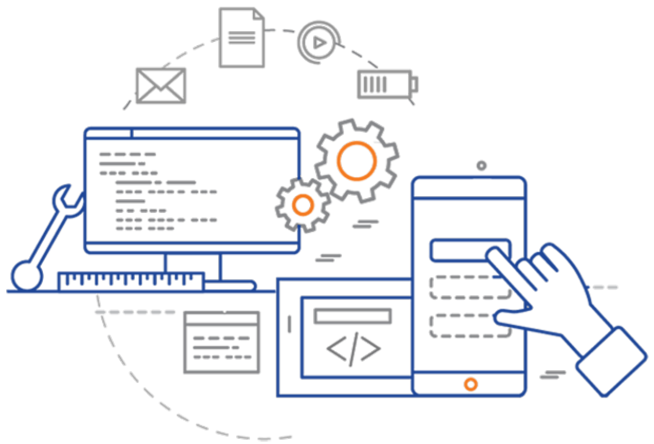


CS 398: Application Development

Building Desktop Applications

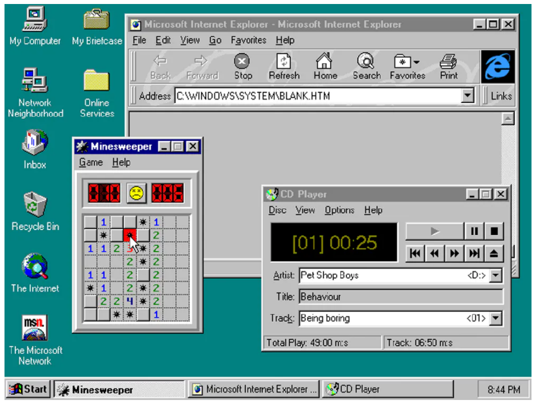
Features; Graphical toolkits; JavaFX

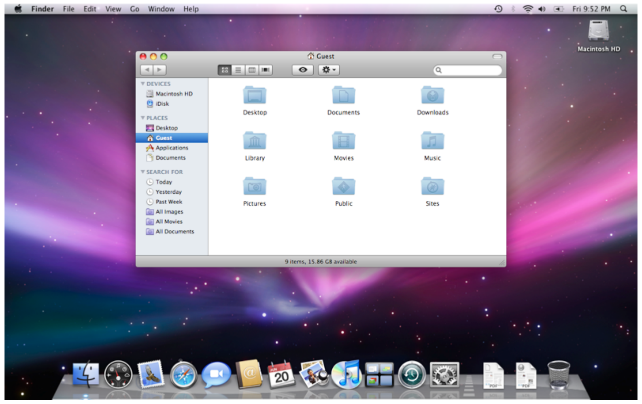
Graphical User Interfaces

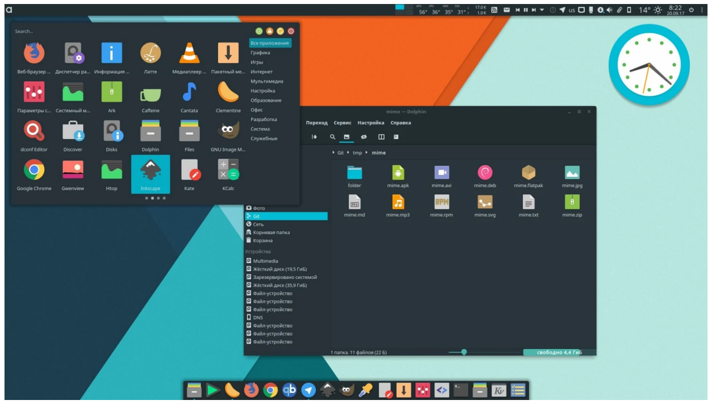
Graphical applications arose in the early 80s as we moved from text-based terminals to more capable graphical systems.

Graphical User Interfaces (GUIs) were thought to be more "user-friendly" to new users.

Different vendors adopted very similar conventions, resulting in very similar interfaces.


Windows 95


Mac OSX Leopard


Linux

Vendors tended to design similar looking systems, with only minor functional and aesthetic differences. This is a "standard" desktop.

Features

Windows

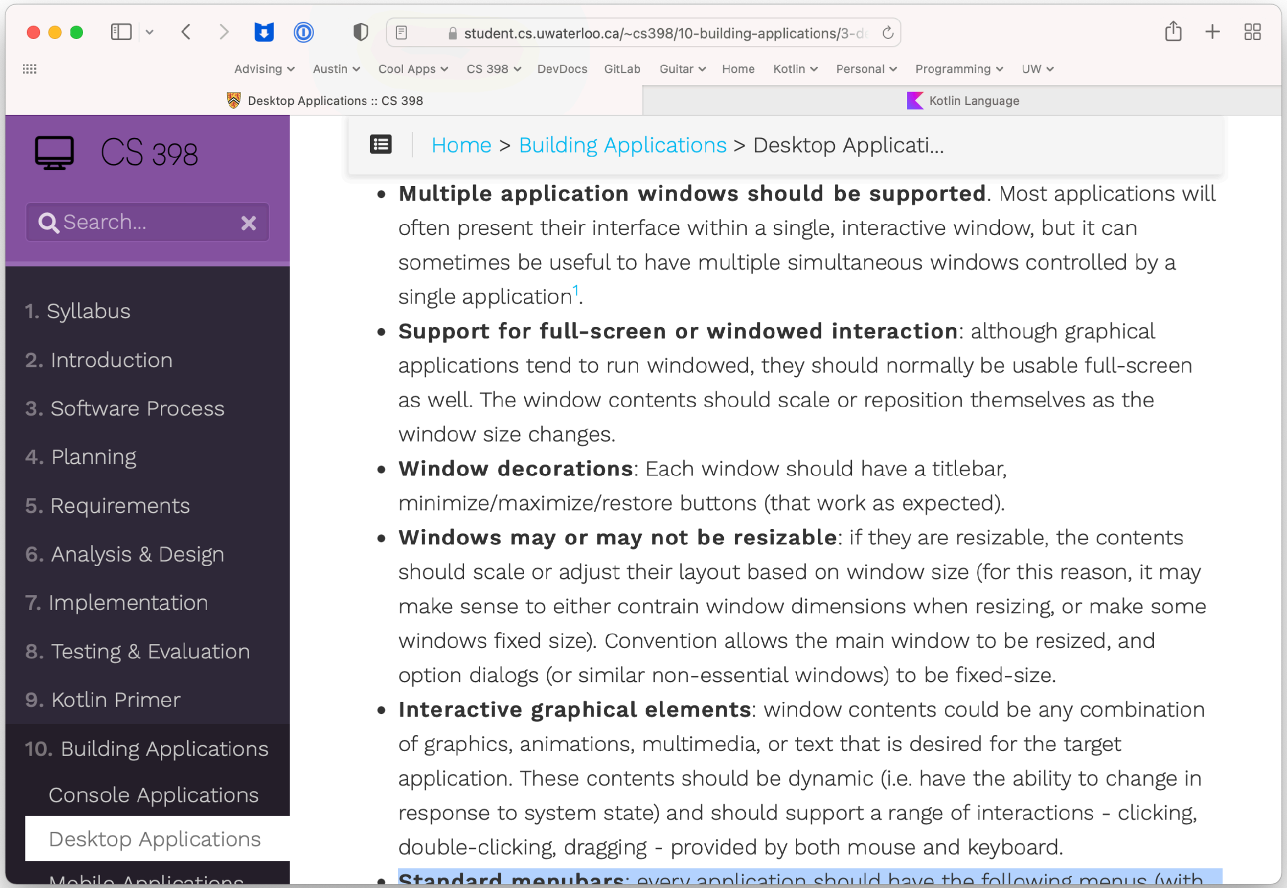
- **Multiple application windows.** Most applications will often present their interface within a single, interactive window, but it can sometimes be useful to have multiple simultaneous windows controlled by a single application[1].
- **Support for full-screen or windowed interaction**: although graphical applications tend to run windowed, they should normally be usable full-screen as well. The window contents should scale or reposition themselves as the window size changes.
- **Window decorations**: Each window should have a titlebar, minimize/maximize/restore buttons (that work as expected).
- **Windows may or may not be resizable**: if they are resizable, the contents should scale or adjust their layout based on window size (for this reason, it may make sense to either contrain window dimensions when resizing, or make some windows fixed size). Convention allows the main window to be resized, and option dialogs (or similar non-essential windows) to be fixed-size.

