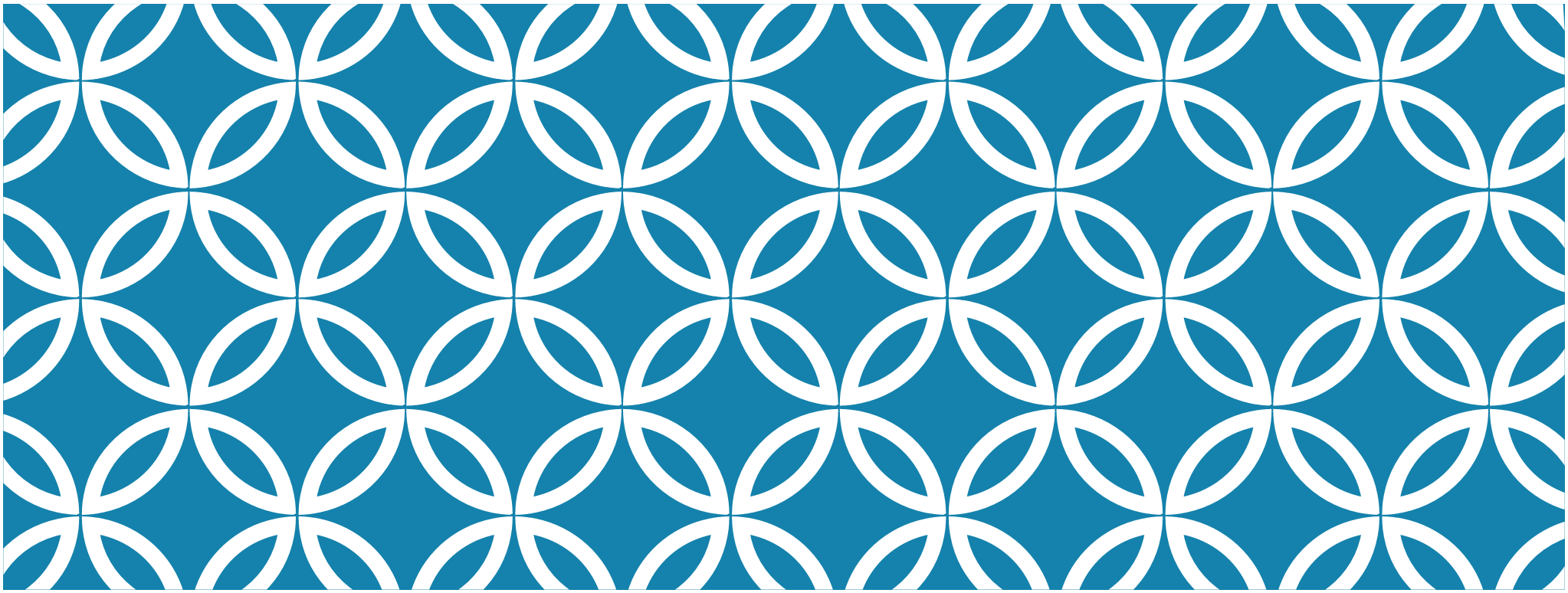


GRAPHICAL USER INTERFACES

CS 346: Application Development

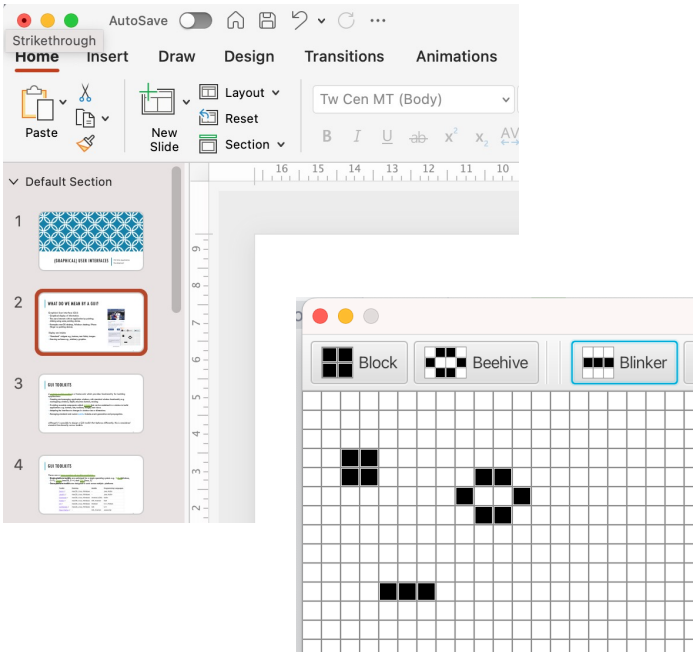
WHAT DO WE MEAN BY A GUI?

Graphical User Interface (GUI)

- The users interacts with an application by pointing-clicking using some pointing device (e.g., mouse, touchpad, finger).
- Also keyboard support for entering text.
- Examples: macOS desktop, Windows desktop, iPhone.

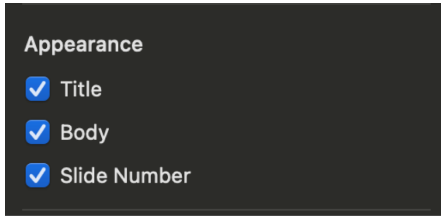
Output can include:

- "Standard" widgets e.g., buttons, text fields, images.
- Drawing surfaces e.g., arbitrary graphics.

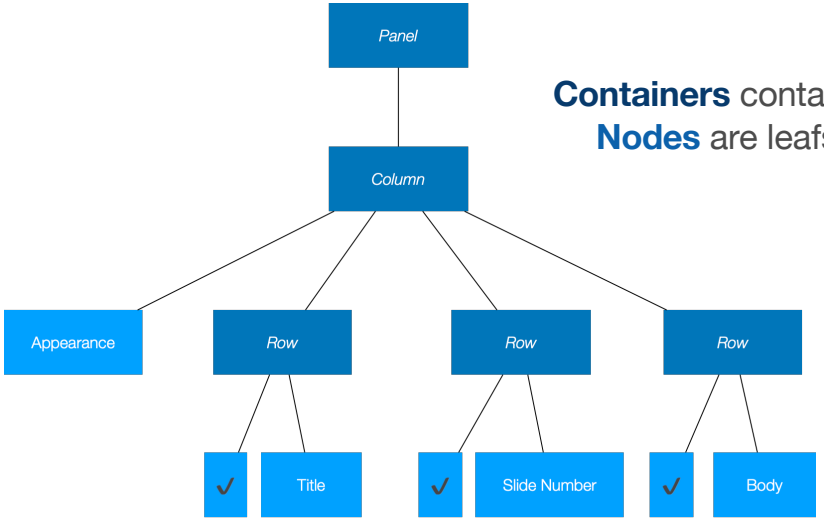


CONCEPT: UI AS A SCENE GRAPH

In GUI design, we use an abstraction called a **scene graph,** to represents graphical content as a *tree where higher level elements manage their children*. Toolkits provide components ("widgets") which developers directly instantiate and place on-screen.



UI Panel



Containers contain other classes.
Nodes are leafs in the graph.

