# User Interfaces

CS 346 Application Development

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

# Contents

# Introduction

# Modern Interfaces

A user interface is the bridge betwen a person and an interactive system. It accepts and handles input, and returns meaningful results back to the user. Modern interfaces tend to be graphical, with rich media interfaces.


Figure 1: A Playstation screen, Windows 11 desktop and terminal are all user interfaces. They may be optimized for different types of information, but they all facilitate user interaction.

# Modern Interfaces

Modern user interfaces are graphical, meaning that we rely on high-resolution graphical devices for output. Most interfaces that we routinely encounter are graphical in nature.

Graphical user interfaces are characterized by:

- Ability to combine text/graphics output.
- Interactive graphical elements e.g., buttons, menus, lists.
- Rich media support e.g., animations, graphics, sound.
- Support for a pointing device for input.

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. e.g.,

- Multi-touch input e.g., iPhone, Pixel phones.
- Joystick or game controller input on a game console. e.g., Xbox.
- Mouse or trackpad on a notebook computer.

# Architecture

Recall that we're going to structure our application into these layers:

- a `UI Layer`, to handle input/output,
- a `Domain Layer` to handle business logic, and
- a `Data Layer` to store and manage data.

The UI layer is comprised of two types of structures:

- **UI elements** that render data to the screen e.g., buttons, images, sliders, radio buttons and other familiar widgets.
- **State holders** that hold data, expose it to the UI, and handle logic to interact with that data.

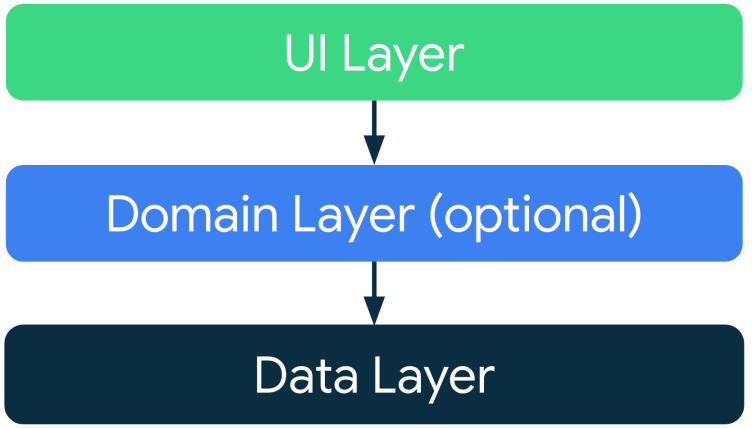



Figure 2: Courtesy of Android Guide to App Architecture

The role of the UI is to present the data that is managed by the other layers.

# GUI Toolkits

A GUI toolkit is a framework which provides the functionality that is required to build and manage a user interface. It is specifically responsible for both displaying visual elements, and managing state and interaction with those elements. Concretely, they provide us with these capabilities:

Graphical capabilities

- Creating and managing application windows, with standard functionality e.g. overlapping windows, 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.

Application state management

- Handling user interaction with hardware e.g., keyboards, touch.
- Promoting state changes so that they can be processed and displayed by the user interface.

# GUI Toolkits

## Imperative Toolkits

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.

# GUI 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 Toolkits

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!