Interaction

Interactive graphical elements: window contents are a combination of txt, images, and interactive elements.

Windows are dynamic, and rearrange contents based on window size and dimensions.

Primary interaction is point-and-click with a mouse.



- **Multiple application windows should be supported**. Most applications will often present their interface within a single, interactive window, but it can sometimes be useful to have multiple simultaneous windows controlled by a single application[1].
- **Support for full-screen or windowed interaction**: although graphical applications tend to run windowed, they should normally be usable full-screen as well. The window contents should scale or reposition themselves as the window size changes.
- **Window decorations**: Each window should have a titlebar, minimize/maximize/restore buttons (that work as expected).
- **Windows may or may not be resizable**: if they are resizable, the contents should scale or adjust their layout based on window size (for this reason, it may make sense to either contrain window dimensions when resizing, or make some windows fixed size). Convention allows the main window to be resized, and option dialogs (or similar non-essential windows) to be fixed-size.
- **Interactive graphical elements**: window contents could be any combination of graphics, animations, multimedia, or text that is desired for the target application. These contents should be dynamic (i.e. have the ability to change in response to system state) and should support a range of interactions - clicking, double-clicking, dragging - provided by both mouse and keyboard.
- **Standard menubars**: every application should have the following menus (with

Interaction

Standard menu bars: every application should have the following menus (with shortcuts). Although some applications choose to eliminate menus (or replace with other controls), most of the time you should include them. Exact contents may vary, but users expect at-least this functionality:

- File: New, Open, Close, Print, Quit.
- Edit: Cut, Copy, Paste.
- Window: Minimize, Maximize.
- Help: About.

Keyboard shortcuts: you should strive to have keyboard shortcuts for common functionality. All standard shortcuts should be supported[2]. e.g.

- Ctrl-N for File-New, Ctrl-O for File-Open, Ctrl-Q for Quit.
- Ctrl-X for Cut, Ctrl-C for Copy, Ctrl-V for Paste.
- F1 for Help.

Toolkits

A widget or GUI toolkit is a UI framework which provides most this functionality. This includes support for:

- Creating and managing application windows, with standard window functionality e.g. overlapping windows, depth, min/max buttons, resizing.
- Reusable components called widgets that can be combined in a window to build typical applications. e.g. buttons, lists, toolbars, images, text views.
- Dynamic layout that adapts the interface to change in window size or dimensions.
- Support for an event-driven architecture[3] i.e. support for standard and custom events. Includes event generation and propagation.

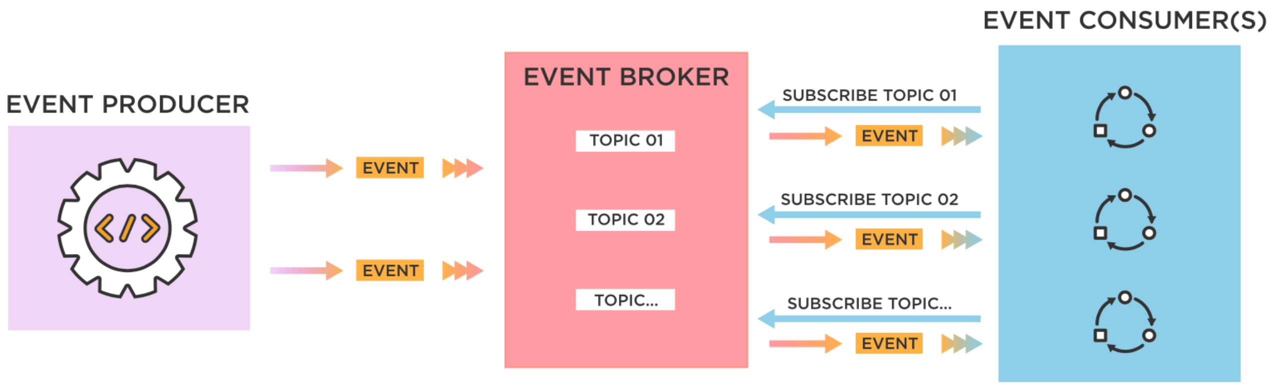
- e.g. Swing, JavaFX (Java or Kotlin Desktop), Compose (Kotlin Android/Desktop)

Architecture

Event-Driven Architecture

User interfaces are designed around the idea of using events or messages as a mechanism for components to indicate state changes to other interested entities.

This type of system, designed around the production, transmission and consumption of events between loosely-coupled components, is called an Event-Driven Architecture.

