Widgets

Purpose of Widgets
Widgets in JavaFX
Properties & Property Binding

UI CS 349

May 23

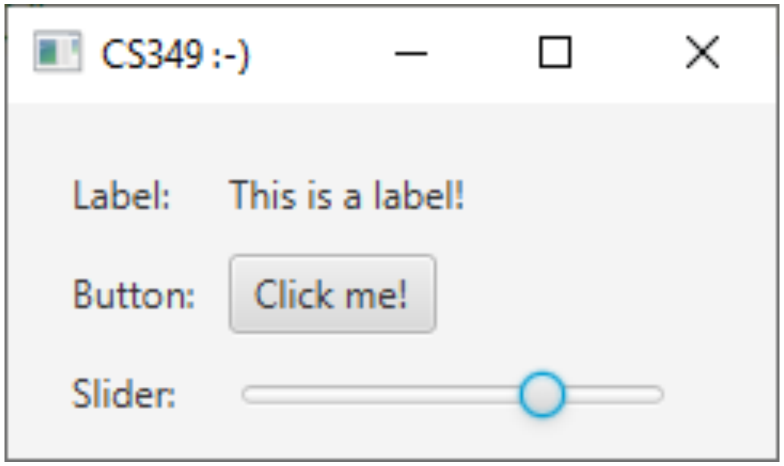
Purpose of Widgets

User Interface Widgets

Widgets are parts of an interface that have their own behavior (e.g., buttons, drop-down menus, spinners, file dialog boxes, progress bars, slider). They are also called *components*, *controls*, or *UI elements*.

They can perform four essential functions:

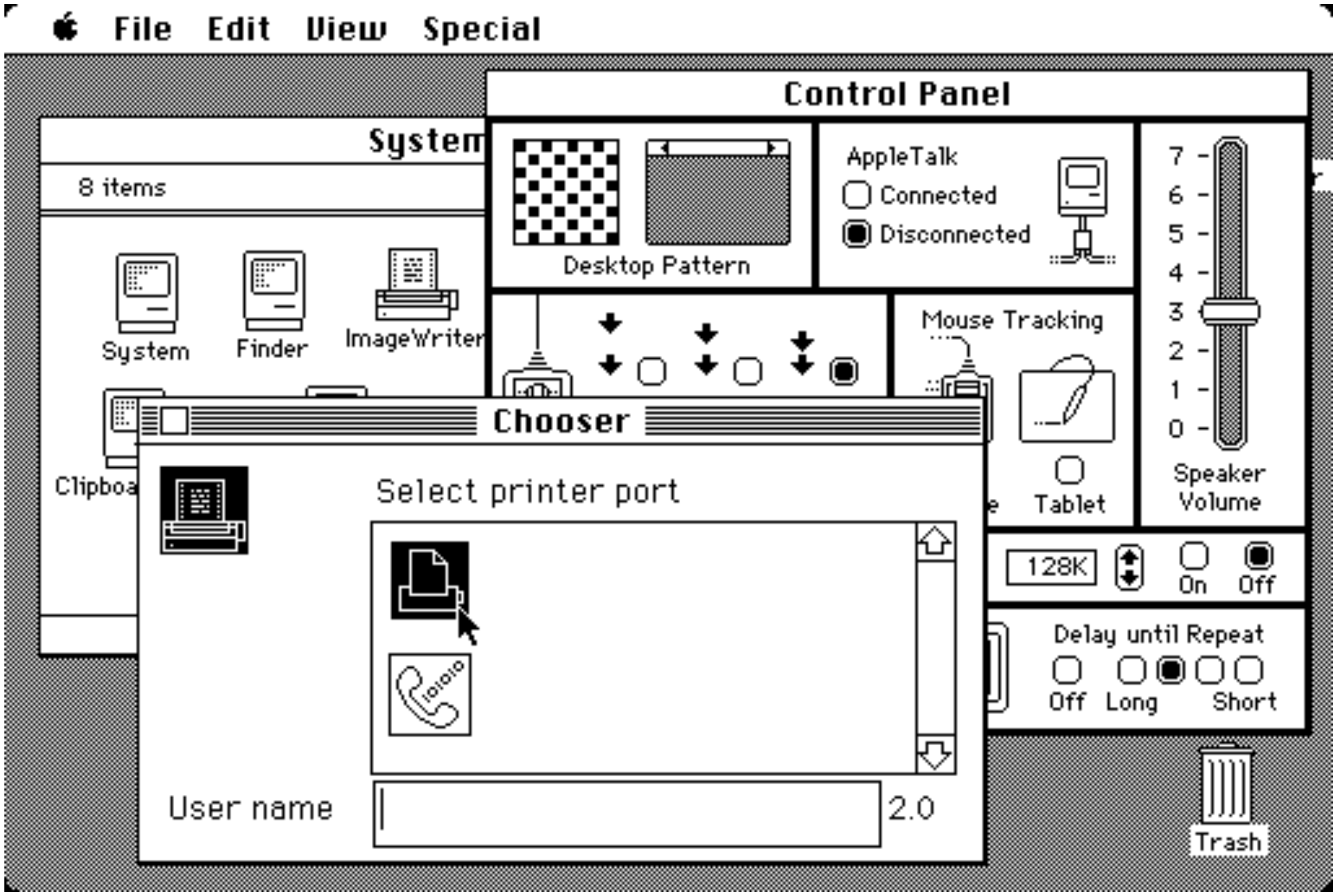
- capture user input
- provide user feedback
- maintain state
- generate events – *more on this later*



User Interface Widgets

The original eight widgets:

