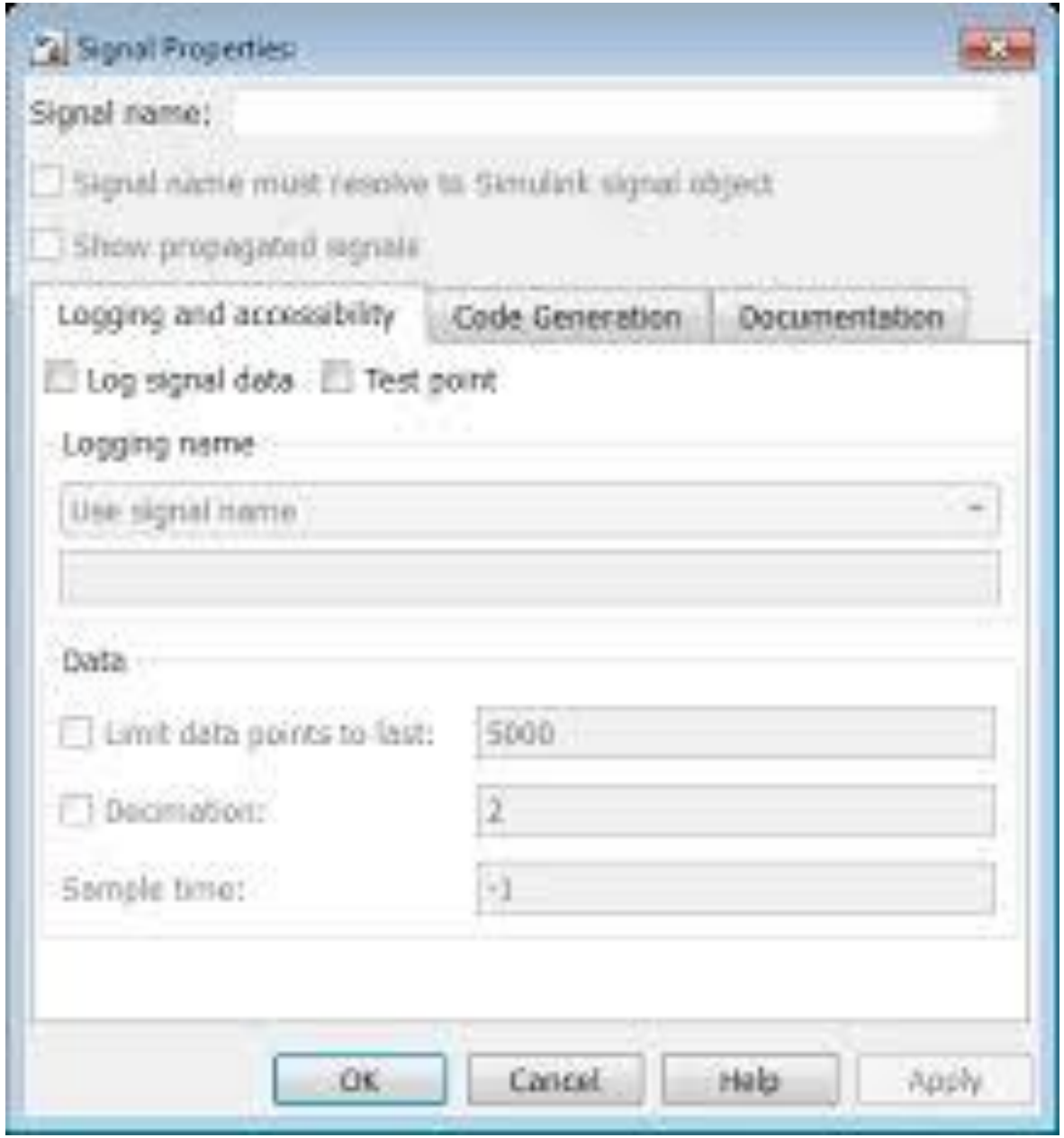
Widgets: Buttons

Common buttons and expected behavior…



Standards are important in making a GUI usable, and as a result your application, consistence also creates a professional appearance.

- Changes applied, close window
- No changes, close window
- Can’t cancel, close window
- Set default, keep window open
- Sometimes changes applied, keep window open


Signal Properties
Signal name:
Signal name must resolve to Simulink signal object
Show propagated signals
Logging and accessibility
Code Generation
Documentation
Log signal data
Test point
Logging name
Use signal name
Data
Limit data points to last:
5000
Decimation:
2
Sample time:
-1
OK
Cancel
Help
Apply

Widgets: Buttons

Keep all buttons the same size

... or have a "short button" and a "long" button size

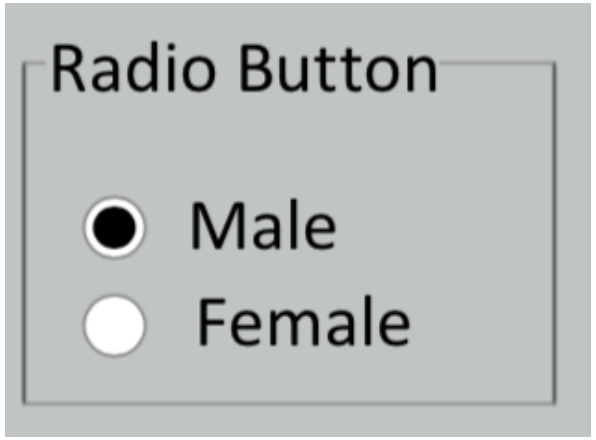
Group buttons

Isolate buttons from the rest (space)

Widgets: Radio Buttons

For several exclusive choices

Usually in a group

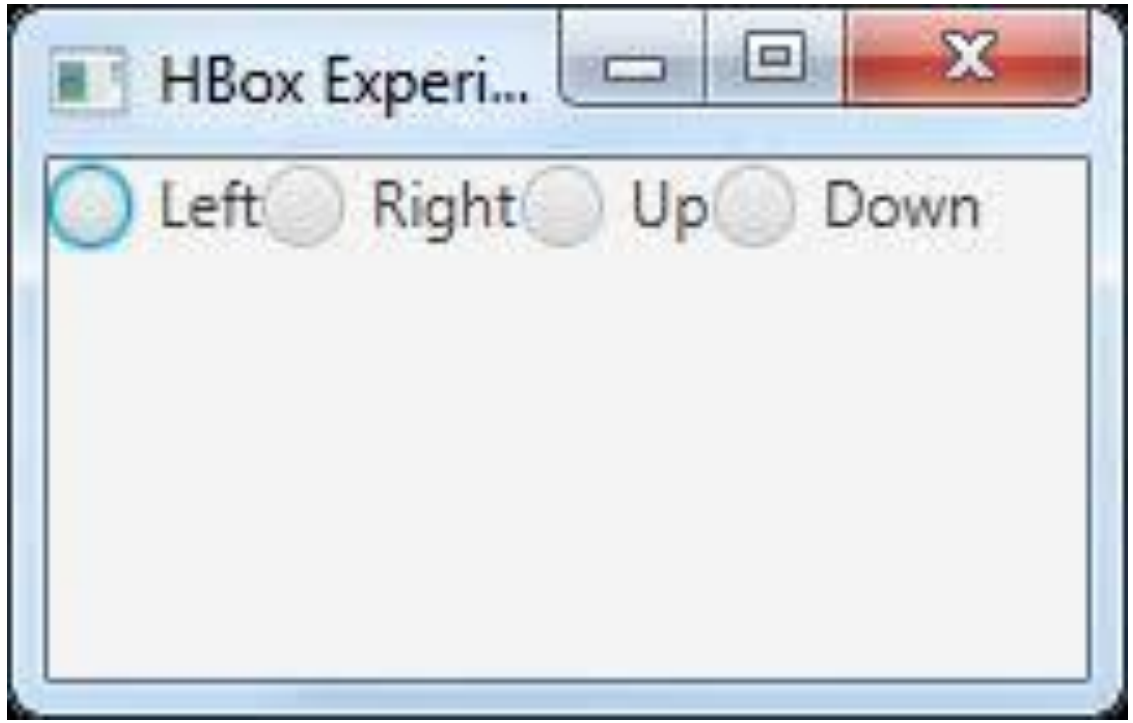


In a quiz a radio button will tell you only one answer is correct….

Toggle group object

Toggle groups can be used to make radio buttons represent a set of on off switches in which only one can be on

javafx.scene.control.ToggleGroup

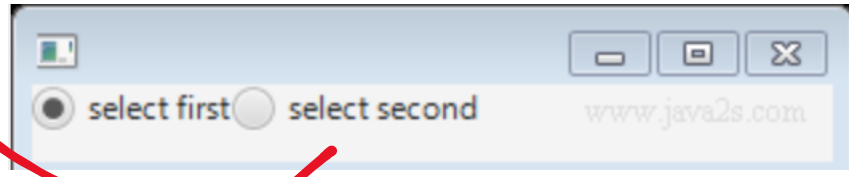


Toggle group object

Set of on off switches in which one can be on.

javafx.scene.control.ToggleGroup

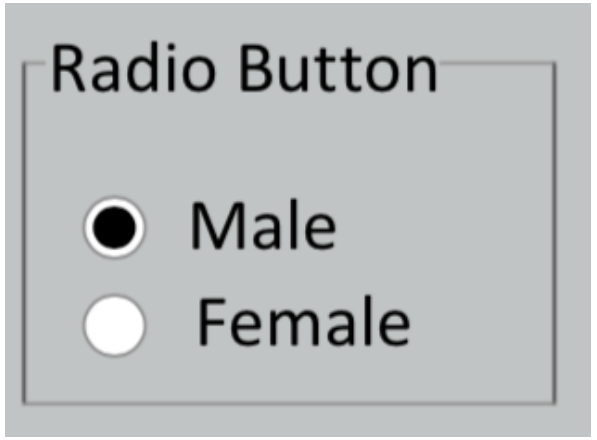
```
ToggleGroup radioGroup = new ToggleGroup();
radioButton1.setToggleGroup(radioGroup);
radioButton2.setToggleGroup(radioGroup);
radioButton3.setToggleGroup(radioGroup);
```

