User Interfaces

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

Introduction

What is a User Interface?

A user interface is the software or hardware mechanism that allows a person to provide input to a machine, and where the machine presents output back to the user. In software, we are focused on software user interfaces, in all of their forms.

Figure 1: A Playstation screen, Windows 11 desktop and terminal shell are all user interfaces. They may present different information, but they all facilitate user interaction.

What is a User Interface?

We can also define user interfaces as “the place where a person expresses intention to an artifact, and the artifact presents feedback to the person”. [1]

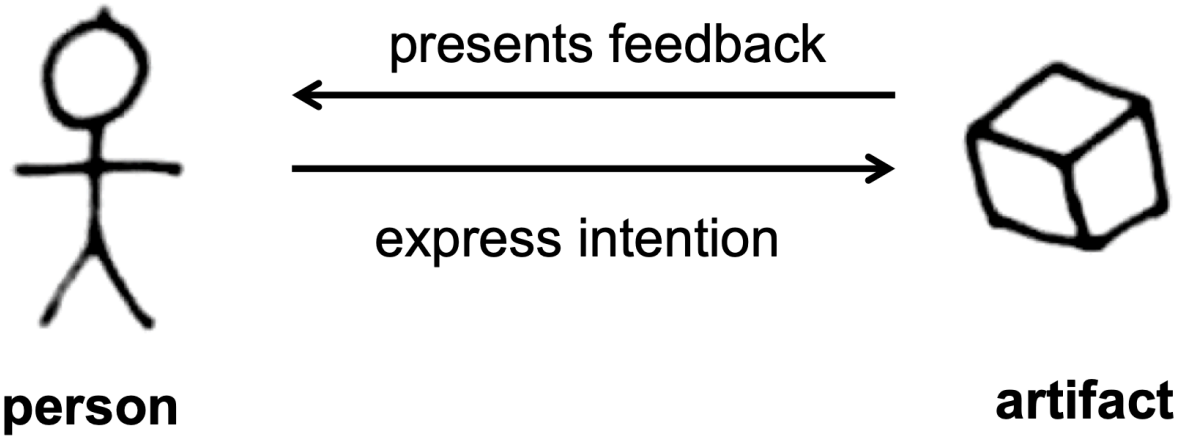


Figure 2: CS 349 talks about user interfaces in the context of interaction, where there is an input and output cycle between the user and device over time.

Graphical User Interfaces

Modern user interfaces are graphical, meaning that we rely on high-resolution graphical devices for output. Mobile phones, desktop computers, airport kiosks, game consoles all leverage graphical output.

Graphical user interfaces are characterized by:

- Graphical environment for text/graphics output.
- Interactive graphical elements e.g., buttons, menus, lists.
- Rich media support e.g., animations, graphics, sound.
- Pointing device.

Interaction often consists of “point-and-click interaction”, where the user manipulates some pointing device to direct input to a specific region of the screen. Devices to support this interaction include:

- Multi-touch input on mobile devices e.g., iPhone, Android.
- Joystick or game controller input on a game console. e.g., Playstation.
- Mouse or trackpad on a notebook computer.

Finally, most systems have a dedicated keyboard for entering text.

Baseline Functionality

As a developer who wishes to build a user interface, where do you start?
We need these capabilities:

1. Some ability to draw arbitrary graphics on the screen. This could be pixel-based graphics, or higher level primitives e.g., rectangles.
2. Ideally, it would be better to have pre-generated objects that we can place on the screen e.g., windows, buttons, scrollbars, panels.
3. An ability to support interaction with these graphical representations. i.e., click, drag, enter/exit region and all of the typical interactions we would expect.

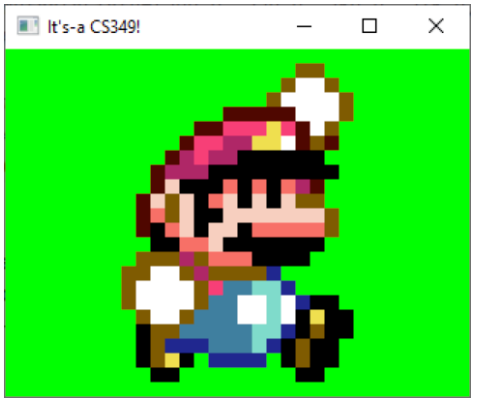


Figure 3: Mario can be drawn as a series of pixels.

GUI Toolkit

A GUI toolkit is a framework which provides the functionality that is required to build and manage a user interface. Although we could write code to do everything ourselves, this lets us design and develop at a higher-level of abstraction.