button, menu, radio buttons, checkbox, slider, textbox, scrollbar, spinner

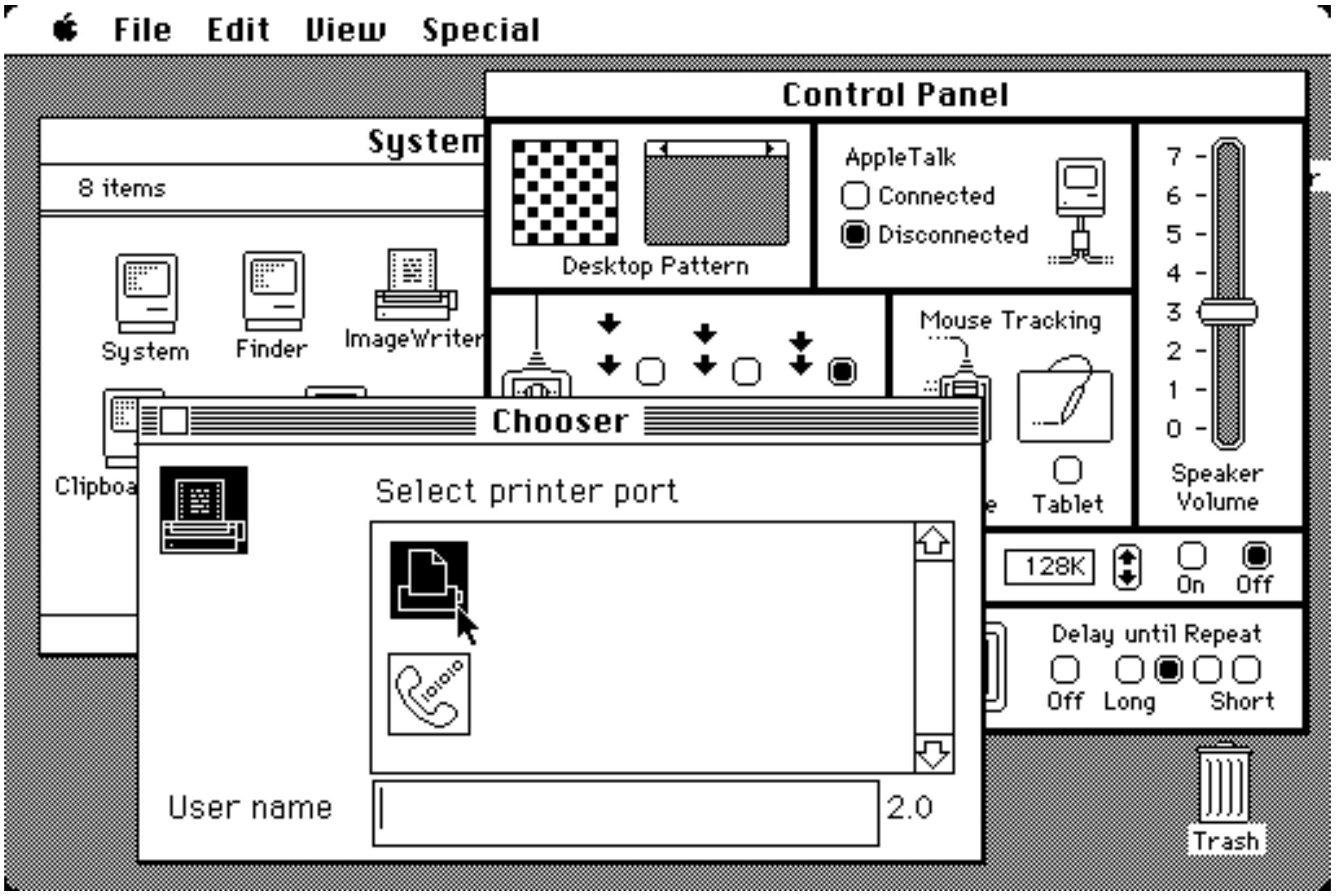


Macintosh System 5, circa 1987

User Interface Widgets

Widgets are often packaged in GUI toolkits, such as, SwiftUI, WinUI, Gtk+, Qt, and JavaFX.

Widget toolkits vary in presentation, but all include “standard” widgets.



User Interface Widgets

Capture user input

- Capture user input in various forms
- The type of input varies with the widget

Generate events

- *They generate events (i.e., messages) that can be sent to other parts of your application to indicate that the user has done something*

Provide feedback

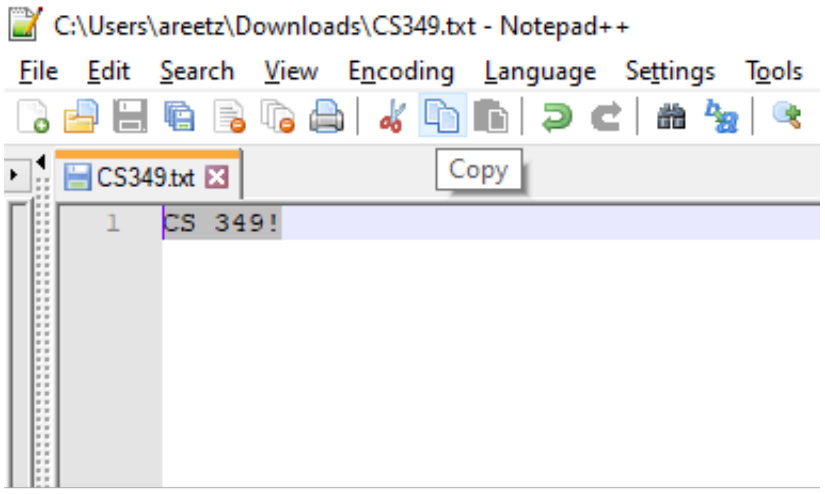
- Provide user feedback indicating that they have been activated (whatever that means for that particular widget)

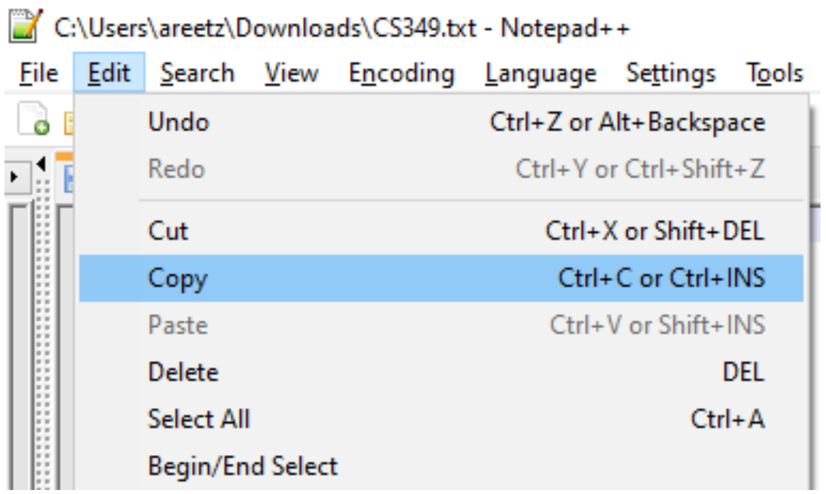
Maintain state

- They may have state or data that they retain and control, that can represent state to the user

User Interface Widgets

Different widgets can provide the same conceptual functionality in different ways.


C:\Users\areetz\Downloads\CS349.txt - Notepad++
File Edit Search View Encoding Language Settings Tools
CS349.txt
Copy
CS 349!


C:\Users\areetz\Downloads\CS349.txt - Notepad++
File Edit Search View Encoding Language Settings Tools
Undo Ctrl+Z or Alt+Backspace
Redo Ctrl+Y or Ctrl+Shift+Z
Cut Ctrl+X or Shift+DEL
Copy Ctrl+C or Ctrl+INS
Paste Ctrl+V or Shift+INS
Delete DEL
Select All Ctrl+A
Begin/End Select

Logical Inputs vs. Widgets

Logical inputs describe the underlying functionality (i.e., the type of input or interaction that they support). This includes state and events.

- State: what data does the widget need to store?
  - e.g., label holds a string, slider holds min / current / max values
- Events: what messages does the widget generate when activated?
  - e.g., buttons generate “activated”; sliders generate ”changed” events

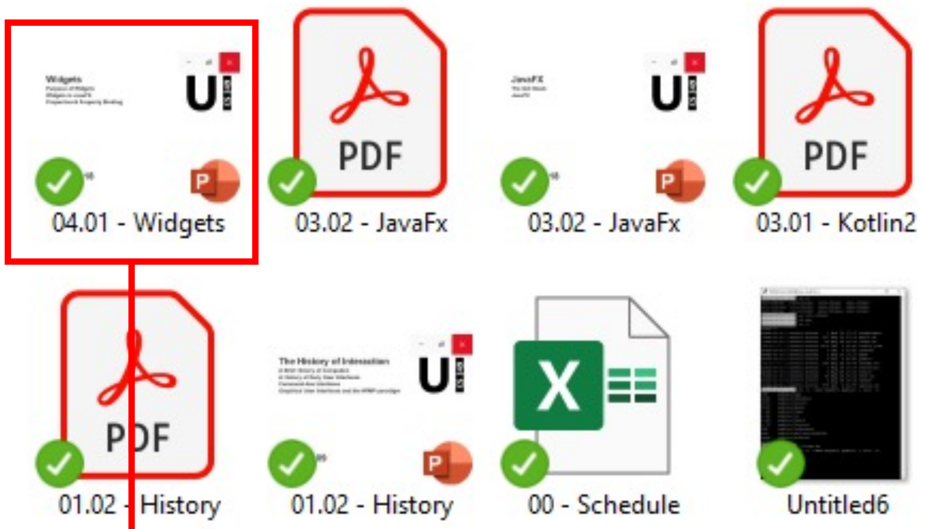
Widgets are implementations of logical inputs, and define their appearance. They add properties to logical input.