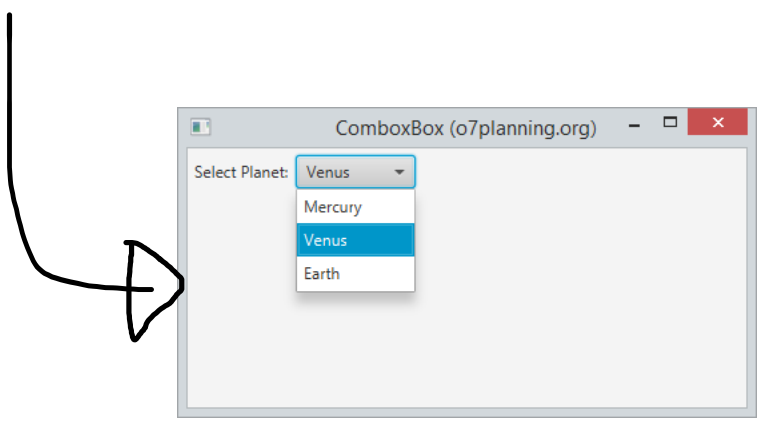



```
public void start(Stage stage) {
  HBox root = new HBox();
  Scene scene = new Scene(root, 300, 150);
  stage.setScene(scene);
  stage.setTitle("");

  ToggleGroup group = new ToggleGroup();
  RadioButton button1 = new RadioButton("select first");
  button1.setToggleGroup(group);
  button1.setSelected(true);
  RadioButton button2 = new RadioButton("select second");
  button2.setToggleGroup(group);

  root.getChildren().add(button1);
  root.getChildren().add(button2);

  scene.setRoot(root);
  stage.show();
}
```

Radio Button Widgets

- One of several exclusive choices
- Usually in a group
- Use vertically
- Six options or less
- If more than six options use a ListBox
- Avoid Yes/No or On/Off

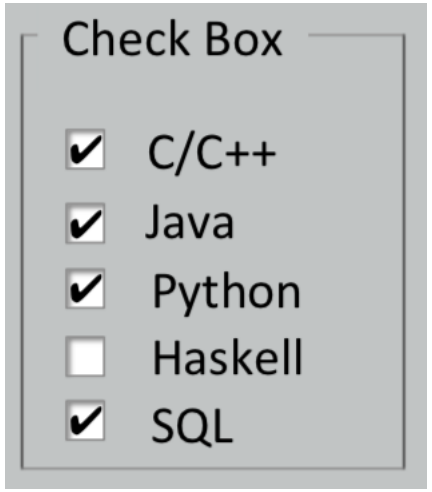




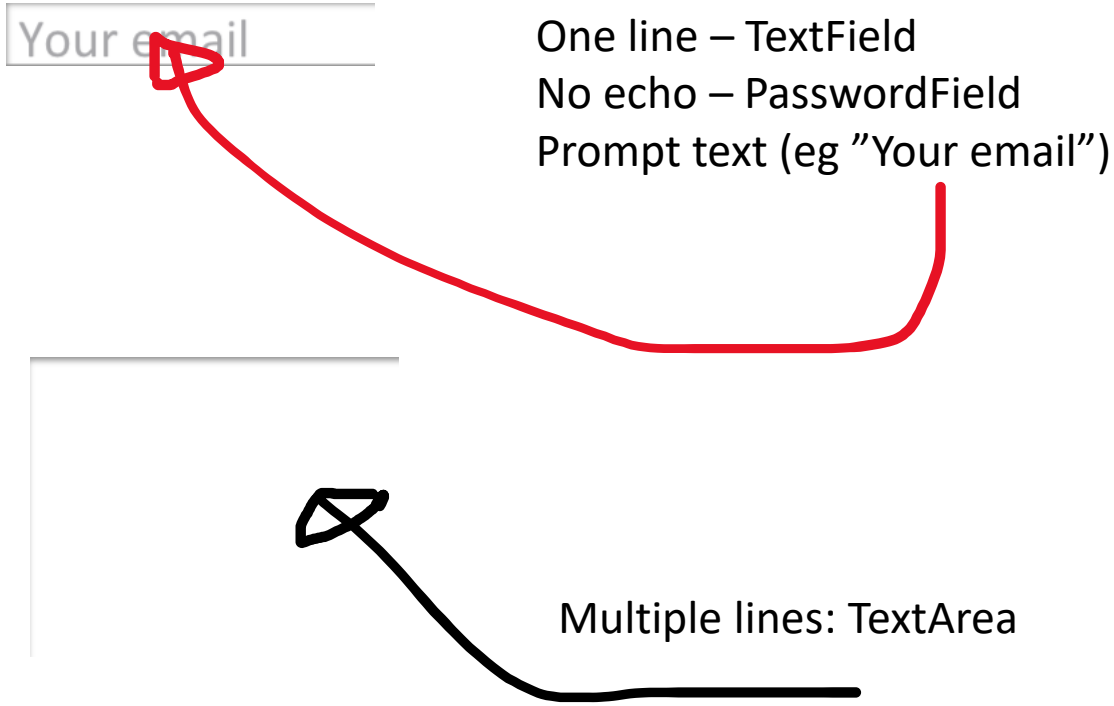
Widgets: CheckBox

- More than several options allowed
- Toggling (Yes/No or On/Off)
- Use vertically
- Ten options or less
- Button for ”select all”

- Alternative is a multiple – select ListBox

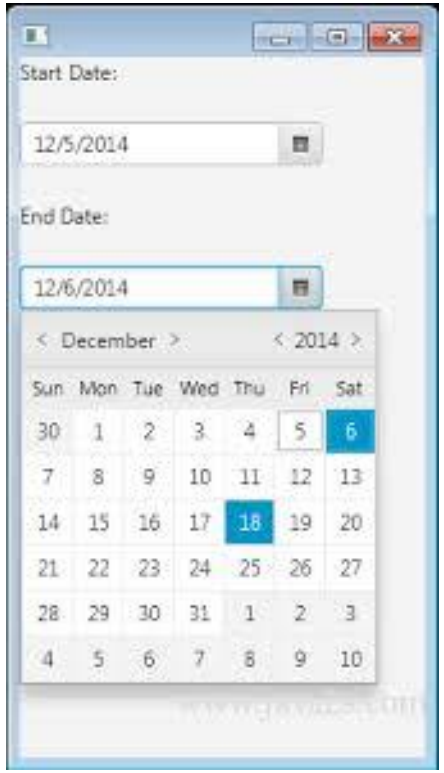


Widgets for getting data from the user

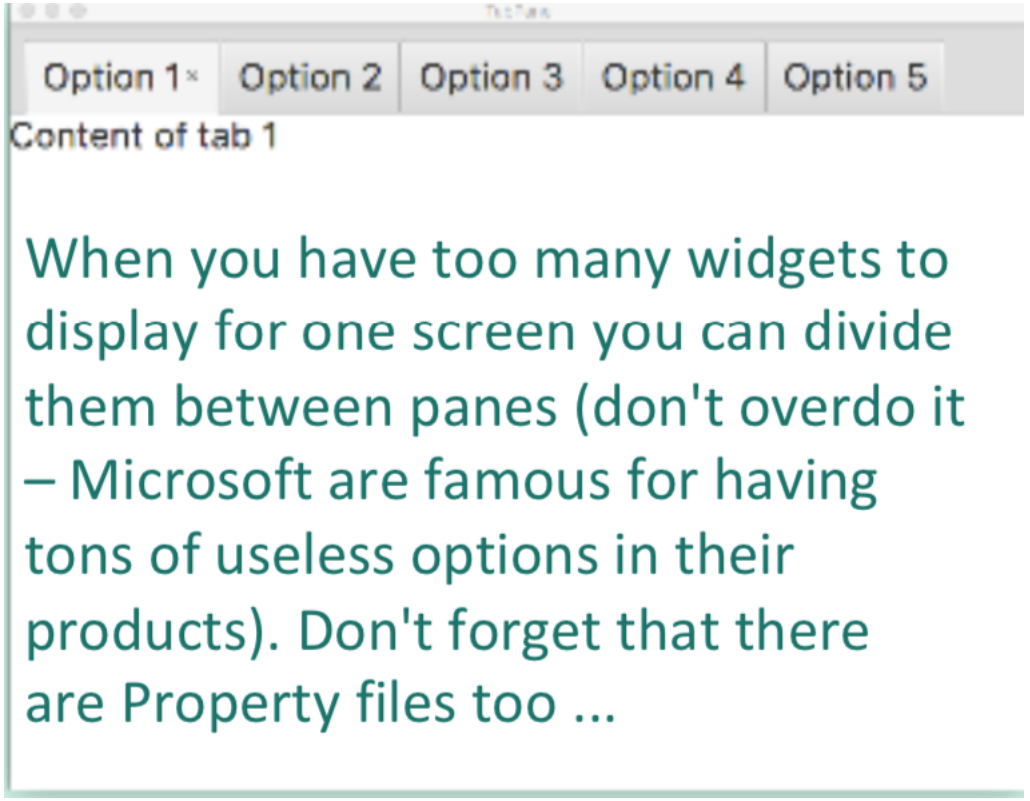


Special Widgets for Special Purposes

- ColorPicker
- DatePicker
- There are others too…



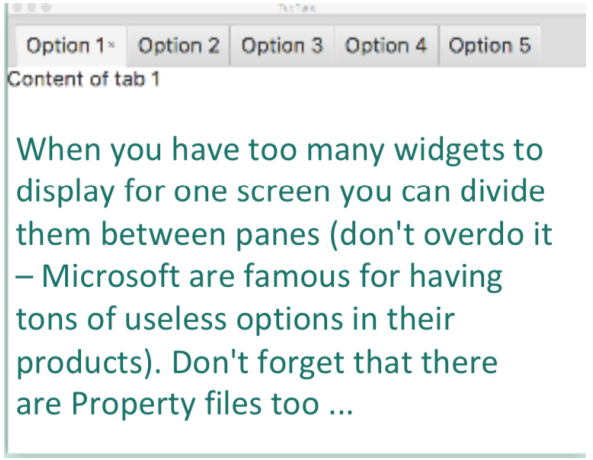
Widget TabPane



- TabPane: useful to avoid clutter

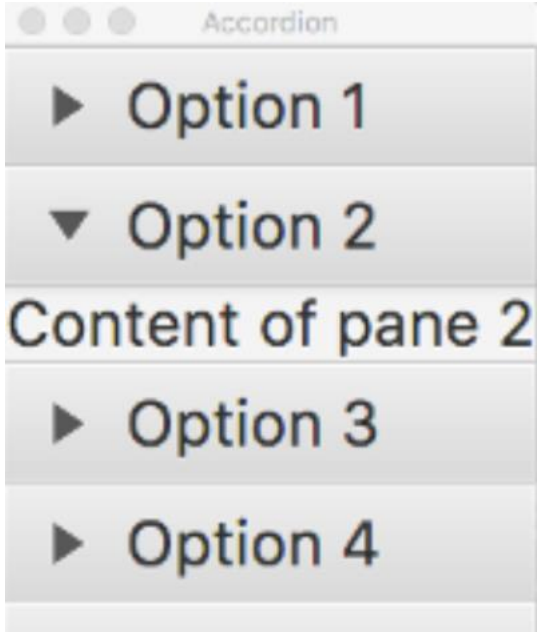
```
...
// Create a TabPane
TabPane pane = new TabPane();
pane.setPrefWidth(800);
pane.setPrefHeight(600);
root.getChildren().add(pane);
Tab tab;


// Create five tabs
for (int i = 1; i <= 5; i++) {
            tab = new Tab();
            tab.setText("Option " + i);
            tab.setContent(new Label("Content of tab " + i));
            pane.getTabs().add(tab);
            }
```