Toolkits provide us with these capabilities:

- Creating and managing application windows, with standard functionality e.g. overlapping windows, min/max buttons, resizing.
- 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.
- The ability to handle user interaction with hardware e.g., keyboards, touch.
- Advanced, platform specific features like screen navigation and animation.

> A GUI toolkit provides literally everything that you need for your platform.

Types of Toolkits

Imperative

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

- UI objects are just classes with properties for position (x,y), dimensions (w,h), visual properties. e.g. Button, Scrollbar, Panel.
- Code places elements on-screen and controls their appearance.
- Code determines how the user interface behaves based on input.

An imperative toolkit relies on custom code to change the user interface in response to application state changes.

- You end up writing a lot of code that reacts to a user's input by changing the state of these components. This is a large part of the application's complexity!

Examples

- Swing, Qt, JavaFX, MFC, Gtk.

Types of Toolkits

This is an example of an imperative UI built using Kotlin and JavaFX. A lambda is called to respond to the user selecting an item on the list. Any state changes need to be explicitly called out.

```
class Main : Application() {
  override fun start(stage: Stage) {
    val list = ListView<String>()
    list.items.addAll("One", "Two", "Three", "Four", "Five")
    list.selectionModel.selectIndices(0)
    list.selectionModel.selectedItem
    list.selectionModel
          .selectedItemProperty()
          .addListener { _, old, new → println("$old → $new") }

    stage.title = "List Demo"
    stage.scene = Scene(StackPane(list), 400.0, 300.0)
    stage.isResizable = false
    stage.show()
  }
}
```

Declarative

Many modern toolkits are moving to a **declarative** appproach:

- 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? if so, change its color and allow it to be clicked.
  - e.g., is there data in the list that the user can select?

A declarative toolkit automatically manages how the UI reacts to state changes. It infers how the UI presents state to the user.

Examples

- React, SwiftUI, Flutter, Compose.

> We'll explore lots of samples in the next slides!

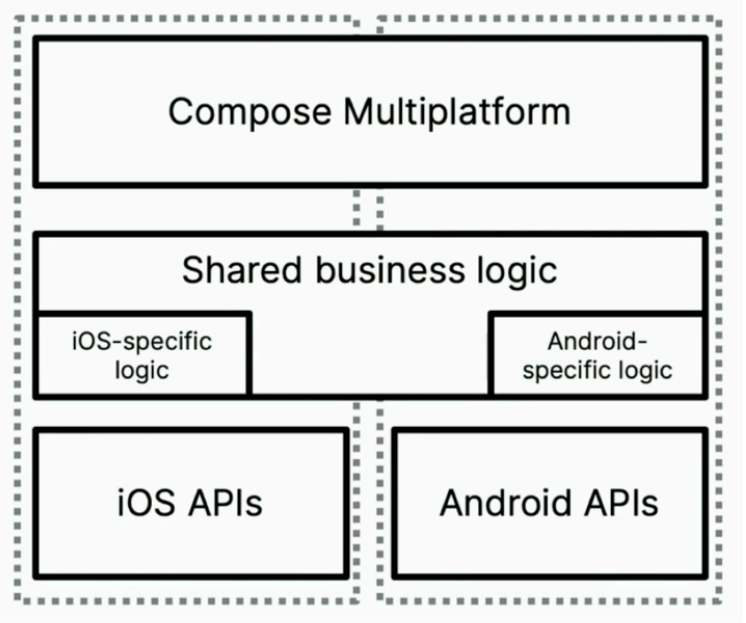
Compose Multiplatform

Introduction

`Compose` is a declarative, cross-platform toolkit.

- It was designed by Google, and released as Jetpack Compose for Android in 2017. [2]
- JetBrains ported it to desktop, and it was released in 2021 as Compose Multiplatform for desktop (macOS, Windows, Linux), and iOS. [3]
- Compose JS and WASM are both under development.

In this course we'll focus on Compose for Desktop and mobile targets including Android and iOS.



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

Figure 4: Compose Multiplatform is a cross-platform GUI toolkit that sits outside of the rest of your application code.

Introduction

With imperative toolkits, you need to monitor user inputs, and write code to determine what state changes result from that input. Much of the complexity of your application comes from managing that state.

Declarative toolkits like Compose seek to manage that relationship for you. Compose monitors state in your application, and automatically emits the user interface that reflects that state. This greatly reduces code complexity [4].