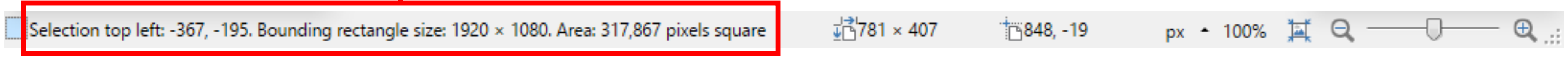
- Properties: values that determine how the widget is presented. Properties may be general (e.g., position and size) or specific (e.g., text)
  - Common properties: position (x,y), size (width, height), color
  - Custom properties: specific to a logical input, e.g., orientation for slider

Logical Display

Displays text or images to the user. Purpose is displaying data or providing feedback.

- States:
  - None
- Events:
  - None
- Examples:
  - Labels
  - Images

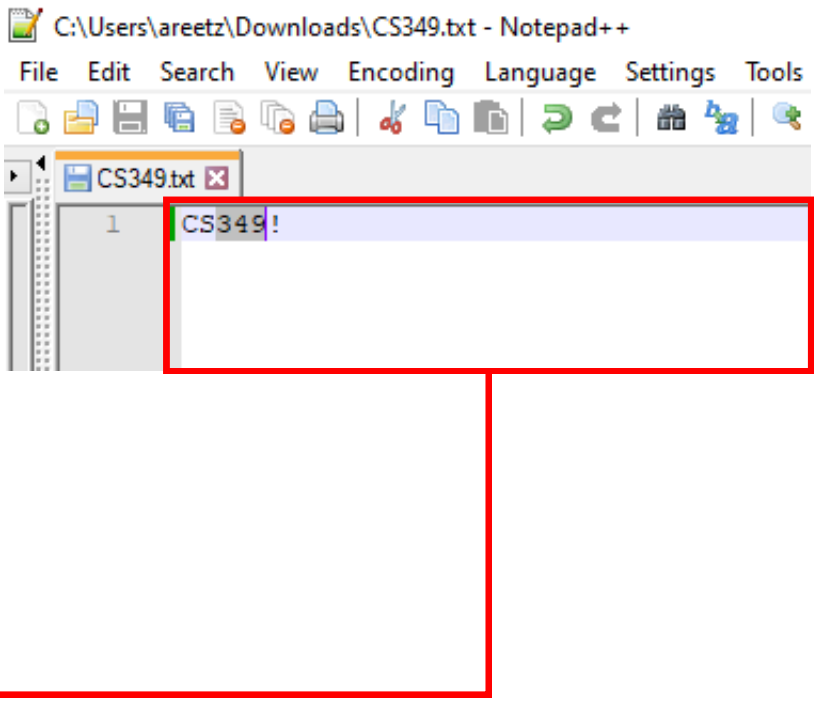



Selection top left: -367, -195. Bounding rectangle size: 1920 × 1080. Area: 317,867 pixels square

Logical Text Entry

Allows users to enter text and displays the current state.

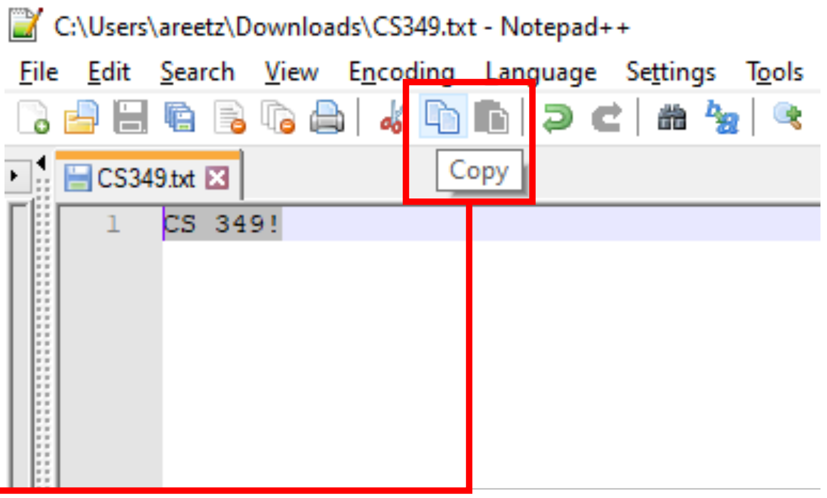
- States:
  - Text: String
  - Selection: Range
- Events:
  - Text changed
  - Entry complete
  - Selection changed
- Examples:
  - Text fields
  - Text areas

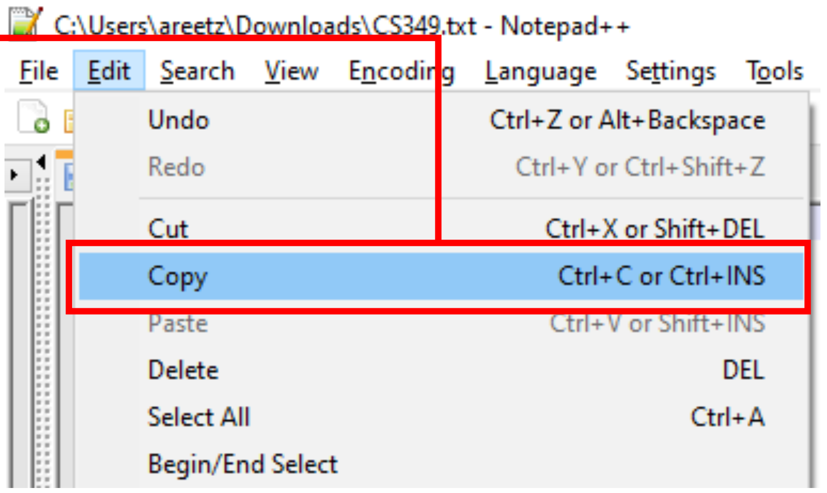


Logical Button

Enables users to perform a simple interaction, with a single fixed action.