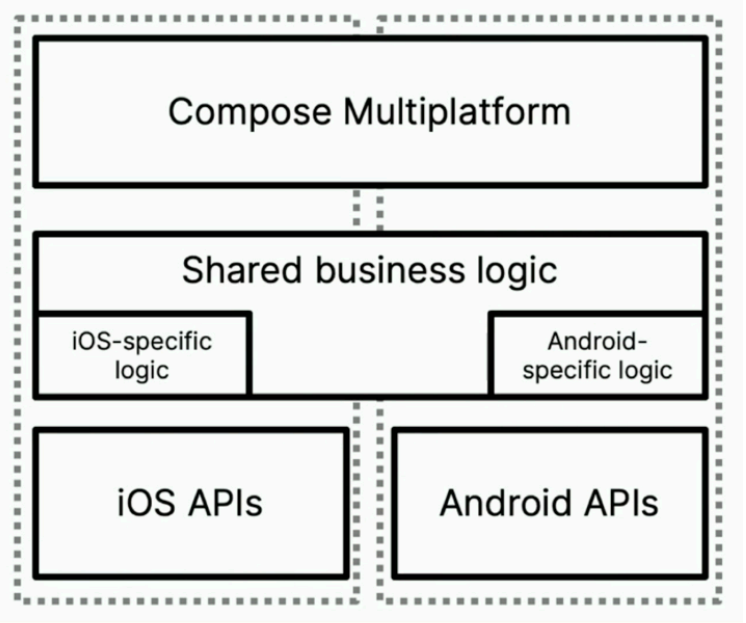
# Compose Multiplatform

# Introduction

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

- It was designed by Google, and released as Jetpack Compose for Android in 2017. [1]
- JetBrains ported it to desktop, and it was released in 2021 as Compose Multiplatform for desktop (macOS, Windows, Linux), and iOS. [2]
- 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 3: Compose Multiplatform is a cross-platform GUI toolkit that sits outside of the rest of your application code.

# Introduction

## What makes Compose declarative?

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 the application state**.

Declarative toolkits like Compose seek to simplify the relationship between the user interface and application state. **Compose monitors the state of your application, and automatically emits a user interface that reflects that state**. This greatly reduces code complexity [3].

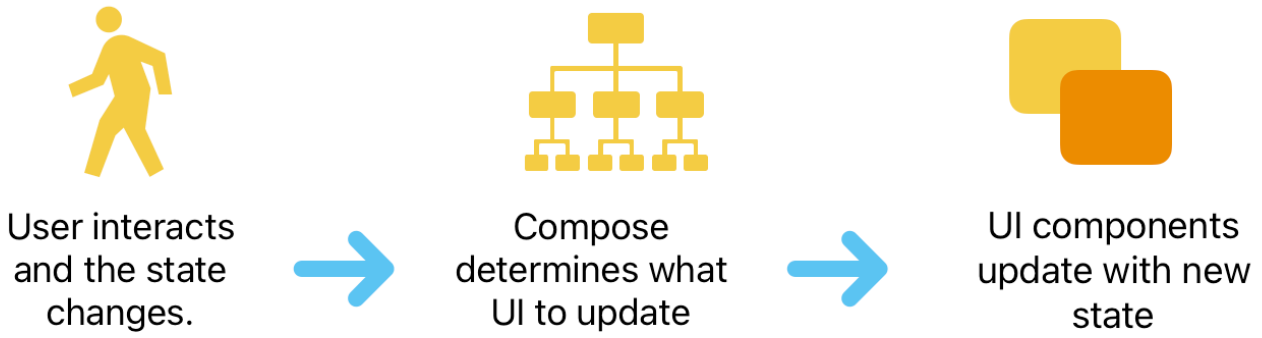



Figure 4: User interface updates are managed at runtime by Compose. This is very efficient because it only updates those on-screen components that are affected by the state change.

# Introduction

## 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.

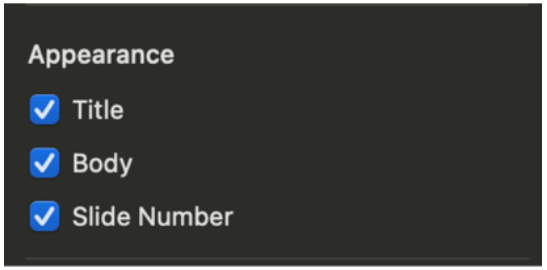

Let’s start by talking about graphical output.