Container of tabs



Nodes can have any container or other widget

Widget Accordion and Titled Panes



- Titled panes are added to an accordion

Too Many Widgets? Can Use Accordion and Titled Panes

```
// Create an Accordion
Accordion accordion = new Accordion();
root.getChildren().add(accordion);
TitledPane pane;


// Create five titled panes
for (int i = 1; i <= 5; i++) {
        pane = new TitledPane();
        pane.setText("Option " + i);
        pane.setContent(new Label("Content of pane " + i));
        accordion.getPanes().add(pane);
        }
```

Padding and Spacing

Padding Distance from the edge

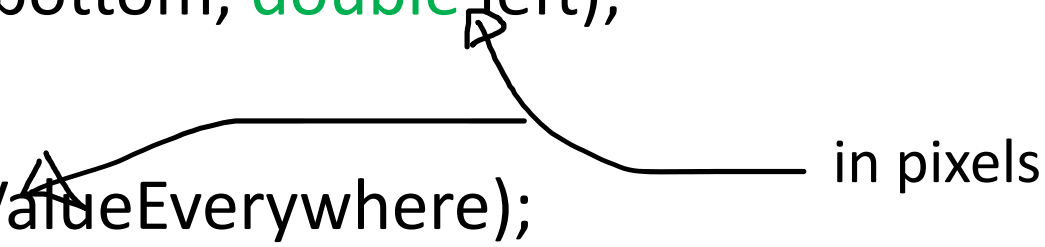
Spacing Distance between widgets

To make everything more readable, there should be “white” space. Two options, padding and spacing (which can change when you resize windows)

Padding and Spacing

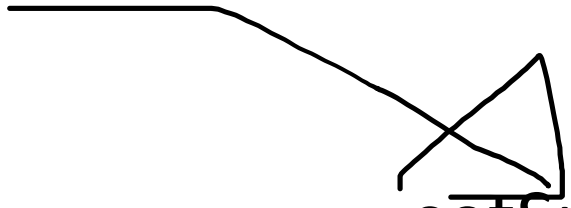
.setPadding(Insets paddingValue)

```
import javafx.geometry.Insets;
Insets(double top, double right,
                double bottom, double left);

Insets(double sameValueEverywhere);
```



Padding and Spacing

- Same distance between all the widgets in a container



.setSpacing(double spacingValue)

- Used to compute initial size
- When the window is first displayed, it may have a size you set, or the size may be computed.
- Of course, a lot of things will change in spacing if you broaden the window for instance.

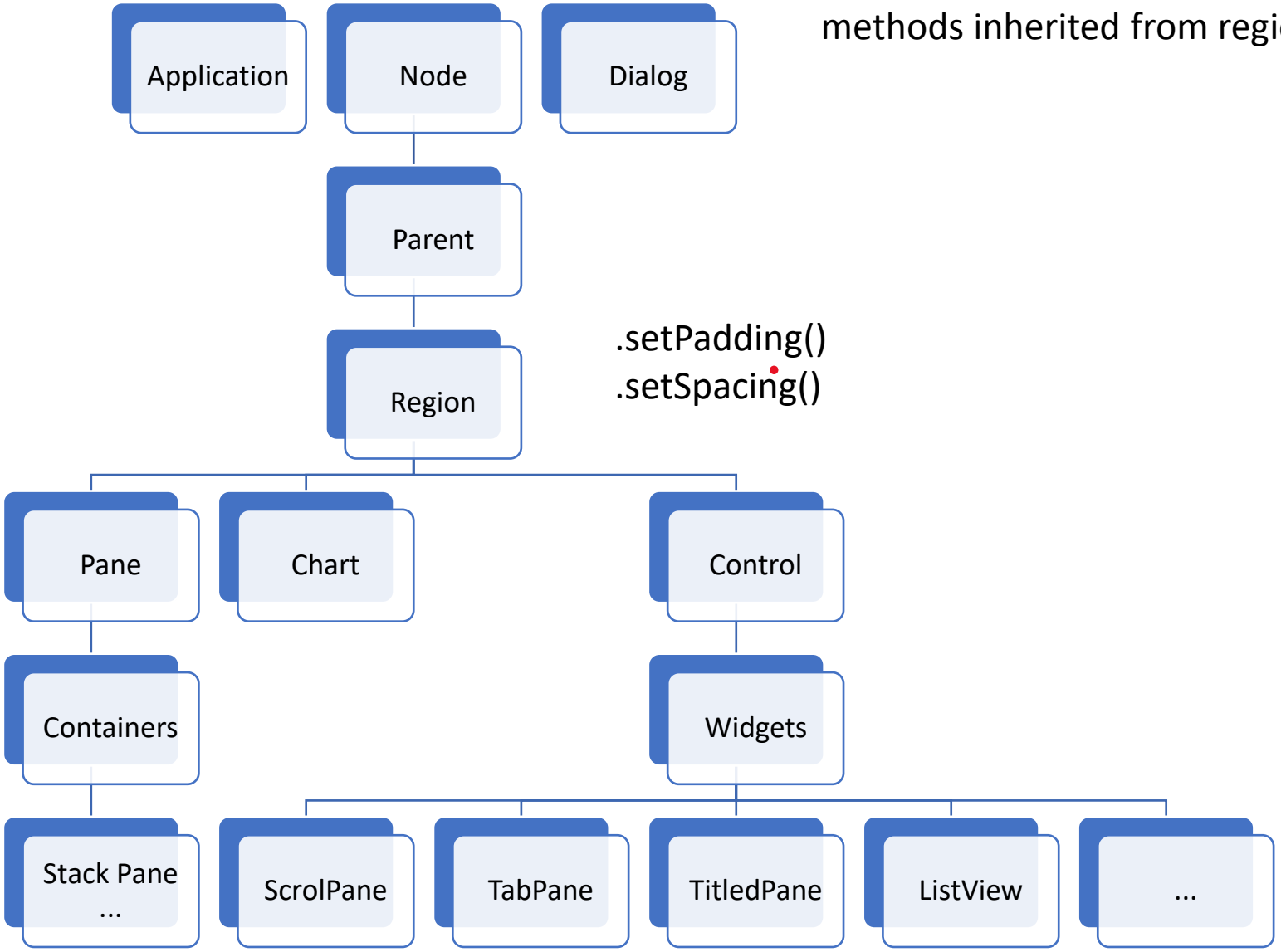
Padding and Spacing

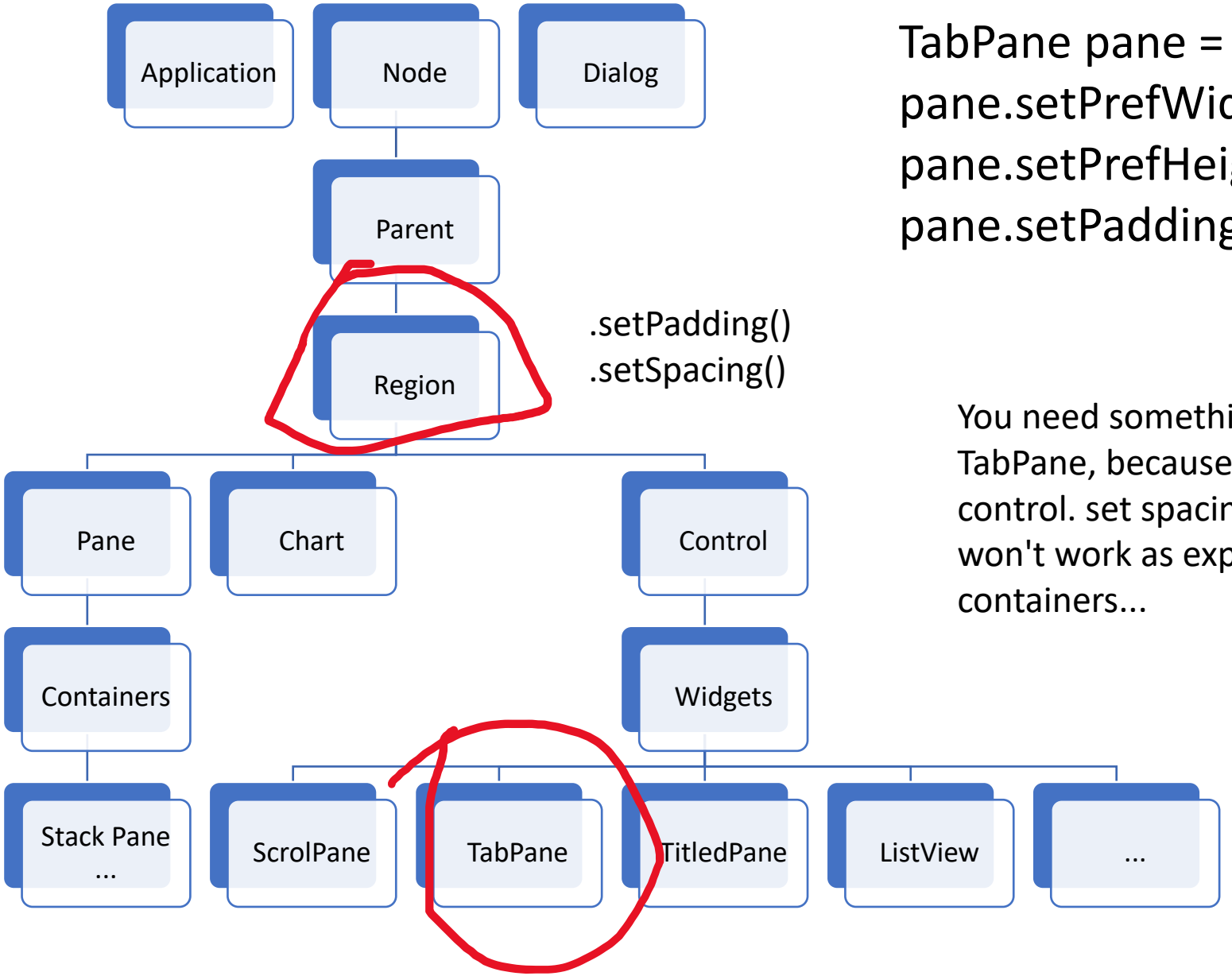
- Some containers (BorderPane, GridPane, HBox, VBox, StackPane, TilePane) implement a static method:

.setMargin(Node child, Insets marginValue)

(allows for setting spacing at the level of the individual widgets)

Be careful: padding and spacing are set with methods inherited from region



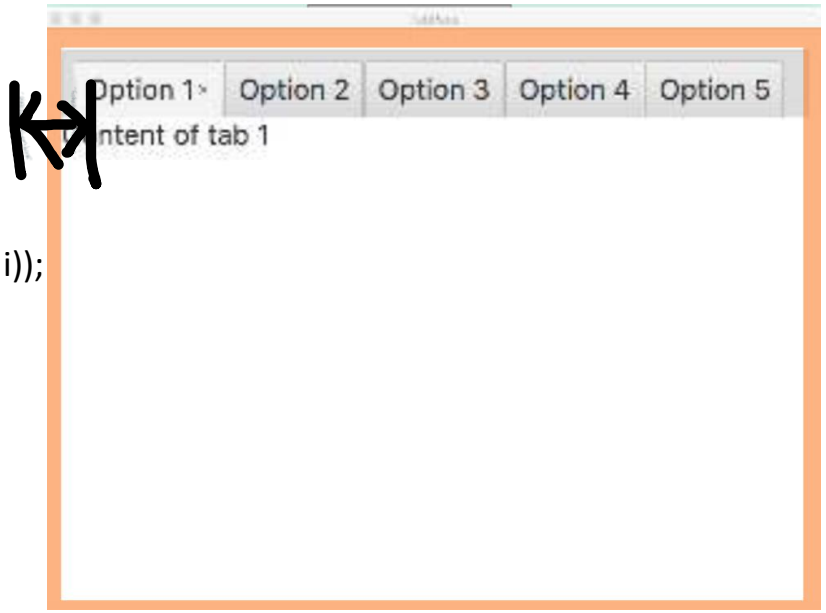



```
TabPane pane = new TabPane();
pane.setPrefWidth(800);
pane.setPrefHeight(600);
pane.setPadding(new Insets(20));
```

You need something special with a TabPane, because it's a composite control. set spacing or padding won't work as expected for containers...

Instead you should add a container (region) to the tab, eg Vbox:

```
Tab tab;
VBox tabBox;
// Create five tabs
for (int i = 1; i <= 5; i++) {
        tab = new Tab(); tab.setText("Option " + i);
        tabBox = new VBox();
        tabBox.setPadding(new Insets(20));
        tabBox.getChildren().add(new Label("Content of tab " + i));
        tab.setContent(tabBox);
        pane.getTabs().add(tab);
        }
```

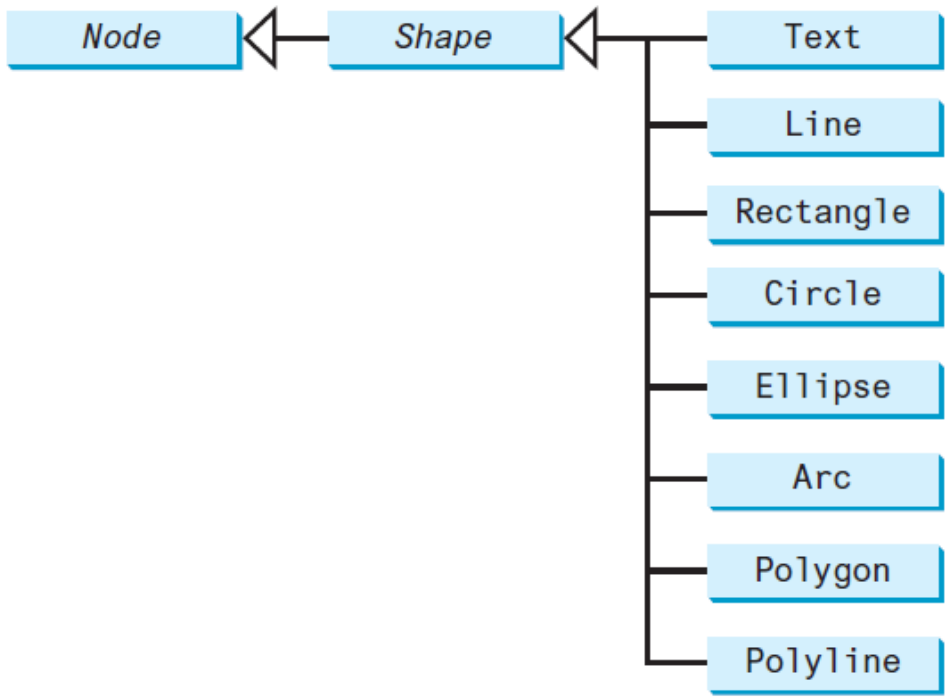


Shapes

Shapes

JavaFX provides many shape classes for drawing texts, lines, circles, rectangles, ellipses, arcs, polygons, and polylines.

The `Shape` class is the abstract base class that defines the common properties for all shapes. Among them are the `fill`, `stroke`, and `strokeWidth` properties. The `fill` property specifies a color that fills the interior of a shape. The `stroke` property specifies a color that is used to draw the outline of a shape. The `strokeWidth` property specifies the width of the outline of a shape. This section introduces the classes `Text`, `Line`, `Rectangle`, `Circle`, `Ellipse`, `Arc`, `Polygon`, and `Polyline` for drawing texts and simple shapes. All these are subclasses of `Shape`, as shown in Figure 14.25.



A shape is a node. The `Shape` class is the root of all shape classes.

The `getter` and `setter` methods for property value and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.text.Text

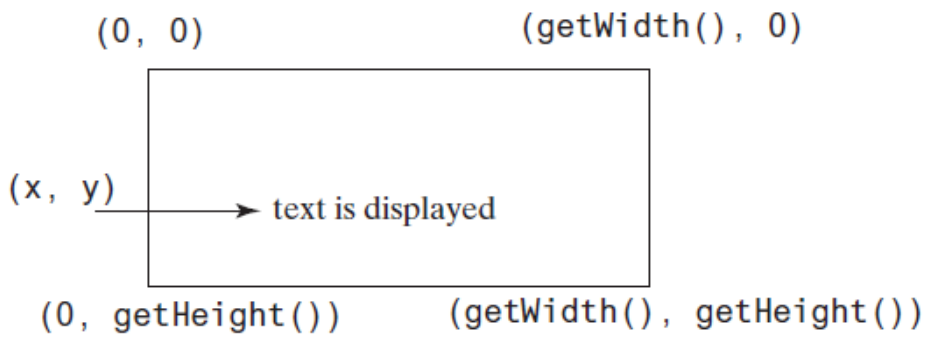
```
-text: StringProperty
-x: DoubleProperty
-y: DoubleProperty
-underline: BooleanProperty
-strikethrough: BooleanProperty
-font: ObjectProperty<Font>
```

```
+Text()
+Text(text: String)
+Text(x: double, y: double,
   text: String)
```

Defines the text to be displayed.
Defines the *x*-coordinate of text (default 0).
Defines the *y*-coordinate of text (default 0).
Defines if each line has an underline below it (default false).
Defines if each line has a line through it (default false).
Defines the font for the text.

Creates an empty `Text`.
Creates a `Text` with the specified text.
Creates a `Text` with the specified *x*-, *y*-coordinates and text.

`Text` defines a node for displaying a text.



(a) `Text`(*x*, *y*, text)

(b) Three `Text` objects are displayed

A Text object is created to display a text.

The `getter` and `setter` methods for property value and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

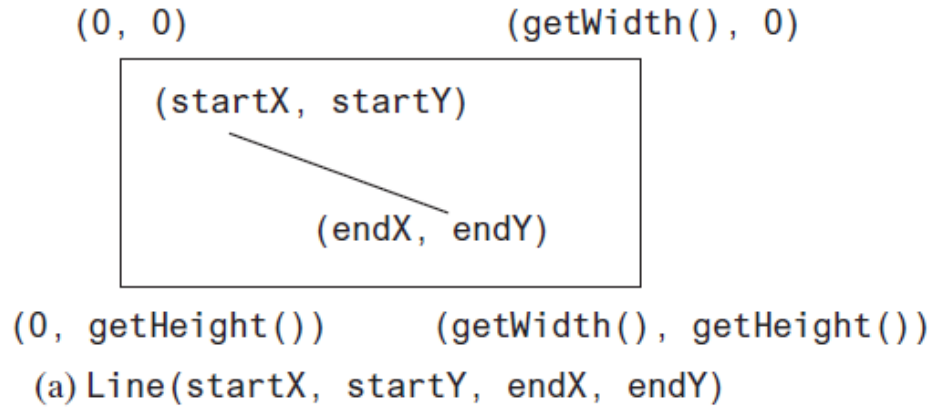
javafx.scene.shape.Line

```
-startX: DoubleProperty
-startY: DoubleProperty
-endX: DoubleProperty
-endY: DoubleProperty

+Line()
+Line(startX: double, startY:
   double, endX: double, endY:
   double)
```

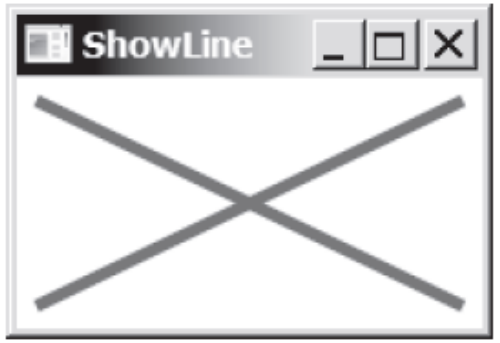
The *x*-coordinate of the start point.
The *y*-coordinate of the start point.
The *x*-coordinate of the end point.
The *y*-coordinate of the end point.