- States:
  - none
- Events:
  - Button activated
- Examples:
  - Buttons
  - Menus

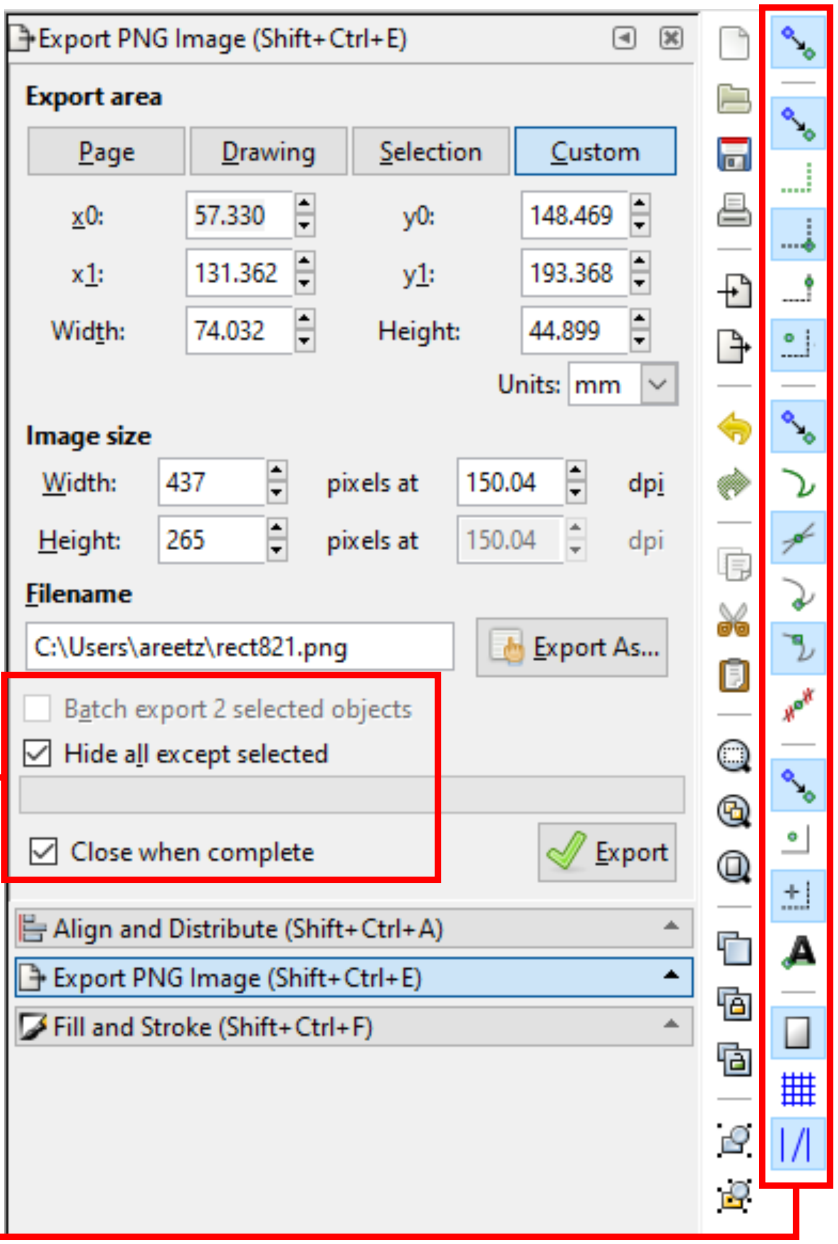




Logical Boolean Selection

Allows users to select between two states and displays the current state.

- States:
  - Selection: Boolean
- Events:
  - Selection changed
- Examples:
  - Checkboxes
  - Toggle buttons



Export PNG Image (Shift+Ctrl+E)

Export area

Page | Drawing | Selection | Custom

x0: 57.330 y0: 148.469

x1: 131.362 y1: 193.368

Width: 74.032 Height: 44.899

Units: mm

Image size

Width: 437 pixels at 150.04 dpi

Height: 265 pixels at 150.04 dpi

Filename

C:\Users\areetz\rect821.png

Export As...

☐ Batch export 2 selected objects

☑ Hide all except selected

☑ Close when complete

Export

Align and Distribute (Shift+Ctrl+A)

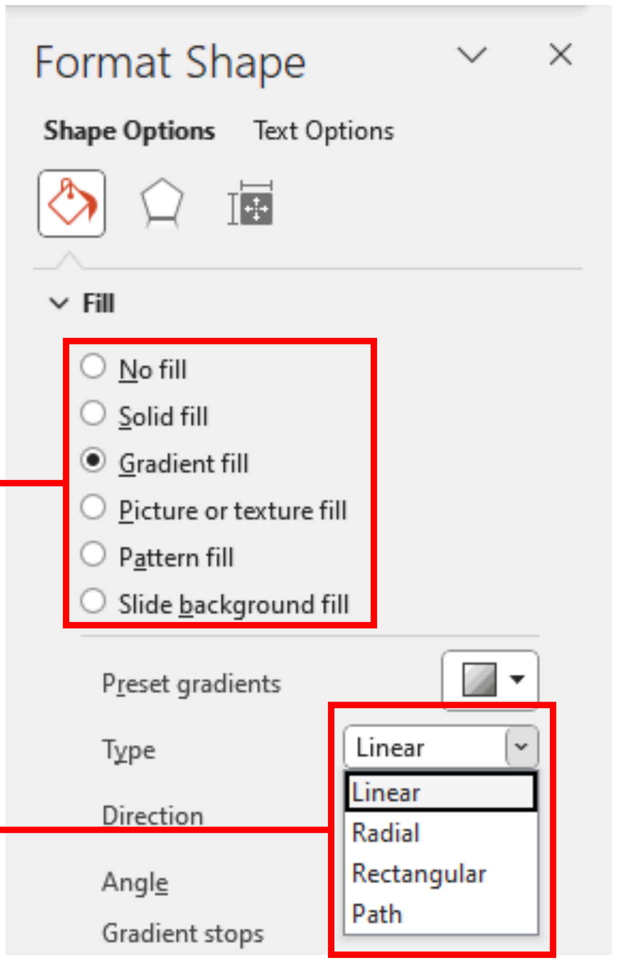
Export PNG Image (Shift+Ctrl+E)

Fill and Stroke (Shift+Ctrl+F)

Logical Discrete Selection

Allows users to select one entry from an arbitrary list of discrete elements and displays the current state.

- States:
  - Selection: Index, Element
- Events:
  - Selection changed
- Examples:
  - Radio buttons
  - Choice boxes



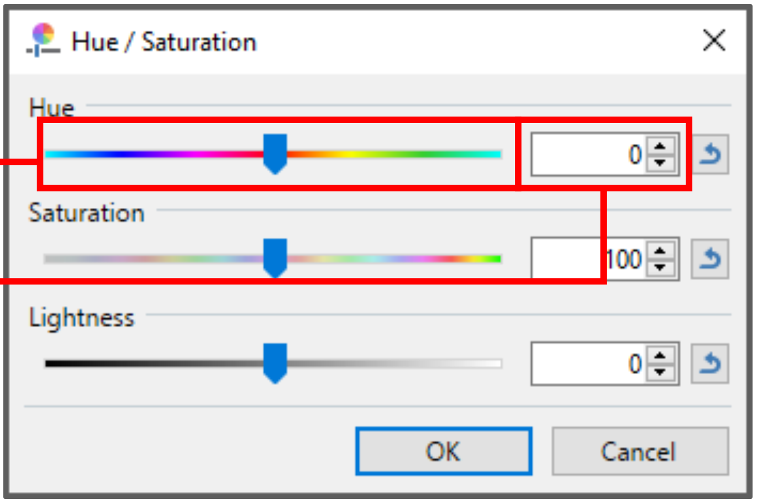
No fill
Solid fill
[✓ Gradient fill]
Picture or texture fill
Pattern fill
Slide background fill

