Layouts

Types of Layouts
Layouts in JavaFX
Designing UI Components using Layouts

UI CS 349

May 24

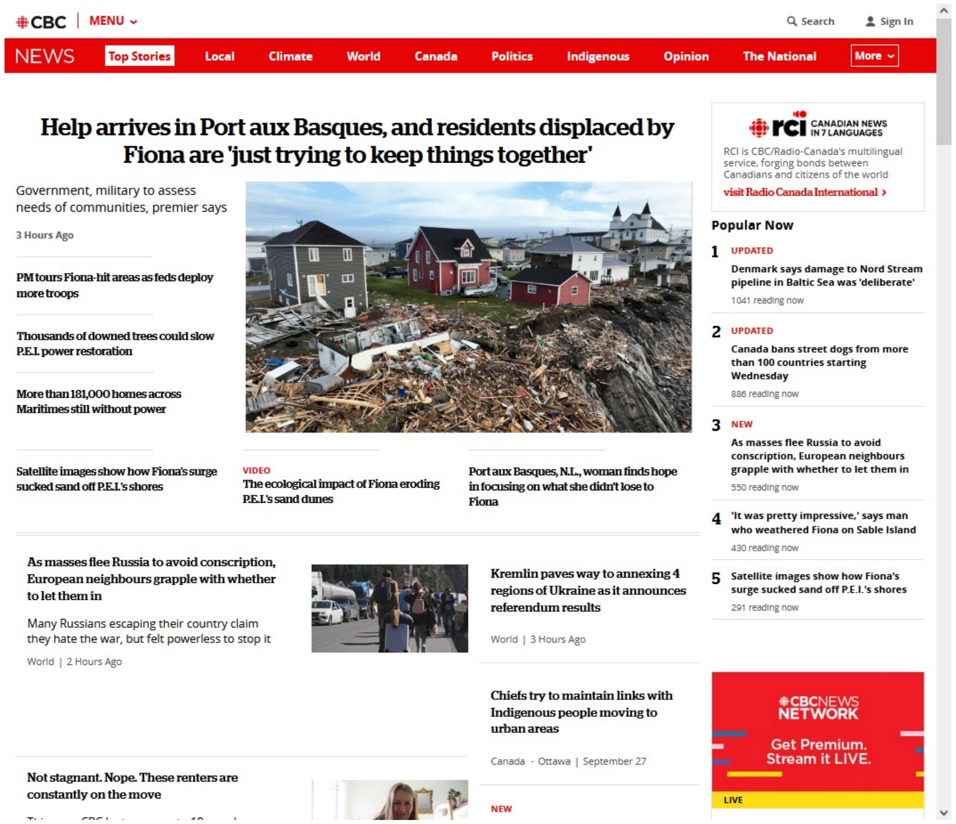
Types of Layouts

Dynamic Layout

Applications need to be able to adjust the presentation of our interfaces.

We need to dynamically reposition and resize our content in response to:

- Change in screen resolution (e.g., different computers or devices).
- Resizing the application window (e.g., user adjustments).



Help arrives in Port aux Basques, and residents displaced by Fiona are 'just trying to keep things together'

Government, military to assess needs of communities, premier says

3 Hours Ago

PM tours Fiona-hit areas as feds deploy more troops

Thousands of downed trees could slow P.E.I. power restoration

More than 181,000 homes across Maritimes still without power

Satellite images show how Fiona's surge sucked sand off P.E.I.'s shores

VIDEO The ecological impact of Fiona eroding P.E.I.'s sand dunes

Port aux Basques, N.L., woman finds hope in focusing on what she didn't lose to Fiona

As masses flee Russia to avoid conscription, European neighbours grapple with whether to let them in

Many Russians escaping their country claim they hate the war, but felt powerless to stop it

World | 2 Hours Ago

Kremlin paves way to annexing 4 regions of Ukraine as it announces referendum results

World | 3 Hours Ago

Chiefs try to maintain links with Indigenous people moving to urban areas

Canada · Ottawa | September 27

Not stagnant. Nope. These renters are constantly on the move

NEW

rci CANADIAN NEWS IN 7 LANGUAGES

RCI is CBC/Radio-Canada's multilingual service, forging bonds between Canadians and citizens of the world

visit Radio Canada International ›

Popular Now

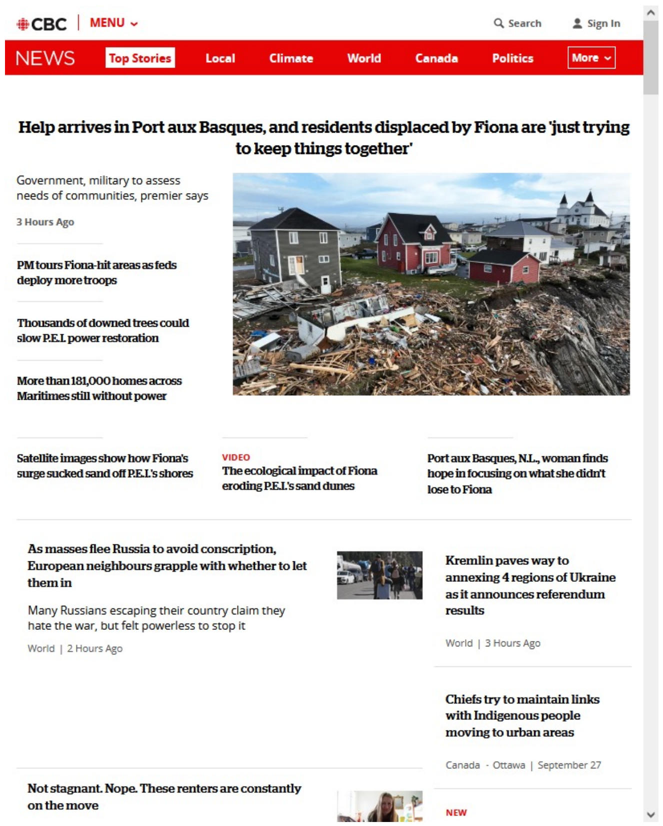
1 UPDATED Denmark says damage to Nord Stream pipeline in Baltic Sea was 'deliberate' 1041 reading now

2 UPDATED Canada bans street dogs from more than 100 countries starting Wednesday 886 reading now

3 NEW As masses flee Russia to avoid conscription, European neighbours grapple with whether to let them in 550 reading now

4 'It was pretty impressive,' says man who weathered Fiona on Sable Island 430 reading now

5 Satellite images show how Fiona's surge sucked sand off P.E.I.'s shores 291 reading now



Help arrives in Port aux Basques, and residents displaced by Fiona are 'just trying to keep things together'

Government, military to assess needs of communities, premier says

3 Hours Ago

PM tours Fiona-hit areas as feds deploy more troops

Thousands of downed trees could slow P.E.I. power restoration

More than 181,000 homes across Maritimes still without power

Satellite images show how Fiona's surge sucked sand off P.E.I.'s shores

VIDEO The ecological impact of Fiona eroding P.E.I.'s sand dunes

Port aux Basques, N.L., woman finds hope in focusing on what she didn't lose to Fiona

As masses flee Russia to avoid conscription, European neighbours grapple with whether to let them in

Many Russians escaping their country claim they hate the war, but felt powerless to stop it

World | 2 Hours Ago

Kremlin paves way to annexing 4 regions of Ukraine as it announces referendum results

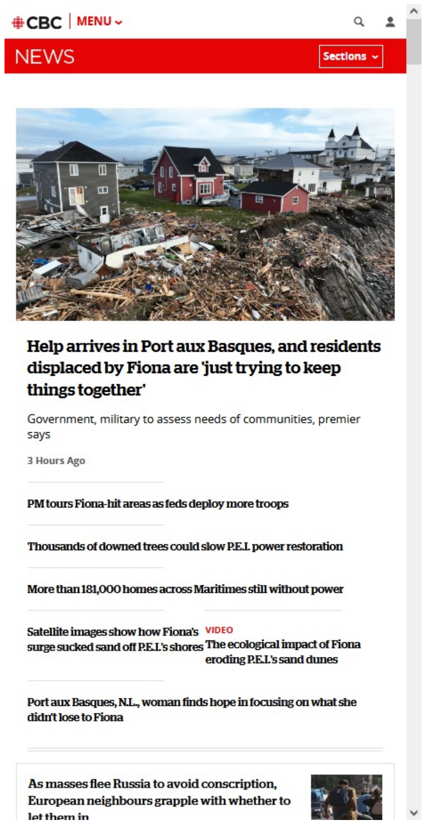
World | 3 Hours Ago

Chiefs try to maintain links with Indigenous people moving to urban areas

Canada · Ottawa | September 27

Not stagnant. Nope. These renters are constantly on the move

NEW



Help arrives in Port aux Basques, and residents displaced by Fiona are 'just trying to keep things together'

