# Graphical User Interfaces

What is 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.
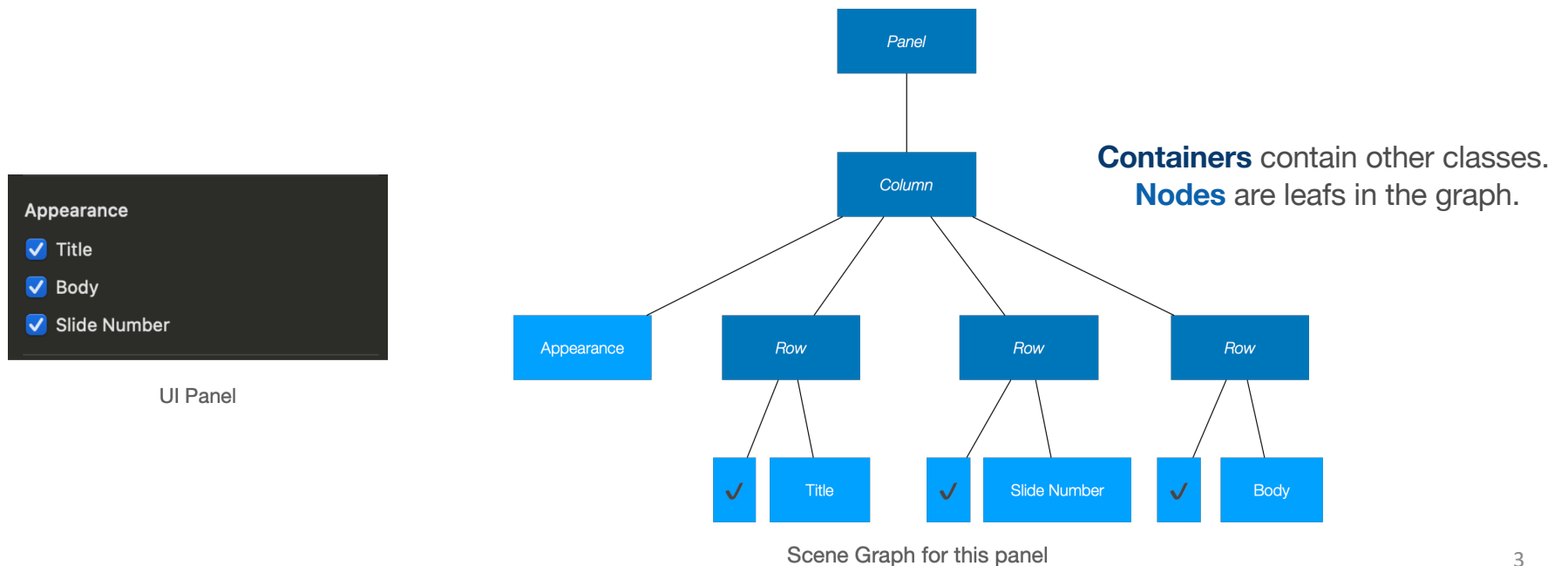- Examples: macOS desktop, Windows desktop, iPhone.

Output combines

- "Standard" widgets
  - e.g., buttons, text fields, images.
- Drawing surfaces
  - e.g., arbitrary graphics.
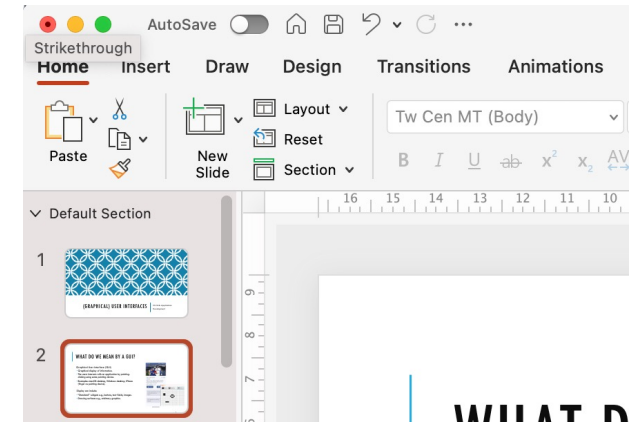
CONCEPT: UI as a Scene Graph

In GUI design, we use an abstraction called a **scene graph**, to represent graphical content as a *tree, where higher level elements manage their children*.

UI Panel

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

Scene Graph for this panel

CONCEPT: Event-Driven Interaction

Interaction relies on **events** being generated and passed around 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 with a mouse.
  - `KeyPressed`: A key on a keyboard has been pressed.