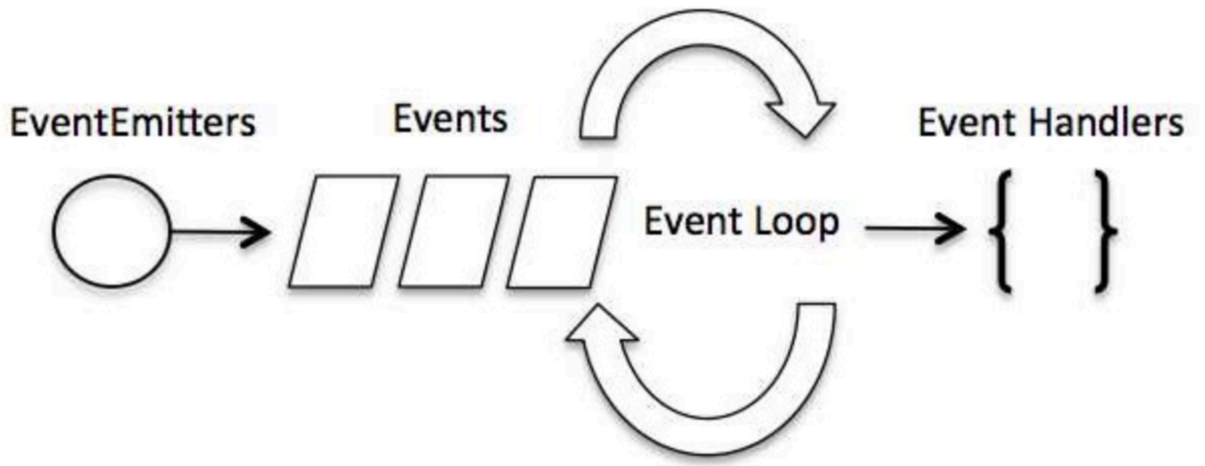


https://www.tibco.com/reference-center/what-is-event-driven-architecture

How does event-driven architecture work?

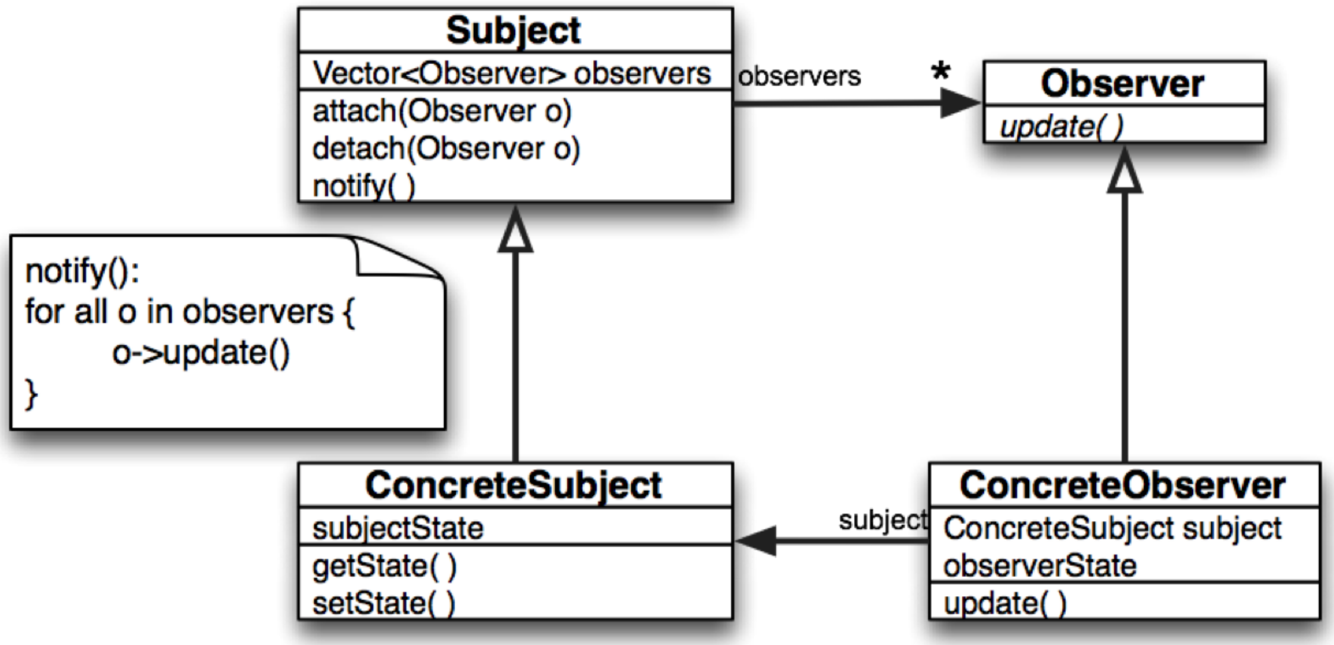
An **event producer** detects or senses the conditions that indicate that something has happened, and creates an event.

The event is transmitted from the event producer to one for more **event consumers** through event channels, where an event processing platform processes the event asynchronously. Event consumers choose how to act.



MVC Pattern

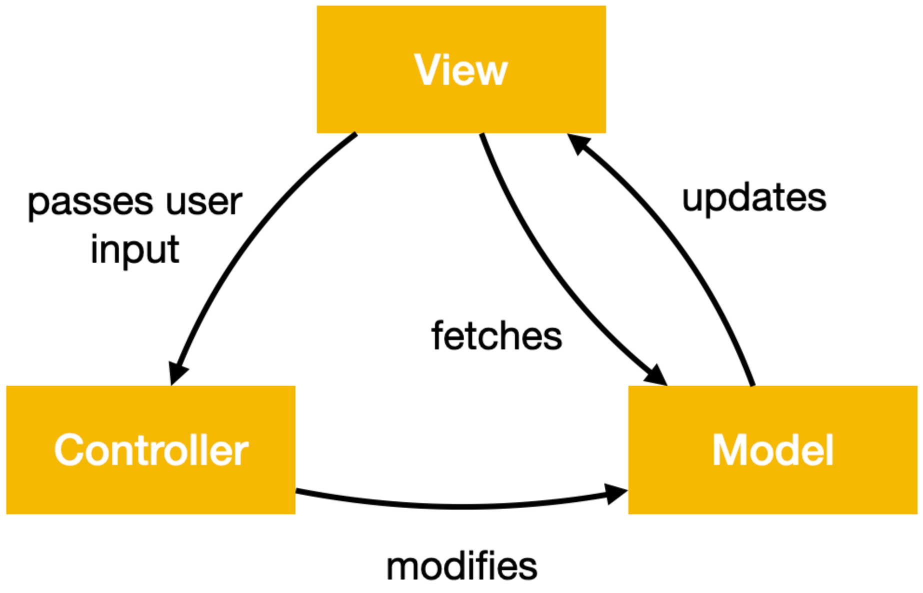
We often choose to model these types of systems using the MVC pattern, which is a specific instance of the observer pattern.