Government, military to assess needs of communities, premier says

3 Hours Ago

PM tours Fiona-hit areas as feds deploy more troops

Thousands of downed trees could slow P.E.I. power restoration

More than 181,000 homes across Maritimes still without power

Satellite images show how Fiona's surge sucked sand off P.E.I.'s shores

VIDEO The ecological impact of Fiona eroding P.E.I.'s sand dunes

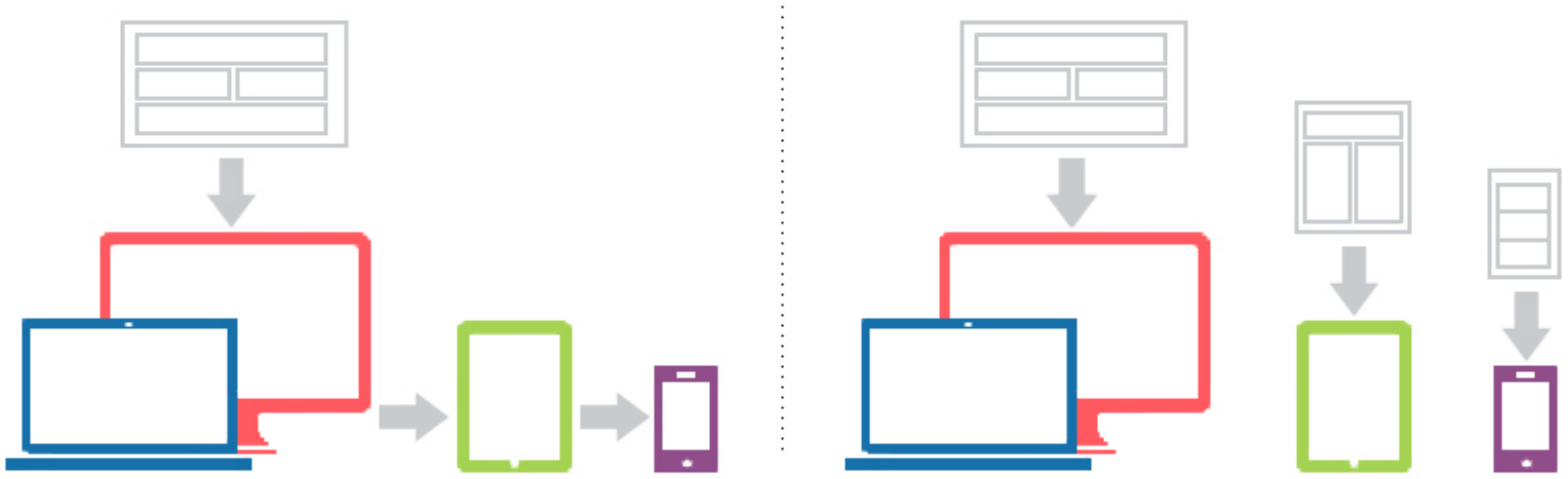
Port aux Basques, N.L., woman finds hope in focusing on what she didn't lose to Fiona

As masses flee Russia to avoid conscription, European neighbours grapple with whether to let them in

Strategies for Dynamic Layout

There are two main strategies for handling dynamic layout:

- **Responsive**: support a universal design that reflows a spatial layout to fit the dimensions of the current view (i.e. device or window).
- **Adaptive**: design optimized spatial layouts for each of your devices, and dynamically switch to fit the device that is in use.



Responsive Layout

We are going to focus on responsive layout: adapting to changes dynamically.

To dynamically adjust content to a window, we want to:

- maximize use of available space for displaying widgets,
- while maintaining consistency with spatial layout, and
- preserving the visual quality of spatial layout

This requires that our application can dynamically adjust elements:

- re-allocate space for widgets
- adjust location and size of widgets
- perhaps change visibility, look, and / or feel of widgets

The Role of Container Nodes

`Container nodes` describe how their children should be placed.

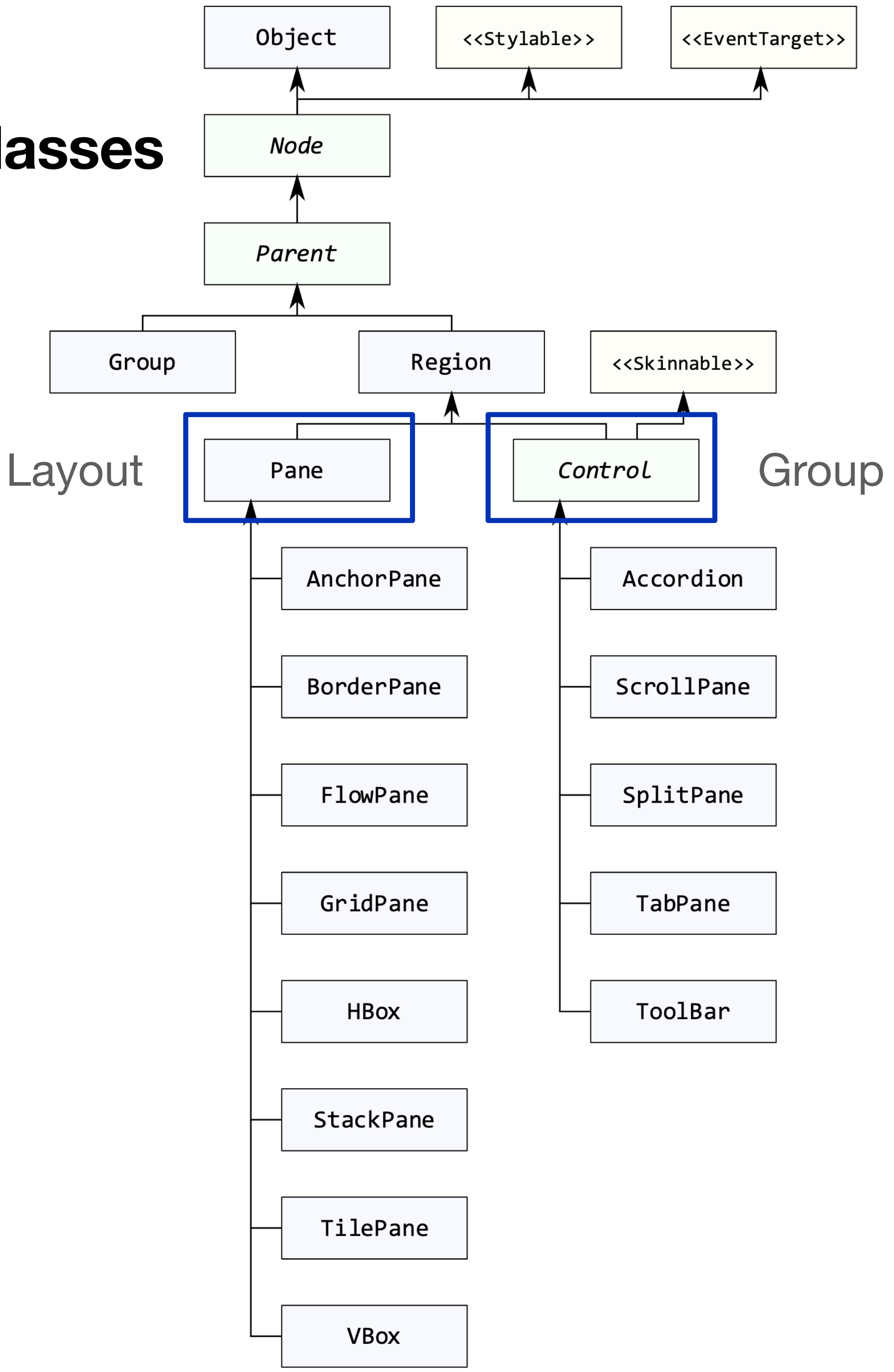
These containers may have different strategies for handling layout. For example, `Group` lets the designer position children directly, while `StackPane` tries to re-position its children in the centre of the screen.

The designer's role is to determine which containers (layouts) to combine to get the desired effect. Often it takes multiple containers, sometimes nested together, to achieve this.

Nodes have properties that control their position, size and other characteristics. The container will set the size and location of its children according to its internal rules, to size and position content appropriately.

Layouts in JavaFX

JavaFX Container Classes



Layout Strategies

The approaches that containers use can be grouped into one of these styles:

- **Fixed Layout**: non-resizable
- **Variable Intrinsic Layout**: adjusting widget size and position
- **Relative Layout**: positioning components relative to one another
- **Custom Layout**: define your own!

Fixed Layouts

The layout does not move or resize nodes by itself. You need to *manually* specify location and position of all nodes within the layout.

This is most suitable for cases when you have a fixed-size window.

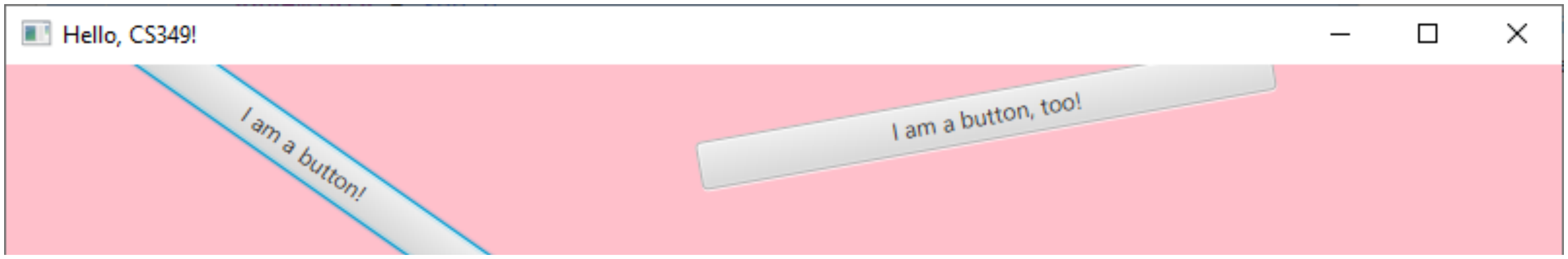
Containers that support fixed layout:

- `Group`
- `Pane`

Container – `Group`

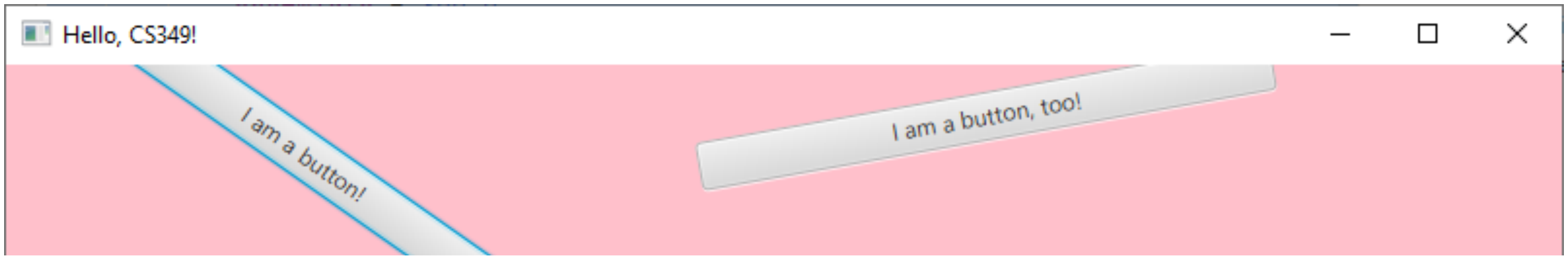
A `Group` contains children that are rendered in order whenever this node is rendered. The `Group` will take on the collective bounds of its children and is *not directly resizable*.