  GUI = Drawing (scene graph) + Interactivity (triggered by events)

GUI Toolkits

A GUI toolkit is a framework which provides this functionality.

- Creating and managing application windows, with standard functionality e.g. overlapping windows, min/max, 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.
  - Generating events and responding to them in code.
  - Handling user interaction with hardware e.g., keyboards, touch.

Toolkit Styles

Historically, most GUI frameworks have been **imperative**:

- UI objects are just classes with properties for `position (x,y)`, `dimensions (w,h)`, other visual properties. e.g. `Button`, `Scrollbar`, `Panel`, `Slider`.

- 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?).
    - i.e. a **declarative** toolkit *automatically manages how the UI reacts to state changes*.

GUI Toolkits

Single-platform are designed for one platform only.
- e.g. WTL (Win, C++), Cocoa (Mac, C++) and GTK (Linux, C).

Cross-platform toolkits are designed for multiple platforms.
- "Holy grail" of software development. Many examples.

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

Compose Toolkit

A declarative, multi-platform toolkit.

# What is Compose?

Compose is a declarative, cross-platform toolkit.

- It was 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, iOS.
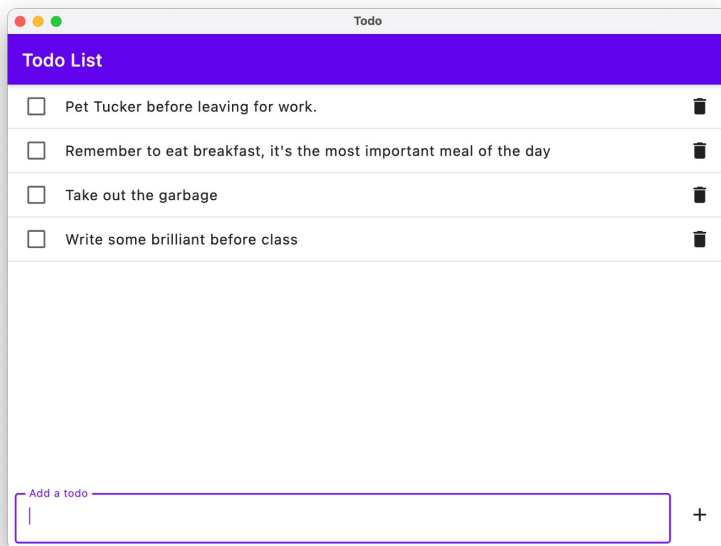
- Compose WASM is "on the way".

In this course we'll focus on Compose for Desktop and Android.

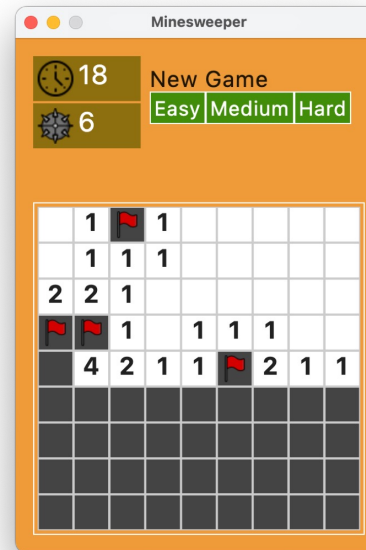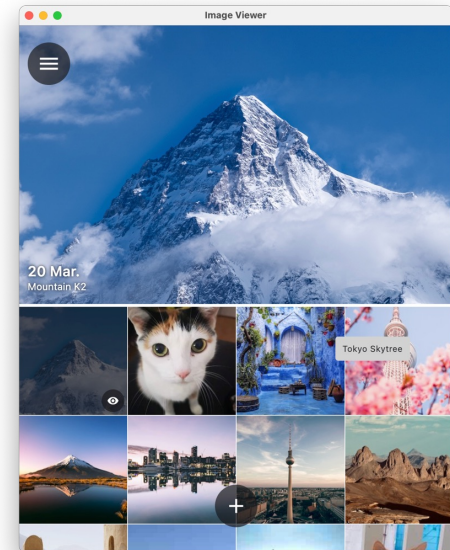This is the rare case where we can use the same toolkit for more than one platform!