Creates an empty `Line`.
Creates a `Line` with the specified starting and ending points.

The `Line` class defines a line.



(a) `Line(startX, startY, endX, endY)`



(b) Two lines are displayed across the pane.

A `Line` object is created to display a line.

The getter and setter me thods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.shape.Rectangle

```
-x: DoubleProperty
-y: DoubleProperty
-width: DoubleProperty
-height: DoubleProperty
-arcWidth: DoubleProperty

-arcHeight: DoubleProperty
```

```
+Rectangle()
+Rectangle(x: double, y:
   double, width: double,
   height: double)
```

The *x*-coordinate of the upper-left corner of the rectangle (default 0).

The *y*-coordinate of the upper-left corner of the rectangle (default 0).

The `width` of the rectangle (default: 0).

The `height` of the rectangle (default: 0).

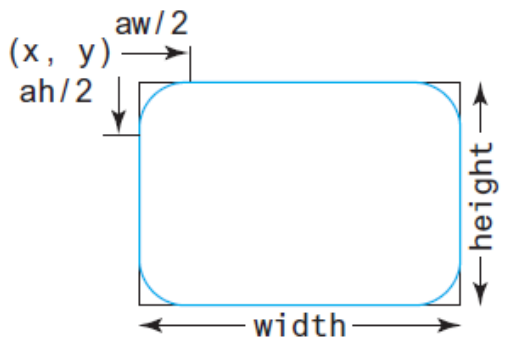
The `arcWidth` of the rectangle (default: 0). arcWidth is the horizontal diameter of the arcs at the corner (see Figure 14.31a).

The `arcWidth` of the rectangle (default: 0). arcHeight is the vertical diameter of the arcs at the corner (see Figure 14.31a).

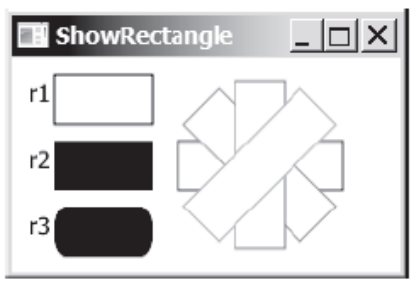
Creates an empty `Rectangle`.

Creates a `Rectangle` with the specified upper-left corner point, width, and height.

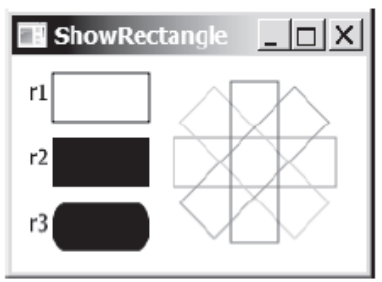
`Rectangle` defines a rectangle.



(a) `Rectangle(x, y, w, h)`

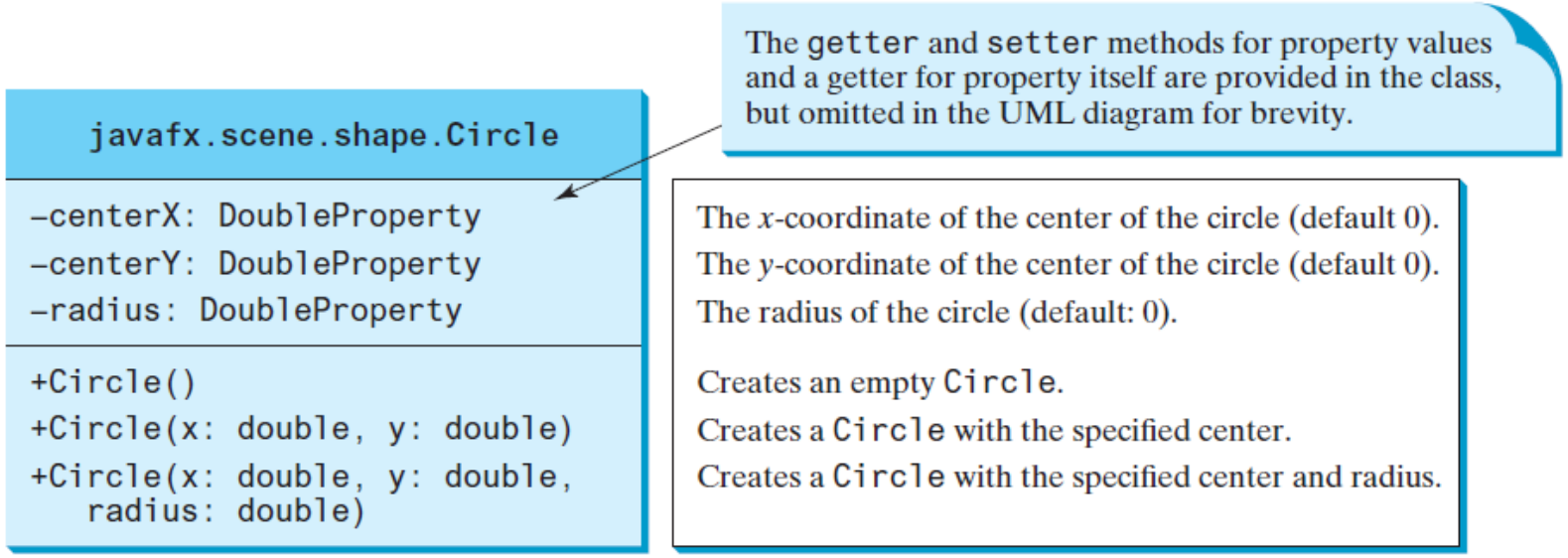


(b) Multiple rectangles are displayed.

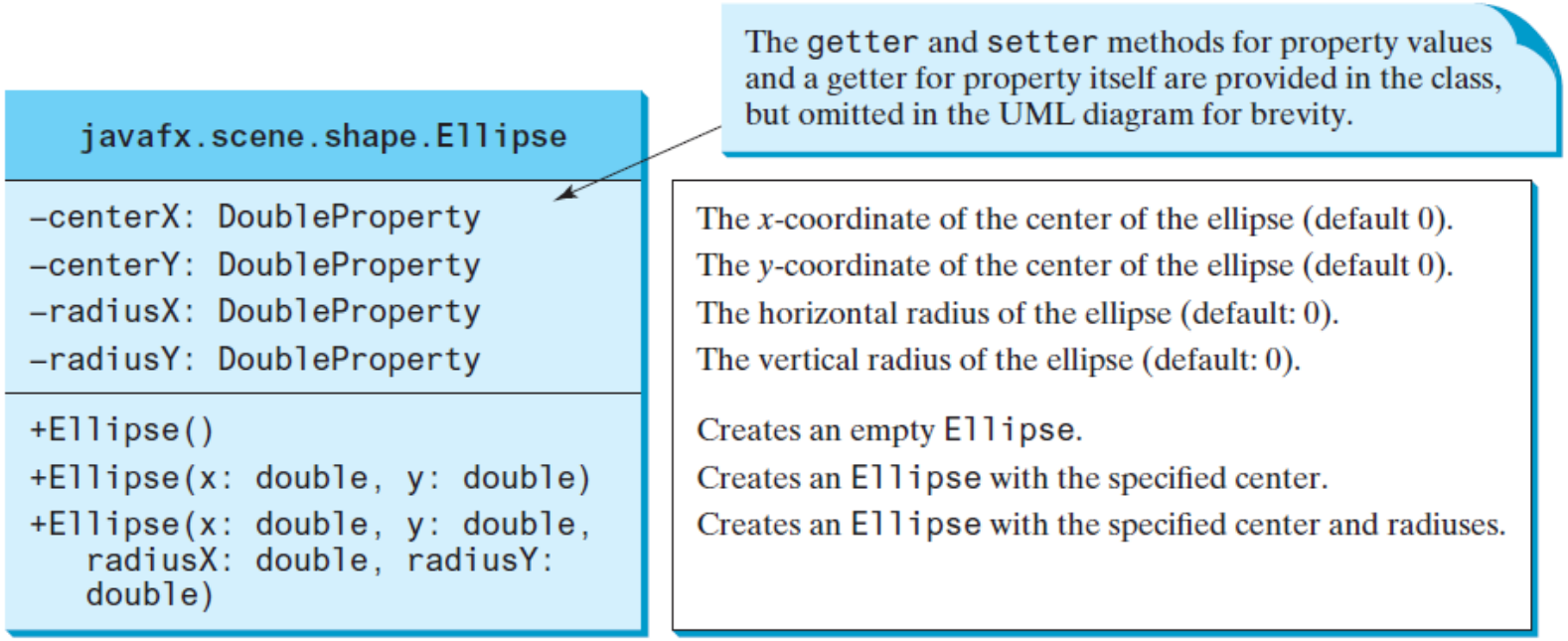


(c) Transparent rectangles are displayed.

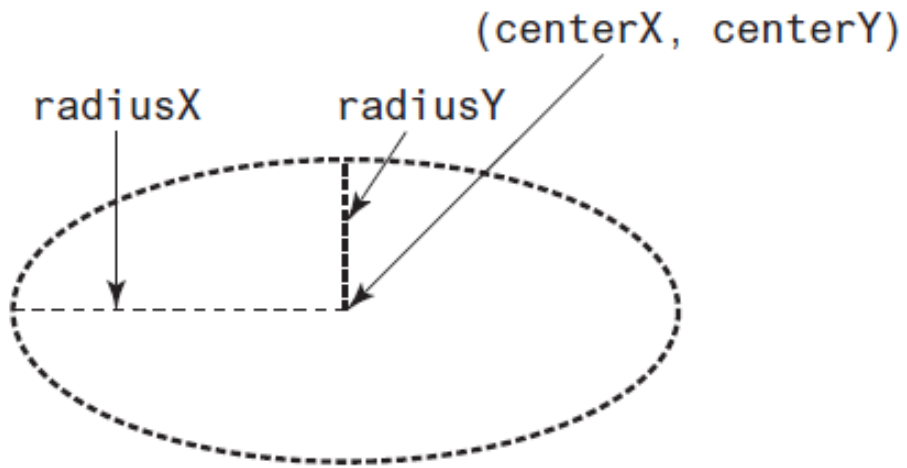
A `Rectangle` object is created to display a rectangle.