Any transform, effect, or state applied to a Group will be applied to all children of that group.



Container – Group

```
override fun start(stage: Stage) {
    val bt1 = Button("I am a button!").apply {
        minWidth = 300.0
        rotate = 45.0
    }
    val bt2 = Button("I am a button, too!").apply {
        minWidth = 300.0
        translateX = 349.0
        translateY = 42.0
    }
    val root = Group(bt1, bt2).apply { rotate = -10.0 }

    stage.apply {
        title = "Hello, CS349!"
        scene = Scene(root , 800.0, 100.0).apply { fill = Color.PINK }
    }.show()
}
```

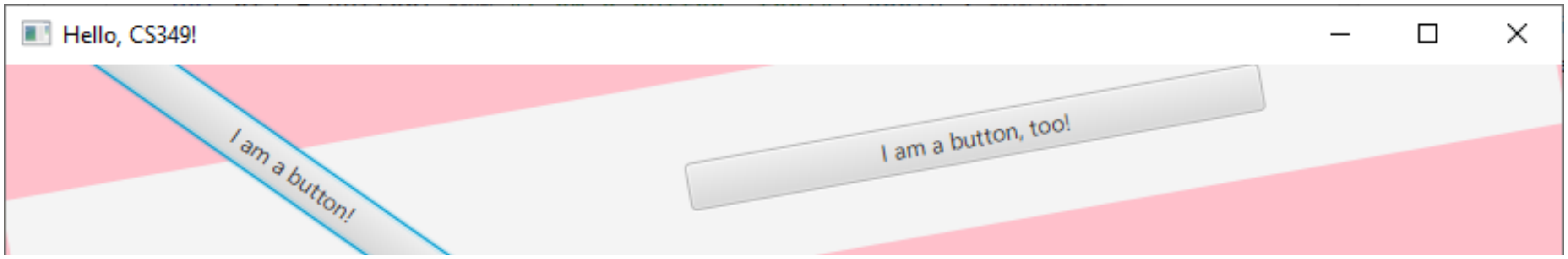


Container – `Pane`

A `Pane` can be used directly in cases where absolute positioning of children is required, since it does not perform layout beyond resizing children to their preferred sizes.

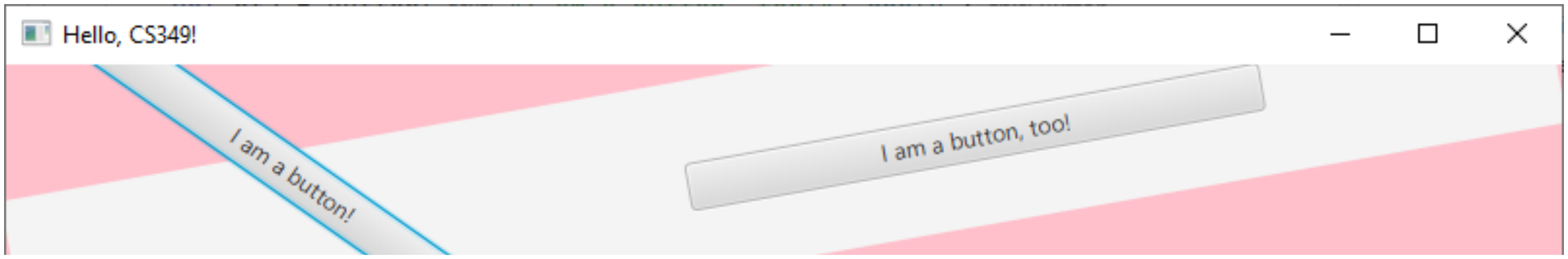
It becomes the application's responsibility to position the children since the pane leaves the positions alone during layout.

Unlike a `Group`, a `Pane` has its own width and height and can be styled independently of its children.



Container – Pane

```
override fun start(stage: Stage) {
    val bt1 = Button("I am a button!").apply {
        minWidth = 300.0
        rotate = 45.0
    }
    val bt2 = Button("I am a button, too!").apply {
        minWidth = 300.0
        translateX = 349.0
        translateY = 42.0
    }
    val root = Pane(bt1, bt2).apply { rotate = -10.0 }

    stage.apply {
        title = "Hello, CS349!"
        scene = Scene(root , 800.0, 100.0).apply { fill = Color.PINK }
    }.show()
}
```



Variable Intrinsic Layouts

This layout algorithm attempts to use the widget's preferred sizes. It recursively queries all widgets and then allocates space to them as a group.

Layout is determined in two-passes (bottom-up, top-down):

1. Get each child widget's preferred size (includes recursively asking all of its children for their preferred size…)
2. Decide on a layout that satisfies everyone's preferences, then iterate through each child, and set its layout (size / position)

Containers that support variable intrinsic layout:

- `VBox`
- `HBox`
- `FlowPane`

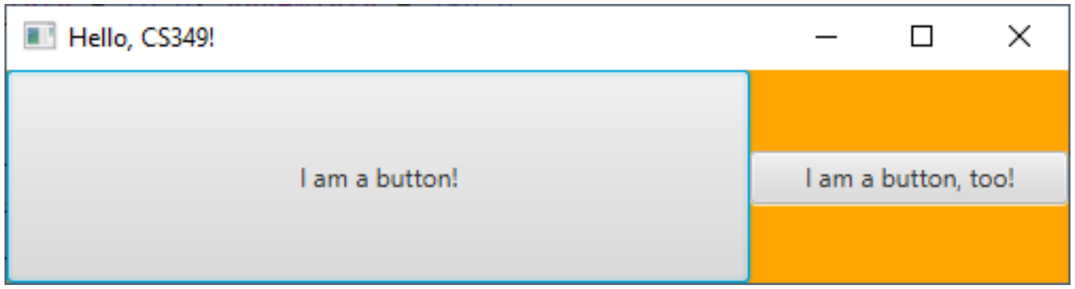
Container – `HBox`, `VBox`

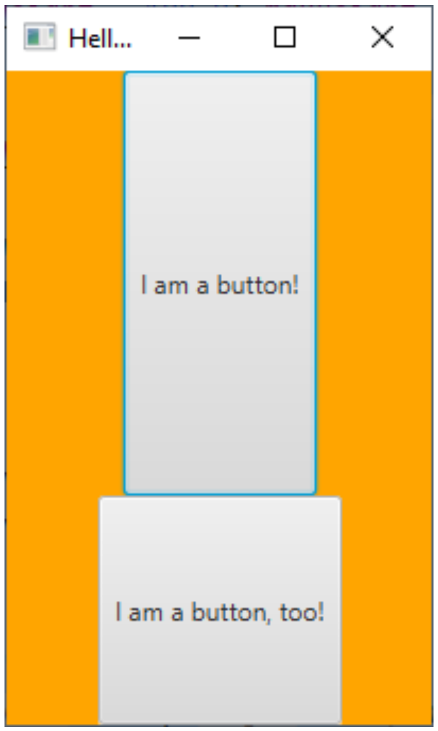
{`HBox`|`VBox`} lays out its children in a single {row|column}.

They will resize children (if resizable) to their preferred {widths|heights} and use its {`fillHeight`|`fillWidth`} property to determine whether to resize their {heights|widths} to fill its own.

The alignment of the content is controlled by the alignment property.

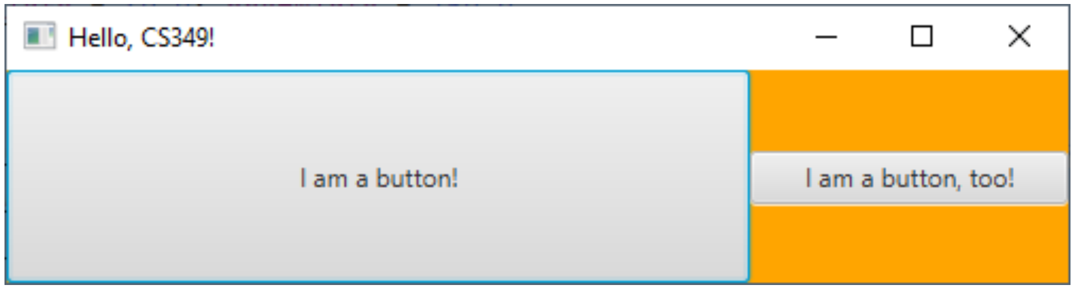
If an {`HBox`|`VBox`} is resized larger than its preferred {width|height}, it will by default leave the extra space unused. To have one or more children be allocated that extra space it may optionally set an {`hgrow`|`vgrow`} constraint on the child.