Scene Graph for this panel

CONCEPT: EVENT-DRIVEN INTERACTION

Graphical user interfaces rely on events being generated and passed to interested parts of your application.

An event is simply a message generated by the system to indicate that *something* has happened.

Examples:

- `MouseMoved`: Indicates that the pointer has been repositioned.
- `MouseClicked`: The user has clicked on something.
- `KeyPressed`: Key interaction.

Traditionally, writing a user interface requires you to intercept and process these messages.

GUI TOOLKITS

A GUI toolkit is a framework which provides the required functionality for building graphical applications:

- Creating and managing application windows, with standard window functionality e.g. overlapping windows, depth, min/max buttons, resizing.
- Providing reusable widgets that can be combined in a window to build applications. e.g. buttons, lists, toolbars, images, text views.
- Adapting the interface to changes in window size or dimensions.
- Managing standard and custom events. Includes event generation and propagation.

Although it is possible to design a GUI toolkit that behaves differently, this is considered standard design across modern toolkits.

GUI TOOLKITS

There are a large number of toolkits available.

- **Single-platform toolkits** are optimized for a single operating system.
  - e.g. WTL (Windows, C++), Cocoa (macOS, C++) and GTK (Linux, C).
- **Cross-platform toolkits** are designed to work across multiple platforms.

Toolkit	Desktop	Mobile	Programming Languages
Swing	macOS, Linux, Windows	-	Java, Kotlin
JavaFX	macOS, Linux, Windows	-	Java, Kotlin
Compose	macOS, Linux, Windows	Android, (iOS)	Kotlin
Flutter	macOS, Linux, Windows	iOS, Android	Dart
Qt	macOS, Linux, Windows	Android	C++, Python
wxWidgets	macOS, Linux, Windows	iOS	C++
React Native	-	iOS, Android	Javascript

TOOLKIT STYLES

Most GUI frameworks are **imperative**:

- UI objects are just classes with properties for `position (x, y)`, `dimensions (w, h)`, other visual properties. e.g. `Button`, `Scrollbar`, `Panel`, `Slider`, `Image` classes.
- Underlying code places elements on-screen and controls their appearance.
- Code determines **how** the user interface behaves based on user input.
  - i.e. an imperative toolkit *relies on custom code to change the user interface in response to state changes*. This is a large part of the application's complexity.

Modern toolkits are **declarative**:

- A declarative paradigm explains **what** to display. The compiler figures out how to display it based on the current state (e.g. is the button enabled/disabled?).
  - i.e. a declarative toolkit *automatically manages how the UI reacts to state changes*.

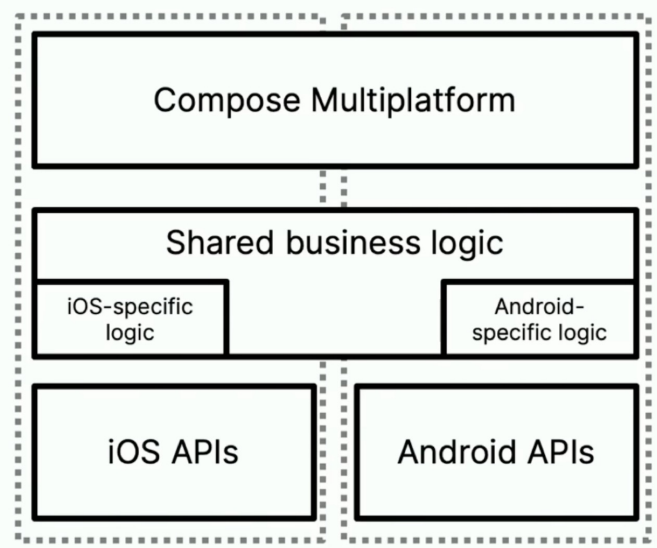
WHAT IS COMPOSE?

Compose is a declarative, cross-platform toolkit.

- It was originally designed by Google, and released as **JetPack Compose** for Android in 2017.
- JetBrains ported Jetpack Compose to desktop, and released it in 2021 as **Compose Multiplatform**, which supports macOS, Windows, Linux desktop.
- Compose iOS and Web are "on the way".

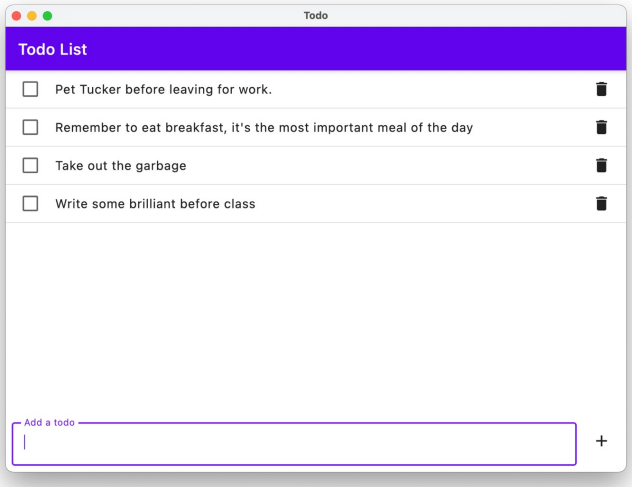
In this course we'll focus on Compose for Desktop and Android, since these are the most stable.

This is the rare case where we can use the same toolkit for more than one platform!

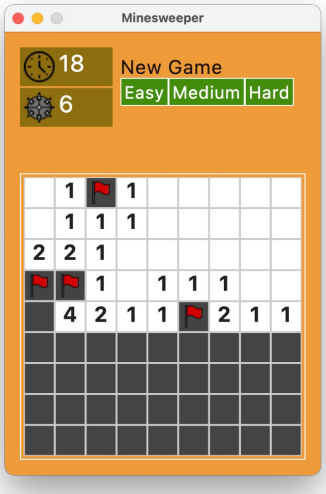


https://www.jetbrains.com/lp/compose-multiplatform/

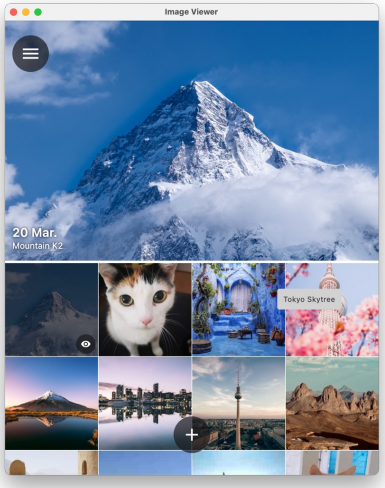
WHAT CAN COMPOSE DO?