# Graphical Output

## Scene Graph

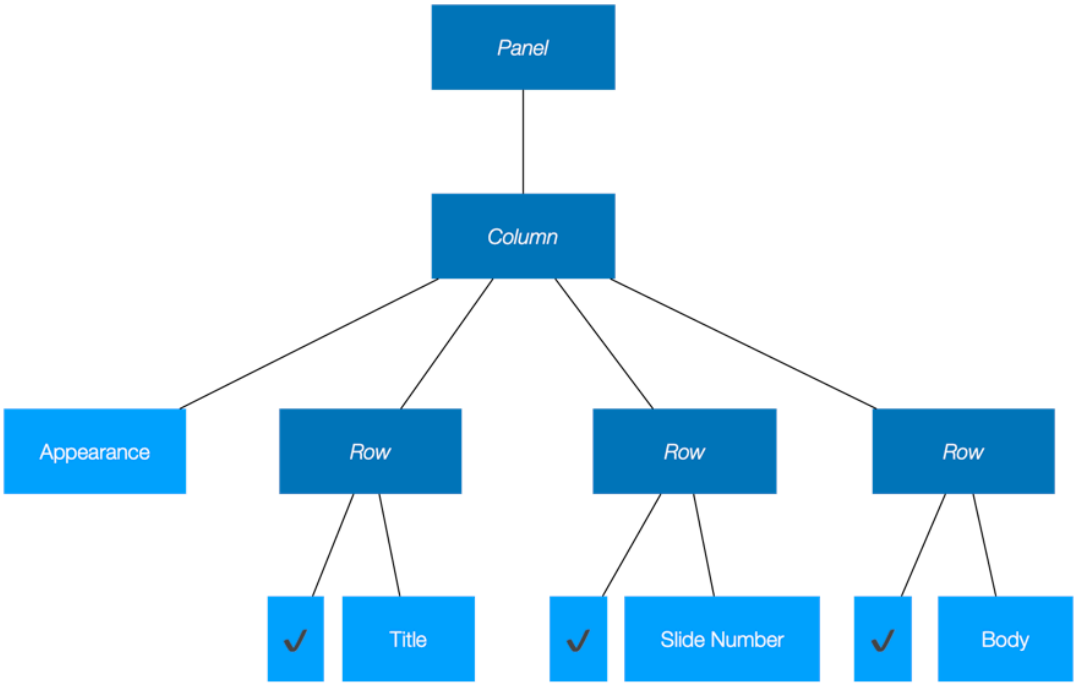
In GUI development, it's common to represent graphical content as a tree of nodes (e.g., `components` or `widgets`). This is called a `component tree` or `scene graph`. As we will see, using a tree makes traversing the UI for drawing and other operations very efficient.

Composes uses this structure for even the simplest user interface.




UI Panel

Figure 5: This panel consists of a title, and three checkboxes with labels.




Scene Graph for this panel

## Composable Functions

In Compose we use composable functions (also called a `composables`) to describe user interfaces interfaces. These are a special kind of function that accepts state and directly emits a user interface element.

- Composables emit output directly into the scene graph. They have no return valid, since they `emit` portions of the UI directly.
- Your scene graph is determined by a hierarchy of function calls.
- Composables are fast and idempotent i.e., free of side effects.

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!")
}
```

# Graphical Output

## Composable Scope

We can build a hierarchy of UI elements like this.

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

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

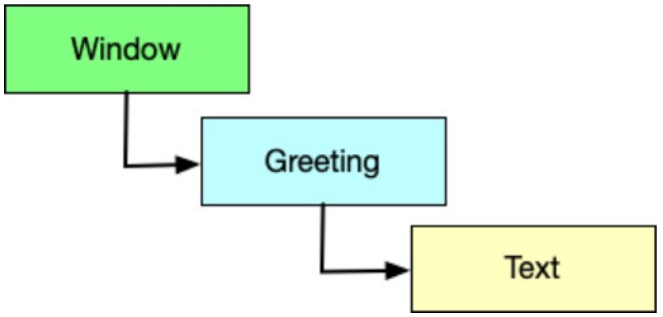



Figure 7: This is a hierarchical relationship, consisting of three nested composables.

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

# Graphical Output

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

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

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

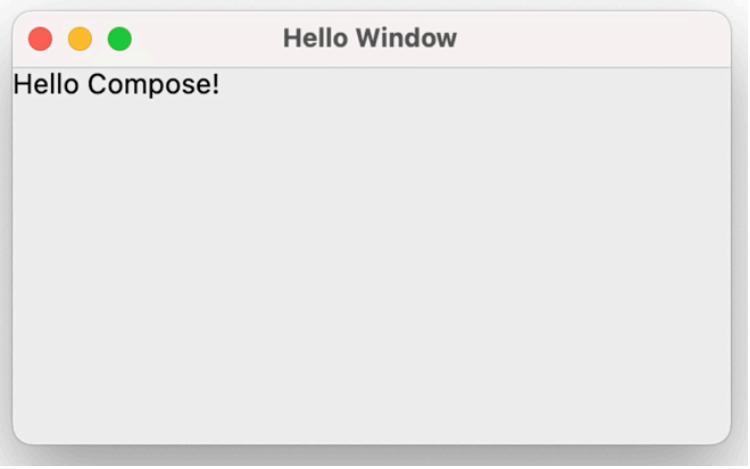



Figure 8: Here's the resulting window. Compose handles decorations like `min`, `max`, `restore` buttons.

# Graphical Output

## 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.

# Graphical Output

## Built-In Composables

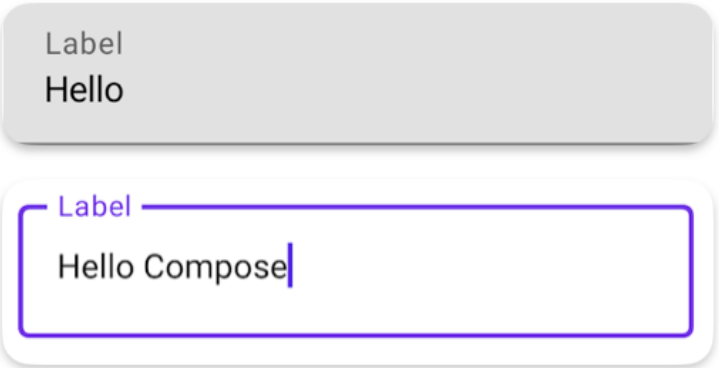
A Text composable displays text.

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

# Graphical Output

A Textfield composable displays text with an optional label.