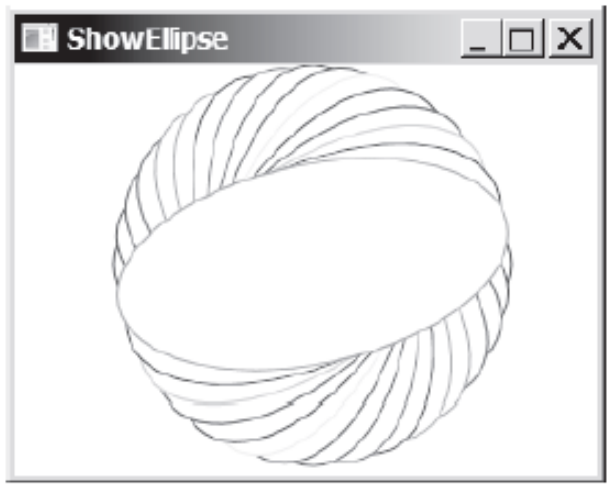
The `Circle` class defines circles.



The `Ellipse` class defines ellipses.



(a) `Ellipse(centerX, centerY, radiusX, radiusY)`



(b) Multiple ellipses are displayed.

An `Ellipse` object is created to display an ellipse.

The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.shape.Arc

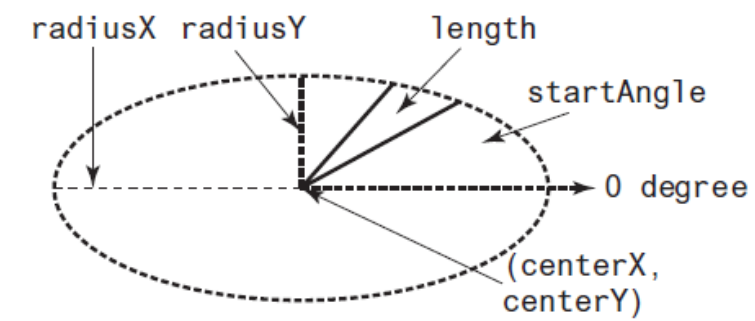
```
-centerX: DoubleProperty
-centerY: DoubleProperty
-radiusX: DoubleProperty
-radiusY: DoubleProperty
-startAngle: DoubleProperty
-length: DoubleProperty
-type: ObjectProperty<ArcType>
```

```
+Arc()
+Arc(x: double, y: double,
   radiusX: double, radiusY:
   double, startAngle: double,
   length: double)
```

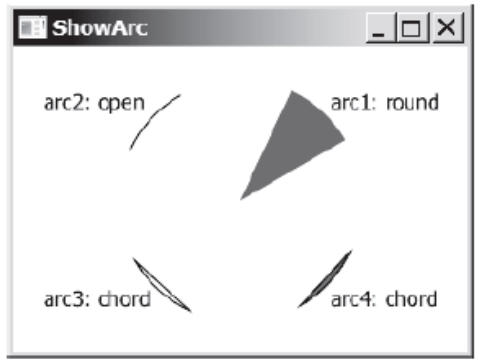
The *x*-coordinate of the center of the ellipse (default 0).
The *y*-coordinate of the center of the ellipse (default 0).
The horizontal radius of the ellipse (default: 0).
The vertical radius of the ellipse (default: 0).
The start angle of the arc in degrees.
The angular extent of the arc in degrees.
The closure type of the arc (`ArcType.OPEN`, `ArcType.CHORD`, `ArcType.ROUND`).

Creates an empty `Arc`.
Creates an `Arc` with the specified arguments.

The `Arc` class defines an arc.

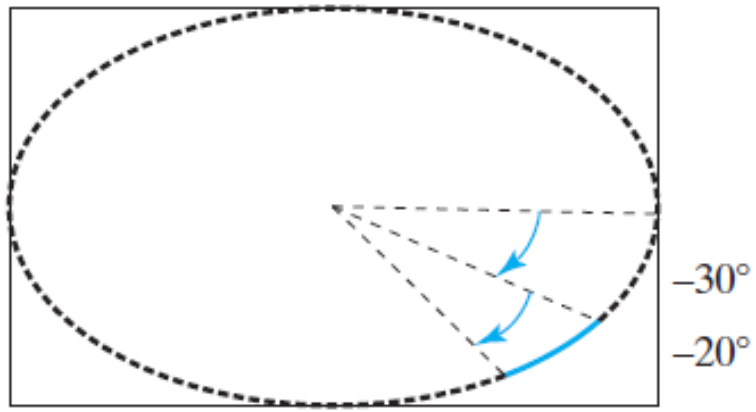


(a) `Arc(centerX, centerY, radiusX, radiusY, startAngle, length)`

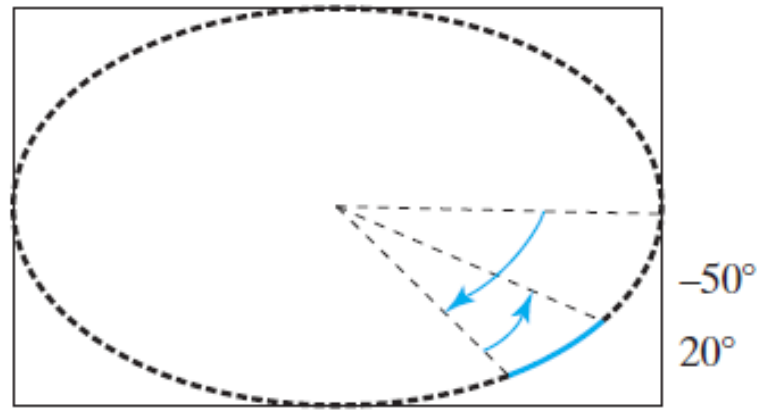


(b) Multiple ellipses are displayed.

An `Arc` object is created to display an arc.

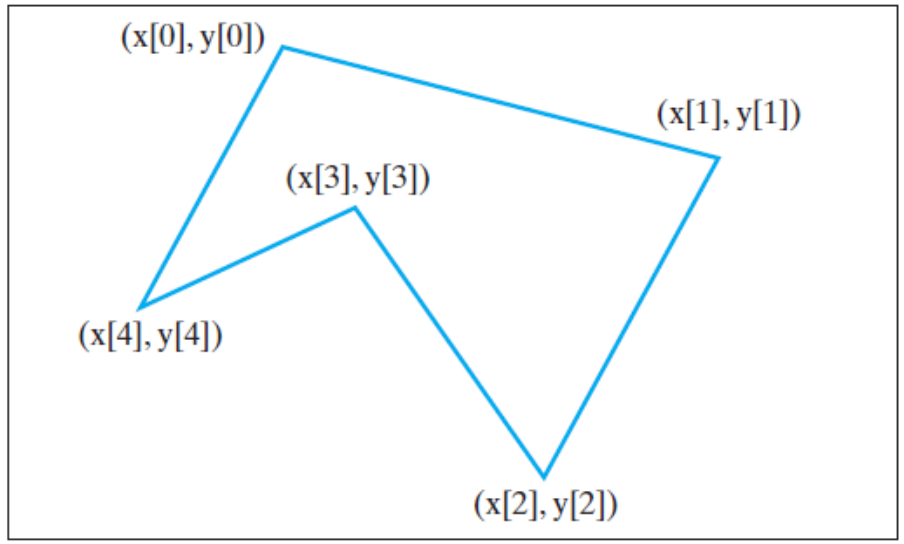


(a) Negative starting angle –30° and negative spanning angle –20°

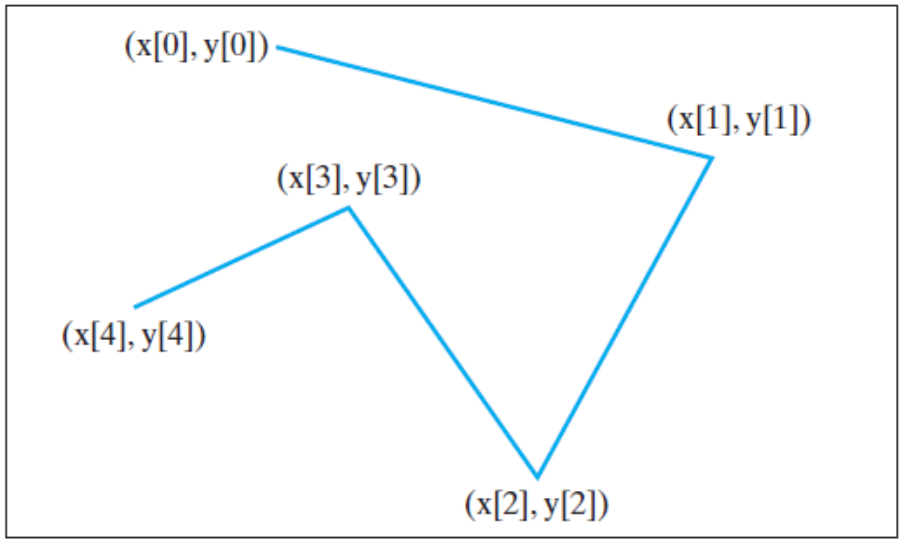


(b) Negative starting angle –50° and positive spanning angle 20°

Angles may be negative.