Todo



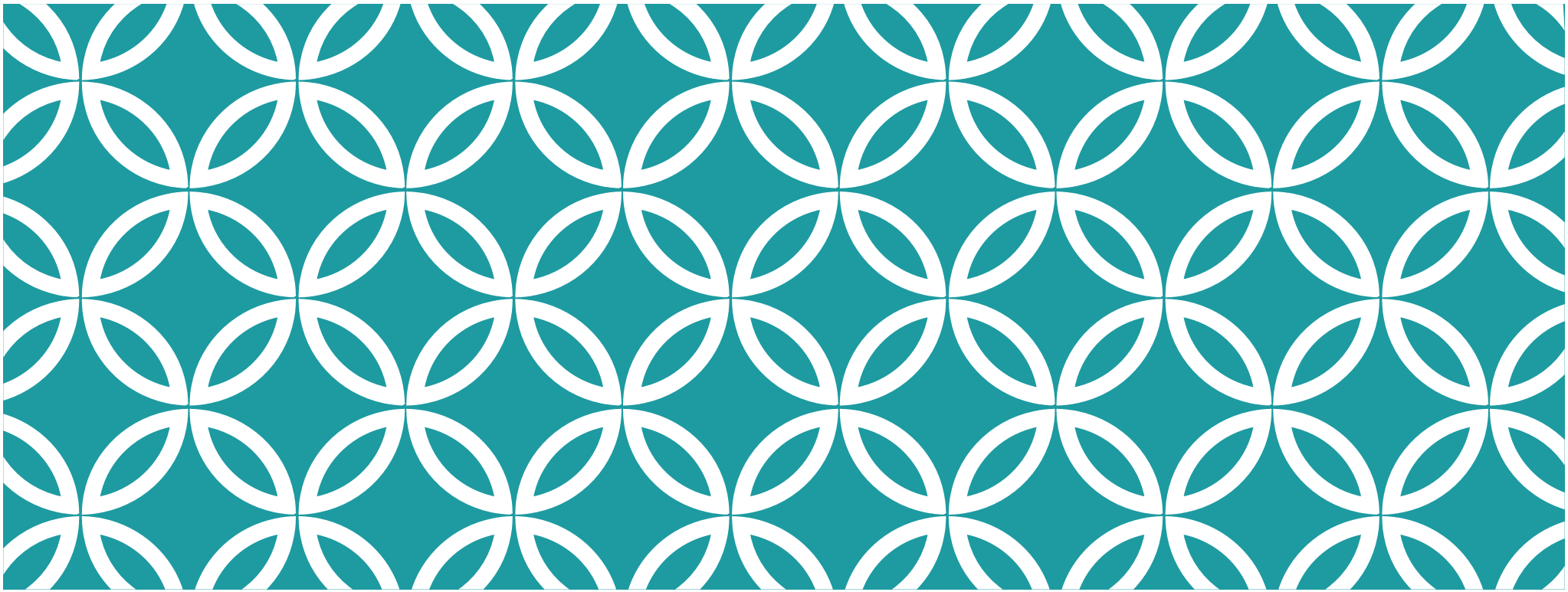
Minesweeper



ImageViewer

https://github.com/JetBrains/compose-multiplatform/

https://github.com/android/compose-samples



COMPOSE > COMPOSABLES

CS 346: Application Development

CONCEPT: COMPOSABLE FUNCTION

A key concept in Compose is the idea of a **composable function** (also just called a **composable**). This is a small function that describes a part of your user interface.

Think of a composable function as a special kind of function that accepts state, and emits a user interface element.

For example, this function takes in a String and displays it on-screen by emitting a Text element that will be displayed.

```
@Composable
fun Greeting(name: String) {
  Text("Hello $name!")
}
```

CHARACTERISTICS OF COMPOSABLES

The function must be annotated with the @Composable annotation.

- Composable functions are fast, idempotent, and free of side effects!
- Composables do not return a value – they emit output directly into the scene graph.
- Composable functions will often accept parameters, which are used to format the composable before displaying it.

```
@Composable
fun Greeting(name: String) {
  Text("Hello $name!")
}
```

COMPOSABLE SCOPE (1/2)

composable scope

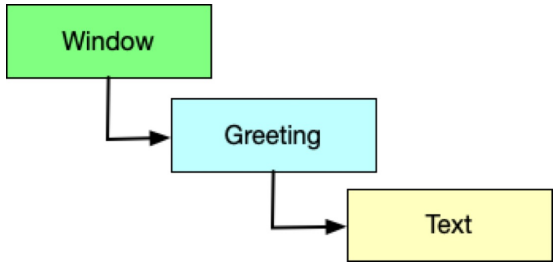
Let's display a window.

```
fun main() = application {
    Window(
      title = "Hello Window",
      onCloseRequest = ::exitApplication
    ) {
      Greeting("Compose")
    }
}

@Composable
fun Greeting(name: String) {
  Text("Hello $name!")
}
```

The `application` function defines a **Composable Scope** – think of it like a wrapper.

Composable functions must be called from a Composable Scope, or from other Composables.

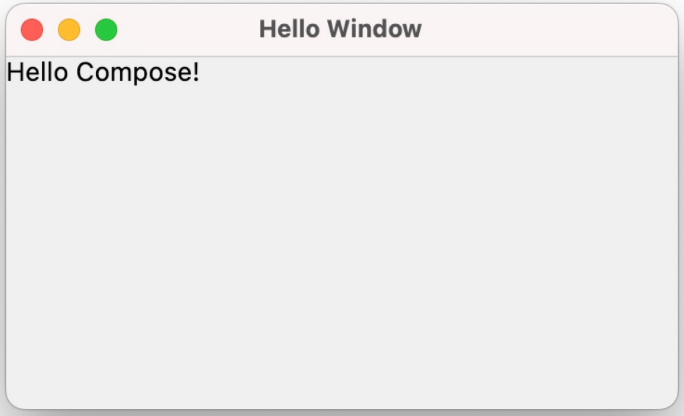
These composables describe a scene graph.



COMPOSABLE SCOPE (2/2)

Here's the resulting window.

```
fun main() = application {
    Window(
      title = "Hello Window",
      onCloseRequest = ::exitApplication
    ) {
      Greeting("Compose")
    }
}

@Composable
fun Greeting(name: String) {
  Text("Hello $name!")
}
```



The Compose toolkit handles standard functionality e.g. min/max buttons, titlebar. You customize the composables by passing in parameters.

See GL Public repo: /samples/lectures/compose-desktop

USING COMPOSABLES

Construct user interfaces by combining composables together to form a scene graph. These can be built-in composables, or ones that you create.

There are many **built-in** composables:

- Some composables act as containers and manage children composables in the scene graph.
- Other composables display data, and (some) provide interactivity for users.

Because Compose is cross-platform, most of these composables will work across all supported platforms.

- e.g. the `Text` composable exists on both desktop and Android (it hasn't been reimplemented - it's the same code).
- Composable Scope differs by platform e.g. `application` is desktop specific.
- We'll continue to demo using Compose Multiplatform/desktop for now.

PROPERTIES

Each composable has its own parameters that can be supplied to affect its appearance and behaviour.

These are exposed in the constructor as named parameters.

Examples:

- **Text, textAlign, lineHeight, fontName, fontSize** are common with text.
- **Color** is a property shared by most Composables.
- **Style** lets you use a particular design attribute that is included in the theme.
- **Modifier** is a class that contains parameters that are commonly used across elements. This allows us to set a number of parameters within an instance of a Modifier.

EXAMPLE: TEXT

A Text composable displays text.