Type: Linear
[✓ Linear]
Radial
Rectangular
Path

Logical Continuous Selection

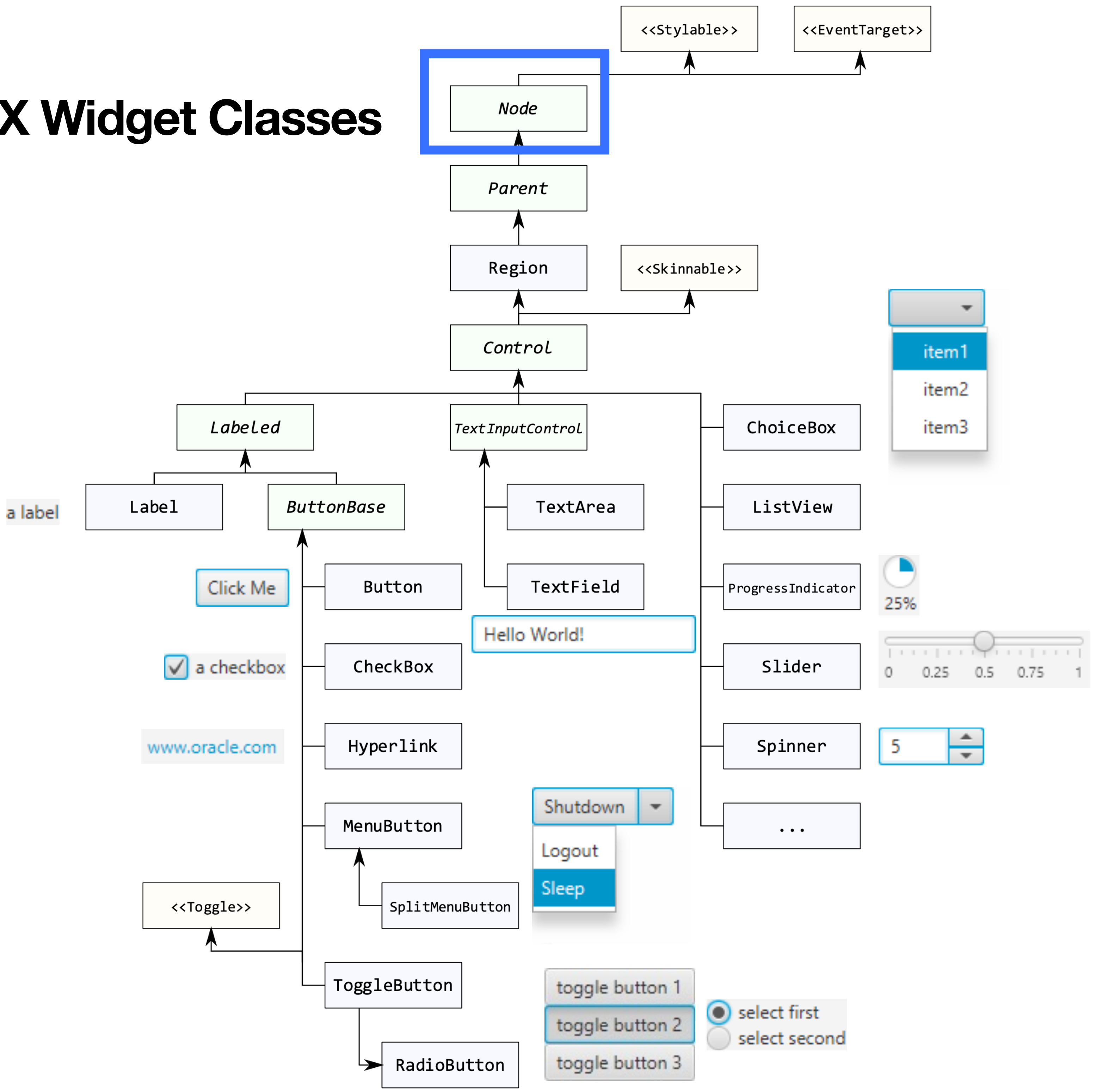
Allows users to select one value from a continuous range of values and displays the current state.

- States:
  - Value: integer, real number
- Events:
  - Value changed
- Examples:
  - Sliders
  - Spinners



Widgets in JavaFX

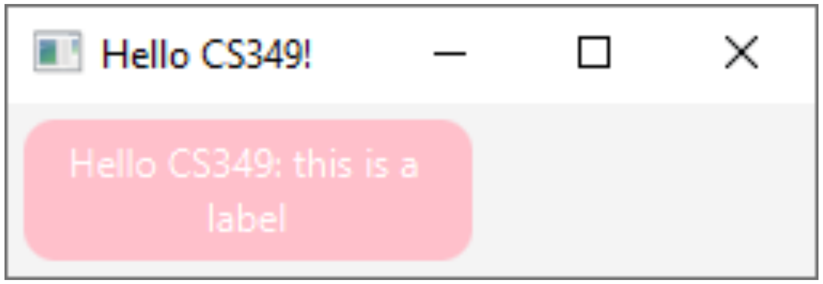
JavaFX Widget Classes



Display Widgets

Display widgets display some (static) information, such as, text or images. These widgets include `Label` and `ImageView`.

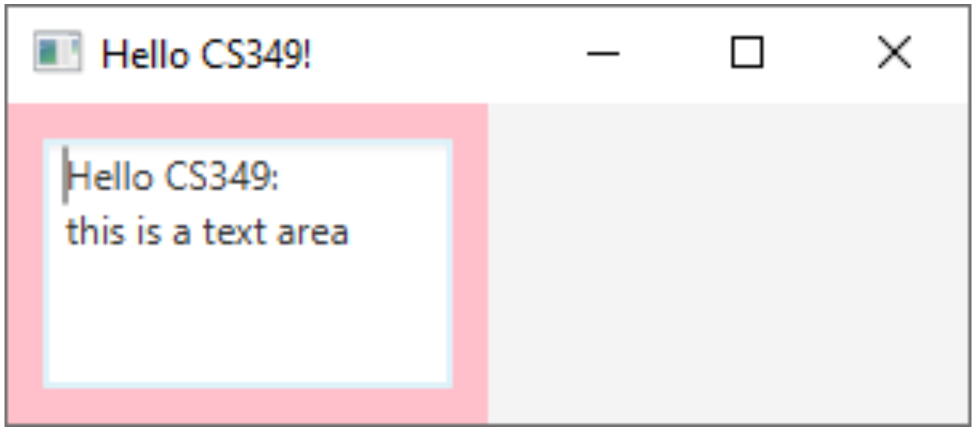
```
val myText = Label("Hello CS349: this is a label").apply {
    padding = Insets(10.0)
    background = Background(BackgroundFill(Color.PINK,
                                           CornerRadii(10.0),
                                           Insets(5.0)))
    textFill = Color.WHITE
    textAlignment = TextAlignment.CENTER
    isWrapText = true
    maxWidth = 150.0
}
```