MVC Pattern

MVC divides an application into three distinct parts:

- **Model**: the core component of the application that handles state.
- **View**: a representation of the application state, often as a user-interface ("presentation")
- **Controller**: a component that accepts input, interprets user actions and converts to commands for the model or view ("business logic").



```
class Main {
  val model = Model()
  val controller = Controller(model)
  val view = View(controller, model)
    model.addView(model)
}
```

```
class Controller(val model: Model) {
  fun handle(event: Event) {
    // pass event data to model
  }
}
```

```
class Model {
  val views = listOf()
  fun addView(view: IView) {
    views.add(view)
  }
  fun update() {
    for (view : views) {
      view.update()
    }
  }
}
```

```
interface IView {
  fun update()
}

class View(
  val controller: Controller, val model: Model): IView
  {
  override fun update() {
    // fetch data from model
  }
}
```

JavaFX

JavaFX

JavaFX is a popular Java and Kotlin desktop toolkit.

- It was developed as a replacement to the aging Swing toolkit in Java, and has advanced features (e.g. 2D and 3D graphics, video playback, charts).
- It’s cross-platform on Windows, Mac, Linux, providing a native look and feel. It supports hardware acceleration, so it performs well!
- It consists of a small set of Java libraries that can be freely distributed.
- It’s easy to import into a Gradle project.

Create a JavaFX project

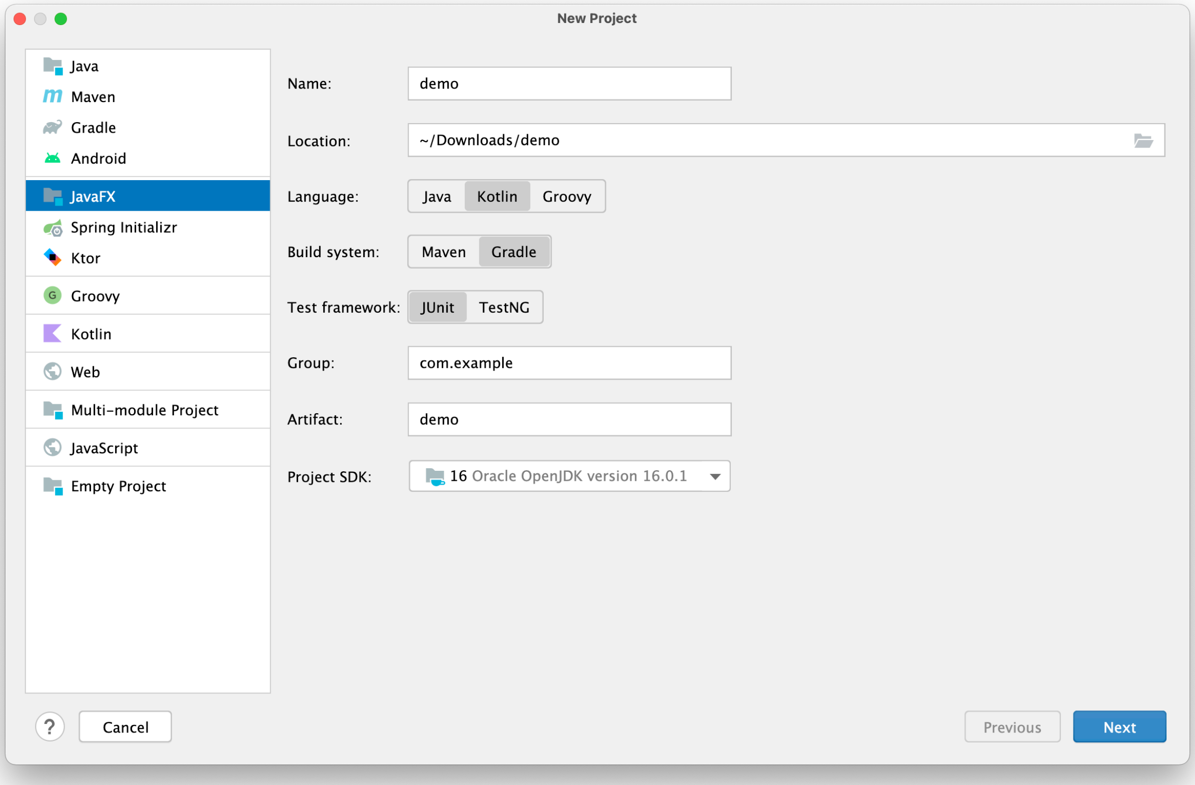
File, New Project

Choose JavaFX.

Options:

- Kotlin, Gradle, JUnit.
- (Optional) Other libraries.

Pick Gradle! It will automatically manage the JavaFX libraries.



Name: demo

Location: ~/Downloads/demo

Language: Java [✓ Kotlin] Groovy

Build system: Maven [✓ Gradle]