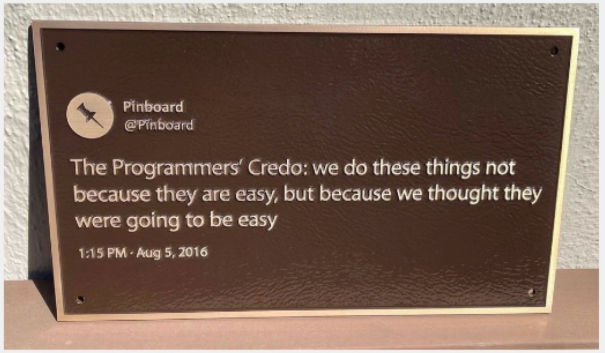
```
@Composable
fun SimpleText() {
  Text(
      text = “Widget Demo”,
      color = Color.Blue,
      fontSize = 30.sp,
      style = MaterialTheme.typography.h2,
      maxLines = 1
  )
}
```

See GitLab repo: public/code/compose-desktop

EXAMPLE: IMAGE

An Image composable displays an image (by default, image is loaded from your Resources folder).

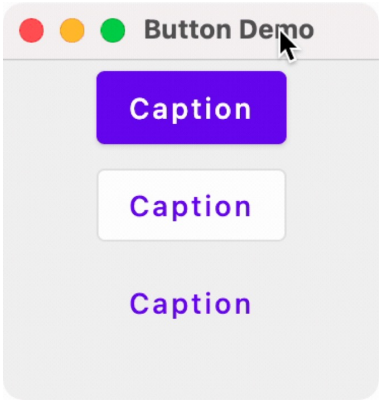
```
@Composable
fun SimpleImage() {
    Image(
        painter = painterResource("credo.jpg"),
        contentDescription = null,
        contentScale = ContentScale.Fit,
        modifier = Modifier
            .height(150.dp)
            .fillMaxWidth()
            .clip(shape = RoundedCornerShape(10.dp))
    )
}
```



EXAMPLE: BUTTON

There are three main Button composables:

- Button: A standard button with no caption.
- OutlinedButton: A button with an outline. Secondary.
- TextButton: A button with a caption.



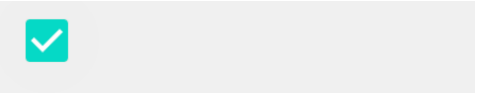

```
fun main() {
    application{
        Window(onCloseRequest = ::exitApplication, title = "Button Demo")
        {
            Column(modifier = Modifier.fillMaxSize(),
                horizontalAlignment = Alignment.CenterHorizontally)
            {
                Button(onClick = { println("Button clicked") }) { Text("Caption") }
                OutlinedButton(onClick = { println("OutlinedButton clicked") }) { Text("Caption") }
                TextButton(onClick = { println("TextButton clicked") }) { Text("Caption") }
            }
        }
    }
}
```

EXAMPLE: CHECKBOX

A checkbox is a toggleable control that presents a true/false state.

- The `OnCheckedChange` function is called when the user interacts with it (and in this case, the state represented by it is being stored in a MutableState variable named `isChecked`).

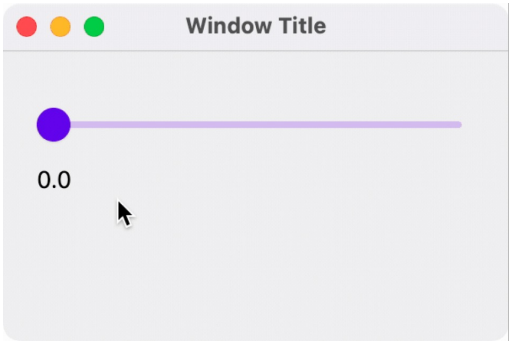
```
@Composable
fun SimpleCheckbox() {
    val isChecked = remember { mutableStateOf(false) }

    Checkbox(
        checked = isChecked.value ,
        enabled = true,
        onCheckedChange = { isChecked.value  = it }
    )
}
```



EXAMPLE: SLIDER

A slider lets the user make a selection from a continuous range of values. It's useful for things like adjusting volume or brightness or choosing from a range of values.

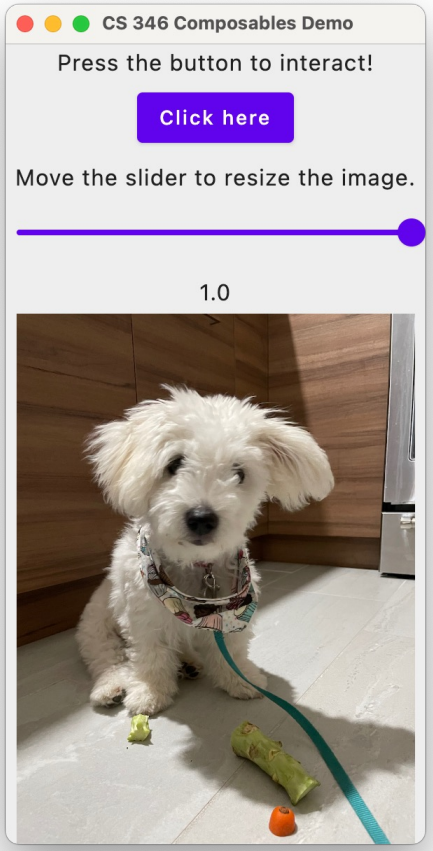
```
@Composable
fun SliderMinimalExample() {
    var sliderPosition by remember
        { mutableFloatStateOf(0f) }

    Column {
        Slider(
            value = sliderPosition,
            onValueChange = { sliderPosition = it }
        )
        Text(text = sliderPosition.toString())
    }
}
```