(a) Polygon

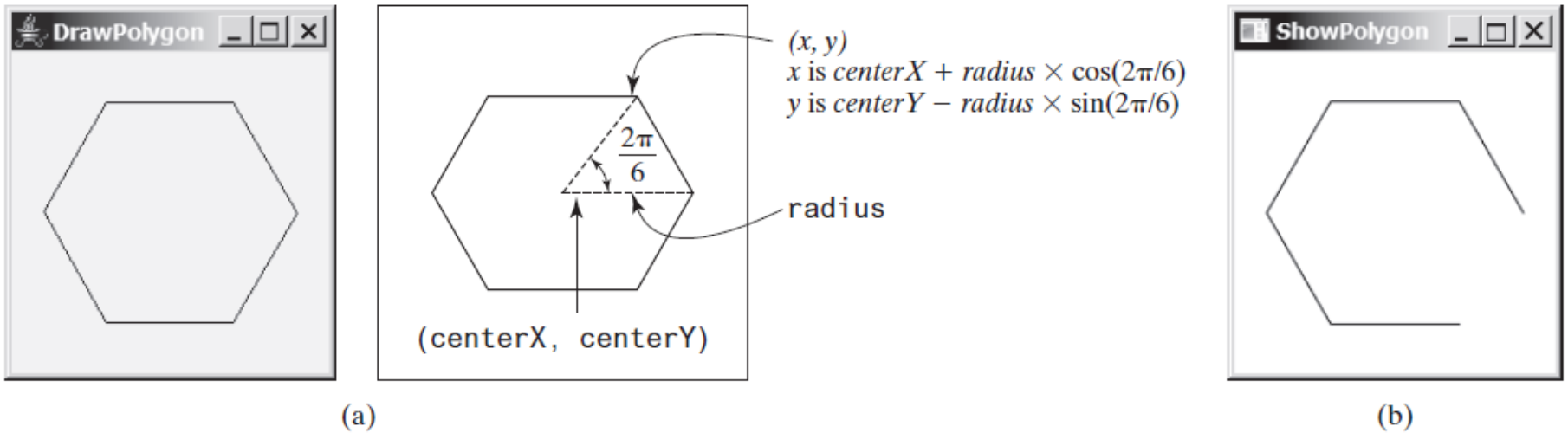


(b) Polyline

`Polygon` is closed and `Polyline` is not closed.

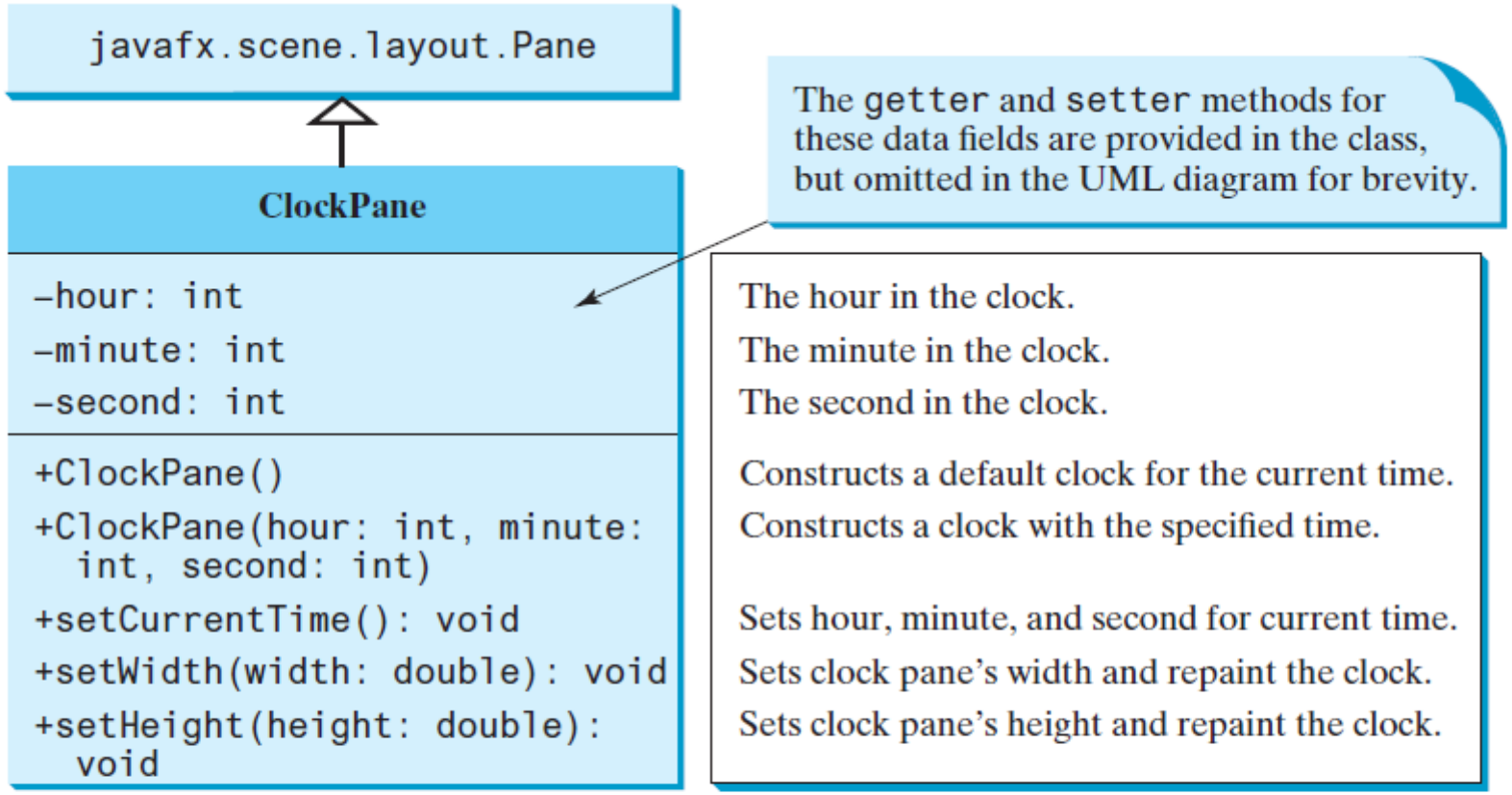
javafx.scene.shape.Polygon	
+Polygon()	Creates an empty Polygon.
+Polygon(double... points)	Creates a Polygon with the given points.
+getPoints(): ObservableList<Double>	Returns a list of double values as *x*- and *y*-coordinates of the points.

The `Polygon` class defines a polygon.



(a) A `Polygon` is displayed. (b) A `Polyline` is displayed.

If you replace `Polygon` by `Polyline` (line 23), the program displays a polyline as shown in Figure 14.40b. The `Polyline` class is used in the same way as `Polygon`, except that the starting and ending points are not connected in `Polyline`.



`ClockPane` displays an analog clock.

Assume `ClockPane` is available; we write a test program in Listing 14.20 to display an analog clock and use a label to display the hour, minute, and second, as shown in Figure 14.42.

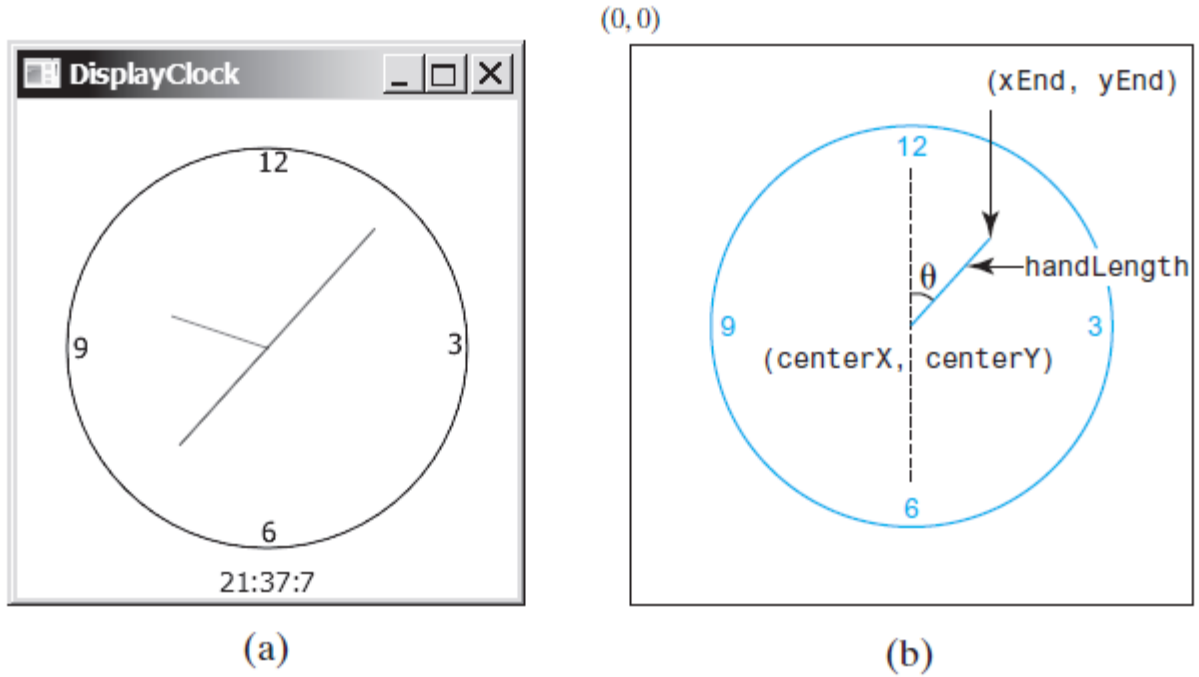


FIGURE 14.42 (a) The `DisplayClock` program displays a clock that shows the current time. (b) The endpoint of a clock hand can be determined, given the spanning angle, the hand length, and the center point.

DisplayClock.java

```
1  import javafx.application.Application;
2  import javafx.geometry.Pos;
3  import javafx.stage.Stage;
4  import javafx.scene.Scene;
5  import javafx.scene.control.Label;
6  import javafx.scene.layout.BorderPane;
7
8  public class DisplayClock extends Application {
9    @Override // Override the start method in the Application class
10   public void start(Stage primaryStage) {
11     // Create a clock and a label
12     ClockPane clock = new ClockPane();                          create a clock
13     String timeString = clock.getHour() + ":" + clock.getMinute()
14       + ":" + clock.getSecond();
15     Label lblCurrentTime = new Label(timeString);               create a label
16
17     // Place clock and label in border pane
18     BorderPane pane = new BorderPane();
19     pane.setCenter(clock);                                      add a clock
20     pane.setBottom(lblCurrentTime);                             add a label
21     BorderPane.setAlignment(lblCurrentTime, Pos.TOP_CENTER);
22
23     // Create a scene and place it in the stage
24     Scene scene = new Scene(pane, 250, 250);
25     primaryStage.setTitle("DisplayClock"); // Set the stage title
26     primaryStage.setScene(scene); // Place the scene in the stage
27     primaryStage.show(); // Display the stage
28   }
29 }
```

Event Listeners

Similar to Exceptions: “bubbles up”

Event Handlers are like “catch”

There are different rules for the system to find the target widget that has the focus for a particular events. EG cursor position for mouse events, button presses, etc. Consider issues such as if an element is hidden by another it's the one on top and it is considered to be the target.