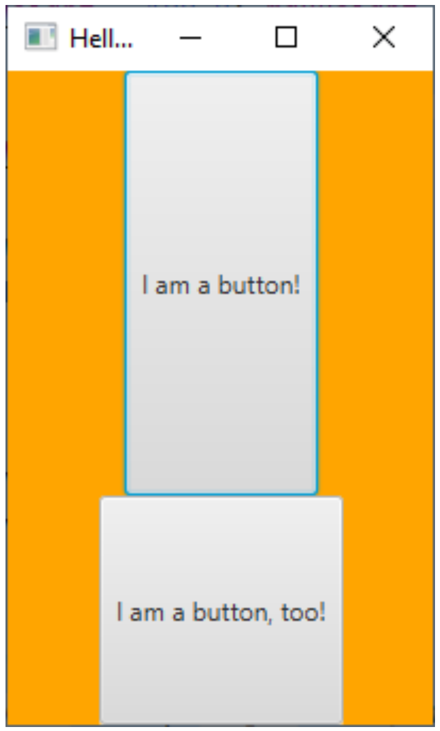
Container – HBox, VBox

```
override fun start(stage: Stage) {
    val bt1 = Button("I am a button!").apply {
        minWidth = 60.0; prefWidth = 100.0;
        maxWidth = Double.MAX_VALUE; maxHeight = Double.MAX_VALUE }
    val bt2 = Button("I am a button, too!").apply {
        minWidth = 40.0; prefWidth = 150.0 }
    val root = HBox(bt1, bt2).apply {
        background = Background(BackgroundFill(Color.ORANGE, null, null))
        alignment = Pos.CENTER
        isFillHeight = true }

    HBox.setHgrow(bt1, Priority.ALWAYS)
    HBox.setHgrow(bt2, Priority.NEVER)

    stage.apply {
        title = "Hello, CS349!"
        scene = Scene(root , 500.0, 100.0)
    }.show()
}
```



Container – HBox, VBox

```
override fun start(stage: Stage) {
    val bt1 = Button("I am a button!").apply {
        minHeight = 60.0; prefHeight = 100.0; maxHeight = 200.0 }

    val bt2 = Button("I am a button, too!").apply {
        minHeight = 40.0; prefHeight = 100.0; maxHeight = 200.0 }
    val root = VBox(bt1, bt2).apply {
        background = Background(BackgroundFill(Color.ORANGE, null, null))
        alignment = Pos.CENTER }


    VBox.setVgrow(bt1, Priority.ALWAYS)
    VBox.setVgrow(bt2, Priority.SOMETIMES)

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



Widget Dimensions

Widgets need to be flexible in size and position

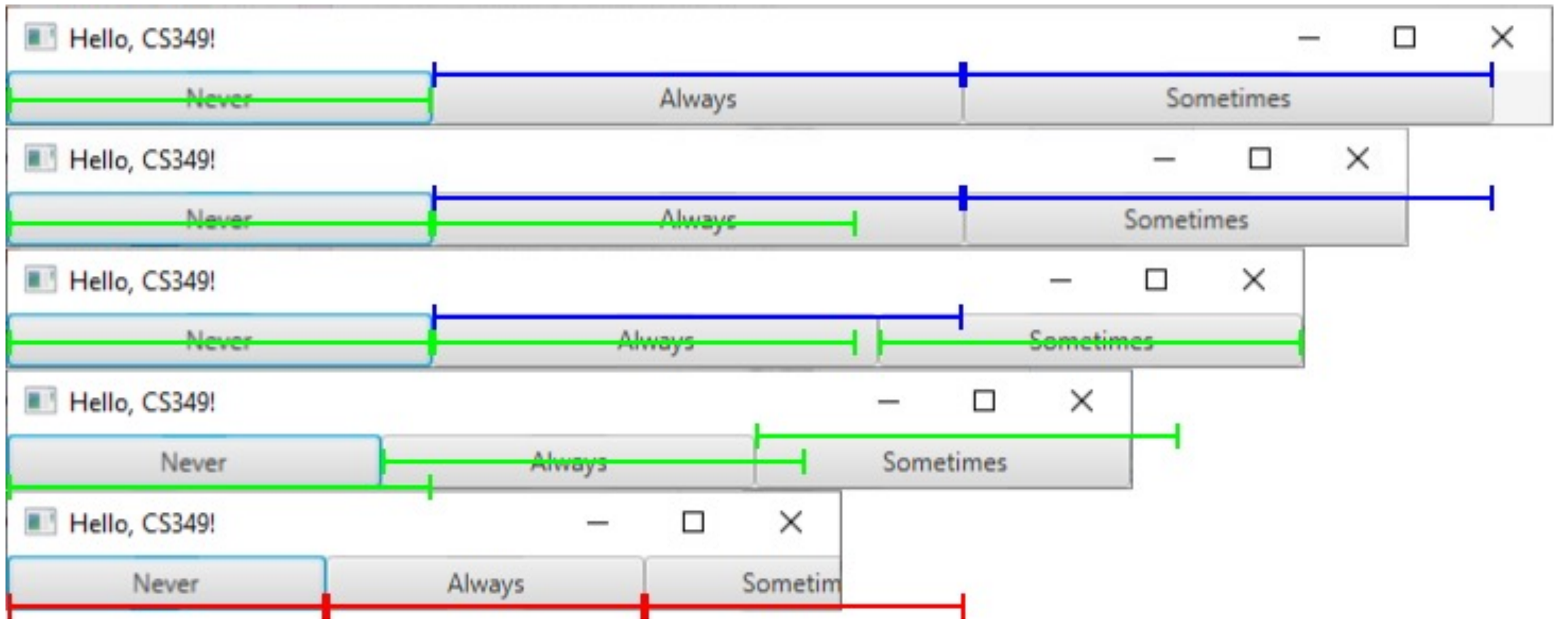
- Widgets store their own position and width / height, but containers have the ability to change these properties.
- Other properties may also be changed by containers (e.g., reducing font size for a caption)

Widgets give the layout algorithm a range of preferred values as “hints”, and containers considers the size hints of nodes in determining layout.

- **minWidth | minHeight**: parent should not resize node’s {width|height} smaller than this value
- **prefWidth | prefHeight**: parent should treat this value as the node's ideal {width|height}
- **maxWidth | maxHeight**: parent should not resize node’s {width|height} larger than this value

Widget Dimensions

```
val root = HBox(Button("Never").apply {
    minWidth = 150.0; prefWidth = 200.0; maxWidth = 250.0
    HBox.setHgrow(this, Priority.NEVER)
}, Button("Always").apply {
    minWidth = 150.0; prefWidth = 200.0; maxWidth = 250.0
    HBox.setHgrow(this, Priority.ALWAYS)
}, Button("Sometimes").apply {
    minWidth = 150.0; prefWidth = 200.0; maxWidth = 250.0
    HBox.setHgrow(this, Priority.SOMETIMES)
})
```

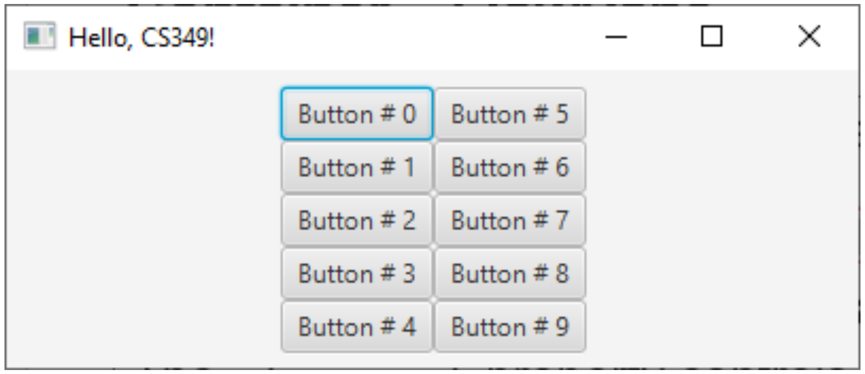


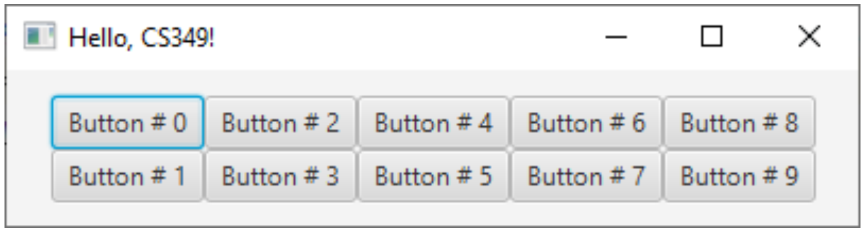
Container – `FlowPane`

`FlowPane` lays out its children in a flow that wraps at the flowpane's boundary. A {horizontal|vertical} flowpane will layout nodes in {rows|columns}, wrapping at the flowpane’s {width|height}.

The prefWrapLength property establishes its preferred {width| height}.

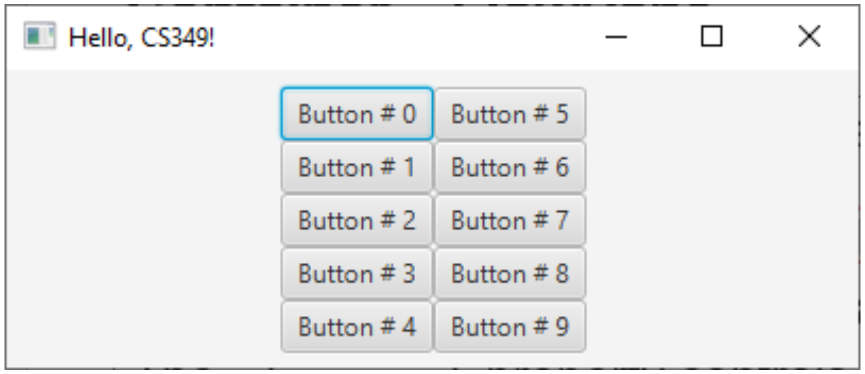
The alignment property controls how the {rows|columns} are aligned within the bounds of the flowpane.





Container – `FlowPane`