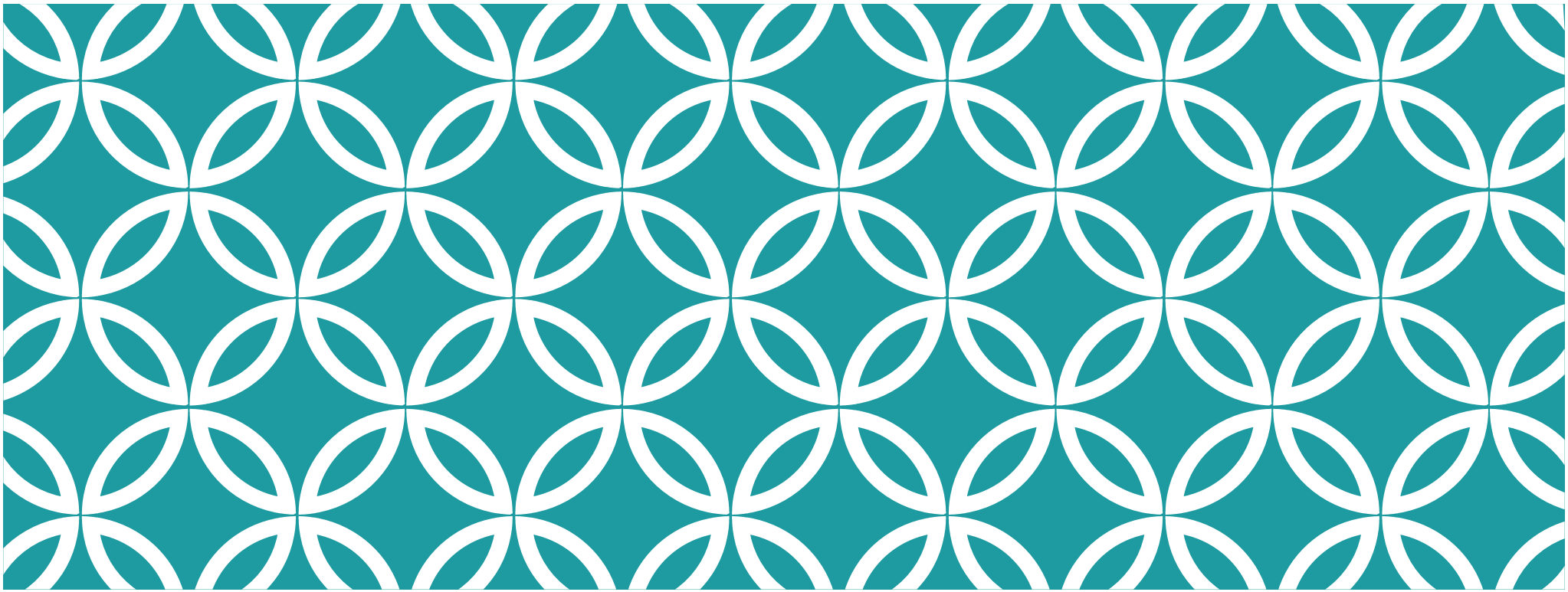


DEMO

See GL Public:
/samples/lectures/compose-desktop

Run the Composables.kt main





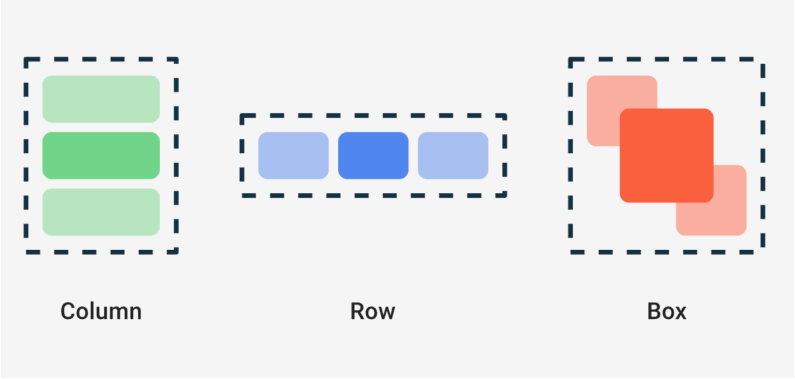
COMPOSE > LAYOUT

CS 346: Application Development

LAYOUT COMPOSABLES

Compose includes Layout Composables, whose purpose is to act as a container to other composables. There are three main layout composables:

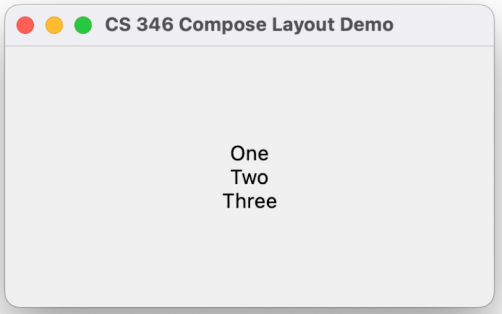
- **Column**, used to arrange widget elements vertically
- **Row**, used to arrange widget elements horizontally
- **Box**, used to arrange objects in layers



https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary

COLUMN COMPOSABLE

```
fun main() = application {
  Window(
    title = "CS 346 Compose Layout Demo",
    onCloseRequest = ::exitApplication
  ) {
      SimpleColumn()
  }
}


@Composable
fun SimpleColumn() {
  Column(
      modifier = Modifier.fillMaxSize(),
      verticalArrangement = Arrangement.Center,
      horizontalAlignment = Alignment.CenterHorizontally
  ) {
      Text("One")
      Text("Two")
      Text("Three")
  }
}
```

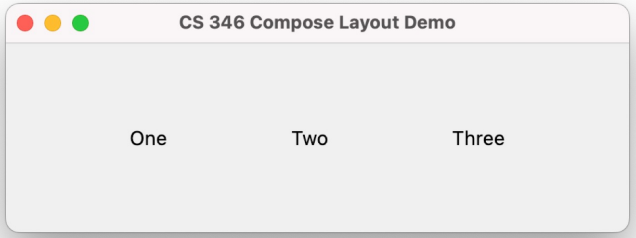


Arrangement: The direction the composable flows.

Alignment: Orthogonal to the arrangement.

ROW COMPOSABLE

```
fun main() = application {
  Window(
    title = "CS 346 Compose Layout Demo",
    onCloseRequest = ::exitApplication
  ) {
      SimpleRow()
  }
}

@Composable
fun SimpleRow() {
  Row(
      modifier = Modifier.fillMaxSize(),
      horizontalArrangement = Arrangement.SpaceEvenly,
      verticalAlignment = Alignment.CenterVertically
  ) {
      Text("One")
      Text("Two")
      Text("Three")
  }
}
```

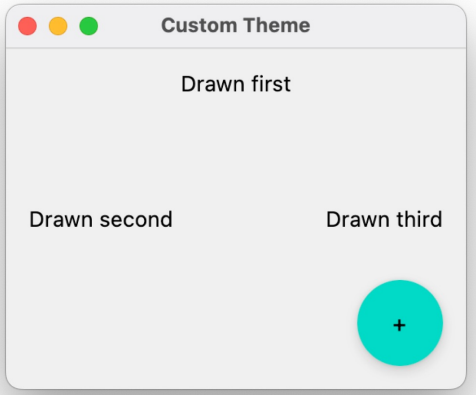


Arrangement: The direction the composable flows.

Alignment: Orthogonal to the arrangement.

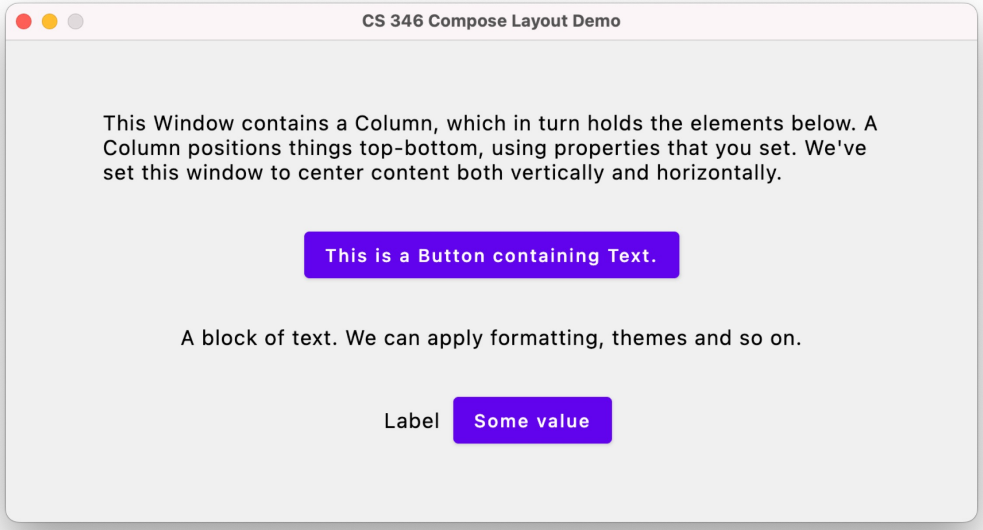
BOX COMPOSABLE

```
fun main() = application {
  Window(
      title = "Custom Theme",
      onCloseRequest = ::exitApplication,
      state = WindowState(
         width = 300.dp, height = 250.dp,
         position = WindowPosition(50.dp, 50.dp)
      )
  ){
      SimpleBox()
  }
}


@Composable
fun SimpleBox() {
  Box(Modifier.fillMaxSize().padding(15.dp)) {
      Text("Drawn first", modifier = Modifier.align(Alignment.TopCenter))
      Text("Drawn second", modifier = Modifier.align(Alignment.CenterStart))
      Text("Drawn third", modifier = Modifier.align(Alignment.CenterEnd))
      FloatingActionButton(
          modifier = Modifier.align(Alignment.BottomEnd),
          onClick = {println("+ pressed")}
      ) {
          Text("+")
      }
  }
}
```