Text Entry Widgets

Text entry widgets allow users to input text. These widgets include `TextField` (single-line) and `TextArea` (multi-line).

```
val myText = TextArea("Hello CS349:\nthis is a text area").apply {
    padding = Insets(10.0)
    background = Background(BackgroundFill(Color.PINK, null, null))
    maxWidth = 150.0
    maxHeight = 100.0
    textProperty().addListener { _, _, newValue -> // String
        stage.title = newValue
    }
}
```

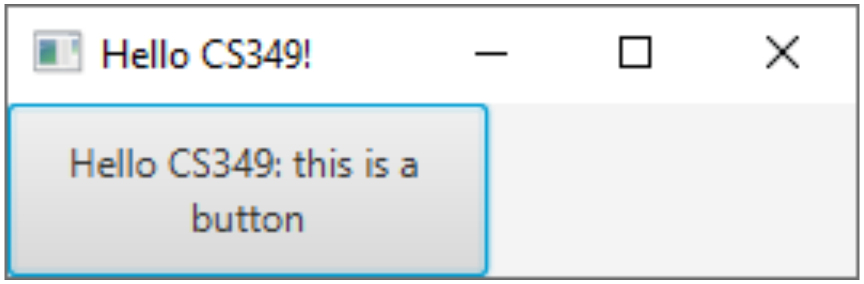


Hello CS349:
this is a text area

Button Widgets

Button widgets allow users to perform a single action. These widgets include `Button` and `MenuItem`.

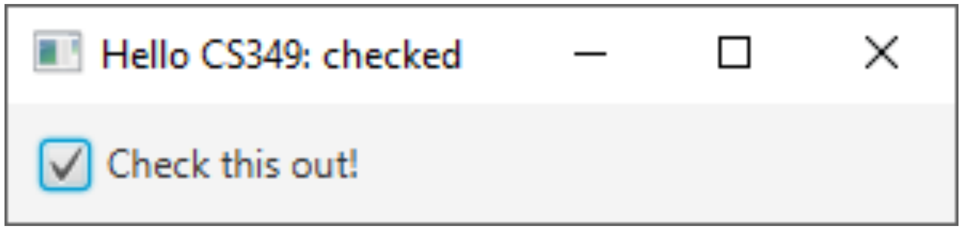
```
val myButton = Button("Hello CS349: this is a button").apply {
    padding = Insets(10.0)
    isWrapText = true
    textAlignment = TextAlignment.CENTER
    maxWidth = 150.0
    maxHeight = 100.0
    onAction = EventHandler {
        stage.title = ”Title text"
    }
}
```

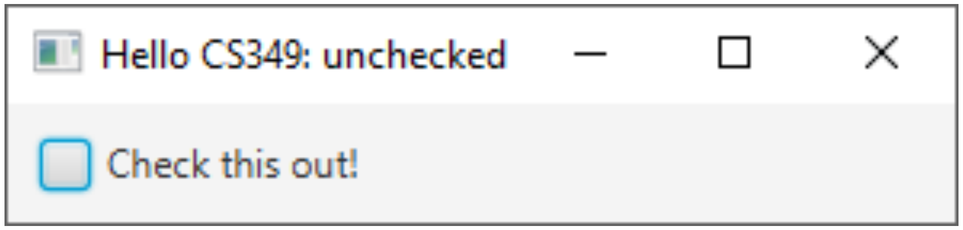


Boolean Selection Widgets

Boolean selection widgets allow users to select between two states. These widgets include `CheckBox` and `ToggleButton`.

```
val myCheck = CheckBox("Check this out!").apply {
    isSelected = true
    padding = Insets(10.0)
    selectedProperty().addListener { _, _, newValue -> // Boolean
        stage.title =
            "Hello CS349: ${if (newValue.not()) "un" else ""}checked"
    }
}
```

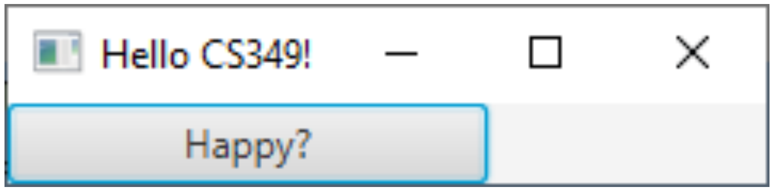


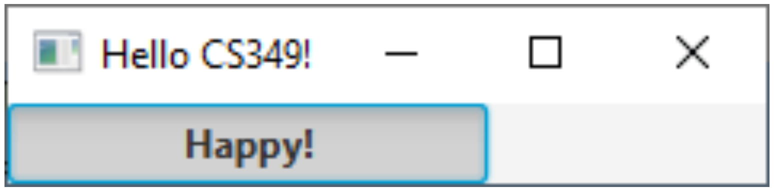


Boolean Selection Widgets

Boolean selection widgets allow users to select between two states. These widgets include `CheckBox` and `ToggleButton`.

```
val myToggle = ToggleButton("Happy?").apply {
    minWidth = 150.0
    selectedProperty().addListener { _, _, newValue -> // Boolean
        text = "${text.dropLast(1)}${if (newValue) "!" else "?"}"
        font = Font.font(null,
                         if(newValue) FontWeight.BOLD
                         else FontWeight.NORMAL,
                         -1.0)
    }
}
```

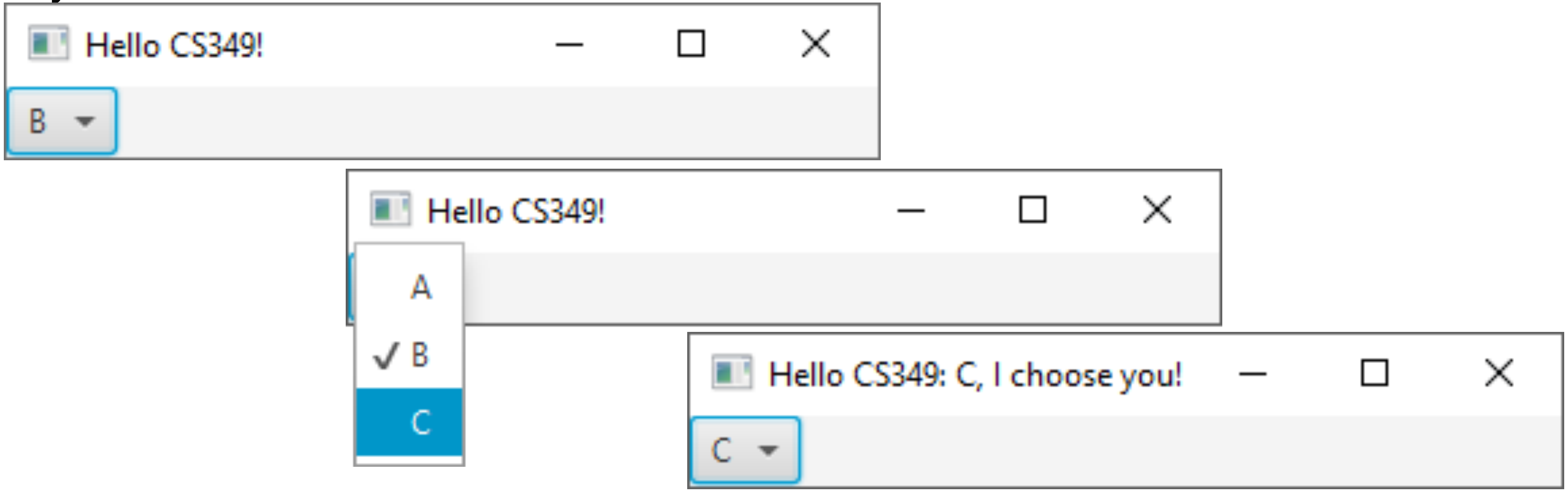




Discrete Selection Widgets

Discrete selection widgets allow users to select **one** of an arbitrary number of entries. These widgets include `ChoiceBox` and `RadioButton`.