Events and Change Listeners

1. Determine the Event Target

EventTarget interface

- Window
- Scene
- Node

Events and Change Listeners

.setOn SomeAction**()** methods for nodes:

"KeyPressed"
"KeyReleased"
"KeyTyped"
"MouseClicked"
"MouseExited"
many more...

Events and Change Listeners

.setOnAction() method for

- Buttons
- RadioButtons
- CheckBoxes

Events and Change Listeners

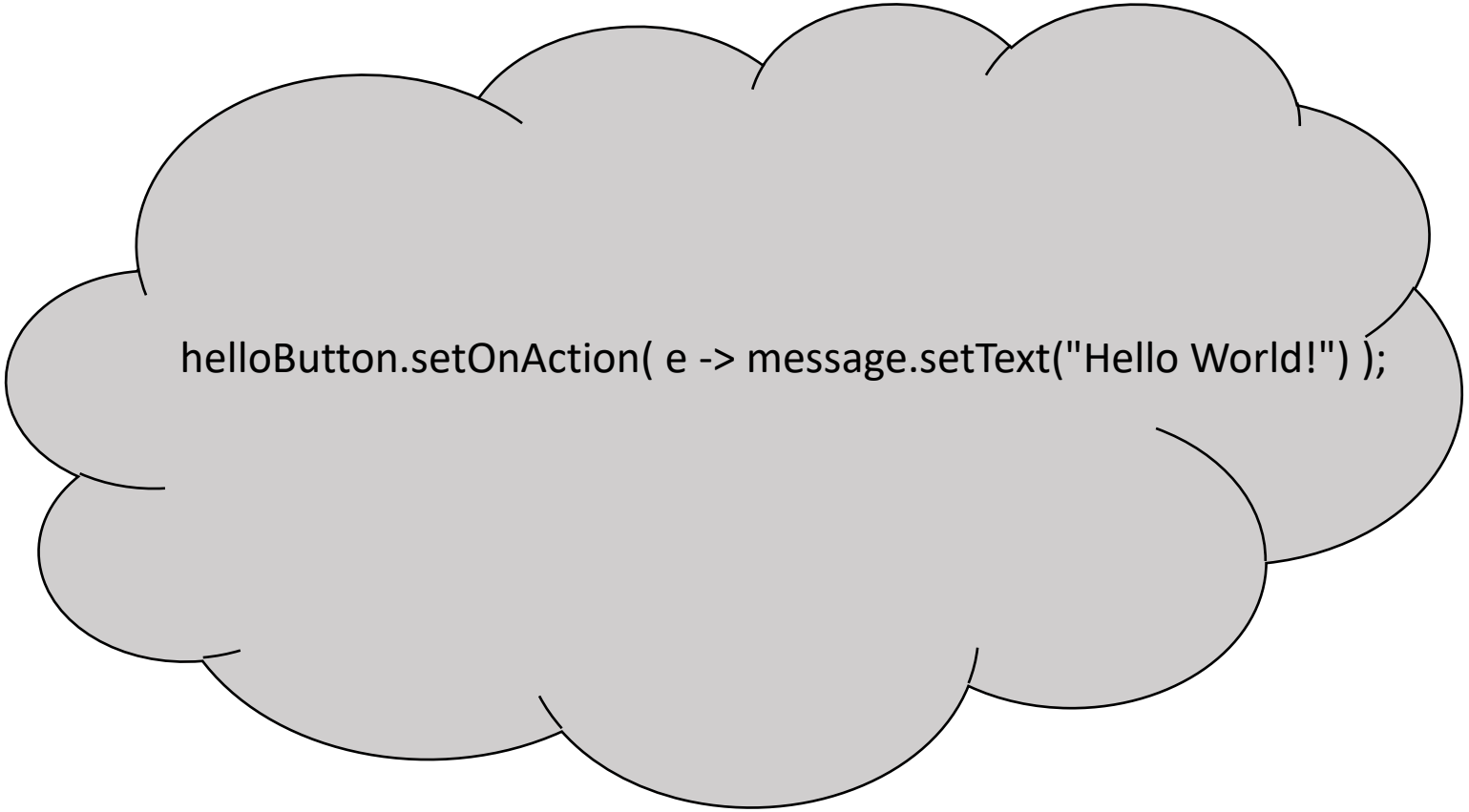
Use Lambda Expressions:

```
Button btn = new Button();
btn.setText("Say 'Hi'");
btn.setOnAction(e -> System.out.println("Hi!") );
```

```
public class HelloWorldFX extends Application {

    public void start(Stage stage) {

        Label message = new Label("First FX Application!");
        message.setFont( new Font(40) );

        Button helloButton = new Button("Say Hello");
        helloButton.setOnAction( e -> message.setText("Hello World!") );
        Button goodbyeButton = new Button("Say Goodbye");
        goodbyeButton.setOnAction( e -> message.setText("Goodbye!!") );
        Button quitButton = new Button("Quit");
        quitButton.setOnAction( e -> Platform.exit() );

        HBox buttonBar = new HBox( 20, helloButton, goodbyeButton, quitButton );
        buttonBar.setAlignment(Pos.CENTER);
        BorderPane root = new BorderPane();
        root.setCenter(message);
        root.setBottom(buttonBar);

        Scene scene = new Scene(root, 450, 200);
        stage.setScene(scene);
        stage.setTitle("JavaFX Test");
        stage.show();

    } // end start();

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

} // end class HelloWorldFX
```

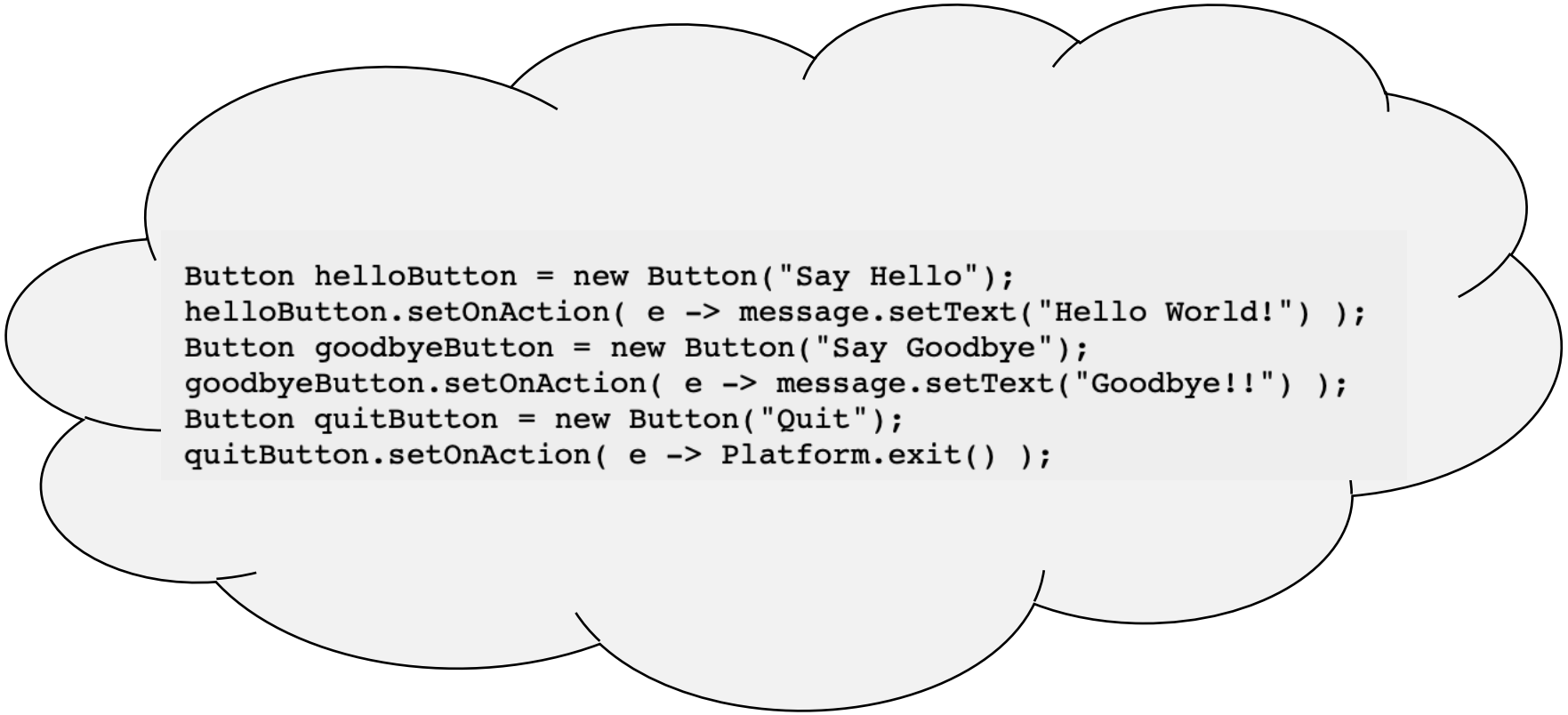
Events and Change Listeners

Use Lambda Expressions:



Events and Change Listeners

Use Lambda Expressions:



```
Button helloButton = new Button("Say Hello");
helloButton.setOnAction( e -> message.setText("Hello World!") );
Button goodbyeButton = new Button("Say Goodbye");
goodbyeButton.setOnAction( e -> message.setText("Goodbye!!") );
Button quitButton = new Button("Quit");
quitButton.setOnAction( e -> Platform.exit() );
```

Exercise

- Download the HelloWorldFX Application that was discussed in this presentation from blackboard resources section or the Java8 online text
- Run the program in your IDE
- Test some changes and familiarize yourself with the methods used (e.g. change the text in the action listener, remove the the quit button action, etc)

You can also define Event Filters to intercept (block) some events or override them

Event Filters are activated before event handlers.

When an event occurs it is first passed through the chain to the target (for example if you have a scene with a button, or even a shape) the event would normally bubble up from the target through a pane -> scene -> stage and be acted on by an event handler if there are any.

Before the event is passed to **any** event handlers, it is first passed to any event filters that exist in the pane, scene, stage or target would be applied first and could prevent the child nodes from receiving the event (block) and possibly do an alternative action (override).

Events and Change Listeners

Instead of .setOnxxxx() methods, you can use addEventHandler():