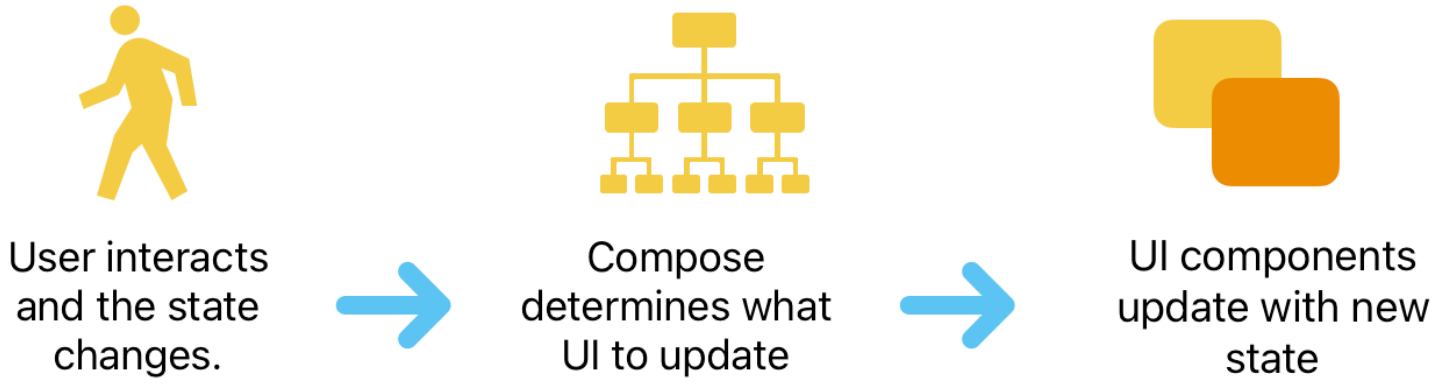


Figure 5: User interface updates are managed at runtime by Compose. This is very efficient because it only updates those on-screen components that are affected by the state change.

How It Works

The Composition Cycle

Compose works by transforming state into UI elements, determining how they should be structured and then drawing them efficiently. This needs to be done continuously as the state of the application changes.

Steps include.

1. **Collecting state**, often through user input.
2. **Composition of elements**, from our composables in code.
2. **Layout of elements**, determining where and how to position them.
3. **Drawing** elements on the screen.

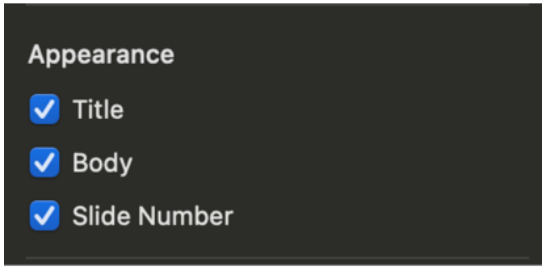
Let’s assume a starting state and focus on the last three steps.

How It Works

Scene Graph

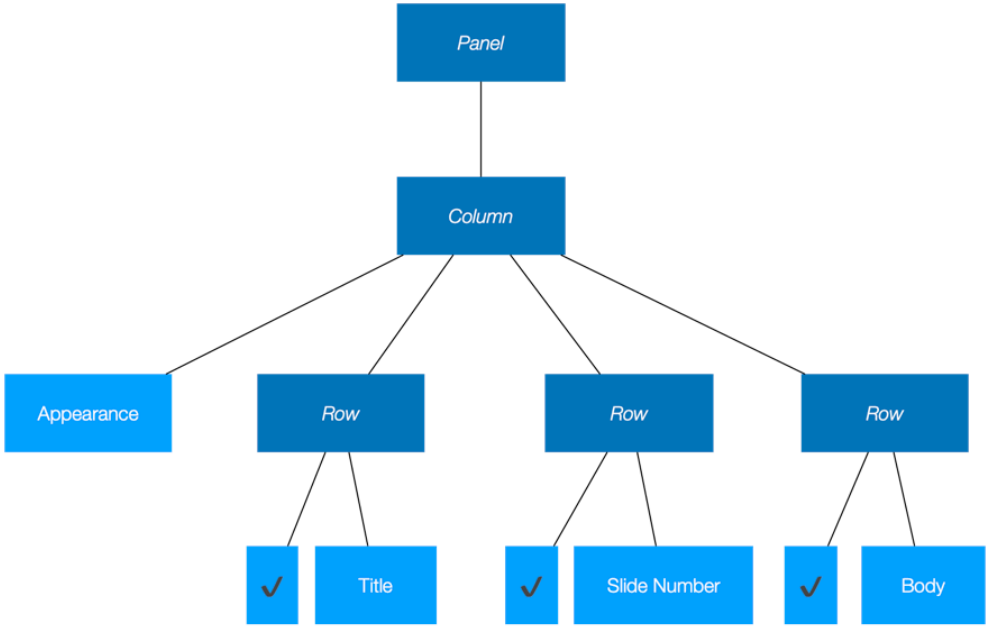
In GUI development, it's common to represent graphical content as a tree of nodes (e.g., `components` or `widgets`). This is called a `component tree` or `scene graph`. As we will see, using a tree makes traversing the UI for drawing and other operations very efficient.