https://www.jetbrains.com/lp/compose-multiplatform/
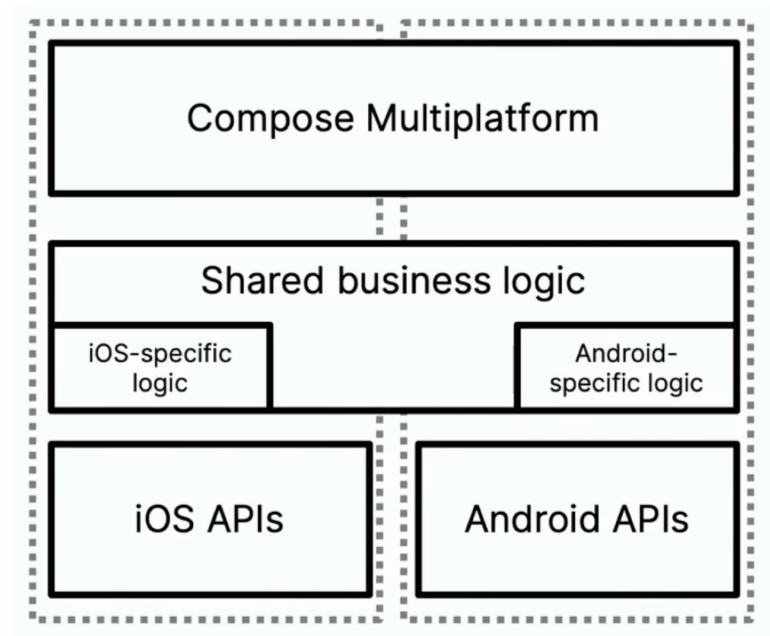
What can Compose do?

Todo

Minesweeper

ImageViewer

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

Creating a Compose project?

Desktop:

- IntelliJ > New Project > Compose Multiplatform

Android:

- Android Studio > New Project > Phone and Tablet > (Empty Activity)

Composables

The building blocks of any user interface.

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.*

- e.g., 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.

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

Composable Scope (1/2)

Let's display a window.

composable scope

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

@Composable
fun Greeting(name: String) {
  Text("Hello $name!")
}
```
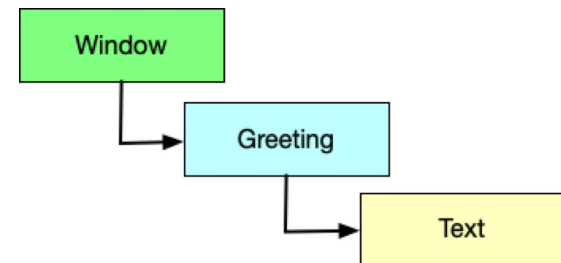
The `application` function defines a **Composable Scope** – think of it like a wrapper.

Composable functions must be called from a Composable Scope, or from other Composables.
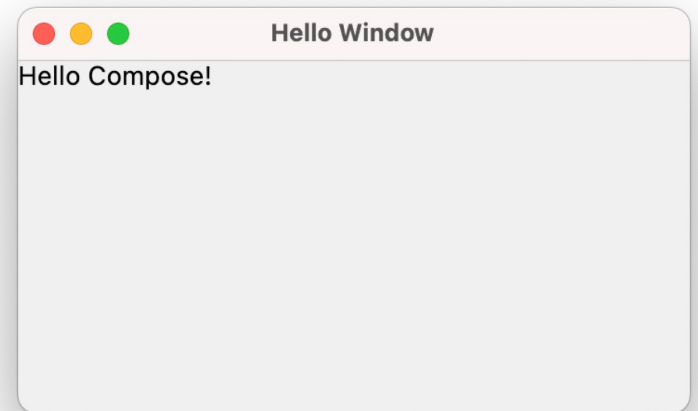
These composables describe a scene graph.

15

Composable Scope (2/2)

Here's the resulting window.

```kotlin
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 GitLab repo: /samples/compose-demo

Using Composables

- **With compose, you 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.
  - Other composables display data, and (some) provide interactivity for users.