```
override fun start(stage: Stage) {
    val root = FlowPane(Orientation.VERTICAL).apply {
        (0..9).forEach() { children.add(Button("Button # $it")) }
        alignment = Pos.CENTER
    }

    stage.apply {
        title = "Hello, CS349!"
        scene = Scene(root , 400.0, 200.0)
    }.show()
}
```



Relative Layouts

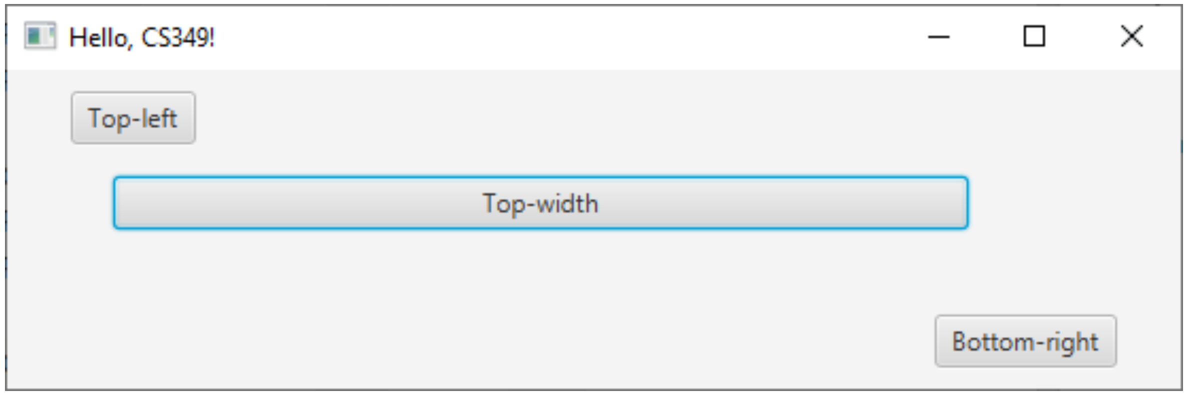
The layout constrains child positions into a specific layout.

Containers that support relative layout:

- `AnchorPane`
- `BorderPane`
- `GridPane`
- `TilePane`

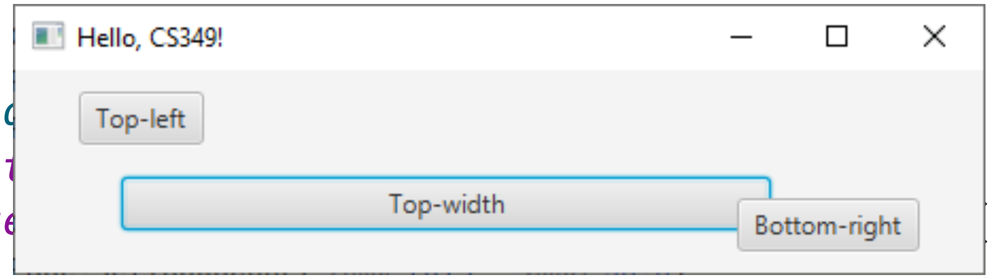
Container – `AnchorPane`

`AnchorPane` allows the edges of child nodes to be anchored to an offset from the anchor pane's edges.



Container – AnchorPane

```
override fun start(stage: Stage) {
    val root = AnchorPane().apply {
        children.add(0, Button("Top-left").apply {
            AnchorPane.setTopAnchor(this, 10.0)
            AnchorPane.setLeftAnchor(this, 30.0)
        })
        children.add(1, Button("Bottom-right").apply {
            AnchorPane.setBottomAnchor(this, 10.0)
            AnchorPane.setRightAnchor(this, 30.0)
        })
        children.add(0, Button("Top-width").apply {
            AnchorPane.setTopAnchor(this, 50.0)
            AnchorPane.setLeftAnchor(this, 50.0)
            AnchorPane.setRightAnchor(this, 100.0)
        })
    }

    stage.
        ti
        sce                                             )
}
```



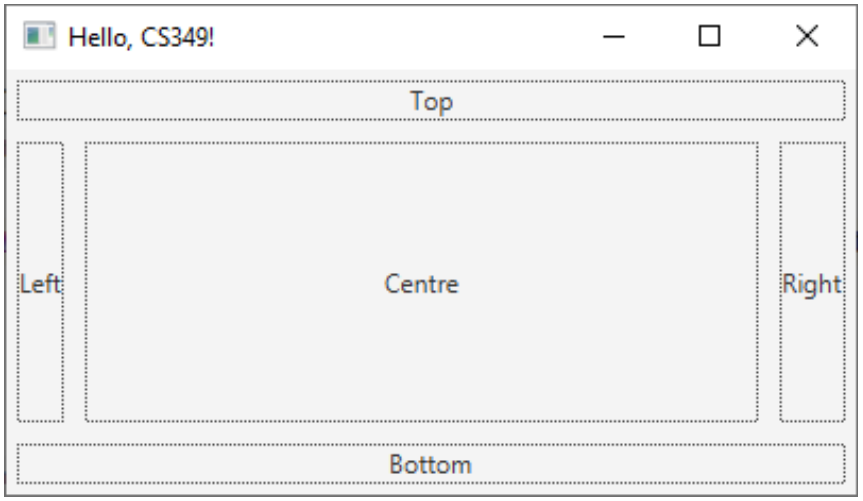
Container – `BorderPane`

`BorderPane` lays out children in `top`, `left`, `right`, `bottom`, and `center`.

The `top` and `bottom` children will be resized to their preferred heights and extend the width of the border pane.

The `left` and `right` children will be resized to their preferred widths and extend the length between the top and bottom nodes.

The center node will be resized to fill the available space in the middle. Any of the positions may be `null`.



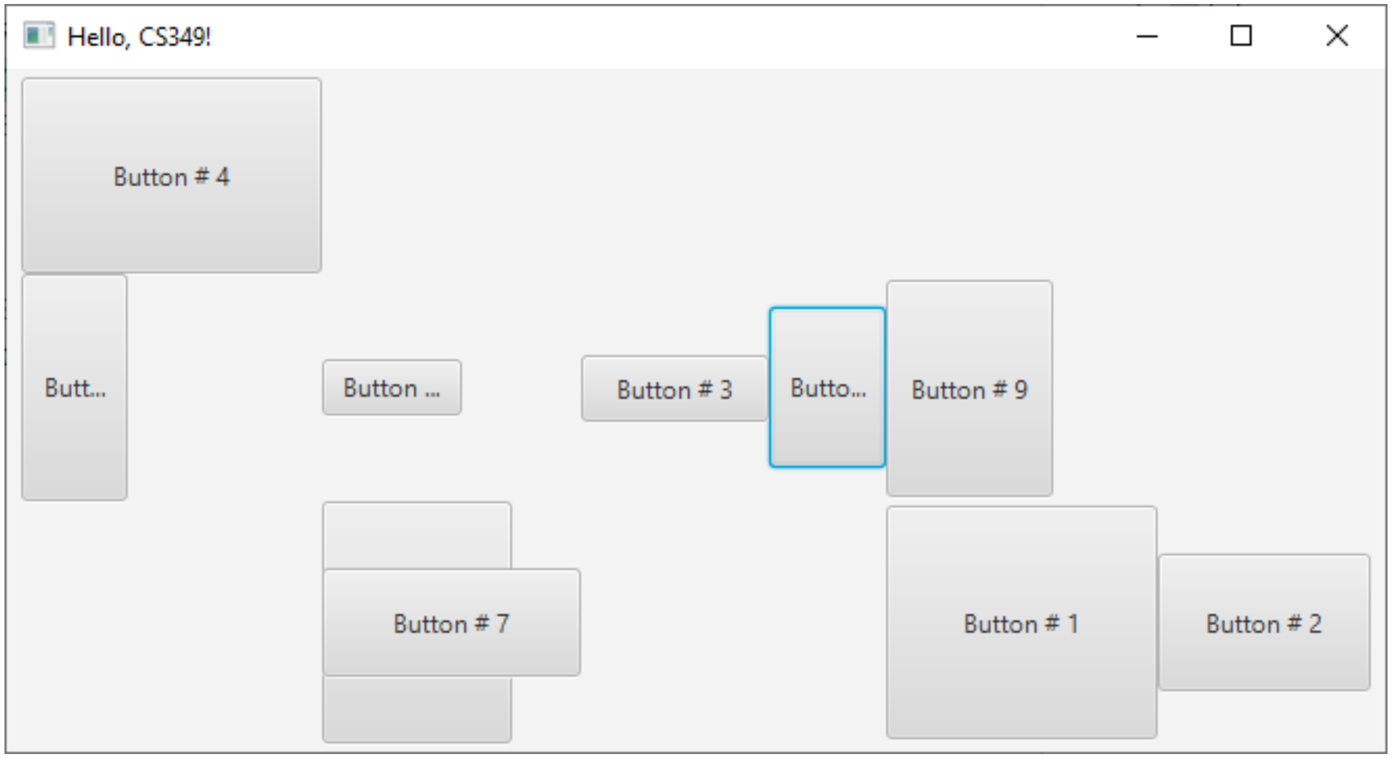
Container – BorderPane