Like other toolkits, Compose also uses this type of hierarchical structure.



UI Panel

Figure 6: This panel consists of a title, and three checkboxes with labels. The scene graph to the right represents this UI component.



Scene Graph for this panel

Composable Functions

The key “building block” in Compose is a composable function (also called a `composable`). A composable function is a special kind of function that accepts state and emits a user interface element.

- They are guaranteed to be fast, idempotent, and free of side effects.
- Composables emit output directly into the scene graph.
- They have no return value.

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

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

> Composables are the fundamental building blocks of our user interfaces!

Composable Functions

Composable Scope

We can build a hierarchy of UI elements like this.

```
fun main() = application {
  Window(
    title = "Hello Window"
  ) {
      Greeting("Compose")
    }
}

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

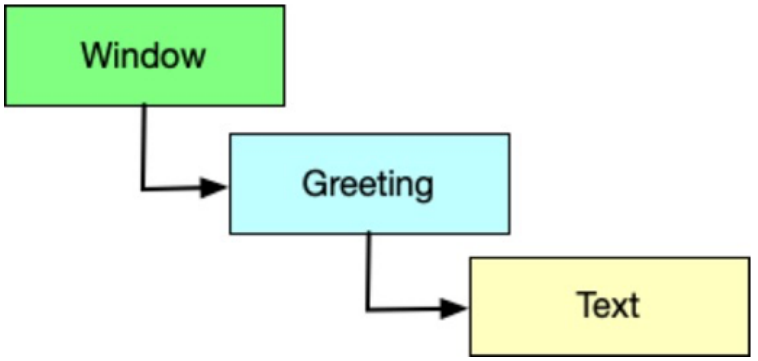


Figure 8: This is a hierarchical relationship, consisting of three nested composables.

The `application` function is a built-in function to define a `Composable Scope`. Composables can only be called from within a Composable Scope.

Composable Functions

The resulting window is a standard desktop window, created and managed by Compose at runtime.

```
fun main() = application {
  Window(
    title = "Hello Window"
  ) {
      Greeting("Compose")
    }
}

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

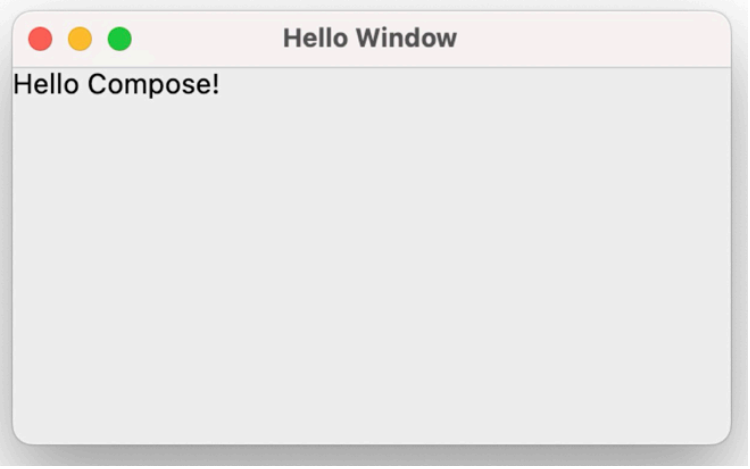


Figure 9: Here's the resulting window. Compose handles decorations like `min`, `max`, `restore`.

Composable Functions

Properties

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

These are exposed as named parameters.

Examples:

- `Text` has properties for `textAlign`,`lineHeight`, `fontName`, `fontSize`.
- `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.

Composable Functions

Built-In Composables

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

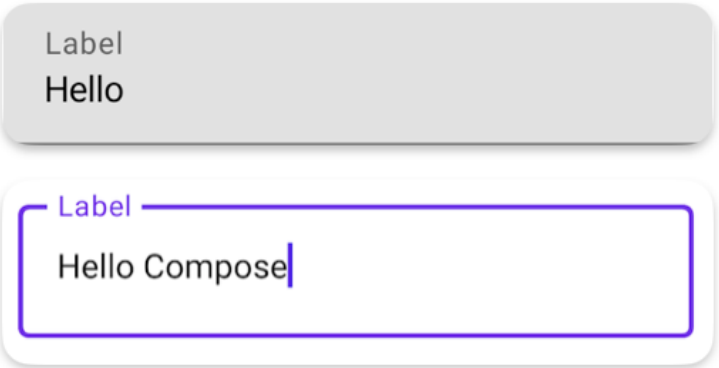
Composable Functions

A Textfield composable displays text with an optional label.