- Because Compose is cross-platform, most composables 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
  )
}
```

Widget Demo

Example: TextField, OutlinedText

A labelled text field

```
val text = remember { mutableStateOf("Hello") }

TextField(
    value = text.value
    label = { Text("Label") }
)

OutlinedTextField(
    value = text.value,
    label = { Text("Label") }
)
```

Label
Hello

Label
Hello Compose

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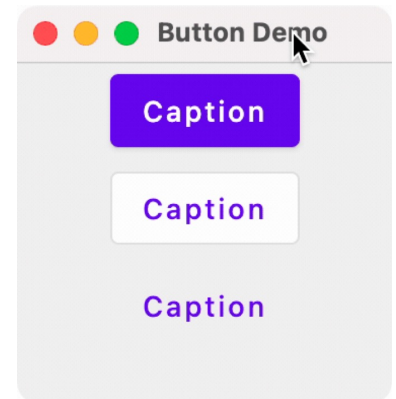
```kotlin
fun main() {
  application{
    Window(onCloseRequest = ::exitApplication, title = "Button Demo")
    {
      Column(modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally)
      {
        Button(onClick = { println("Button clicked") }) { Text("Caption")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

22

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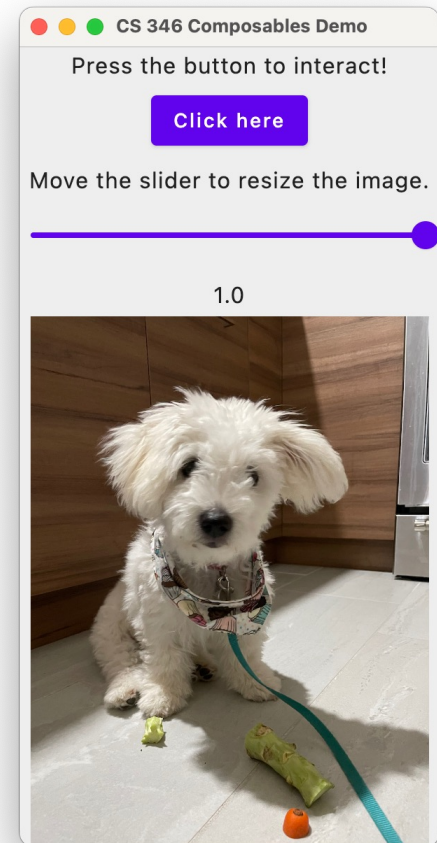
```kotlin
@Composable
fun SliderMinimalExample() {
    var sliderPosition by remember
        { mutableFloatStateOf(0f) }

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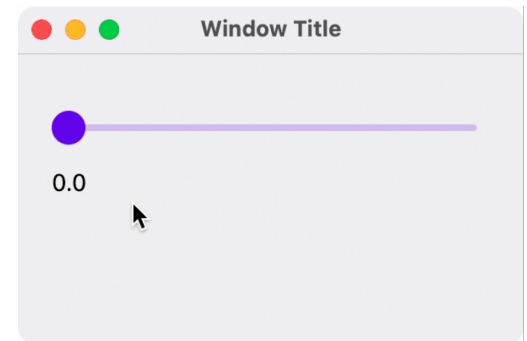
Demo

- GitLab: /sample-code/compose-demo
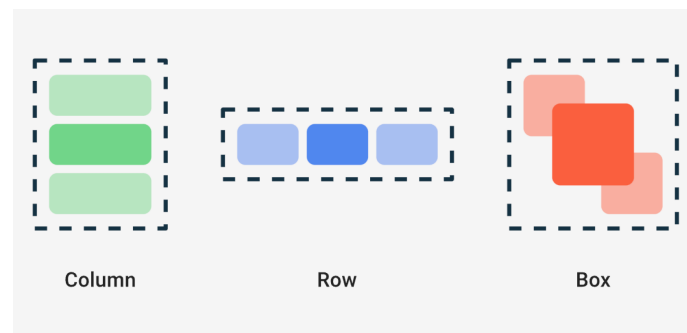- Run the main method

Layout

How do you control the placement of component?
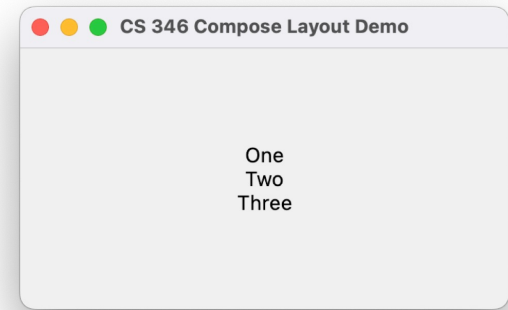
Layout Composables

- Compose includes Layout Composables, whose purpose is to act as a container to other composables. The three main layouts:
  - **Column**, used to arrange widget elements vertically
  - **Row**, used to arrange widget elements horizontally
  - **Box**, used to arrange objects in layers
- Platforms may also have specific layouts e.g., **Scaffold** on Android.

Column   Row   Box

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

Column Composable