```kotlin
override fun start(stage: Stage) {
    val lc = Label("Centre").apply { maxHeight = Double.MAX_VALUE;
                                     maxWidth = Double.MAX_VALUE }
    val lt = Label("Top").apply { maxWidth = Double.MAX_VALUE }
    val lb = Label("Bottom").apply { maxWidth = Double.MAX_VALUE }
    val ll = Label("Left").apply { maxHeight = Double.MAX_VALUE }
    val lr = Label("Right").apply { maxHeight = Double.MAX_VALUE }

    val root = BorderPane(lc, lt, lr, lb, ll)

    root.children.forEach {
        (it as Label)
        BorderPane.setAlignment(it, Pos.CENTER)
        BorderPane.setMargin(it, Insets(5.0))
        it.alignment = Pos.CENTER;
        it.border= Border(BorderStroke(Color.BLACK,
                  BorderStrokeStyle.DOTTED, null, BorderStroke.THIN))
    }

    stage.apply {
        title = "Hello, CS349!"
        scene = Scene(root , 400.0, 200.0).apply { fill = Color.PINK }
    }.show()
```

Container – `GridPane`

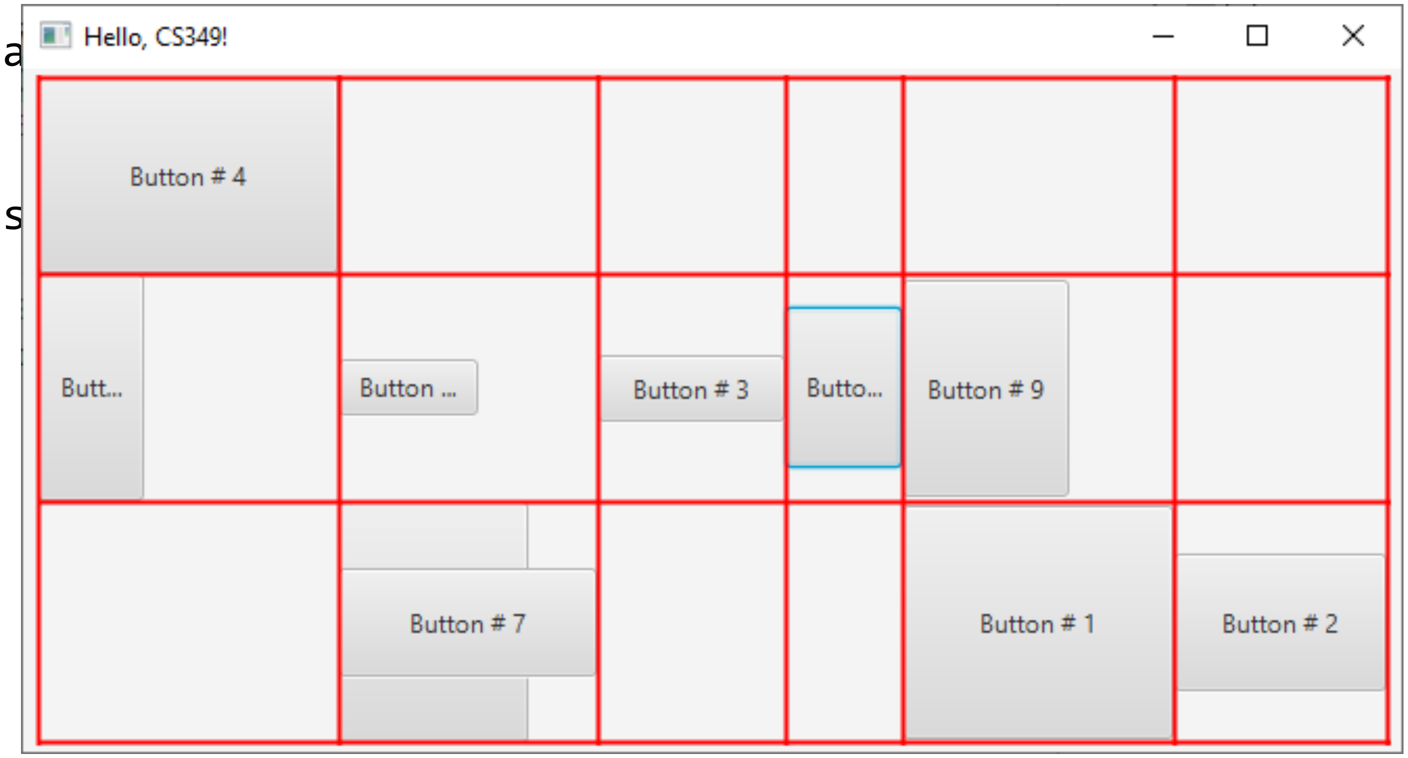
`GridPane` lays out its children within a flexible grid of rows and columns. A child may be placed anywhere within the grid and may span multiple rows / columns.

Children may freely overlap within rows / columns and their stacking order will be defined by the order of the gridpane's children list.



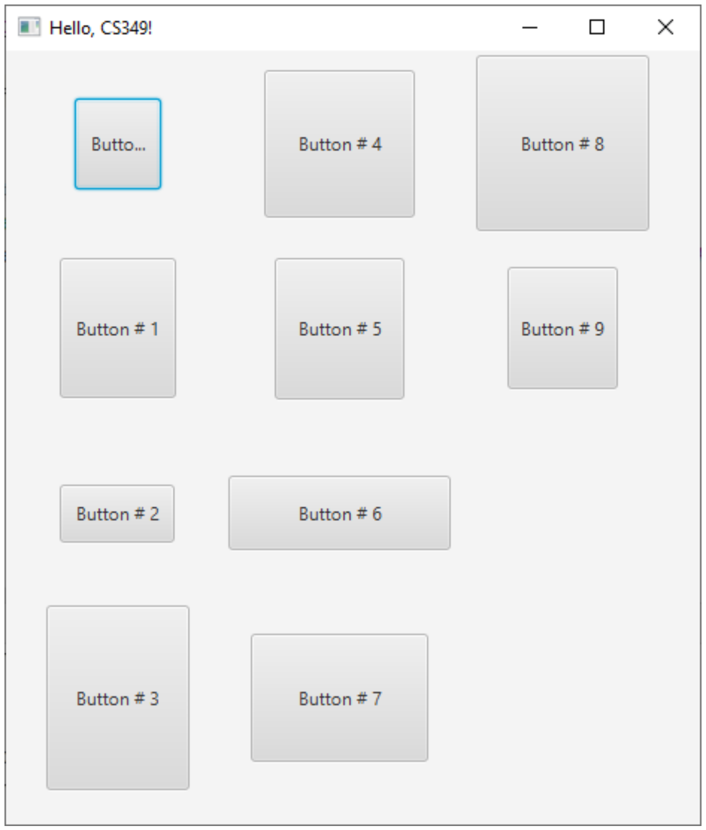
Container – GridPane

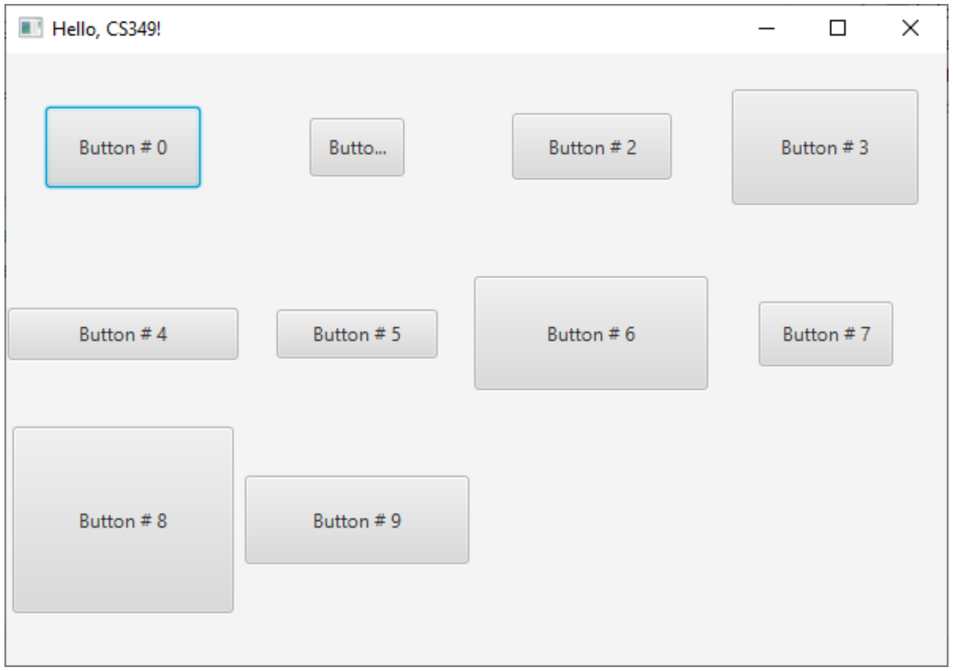
```
override fun start(stage: Stage) {
    val root = GridPane().apply {
        (0..9).forEach() { add(Button("Button # $it").apply {
            prefWidth = Random.nextDouble() * 100 + 50
            prefHeight = Random.nextDouble() * 100 + 25
        }, Random.nextInt(0, 6), Random.nextInt(0, 3)) }
        alignment = Pos.CENTER
    }

    sta

    }.s
}
```



Container – `TilePane`

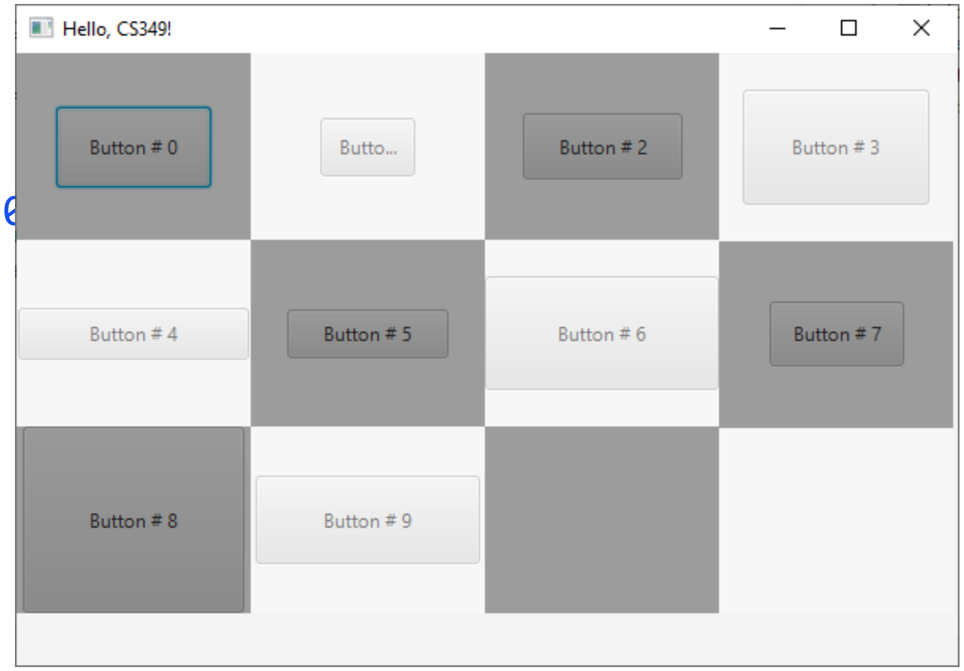
`TilePane` lays out its children in a grid of uniformly sized "tiles". A {horizontal|vertical} tilepane will tile nodes in {rows|columns}, wrapping at the tilepane's {width|height}.





Container – TilePane