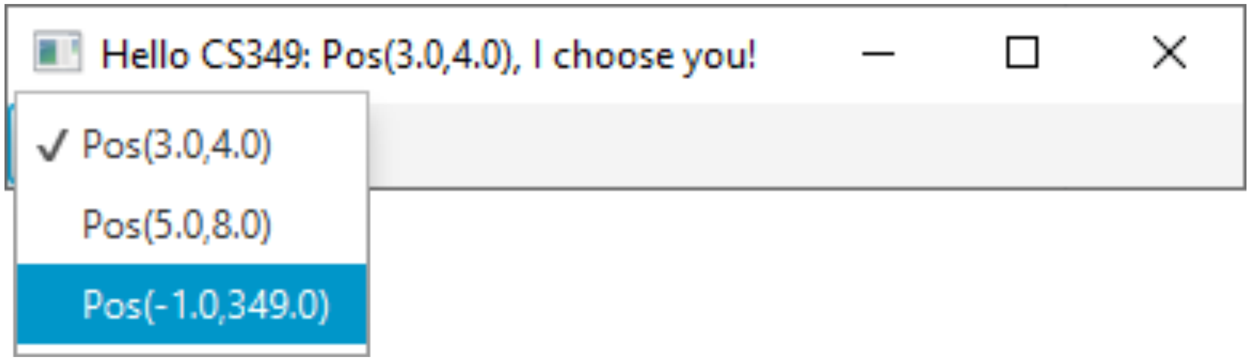
```
val myChoice = ChoiceBox<String>().apply {
    items.addAll("A", "B", "C")
    value = items[1]
    maxWidth = 150.0
    valueProperty().addListener { _, _, newValue -> // String
        stage.title = "Hello CS349: $newValue, I choose you!"
    }
}
```



Discrete Selection Widgets

The following approach uses the custom class `Posn` and accesses the underlying `SelectionModel` of the `ChoiceBox`.

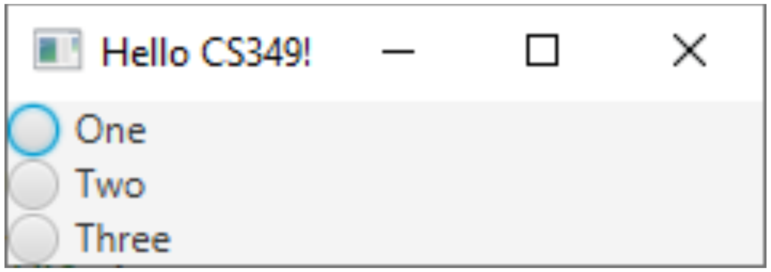
```
class Posn(val x: Double, val y: Double) {
    override fun toString(): String { return "Pos($x,$y)" }
}

val cbItems = listOf(Posn(3.0, 4.0), Posn(5.0, 8.0), Posn(-1.0, 349.0))
val myDrop = ChoiceBox(FXCollections.observableList(cbItems)).apply {
    maxWidth = 150.0
    selectionModel.select(1)
    selectionModel.selectedItemProperty().addListener
    { _, _, newValue -> // Posn
        stage.title = "Hello CS349: $newValue, I choose you!"
    }
}
```

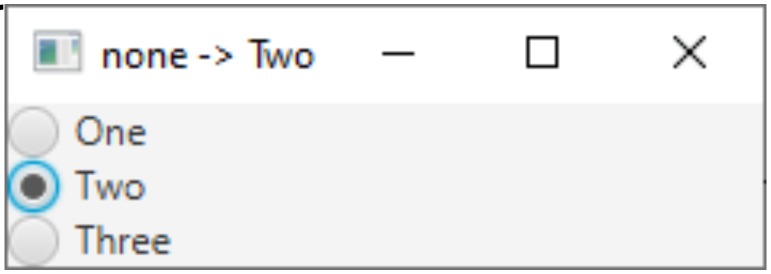

Hello CS349: Pos(3.0,4.0), I choose you!

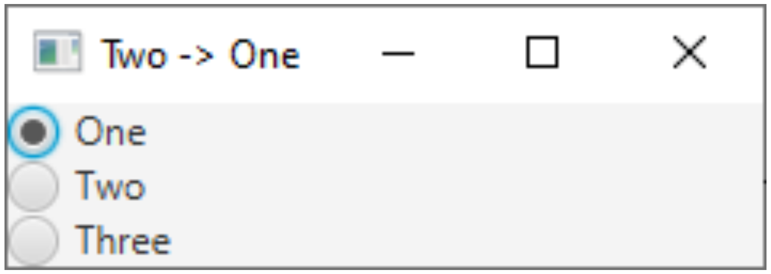
Discrete Selection Widgets

Radio buttons only show the 1-of-n-behaviour if they are grouped within a ToggleGroup.

```
val myRadioA = RadioButton("One")
val myRadioB = RadioButton("Two")
val myRadioC = RadioButton("Three")
ToggleGroup().apply {
    myRadioA.toggleGroup = this;
    myRadioB.toggleGroup = this;
    myRadioC.toggleGroup = this;
    selectedToggleProperty().addListener { _, oldValue, newValue ->
        stage.title = "${(oldValue as RadioButton?)?.text ?: "none"} ->
                       ${(newValue as RadioButton).text}"
    }
}
```



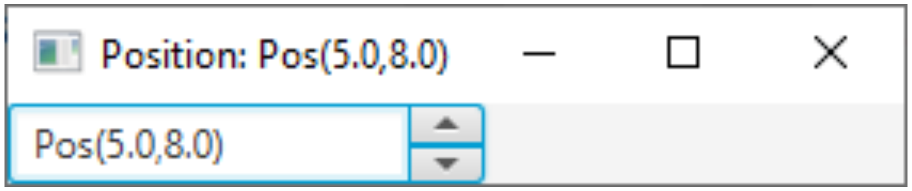




Discrete Selection Widgets

Spinners can be used both for selection between discrete elements or selection for a continuous range of values.

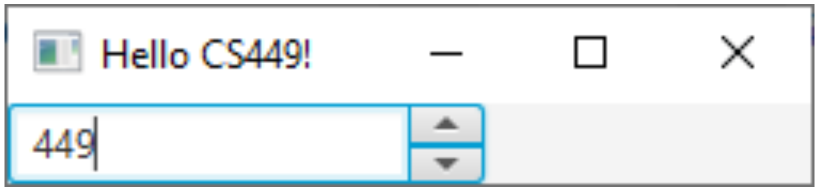
```
class Posn(val x: Double, val y: Double) {
    override fun toString(): String { return "Pos($x,$y)" }
}

val spItems = listOf(Posn(3.0, 4.0), Posn(5.0, 8.0), Posn(-1.0, 349.0))
val mySpinner = Spinner<Posn>().apply {
    valueFactory =
        ListSpinnerValueFactory(FXCollections.observableList(spItems))
    valueFactory.value = spItems.last()
    maxWidth = 150.0
    valueProperty().addListener { _, _, newValue -> // Posn
        stage.title = "Position: ${newValue}!"
    }
}
```



Continuous Selection Widgets

Spinners can be used both for selection between discrete elements or selection for a continuous range of values.