```kotlin
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")
  }
}
```
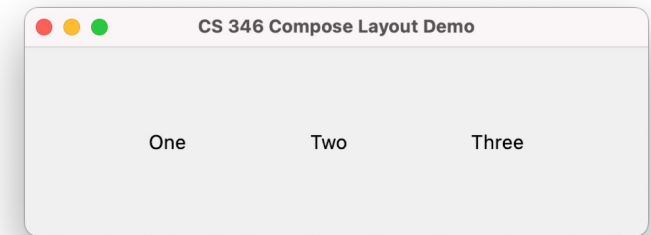
CS 346 Compose Layout Demo

One
Two
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")
  }
}
```

CS 346 Compose Layout Demo

One   Two   Three

**Arrangement**: The direction the composable flows.

Alignment: Orthogonal to the arrangement.
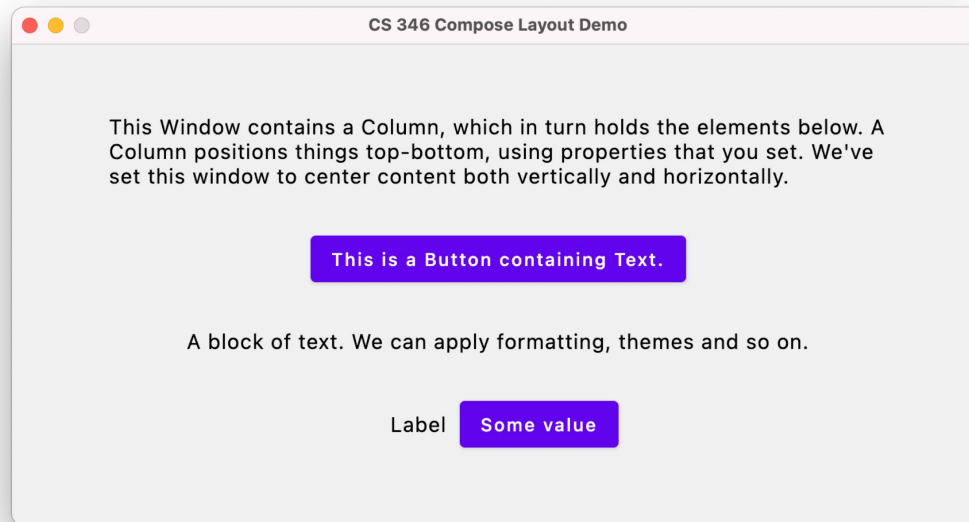
Box Composable

```kotlin
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("+")
      }
  }
}
```
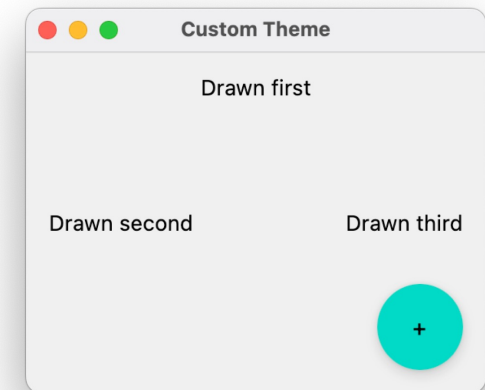
Nesting Layouts

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).

What is a view? A screen?

A screen is just a top-level composable, typically in its own View file.

```kotlin
@Composable
fun MainView() {
    TextRow()
}

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

Navigation

How to transition between screens?

Why is navigation important?

- Complex applications will need multiple screens.

- The typical paradigm is navigating forward and backwards through a series of screens e.g., web browser, mobile applications.

- We need programmatic support for:
  - Switching to a new screen (and potentially passing data forward).
  - Keeping track of the user's navigation history, so that we can go forward/backward through screens.
  - Deep-linking so that we can "jump" to a particular place in the navigation list.
  - Animations and transitions between screens.

Option 1: Simple navigation

What if you just want to switch between two screens?

- Easy solution: composable functions for each screen, and you just choose which one to load based on application state.

- Involves `state-hoisting`.