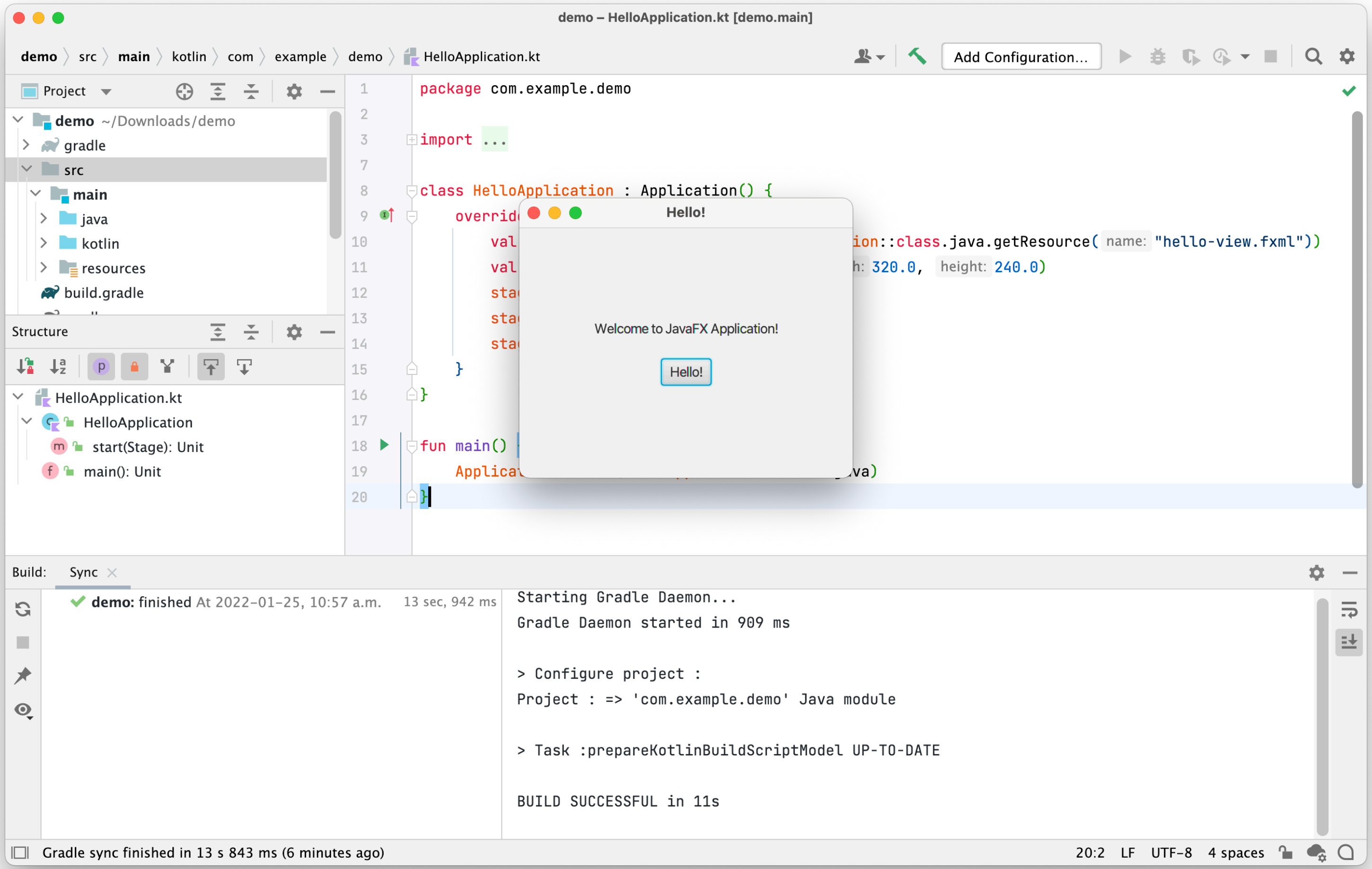
Test framework: [✓ JUnit] TestNG

Group: com.example

Artifact: demo

Project SDK: 16 Oracle OpenJDK version 16.0.1

Hello World!



```
package com.example.demo

import ...

class HelloApplication : Application() {
    overrid
        val                ion::class.java.getResource( name: "hello-view.fxml"))
        val                h: 320.0,  height: 240.0)
        sta
        sta
        sta
    }
}

fun main()
    Applicat                                                  va)
}
```

Hello!

Welcome to JavaFX Application!

Hello!

✓ **demo:** finished At 2022-01-25, 10:57 a.m.   13 sec, 942 ms

```
Starting Gradle Daemon...
Gradle Daemon started in 909 ms

> Configure project :
Project : => 'com.example.demo' Java module

> Task :prepareKotlinBuildScriptModel UP-TO-DATE

BUILD SUCCESSFUL in 11s
```

Gradle sync finished in 13 s 843 ms (6 minutes ago)   20:2 LF UTF-8 4 spaces

Application Lifecycle

JavaFX applications extend the `Application` class. The JavaFX runtime does the following when an application is launched:

1. Creates an instance of the specified Application class.
2. Calls the instance's `init()` method
3. Calls its `start()` method
4. Waits for the application to finish, when either (a) the application calls `Platform.exit()` or (b) the last application window has been closed.
5. Calls its `stop()` method.

Most time is spent here, waiting for things to happen

The `start()` method is abstract and MUST be overridden. The `init()` and `stop()` methods are optional, but MAY be overridden.

Application Flow

```
import javafx.application.Application
import javafx.stage.Stage

class Stages : Application() {
    override fun init() {
        super.init()
        println("init")
    }

    override fun start(stage: Stage) {
        println("start")
    }

    override fun stop() {
        super.stop()
        println("stop")
    }
}
```

required

Methods are invoked in this order.

Note that all are abstract base class methods and have default implementations.

- Step 0: main() — optional
- Step 1: init()   — optional
- Step 2: start() — required
- Step 3: stop() — optional

Hello World - Code

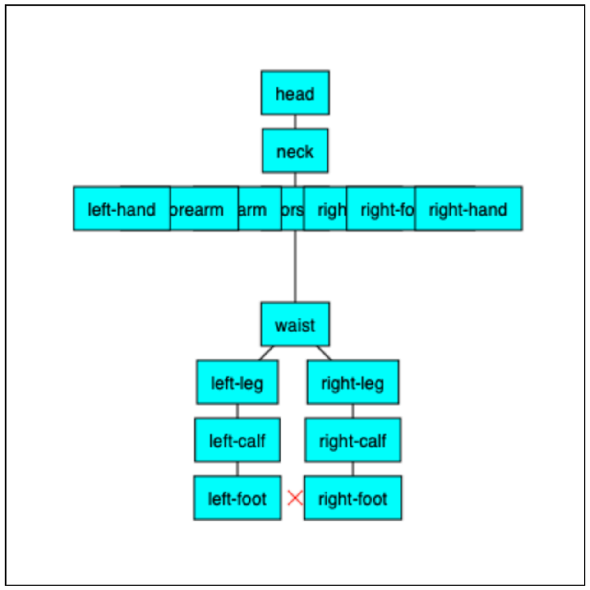
See public repo:
public / desktop / JavaFX / HelloFX

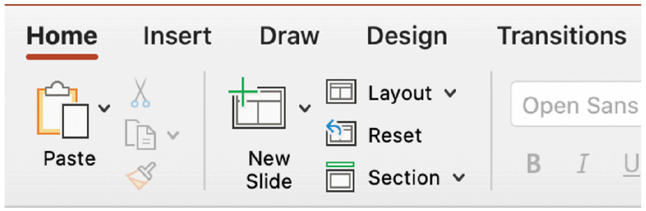
```
import javafx.application.Application
import javafx.scene.Scene
import javafx.scene.control.Label
import javafx.scene.layout.StackPane
import javafx.stage.Stage

class HelloFX: Application() {
    override fun start(stage: Stage?) {
       val pane = StackPane(Label("Hello Java FX"))
        val scene = Scene(pane, 150.0, 75.0)

       stage?.setScene(scene)
        stage?.setTitle("Hello FX")
        stage?.show()
    }
}
```

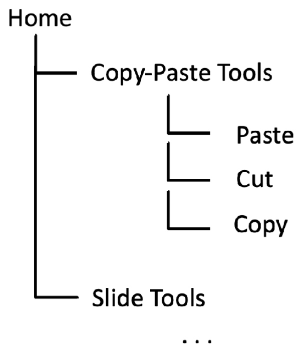
This is a fully-functioning app.