```
override fun start(stage: Stage) {
    val root = TilePane(Orientation.HORIZONTAL).apply {
        (0..9).forEach() { children.add(Button("Button # $it").apply {
            prefWidth = Random.nextDouble() * 100 + 50
            prefHeight = Random.nextDouble() * 100 + 25
        })}
        alignment = Pos.TOP_LEFT
        prefRows = 3
    }

    stage.apply {
        title = "Hello, CS349!"
        scene = Scene(root , 400.0
    }.show()
}
```



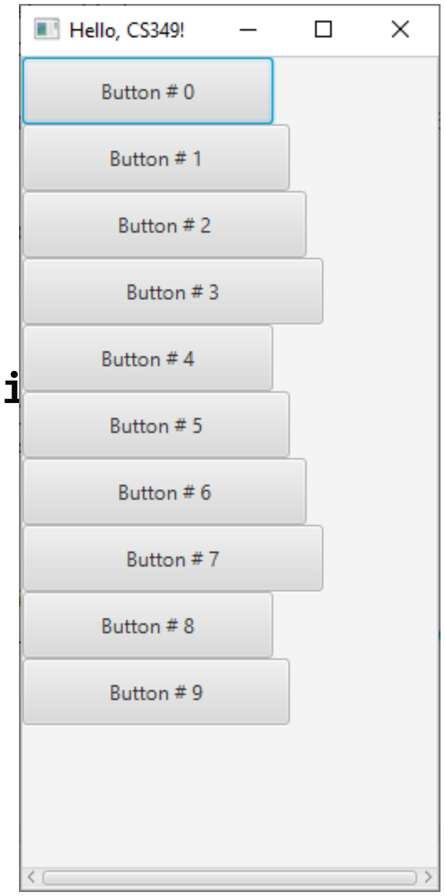
Container – `ScrollPane`

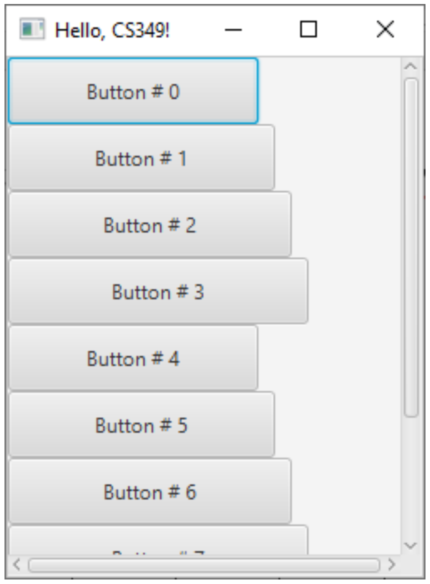
`ScrollPane` provides a scrolled, clipped viewport of its contents. It allows the user to scroll the content around either directly (panning) or by using scroll bars.

It also allows specification of the scroll bar policy, which determines when scroll bars are displayed: always, never, or only when they are needed.

Container – ScrollPane

```
override fun start(stage: Stage) {
    val root = VBox().apply {
        (0..9).forEach() { children.add(Button("Button # $i
            prefWidth = 150.0 + it * 10 % 40
            prefHeight = 40.0
            maxHeight = Double.MAX_VALUE
            VBox.setVgrow(this, Priority.ALWAYS)
        })}
    }

    val scroll = ScrollPane(root).apply {
        hbarPolicy = ScrollPane.ScrollBarPolicy.ALWAYS
        isFitToWidth = true
    }