NESTING LAYOUTS

This example contains a Column as the top-level composable, and a Row at the bottom that contains Text and Button composables (which is how we have the layout flowing both top-bottom and left-right).



LAZY LAYOUTS

Columns and rows work fine for a small amount of data that fits on the screen. What do you do if you have large lists that might be longer or wider than the space that you have available?

Ideally, we would like that content to be scrollable. For performance reasons, we also want large amounts of data to be *lazy loaded*: only the data that is being displayed needs to be in-memory and other data is loaded only when it needs to be displayed.

Compose has a series of lazy components that work like this:

- LazyColumn
- LazyRow
- LazyVerticalGrid
- LazyHorizontalGrid

https://developer.android.com/jetpack/compose/lists

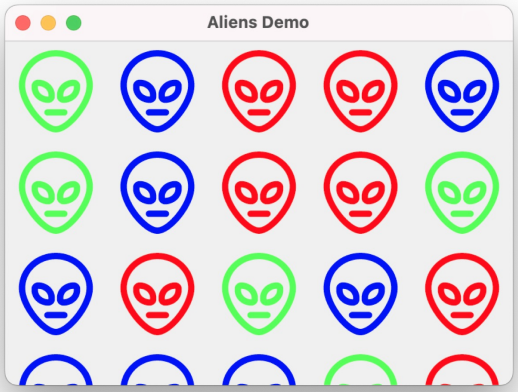
LAZYROW COMPOSABLE

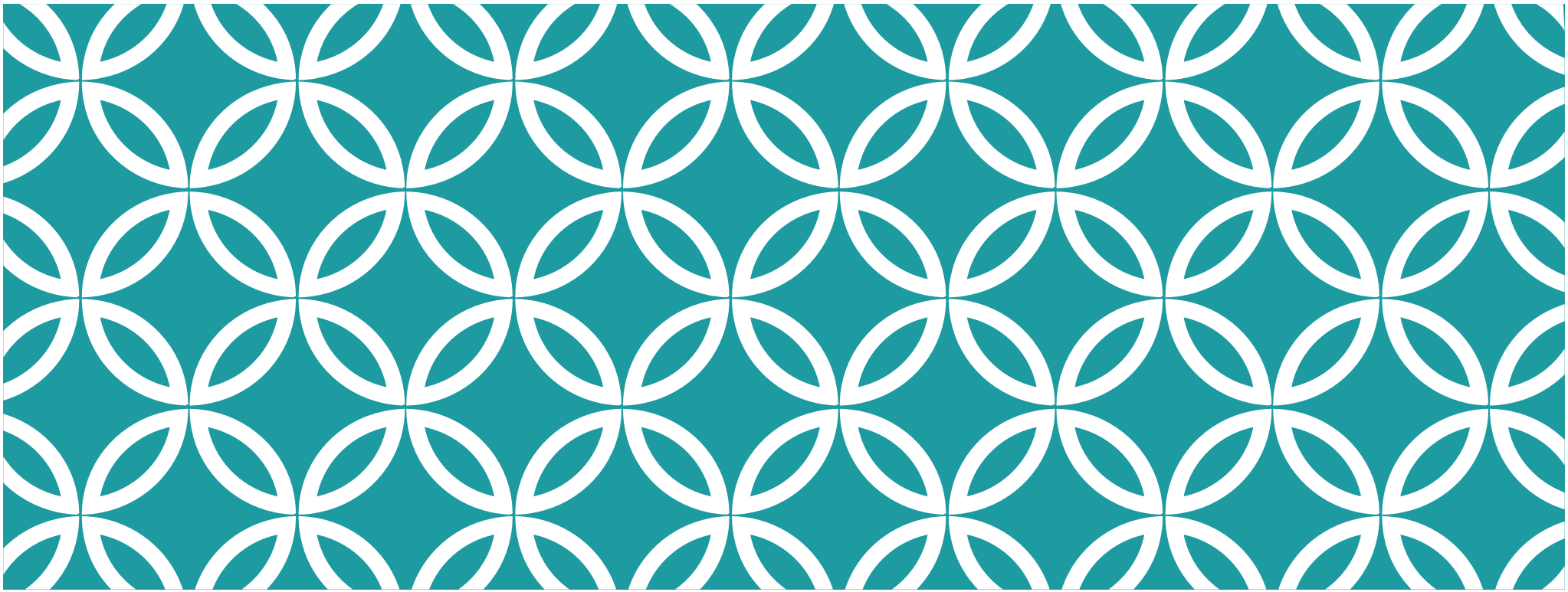
```
fun main() = application {
  Window(
      title = "LazyColumn",
      state = WindowState(width = 500.dp, height = 100.dp),
      onCloseRequest = ::exitApplication
  ) {
      LazyRowDemo()
  }
}


@Composable
fun LazyRowDemo(modifier: Modifier = Modifier) {
  LazyRow(
      modifier = modifier.padding(4.dp).fillMaxSize(),
      verticalAlignment = Alignment.CenterVertically
  ) {
      items(45) {
          Button(
              onClick = { },
              modifier = Modifier
                  .size(100.dp, 50.dp)
                  .padding(4.dp)
          ) {
              Text(it.toString())
          }
      }
  }
}
```

LAZYGRID COMPOSABLE

```
@Composable
fun AndroidLazyGrid(modifier: Modifier = Modifier) {
    LazyVerticalGrid(modifier = modifier, columns = GridCells.Fixed(5)) {
        val colors = listOf<Color>(Color.Blue, Color.Red, Color.Green)
        items(45) {
            AndroidAlien(color = colors.get(Random.nextInt(0,3)) )
        }
    }
}
```

Vertical Grid with fixed number of columns, scrolls up/down.