Windows Handling

In JavaFX, each application can own one or more windows. Within each window, we use abstraction called a **scene graph**, to represents graphical content as *tree, where higher level elements manage their children*.



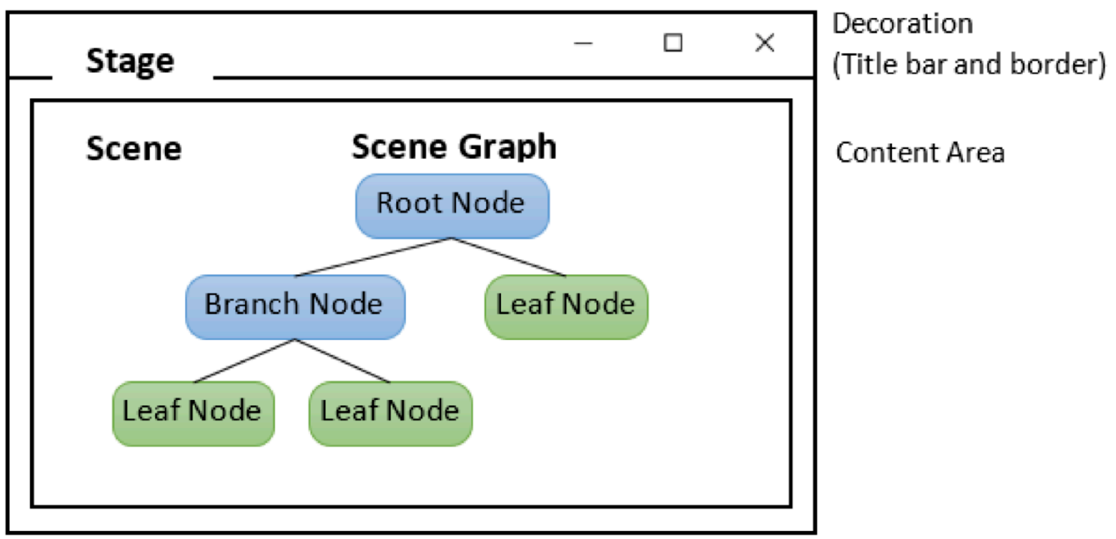




A scene graph is a hierarchy of components that describes graphical content.

Scene Graphs in JavaFX

- The **Stage** class represents the main window
- The **Scene** class contains a scene graph: a non-cyclical tree, with a single root, representing the contents of the scene.
- **Nodes** represent graphical elements in the scene graph.



Stage

The “Stage“ is the top-level container or application window. You can have multiple stages, representing multiple windows.

```
javafx.stage.Window
    javafx.stage.Stage
```

A Stage instance is automatically created by the runtime, and passed into the `start()` method.

Stage methods operate at the window level:

- `setMinWidth(), setMaxWidth()`
- `setResizable()`
- `setTitle()`
- `setScene()`
- `show()`

Scene

The “Scene“ is a container for the content in a scene-graph.

`javafx.scene.Scene`

You must manually construct the scene, and set it up:

- Create a scene graph and specify the root-node.
- Add the scene to a stage and make the stage visible.

Scene methods manipulate the scene graph, or attempt to set properties for the entire graph:

- `setRoot(Node)`
- `setFill(Paint)`
- `getX(), getY()`

A Scene can exist independent of a Stage, but it needs to be attached to a Stage to be visible.

Nodes

“Node“ is the base class for all elements of a Scene graph.

There are two types of nodes:

Branch nodes: These are nodes that can hold other nodes in the tree. Examples of subclasses include `Group`, `Region`.

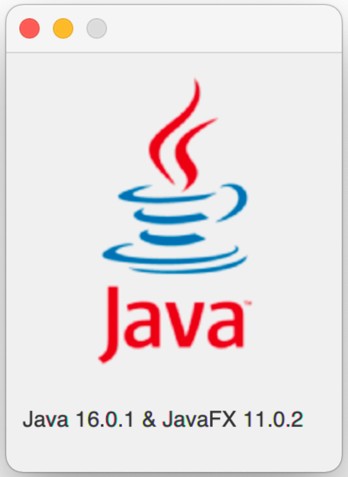
```
javafx.scene.Node
    javafx.scene.Parent
```

Leaf nodes: These are low-level nodes that cannot contain other nodes. Examples include `Circle`, `Rectangle`, `Button` and `Label`.

```
javafx.scene.Node
    javafx.scene.Parent
       javafx.scene.layout.Region
         javafx.scene.control.Control
```

Building a User Interface

```
class App: Application() {
    override fun start(stage:Stage?) {
        val image = ImageView(Image("java.png", 200.0, 200.0, true, true))
        val label = Label("Java ${System.getProperty("java.version")} & "
                + "JavaFX ${System.getProperty("javafx.version")}")
        label.font = Font.font("Helvetica")

        val box = VBox(image, label)
        VBox.setMargin(label, Insets(10.0))

        val scene = Scene(box, 200.0, 250.0)
        stage?.setResizable(false)
        stage?.setScene(scene)
        stage?.show()
    }
}
```



Java 16.0.1 & JavaFX 11.0.2

public / desktop / JavaFX / JavaVersion

Layout

Layout is how items are arranged on the screen.