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

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

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



Composable Functions

An Image composable displays an image or picture. By default, the image is loaded from your `Resources` folder.

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

Composable Functions

Interactive Composables

So far, our composables have been non-interactive. Let's talk about input.

When the user interacts with a screen:

1. An `event` is generated that indicates what happened e.g., "mouse clicked".
2. Your application contains code (which we call `listeners` or `handlers`) that are called in response to this event.
3. These handlers execute in response to the events. This often results in a state change e.g., "entering text in a `name` field updates a `name` variable".
4. Compose reacts by redrawing parts of the UI that rely on the changed state.

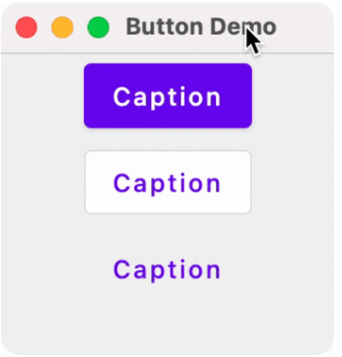
An event is simply a message. Common events include:

- **MouseMove**: Indicates that the pointer has been repositioned.
- **MouseClick**: The user has clicked on something with a mouse.
- **KeyPress**: A key on a keyboard has been pressed.
- **WindowClose**: The window has been closed.

Composable Functions

There are three main Button composables.

- They vary in appearance only.
- `onClick` is a handler that responds to a MouseClick.
- Other handlers exist e.g., `onMouseMove`, `onKeyPress`.
- We commonly use lambda functions for handlers.

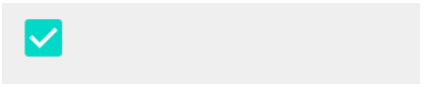



```
Column( modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally)
) {
  Button(onClick = { println("Button") }) {
    Text("Caption")
  }
  OutlinedButton(onClick = { println("OutlinedButton") }) {
    Text("Caption")
  }
  TextButton(onClick = { println("TextButton") }) {
    Text("Caption") }
  }
}
```

Composable Functions

A Checkbox composable is a toggleable control that presents a Boolean state. The `OnCheckedChange` function is called when the user interacts with it.

In this example, we'll write our own Composable function to track the state and emit the appropriate checkbox. This lets us keep the drawing and state together.



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

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

Composable Functions

A Slider composable 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 SliderExample() {
  var sliderPosition by remember
    { mutableStateOf(0f) }

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

State Management

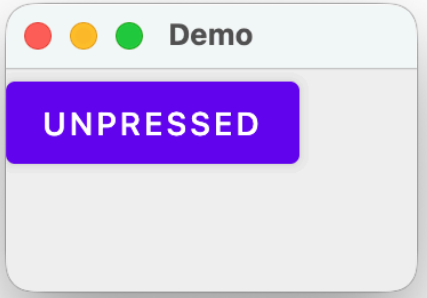
Let's look more carefully at how to manage application state for a UI.

Here's a button captioned "Unpressed". When you click on it, we want it to:

- set the button caption to "Pressed", and
- print "Pressed" to the console.

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

@Composable
fun ToggleButton(caption: String) {
  var str = caption.uppercase()
  Button(onClick = { println("Pressed") }) {
    Text(str)
  }
}
```

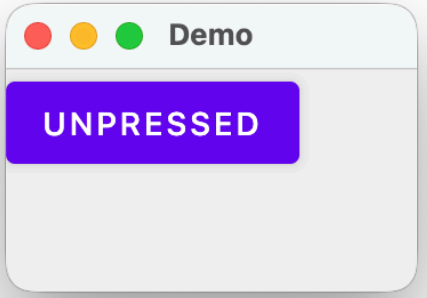


State Management

We added the functionality to change the caption as part of the handler, but it doesn't work. What's happening?

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

@Composable
fun ToggleButton(caption: String) {
  var str = caption.uppercase()
  Button(onClick = {
    str = "Pressed"
    println("Pressed")
  }) {
    Text(str)
  }
}
```



State Management

Recomposition

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

Compose is effectively doing this:

1. Drawing the initial user interface.
2. Monitoring your state (aka variables) directly.
3. When a change is detected, parts of the UI that use that state are updated.
4. Redrawing the affected components by calling their Composable functions.