COMPOSE > STATE

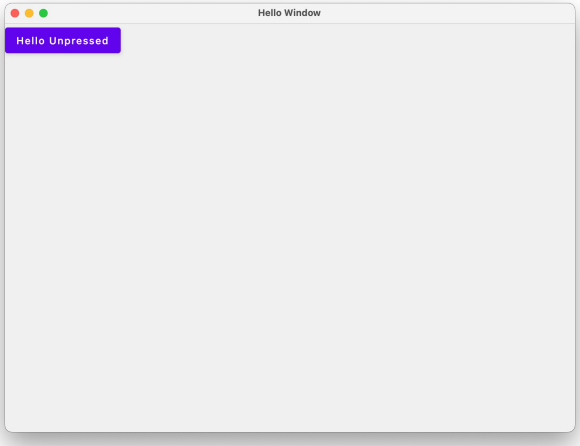
CS 346: Application Development

ADDING INTERACTIVITY (1/4)

onCloseRequest and **onClick** are *event handlers*; we're assigning functions to be called when those events occur.

Let's revisit our Window demo and add an interactive Button.

```
fun main() = application {
    Window(
        title = "Hello Window",
        onCloseRequest = ::exitApplication
    ) {
        Greeting("Unpressed")
    }
}


@Composable
fun Greeting(name: String) {
    Button(onClick = { println("Button pressed") }) {
        Text("Hello $name")
    }
}
```



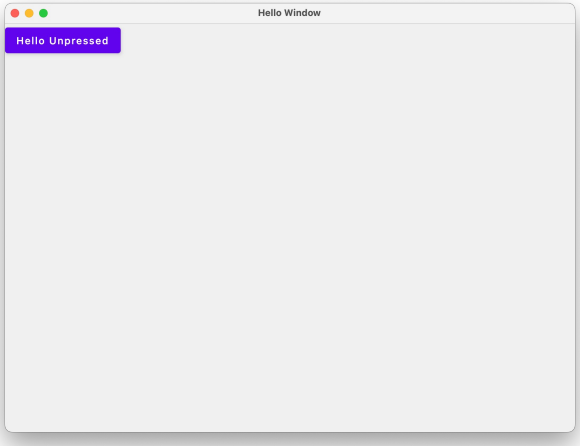
Console Output

```
> Task :run
Button pressed
Button pressed
Button pressed
```

ADDING INTERACTIVITY (2/4)

Let's have it try and update the *emitted* UI.

```
fun main() = application {
  Window(
      title = "Hello Window",
      onCloseRequest = ::exitApplication
  ) {
      Greeting("Unpressed")
  }
}


@Composable
fun Greeting(caption: String) {
  var currentCaption = name
  Button(onClick = { currentCaption = "Pressed" }) {
      Text("Hello $currentCaption")
  }
}
```



It doesn't work. The UI never updates. Why?

CONCEPT: RECOMPOSITION

The declarative design of Compose means that it draws the screen when the application launches, and then *only redraws elements when their state changes*.

Compose is effectively doing this:

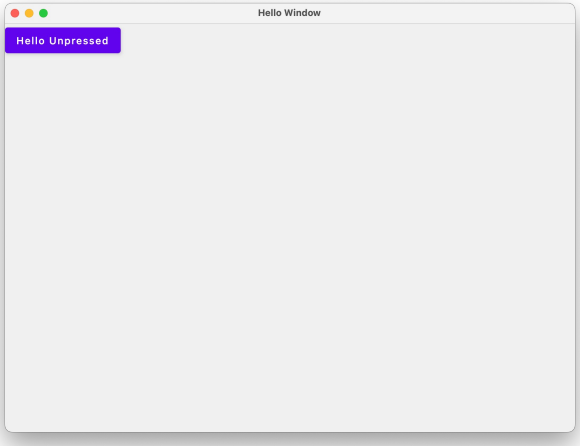
- Drawing the initial user interface.
- Monitoring your state (aka variables) directly.
- When a change is detected in state, the portion of the UI that *relies on that state* is updated.

Compose redraws affected components by calling their Composable functions. This process (detecting a change, and then redrawing the UI) is called **recomposition** and is the main design principle behind Compose.

ADDING INTERACTIVITY (3/4)

Let's revisit our demo. Why doesn't the Button update?

```
fun main() = application {
  Window(
      title = "Hello Window",
      onCloseRequest = ::exitApplication
  ) {
      Greeting("Unpressed")
  }
}


@Composable
fun Greeting(caption: String) {
  var currentCaption = caption
  Button(onClick = {currentCaption = "Pressed" }) {
      Text("Hello $currentCaption")
  }
}
```

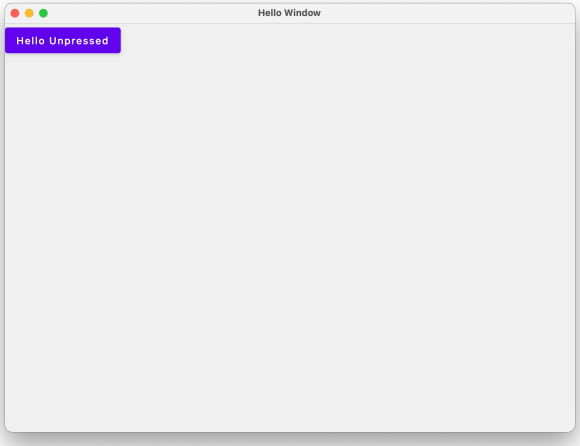


This doesn't work.
The `onClick` handler attempts to change the text property of the Button.
This triggers Compose to call the Window composable, which calls the Button composable, which initializes `text` to its initial value...

ADDING INTERACTIVITY (4/4)

To make state *observable*, use a MutableState class.

```
fun main() = application {
  Window(
      title = "Hello Window",
      onCloseRequest = ::exitApplication
  ) {
      Greeting("Unpressed")
  }
}


@Composable
fun Greeting(caption: String) {
  var currentCaption = remember { mutableStateOf(caption) }
  Button(onClick = {currentCaption.value = "Pressed" }) {
      Text("Hello ${currentCaption.value}")
  }
}
```



This works!
mutableStateOf(name) is an observable String (via type inference).
Remember tells it the @Composable function to NOT re-initialize this state when Recomposition happens.

REMEMBERING STATE

There are multiple classes to handle different *types* of State. Here's a partial list:

Class	Helper Function	State that it represents
MutableState	mutableStateOf()	Primitive
MutableList	mutableListOf	List
MutableMap<K, V>	mutableMapOf(K, V)	Map<K, V>
WindowState	rememberWindowState()	Window parameters e.g. size, position
DialogState	rememberDialogState	Similar to WindowState

```
fun main() = application {
  Window(
      title = "Hello Window",
      onCloseRequest = ::exitApplication,
      state = WindowState(width=300.dp, height=200.dp, position = WindowPosition(50.dp, 50.dp))
  ) {
      val caption = remember { mutableStateOf("Press me") }
      Button(onClick = {caption.value = "Pressed!"}) {
          Text(caption.value)
      }
  }
}
```

STATE HOISTING (1/2)

A composable that uses remember is storing the internal state within that composable, making it *stateful* (e.g. our Greeting composable function above).