    stage.apply {
        title = "Hello, CS349!"
        scene = Scene(scroll , 250.0, 500.0)
    }.show()
}
```



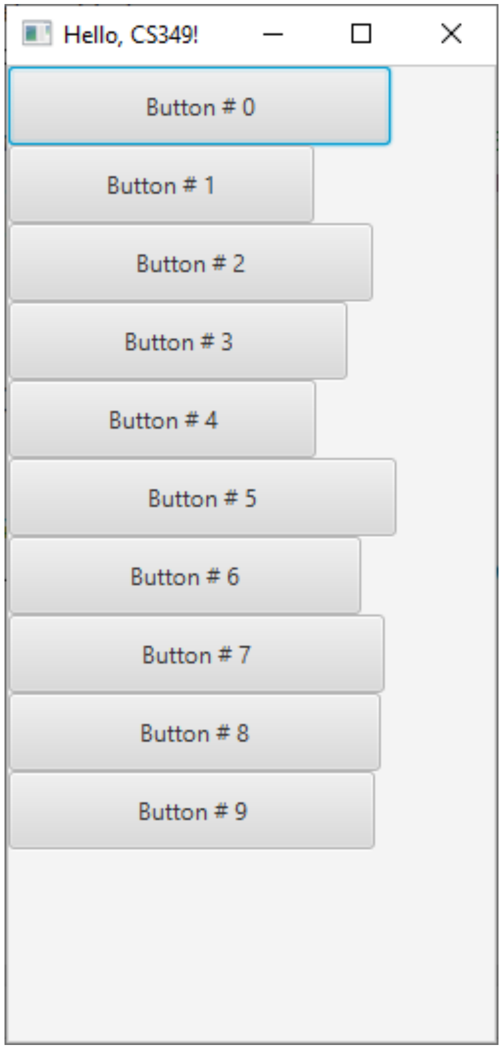


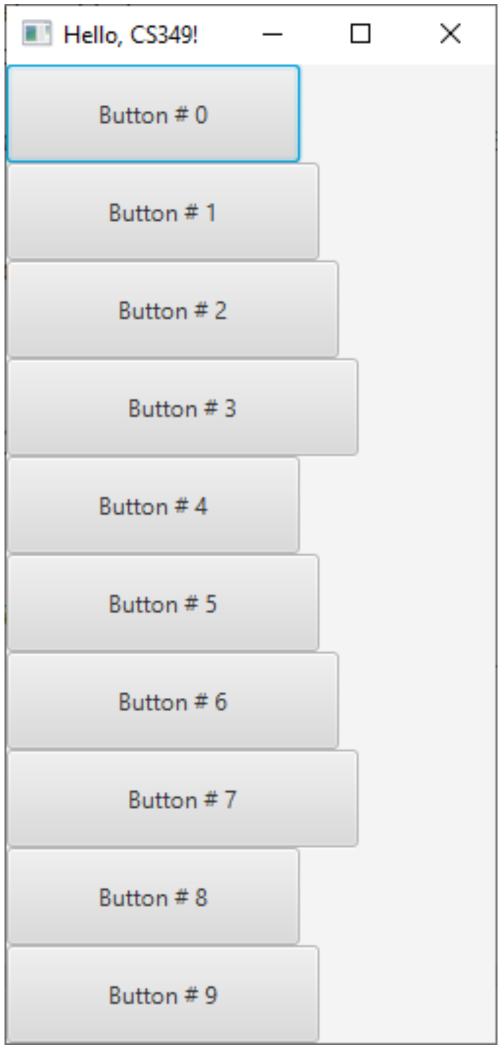
Container – `ScrollPane`

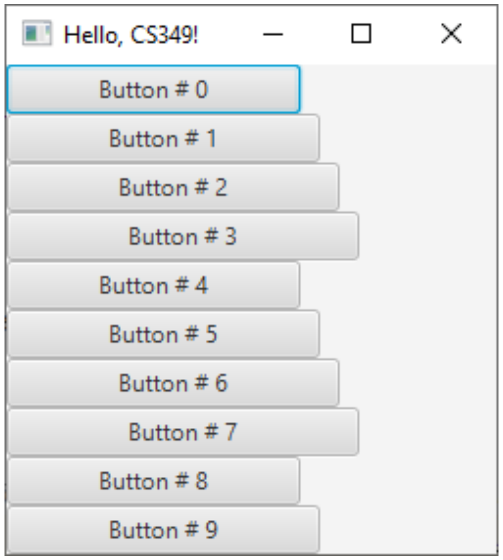
VBox only
enough vertical space
Vgrow NEVER

VBox only
enough vertical space
Vgrow ALWAYS

VBox only
not enough vertical space

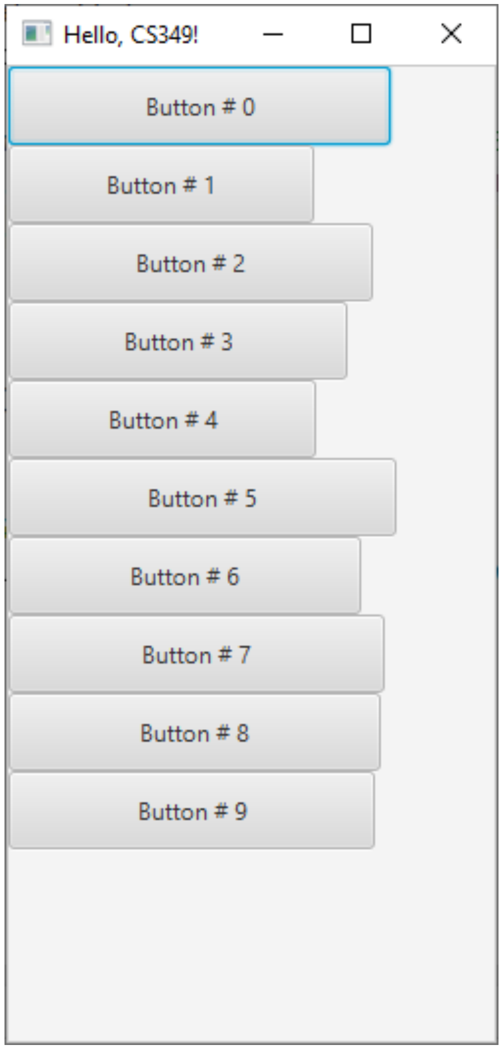




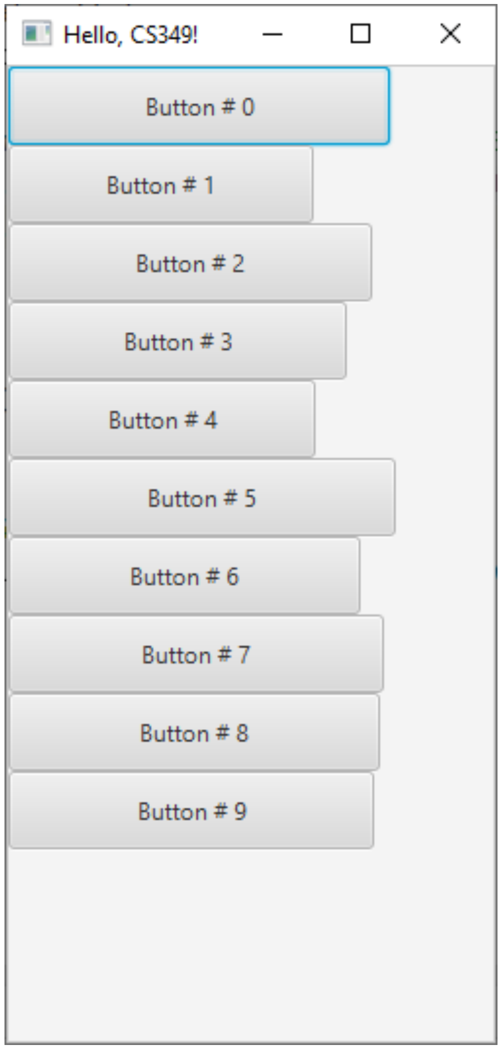


Container – `ScrollPane`

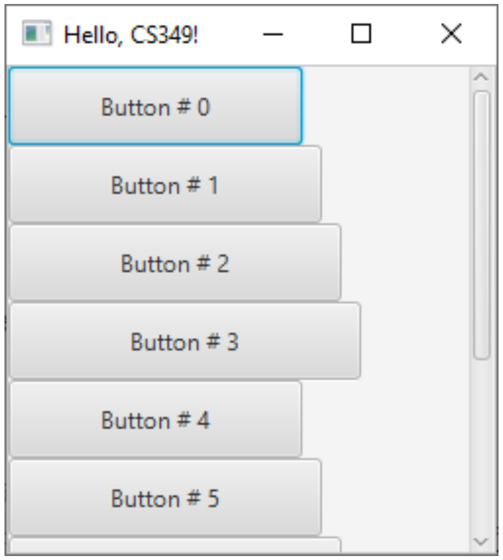
VBox in ScrollPane
enough vertical space
Vgrow NEVER



VBox in ScrollPane
enough vertical space
Vgrow ALWAYS



VBox in ScrollPane
not enough vertical space

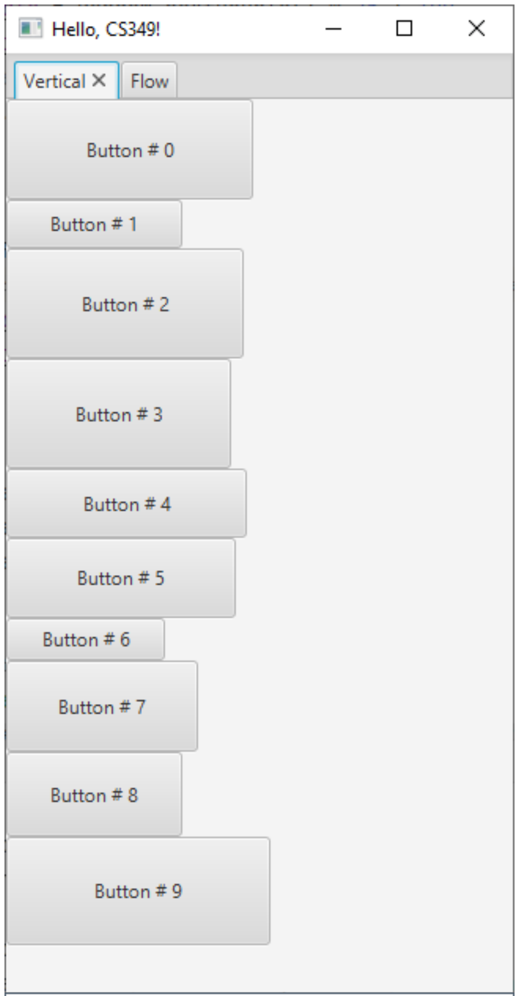


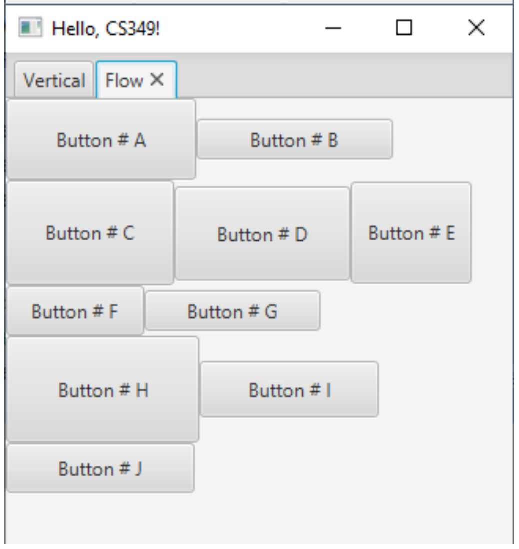
Container – `TabPane`

`TabPane` allows switching between a group of tabs, with only one tab visible at a time. Its tabs can be positioned at any of its four sides. Its default {height|width} will be determined by the largest content {height|width}.

Container – TabPane

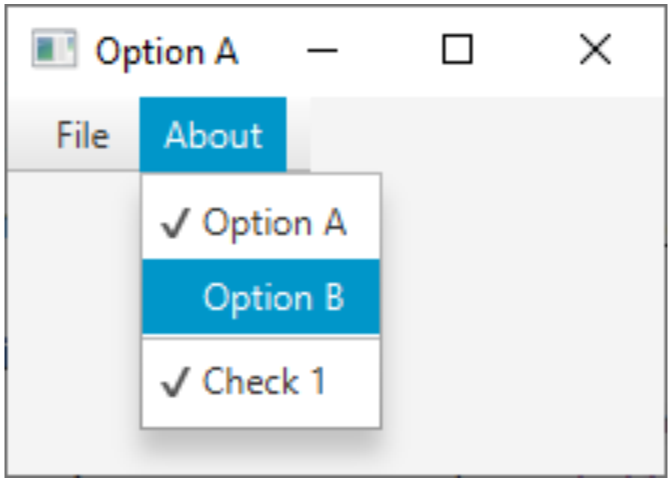
```
override fun start(stage: Stage) {
    val vboxTab = VBox().apply {
        (0..9).forEach() { children.add(Button("Button # $it").apply {
            prefWidth = Random.nextDouble() * 75 + 100
            prefHeight = Random.nextDouble() * 50 + 25
        })}
    }
    val flowTab = FlowPane().apply {
        ('A'..'J').forEach() { children.add(Button("Button # $it").apply
            prefWidth = Random.nextDouble() * 50 + 75
            prefHeight = Random.nextDouble() * 50 + 25
        })}
    }

    val tab = TabPane().apply {
        tabs.add(Tab("Vertical", vboxTab))
        tabs.add(Tab("Flow", flowTab))
        tabsClosingPolicy = TabPane.TabClosingPolicy.ALL_TABS
    }

    stage.apply {
        title = "Hello, CS349!"
        scene = Scene(tab , 400.0, 400.0).apply { fill = Color.PINK }
    }.show()
}
```



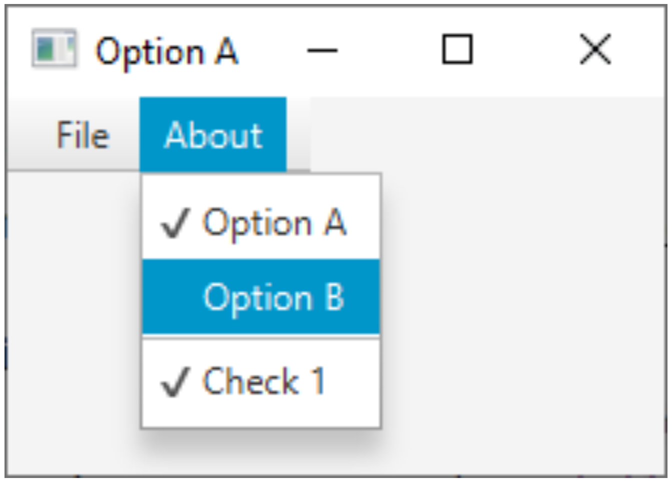


Container – `MenuBar`

`MenuBar` is traditionally placed at the very top of the user interface and embedded within it are Menus. By default, for each menu added to the menu bar, it will be represented as a button with the Menu text value displayed.



Container – MenuBar



```
val menuBar = MenuBar()
menuBar.menus.addAll(
    Menu("File").apply {
        items.add(MenuItem("Quit").apply {
            onAction = EventHandler { Platform.exit() }
        })},
    Menu("About").apply {
        val rm1 = RadioMenuItem("Option A")
        val rm2 = RadioMenuItem("Option B")
        val cm = CheckMenuItem("Check 1").apply {
            selectedProperty().addListener { _, _, new ->
                stage.title = "$text ${if (new) "on" else "off"}"}
        }
        items.addAll(rm1, rm2, SeparatorMenuItem(), cm)
        ToggleGroup().apply {
            rm1.toggleGroup = this; rm2.toggleGroup = this
            selectToggle(rm1)
            selectedToggleProperty().addListener { _, _, new ->
                stage.title = (new as RadioMenuItem).text }}
    })
```

End of the Chapter

Please make sure to

- Remember which layouts are available

Any further questions?