> This process (detecting a change and then redrawing the UI) is called recomposition. It's purpose is to ensure that we only redraw elements when their dependent data changes.

State Management

Why didn't our example work?

The `onClick` handler attempted to change the text property (state) of the Button. However, Compose doesn't know how that state is being used. Remember that it ONLY wants to redraw a part of the UI if it knows that dependent state was changed, otherwise it ignores state changes.

How do we address this? We introduce a `MutableState` object to store data. Think of it as an observable variable. Use it like a regular variable (mostly). Compose will monitor its state and make sure that UI that uses that state gets updated.

All we have to do is

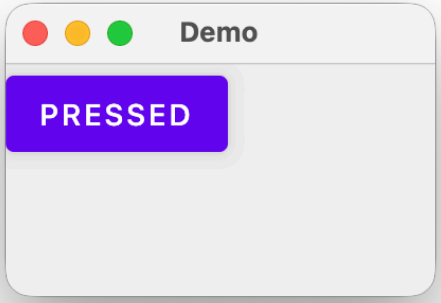
- change our `str` variable to a `MutableState` variable.
- reference the value using the MutableState variable's `value` property.

State Management

This works!

```
fun main() = application {
  Window(title = "Demo", onCloseRequest = ::exitApplication) {
    ToggleButton("unpressed")
  }
}

@Composable
fun ToggleButton(caption: String) {
  var str = mutableStateOf(caption.uppercase())
  Button(onClick = {
    str.value = "PRESSED"
    println("Pressed")
  }) {
    Text(str.value)
  }
}
```



State Management

State Hoisting

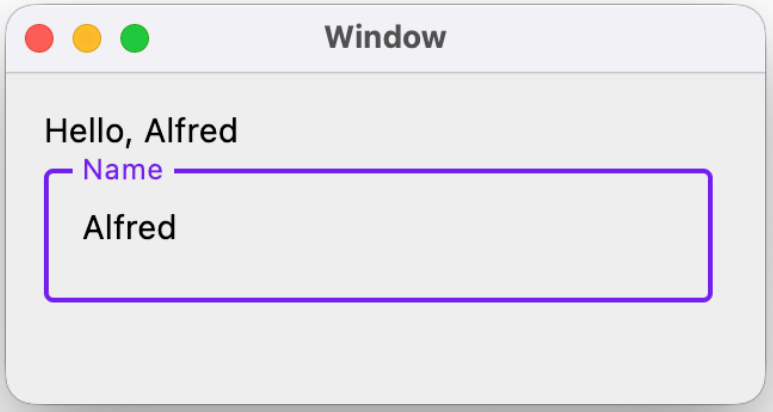
Storing state in a function can make it difficult to test and reuse. It's sometimes helpful to pull state out of a function into a higher-level, calling function. This process is called `state hoisting`.

Example:

We have two high-level composables:

- HelloScreen (top-level window), and
- HelloContent (panel containing the Text composables).

We want to capture the name that the user types in the `OutlinedTextField`, and make it available to the other Text field.



,

State Management

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

@Composable
fun HelloScreen() {
    var name by remember { mutableStateOf("Jeff") }
    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") })
    }
}
```

Layout

While composables describe what to display on the screen, they do not define how those components should be arranged.

For example, this code generates two text elements. Without additional guidance, they display stacked on top of one-another.