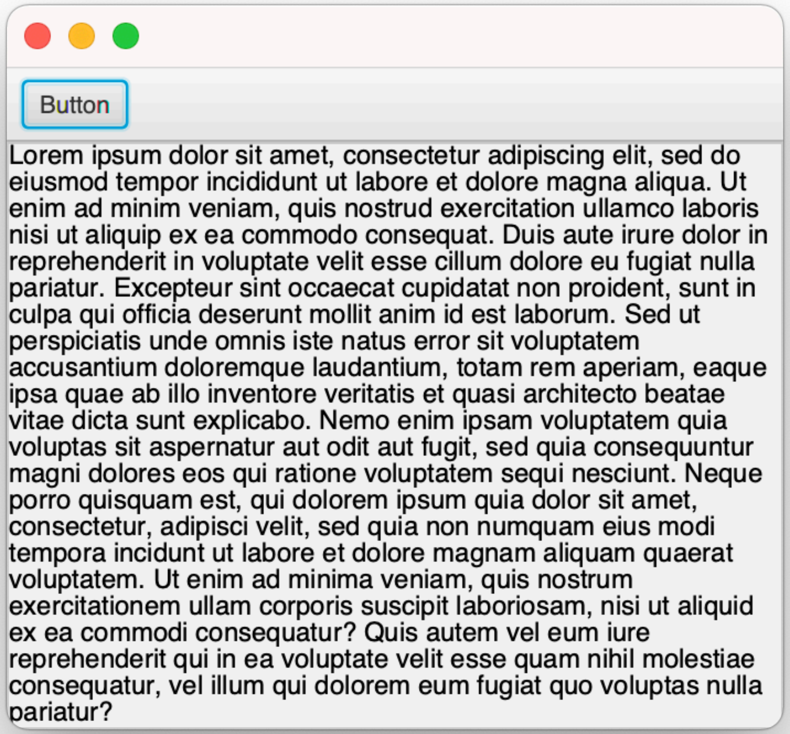
Layout classes are branch nodes that have built-in layout behaviour. Your choice of parent class to hold nodes determines how its children will be laid out.

Layout Class	**Layout Behaviour**
HBox	Layout children horizontally in-order
VBox	Layout children vertically in-order
FlowPane	Layout left-right, top-bottom in-order
BorderPane	Layout across sides, centre in-order
GridPane	2D grid, with cells the same size

Common JavaFX Layout Classes

Layout Example

```
class Main : Application() {
    override fun start(stage: Stage) {
        val toolbar = ToolBar()
        val button = Button("Button")
        button.font = Font("Helvetica", 11.0)
        toolbar.items.add(button)

        val text = Text("Lorem ipsum dolor …”)

        text.font = Font("Helvetica", 12.0)
        text.wrappingWidth = 350.0
        val scroll = ScrollPane()
        scroll.content = text

        stage.scene = Scene(VBox(toolbar, scroll))
        stage.show()
    }
}
```



Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?

public / desktop / JavaFX / Layout

What can we add to a Scene?

Node is the base class for all Leaf nodes in the Scene Graph.

This includes: Camera, **Canvas**, **ImageView**, LightBase, **MediaView**, Parent, **Shape**, **Shape3D**, SubScene, SwingNode

Graphics Primitives (Shape classes for drawing)

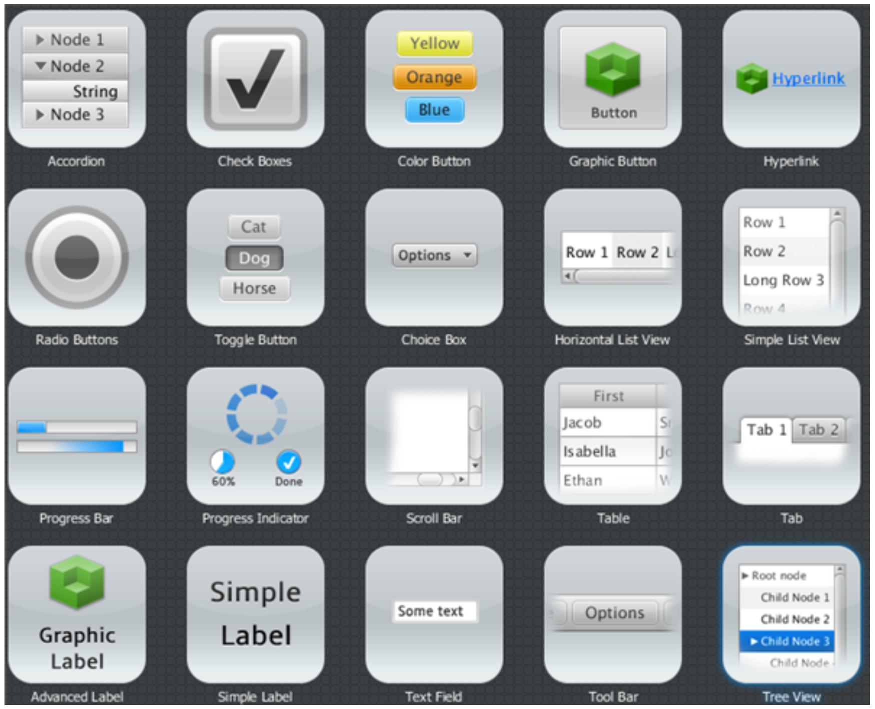
- Arc, Circle, CubicCurve, Ellipse, Line, Path, Polygon, Polyline, QuadCurve, Rectangle, SVGPath, Text

Widgets (interactive components)

- Accordion, ButtonBar, ChoiceBox, ComboBoxBase, HTMLEditor, Labeled, ListView, MenuBar, Pagination, ProgressIndicator, ScrollBar, ScrollPane, Separator, Slider, Spinner, SplitPane, TableView, TabPane, TextInputControl, ToolBar, TreeTableView, TreeView