```kotlin
fun main() = application {
    Window(
        title = "Simple Navigation",
    ) {
        var screen by remember { mutableStateOf<Screen>(Screen.SCREEN_A) }   // track the current screen
        when(screen) {
            Screen.SCREEN_A -> ScreenA({ screen = Screen.SCREEN_B })          // recompose on screen change
            Screen.SCREEN_B -> ScreenB({ screen = Screen.SCREEN_A })          // pass in button press code
        }
    }
}

@Composable
fun ScreenA(clickHandler: () -> Unit) {
    Column {
        Text("Screen A")
        Button(onClick = { clickHandler() }) { Text("Go to Screen B") }      // button invokes lambda
    }
}

@Composable
fun ScreenB(clickHandler: () -> Unit) {
    Column {
        Text("Screen B")
        Button(onClick = { clickHandler() }) { Text("Go to Screen A") }
    }
}
```

See: GitLab > samples > compose-lecture > navigation

Option 2: Complex Navigation

- When just moving between screens isn't sufficient.
- You want an external component to "decide" which screens to load.
  - e.g., navigation bar that chooses what is displayed based on conditions.
- You need to pass complex data between screens.
  - e.g., moving from a summary to detail view (list of customers, to one record).
- We have Navigation libraries to help with this:
  - Jetpack Navigation for Android
  - Compose Navigation for Desktop
  - Voyager multiplatform for Compose (3rd party)

We will revisit in the Platform lectures.

Voyager Navigation

Voyager – works on Android, iOS, desktop. It's simpler to setup and use.

```kotlin
class HomeScreen : Screen {

    @Composable
    override fun Content() {
        val screenModel = rememberScreenModel ()
        // ...
    }
}

class SingleActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Navigator(HomeScreen())
        }
    }
}
```

Voyager makes simple navigation between screens very easy.

If you don't need a more complicated navigation model i.e., deep-linking, then it may be a better choice.

It's also multiplatform so it should work anywhere you can compile a Kotlin application.
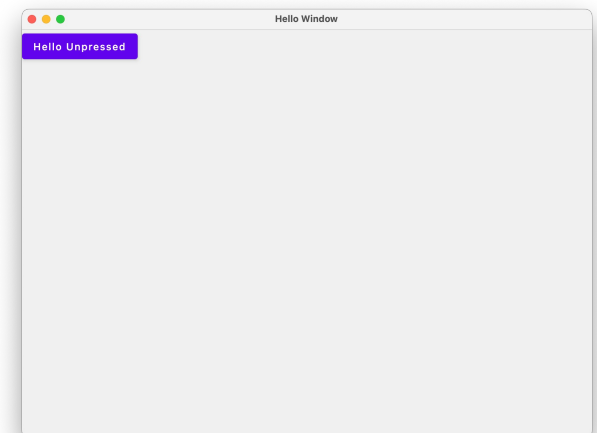
Interactivity & State

How to manage state in your views.

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.

```kotlin
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
```

samples/compose-demo > state/HelloState.kt

43

Adding Interactivity (2/4)

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

```kotlin
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")
  }
}
```
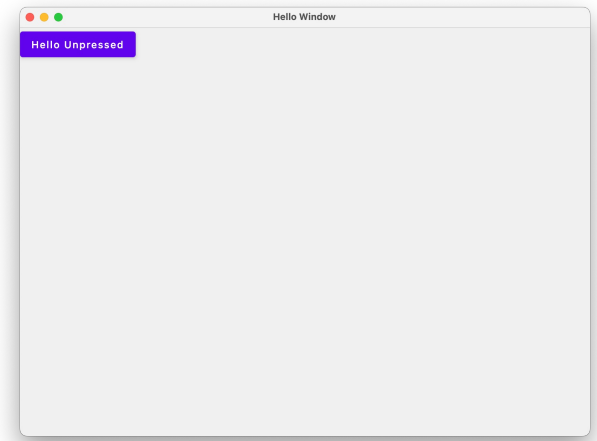
It doesn't work. The UI never updates.
Why?

samples/compose-demo > state/HelloState.kt

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?

```kotlin
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")
  }
}
```
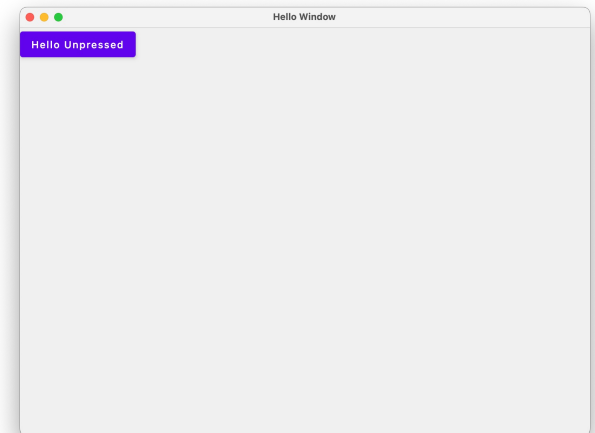

This doesn't work.
The `onClick` handler attempts to change the text property of the Button.
This triggers Compose to call the Window composable, which calls the Button composable, which initializes `text` to its initial value...

Adding Interactivity (4/4)

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