```
@Composable
fun ArtistCard() {
    Text("Alfred Sisley")
    Text("3 minutes ago")
}
```

`Layout` is the step that determines how the composables that we emit in our code should be arranged on screen. We define a layout by writing code to describe how we want things arranged, and the Compose engine applies these rules at runtime.

Layout

The Layout Model

In the layout model, the UI tree is laid out in a single pass.

- Each node is first asked to measure itself, then measure any children recursively, passing size constraints down the tree to children.
- Then, leaf nodes are sized and placed, with the resolved sizes and placement instructions passed back up the tree.
- Parents measure before their children, but are sized and placed after their children.

This algorithm is handled for us at runtime, since components will need to adjust their required size and position based on their contents (i.e. the state of the application).

Layout

Built-In Layouts

Compose provides a collection of ready-to-use layouts. They serve as wrappers for our UI composables.

The main built-in layouts are:

- 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.

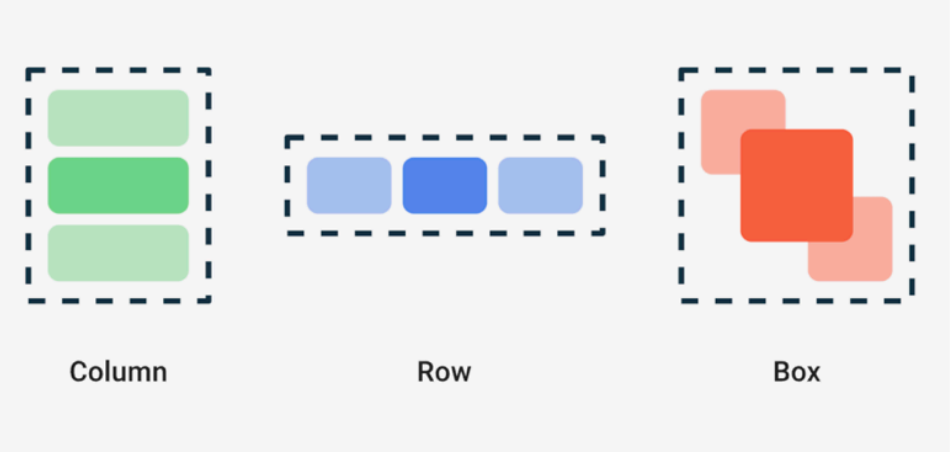


Figure 15: See the Compose Documentation for more details.

Layout

Column

A Column Composable arranges components vertically.

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

Layout

Row

A Row Composable arranges components horizontally.

```
@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.

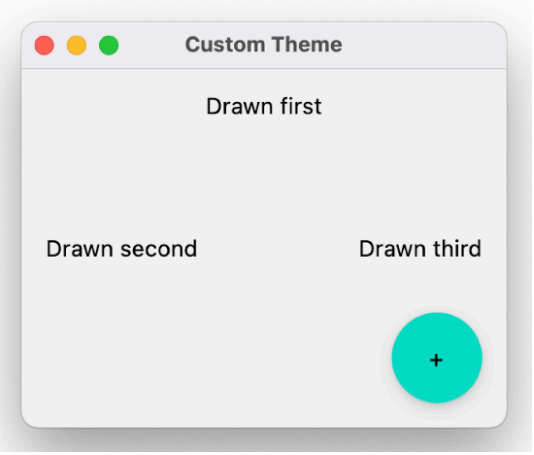
Alignment: Orthogonal to the arrangement.

Layout

Box

A Box Composable lets you arrange components in different areas.

```
@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.BottomE
    ) {
        Text("+")
      }
  }
}
```



Layout

Building a Screen

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

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

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

Navigation

Many applications work perfectly fine with a single screen. That single screen may not even need to change much as the application is running.

However, complex applications will often have multiple screens and the user will need to navigate through them.

- e.g., navigating forward and backwards through screens on a mobile phone.
- e.g., opening multiple windows on a desktop

The user needs to be able to:

- Switch to a new screen (and potentially pass data “forward”).
- Go backward and forward through a set of screens.
- “Jump” to a particular screen directly.

Of course, we also want these transitions to be animated.

Navigation

Option 1: Simple Navigation

What if you just want to switch between two screens?

The easiest solution:

- Replace the UI from your current screen with a new UI.
- This is simply a matter of having different composables for each screen, and programatically decising which one to invoke.
- It involves `state-hoisting`.

You will need to implement functionality yourself

- Not difficult, suggested to just build what you need!
- Great use case for Desktop, which is usually simpler.

Navigation

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

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

Navigation

```
class enum SCREEN {
  SCREEN_A,
  SCREEN_B
}

fun main() = application {
  Window(title = "Simple Navigation") {
    // track the current screen, defaults to SCREEN_A
    var screen by remember {
      mutableStateOf<Screen>(Screen.SCREEN_A)
    }

    // determine screen to display based on current value
    // this re-composes anytime the screen value changes
    when(screen) {
      Screen.SCREEN_A → ScreenA({ screen = Screen.SCREEN_B })
      Screen.SCREEN_B → ScreenB({ screen = Screen.SCREEN_A })
    }
  }
}
```

Navigation

Option 2: Navigation Libraries

When just moving between screens isn't sufficient, we have richer libraries.

Wen to use these?

- 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 for KMP (preferred).