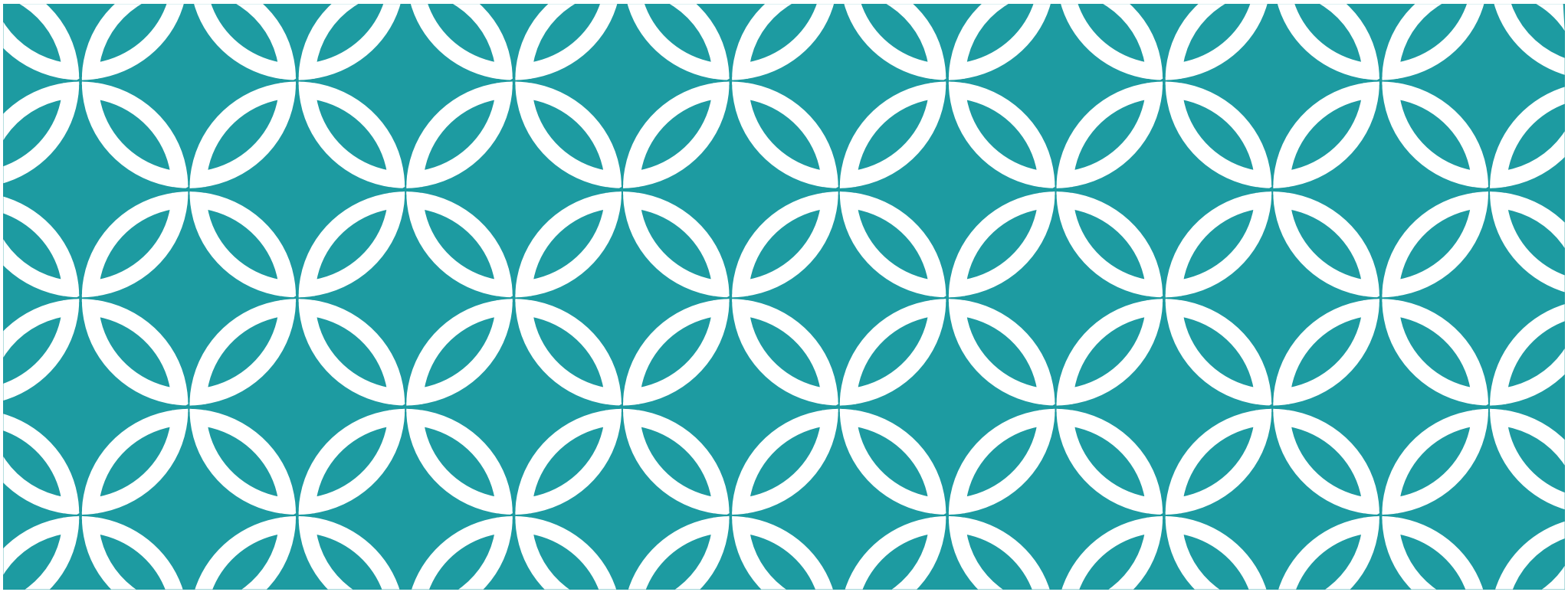
However, storing state in a function can make it difficult to test and reuse. It's sometimes helpful to pull state out of a function into a higher-level, calling function. This process is called *state hoisting*.

STATE HOISTING (2/2)

```
fun main() = application {
  Window( title = "Hello Window", onCloseRequest = ::exitApplication) {
      HelloScreen()
  }
}


@Composable
fun HelloScreen() {
  var name by remember { mutableStateOf("") }
  HelloContent(name = name, onNameChange = { name = it })
}


@Composable
fun HelloContent(name: String, onNameChange: (String) -> Unit) {
  Column(modifier = Modifier.padding(16.dp)) {
      Text(
          text = "Hello, $name",
          modifier = Modifier.padding(bottom = 8.dp),
          style = MaterialTheme.typography.body1
      )
      OutlinedTextField(value = name, onValueChange = onNameChange, label = { Text("Name") })
  }
}
```

Our state is the name that the user is entering in the OutlinedTextField.

Instead of storing that in our HelloContent composable, we keep our state variable in the calling class HelloScreen and pass in the callback function that will set that value.



COMPOSE > THEMES

CS 346: Application Development

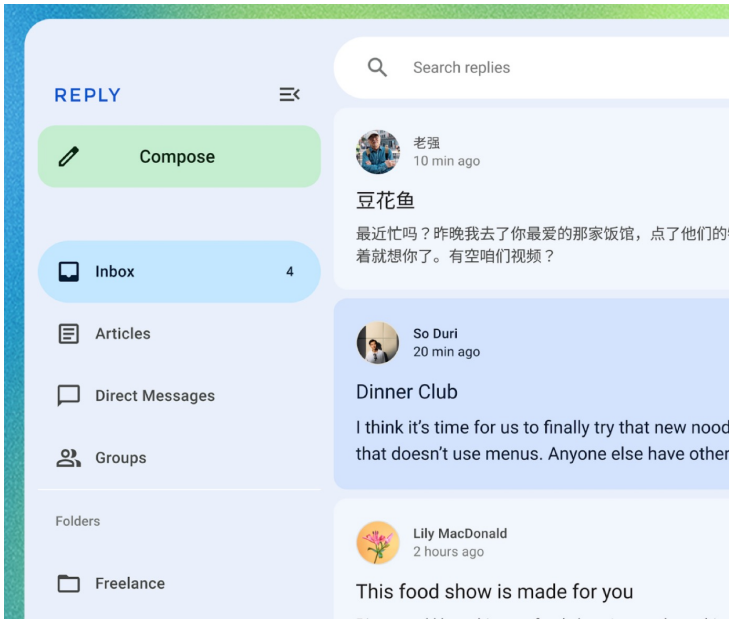
MATERIAL 3 THEME

A theme is a common look-and-feel that is used when building software.

Google includes their **Material Design theme** in Compose, and by default, composables will be drawn using the Material look-and-feel. This includes colors, opacity, shadowing and other visual elements.

https://m3.material.io/

This is fantastic as an Android developer: it's very well specified and complete. It also may not be what you want on desktop, or iOS.



To customize the default theme, we can just extend it and change its properties, and then set our application to use the modified theme.

```
@Composable
fun CustomTheme(
    content: @Composable () -> Unit
) {
    MaterialTheme(
       colors = MaterialTheme.colors.copy(primary = Color.Red, secondary = Color.Magenta),
       shapes = MaterialTheme.shapes.copy(
            small = AbsoluteCutCornerShape(0.dp),
            medium = AbsoluteCutCornerShape(0.dp),
            large = AbsoluteCutCornerShape(0.dp)
        )
    ) { content() }
}


fun main() = application {
  Window(
      title = "Hello Window",
      onCloseRequest = ::exitApplication,
      state = WindowState(width=300.dp, height=250.dp, position = WindowPosition(50.dp, 50.dp))
  ) {
      CustomTheme { … }
```

THIRD-PARTY THEMES

There are third-party themes that you can include to replace the Material theme completely:

- Aurora library allows you to style Compose applications using the Ephemeral design theme.
- JetBrains Jewel changes the look-and-feel to match IntelliJ applications. e.g. IntelliJ IDEA.
- MacOS theme mimics the standard macOS look-and-feel.