```kotlin
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}")
  }
}
```
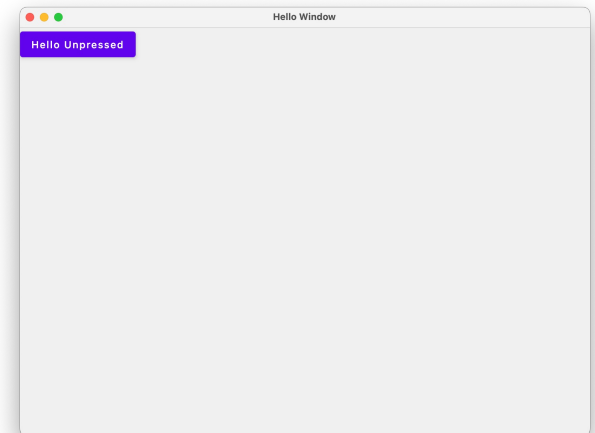

This works!
mutableStateOf(name) is an observable String (via type inference).
Remember tells it the @Composable function to NOT re-initialize this state when Recomposition happens.

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
  ) {
      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)

```kotlin
fun main() = application {
  Window( title = "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")
      OutlinedTextField(value = name,
          onValueChange = onNameChange, label = { Text("Name") })
  }
}
```

Hello Window

Hello, Jeff

Name

Jeff

Our state is the name that the user is entering in the OutlinedTextField.
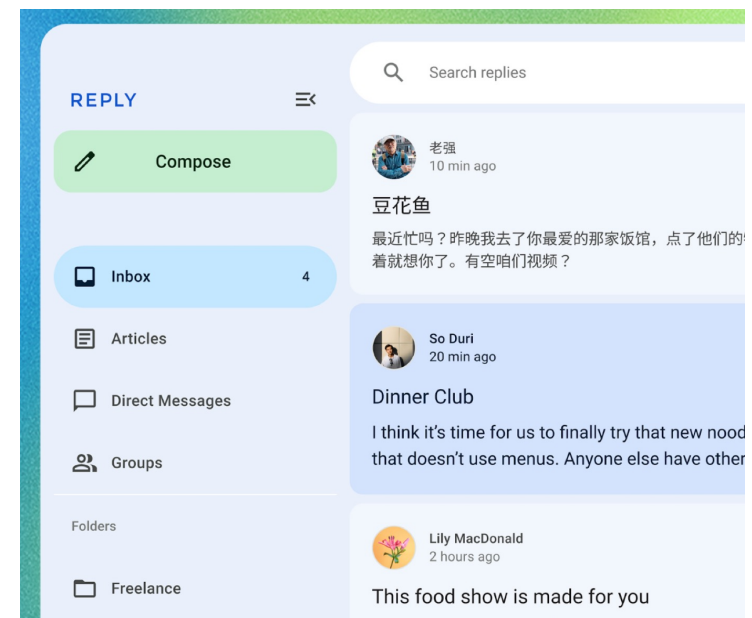
Instead of storing that in our HelloContent composable, we keep our state variable in the calling class HelloScreen and pass in the callback function that will set that value.
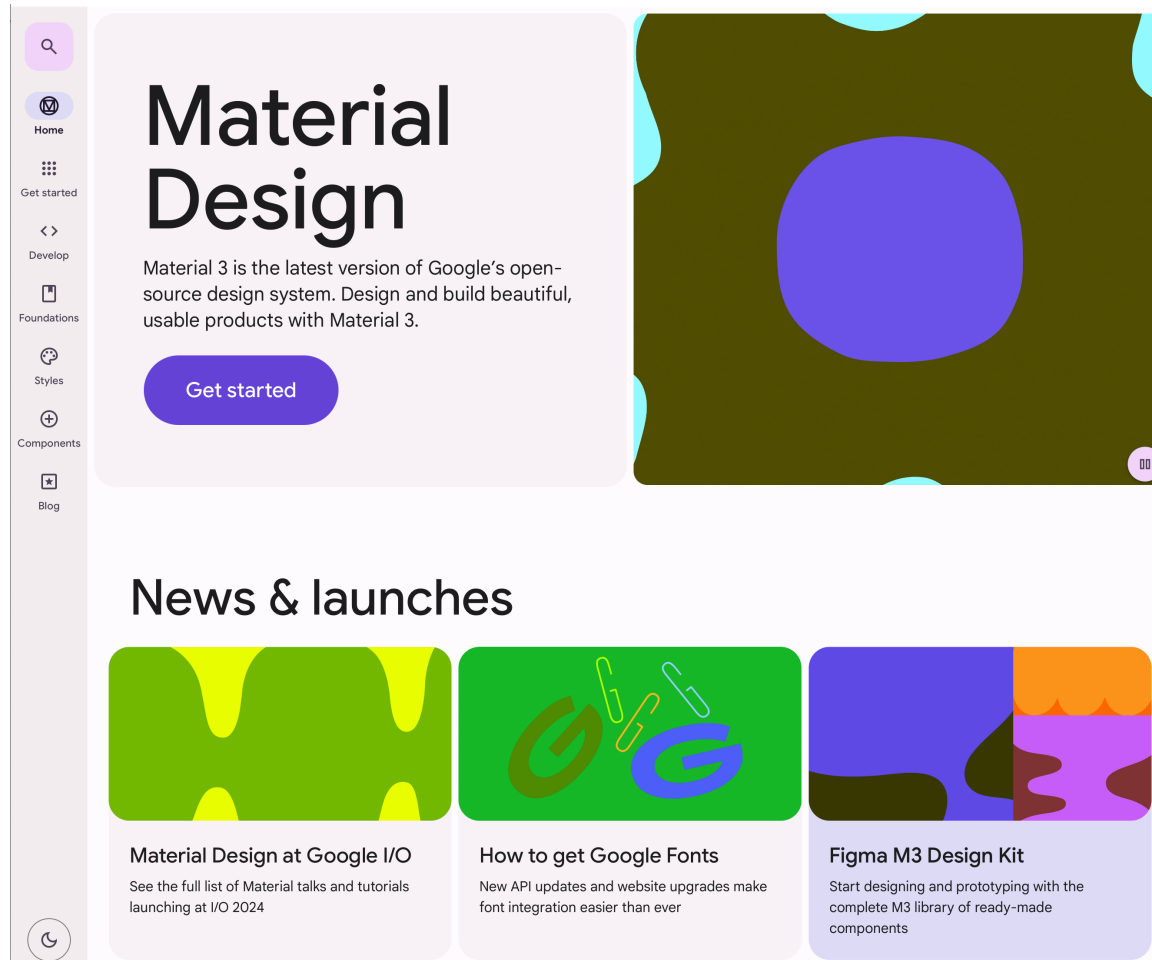
Themes

How to customize it?