Navigation

```
class HomeScreen : Screen {
  @Composable
  override fun Content() {
    val screenModel = rememberScreenModel ()
    // ...
  }
}

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

Themes

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.

This is fantastic as an Android developer: it's very well specified and complete.

You can

- Use the default values (not recommended), or
- Use the default as a starting point and customize typography, color scheme, shape format. (highly recommended).
- design your own theme. (not recommended, unless you have lots of time).

Themes

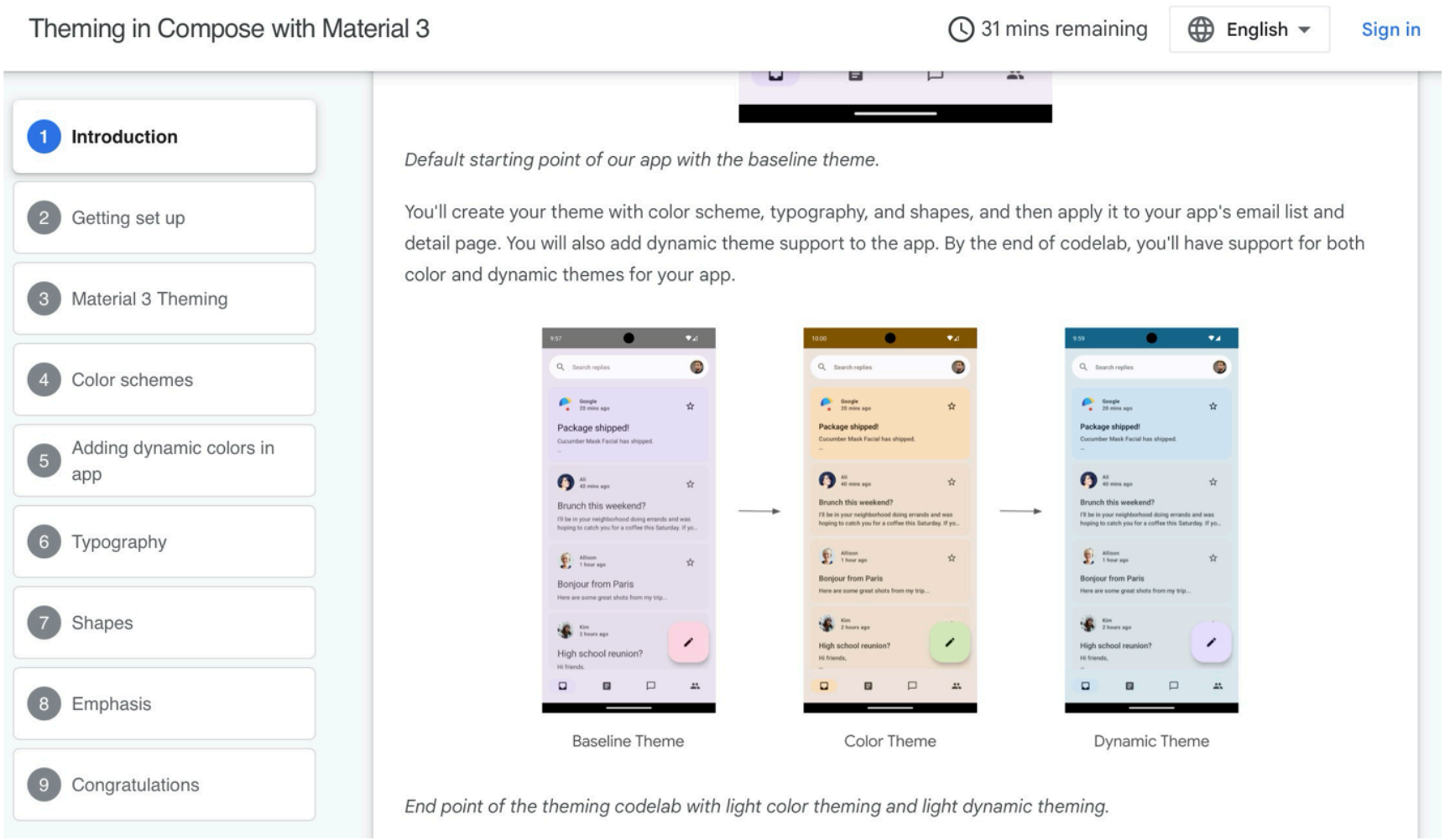


Figure 16: See the [Theming in Compose with Material 3] tutorial for examples of how to customize your application.

,

Bibliography

[1] D. Vogel, “CS 349 User Interfaces Slides.” [Online]. Available: https://student.cs.uwaterloo.ca/~cs349

[2] Google Inc., “Jetpack Compose Documentation.” [Online]. Available: https://developer.android.com/develop/ui/compose/documentation

[3] JetBrains Inc., “Compose Multiplatform.” [Online]. Available: https://kotlinlang.org/compose-multiplatform/

[4] Google Inc., “Thinking in Compose..” [Online]. Available: https://developer.android.com/develop/ui/compose/mental-model