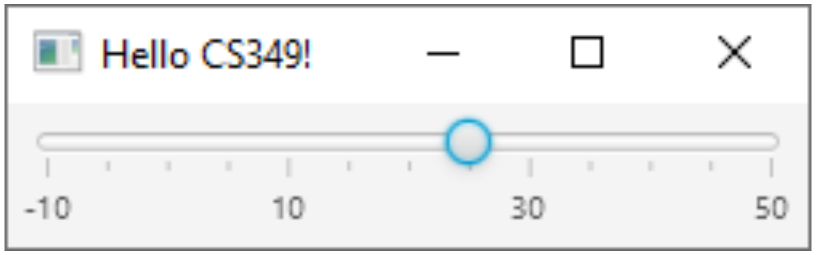
```
val mySpinner = Spinner<Double>(100.0, 499.0, 349.0).apply {
    maxWidth = 150.0
    isEditable = true
    valueProperty().addListener { _, _, newValue -> // Double
        stage.title = "Hello CS${newValue.toInt()}!"
    }
}
```



Continuous Selection Widgets

Continuous selection widgets allow users to select a value from within a continuous range of values, prominently, integers or real numbers.

```
val mySlider = Slider(-10.0, 50.0, 25.0).apply {
    padding = Insets(5.0)
    isShowTickLabels = true
    isShowTickMarks = true
    isSnapToTicks = true
    majorTickUnit = 20.0
    minorTickCount = 3
    minWidth = 250.0
    valueProperty().addListener { _, _, newValue -> // Double
        stage.title = "${newValue.toInt()}"
    }
}
```



Properties & Property Binding

Properties

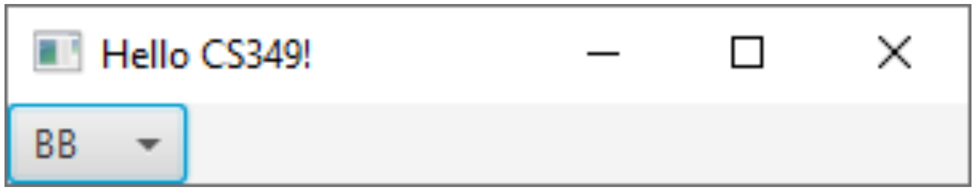
In JavaFX, a property is a special type of class member, that

- stores a value that controls the appearance or behaviour of a widget
- can be set manually or programmatically
- can have a listener attached
- can be bound to a property of another class, so that when one changes, the other is changed automatically

“Binding” Properties via Listener

We can react to state changes by listening to the corresponding property (here: `valueProperty`) and updating another field (here: `stage.title`) manually.

```
override fun start(stage: Stage) {

    val myChoice = ChoiceBox<String>().apply {
        items.addAll("A", "BB", "CCC")
        value = items[1]
        valueProperty().addListener { _, oldValue, newValue ->
            stage.title = $newValue"
        }
    }

    stage.apply {
        scene = Scene(Pane(myChoice), 300.0, 200.0)
        title = "Hello CS349!"
    }.show()
}
```

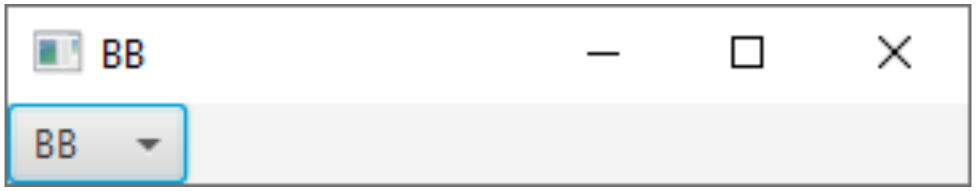


Binding Properties Directly

Alternatively, we can bind two properties together (here: `myChoice.valueProperty` to `stage.titleProperty`). If the first one changes, the other is automatically updated.

```
override fun start(stage: Stage) {

    val myChoice = ChoiceBox<String>().apply {
        items.addAll("A", "BB", "CCC")
        value = items[1]
        stage.titleProperty().bind(valueProperty())
    }

    stage.apply {
        scene = Scene(Pane(myChoice), 300.0, 200.0)
    }.show()
}
```

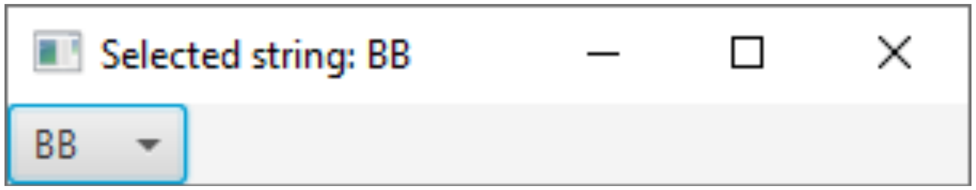
If the data does not need to modified, we can use 1:1 binding.



Binding Properties via Custom Binding

If the data needs to modified, we have to create bindings manually (here: a binding that concatenates two strings).

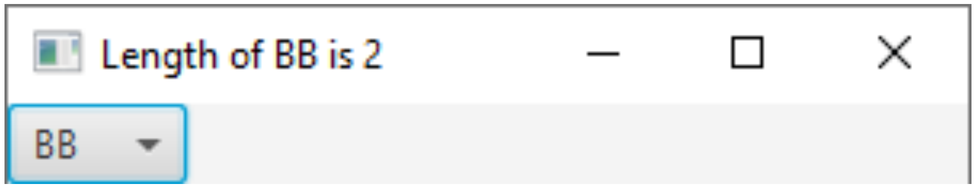
```
override fun start(stage: Stage) {

    val myChoice = ChoiceBox<String>().apply {
        items.addAll("A", "BB", "CCC")
        value = items[1]
        stage.titleProperty().bind(Bindings.concat("Selected string: ",
                                                   valueProperty()))
    }

    stage.apply {
        scene = Scene(Pane(myChoice), 300.0, 200.0)
    }.show()
}
```


Selected string: BB

Binding Properties via Custom Binding

If the data needs to modified, we have to create bindings manually (here: a binding that concatenates multiple strings and converts `valueProperty` into an `Int`).

```
override fun start(stage: Stage) {

    val myChoice = ChoiceBox<String>().apply {
        items.addAll("A", "BB", "CCC")
        value = items[1]
        stage.titleProperty().bind(Bindings.concat(
            "Length of ",
            valueProperty(),
            " is ",
            Bindings.createIntegerBinding({ valueProperty().value.length },
                                          valueProperty())))
    }

    stage.apply {
        scene = Scene(Pane(myChoice), 300.0, 200.0)
    }.show()
}
```


Length of BB is 2

End of the Chapter

Please make sure to

- Be aware of the difference between logic inputs and widgets
- Have a rough understanding about the differences of widgets that implement the same logic input
- Remember which widgets are available
- Properties exist, they can be bound together

Any further questions?