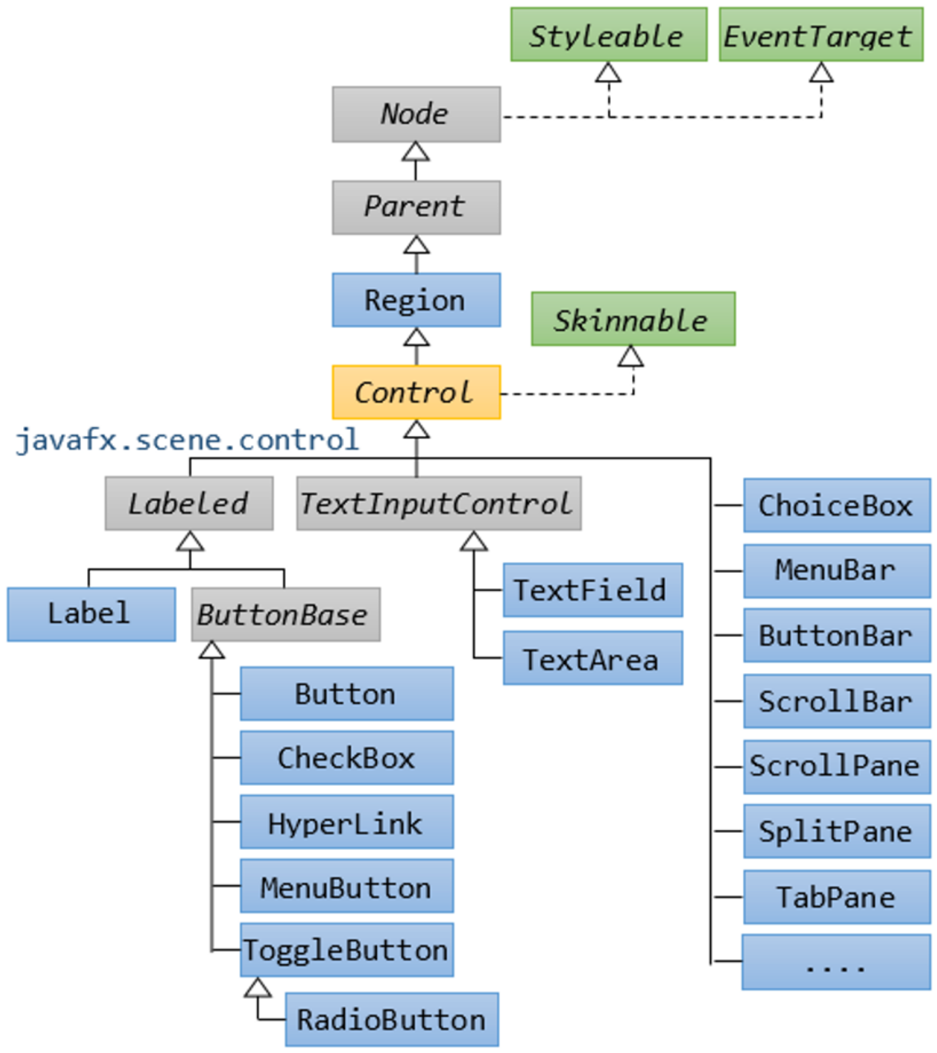
Reusable Components (aka Controls, Widgets)

- JavaFX includes a large collection of components that you can use to create your application.
- All behave the same:
  - Instantiate them.
  - Set properties that describe how they should behave
  - Add them to a layout.



https://openjfx.io/javadoc/17/javafx.controls/javafx/scene/control/package-summary.html

```
java.lang.Object
    javafx.scene.Node
        javafx.scene.Parent
            javafx.scene.layout.Region
                javafx.scene.control.Control
```



Event Handlers

JavaFX is designed to support **event-driven architecture**.

To make a user interface interactive, you write event consumers that can respond to user -generated events. e.g. code to describe what happens when a button is clicked.

What's an event? An event is any significant occurrence or change in state for system hardware or software.

The source of an event can be from internal or external inputs. Events can generate from a user, like a mouse click or keystroke, an external source, such as a sensor output, or come from the system, like loading a program.

How does event-driven architecture work? Events are routed from event producers, to event consumers that can choose to act on them.

Adding listeners (aka event handlers)

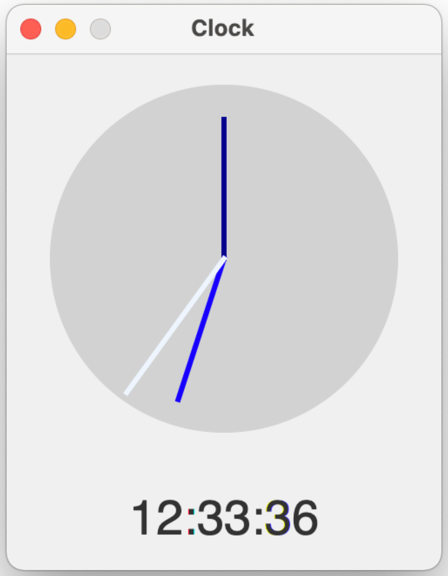
A listener is a form of event handler - a consumer that registers to listen for specific events.

We can attach listeners to any interactive component, which will be executed on events that are produced by that component.

```
// simple handler to respond to user actions
button.setOnMouseClicked {
    resetToDefaultPath()
}

// respond to panel resize
val widthProperty = pane.widthProperty();
widthProperty.addListener(ChangeListener<Number> ()
{
    @override fun changed() {
        // do something
    }
}
```

Events Example

```
class Main : Application() {
    private val width = 250.0
    private val height = 325.0

    override fun start(stage: Stage) {
        // analog time
        val clock = ClockFace(0.0, 0.0, 100.0)
        val clockPane = StackPane(clock.build())
        clockPane.setPrefSize(width, height - 75)

        // digital time
        val time = Label()
        time.font = Font("Helvetica", 28.0)
        val timePane = StackPane(time)
        timePane.setPrefSize(width, 75.0)

        // timer fires every 1/60 seconds, and fetches time
        val dateFormat = SimpleDateFormat("hh:mm:ss")
        val timer = object : AnimationTimer() {
            override fun handle(now: Long) {
                time.text = dateFormat.format(Calendar.getInstance().time)
                clock.setTime(
                    Calendar.getInstance()[Calendar.HOUR_OF_DAY],
                    Calendar.getInstance()[Calendar.MINUTE],
                    Calendar.getInstance()[Calendar.SECOND]
                )
            }
        }
        timer.start()
```



public / desktop / JavaFX / Advanced_Clock

Other Samples

public / desktop / JavaFX / Calculator

```
class Main : Application() {

   class Expr(var num1: Int, var op: OP = OP.NONE, var num2: Int) {
      enum class OP { ADD, SUB, MUL, DIV, NONE }
      fun clear() = run { num1 = 0 ; op = OP.NONE ; num2 = 0 }
      fun set(operation: OP) = run { op = operation }
      fun set(n: Int) = if (op == OP.NONE) num1 = (num1 * 10) + n else num2 = (num2 * 10) + n
   }

override fun start(stage: Stage?) {
    // text output field
    val output = TextField("")
    output.font = Font("Helvetica", 21.0)
    output.isVisible = true
    output.isDisable = true
    output.alignment = Pos.BASELINE_RIGHT

    // numbers
    val button0 = CalcButton("0", { expr.set(0); output.text = output.text.plus("0") } )
    val button1 = CalcButton("1", { expr.set(1); output.text = output.text.plus("1") } )
    val button2 = CalcButton("2", { expr.set(2); output.text = output.text.plus("2") } )
    val button3 = CalcButton("3", { expr.set(3); output.text = output.text.plus("3") } )
    val button4 = CalcButton("4", { expr.set(4); output.text = output.text.plus("4") } )
```

Inner class

Start method for main JFX class