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.

Default starting point of our app with the baseline theme.

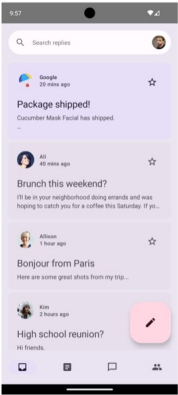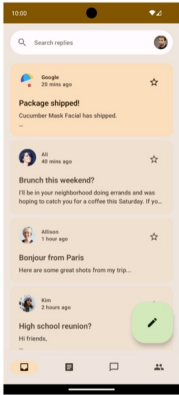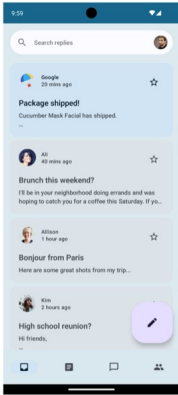You'll create your theme with color scheme, typography, and shapes, and then apply it to your app's email list and detail page. You will also add dynamic theme support to the app. By the end of codelab, you'll have support for both color and dynamic themes for your app.

| Baseline Theme | Color Theme | Dynamic Theme |

End point of the theming codelab with light color theming and light dynamic theming.

https://developer.android.com/codelabs/jetpack-compose-theming#0

54

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

```kotlin
@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 { … }
```

Reference

- Google. 2024. [Jetpack Compose Documentation](#).
- Google. 2024. [Thinking in Compose](#).
- JetBrains. 2024. [Compose Multiplatform Documentation](#).
- Phillip Lackner. 2024. [The Compose Multiplatform Crash Course](#)