```
TextField(
    value = "Hello"
    label = { Text("Label") }
)

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



# Graphical Output

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))
  )
}
```

# Graphical Output

## 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.

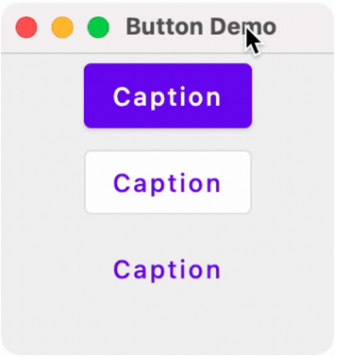
An event is simply a message. Common events include:

- **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.

# Graphical Output

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.

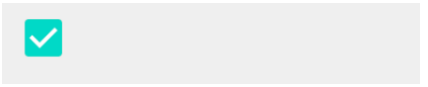



```
Column( modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally)
) {
  Button(onClick = { println("Button") }) {
    Text("Caption")
  }
  OutlinedButton(onClick = { println("OutlinedButton") }) {
    Text("Caption")
  }
  TextButton(onClick = { println("TextButton")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

# Graphical Output

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
  )
}
```

# Graphical Output

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())
  }
}
```

# Graphical Output

## Component 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.

# Graphical Output

## 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).

# Graphical Output

## 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.

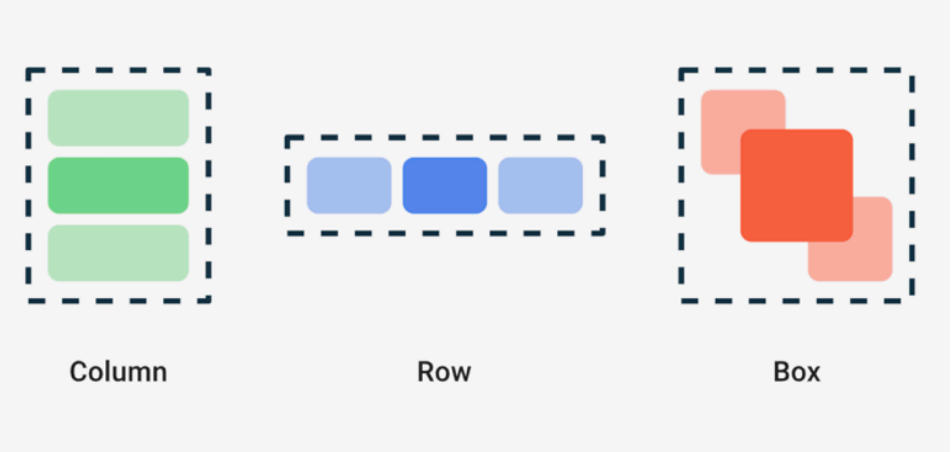



Figure 13: See the Compose Documentation for more details.

# Graphical Output

## 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
  }
}
```

# Graphical Output

## 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.

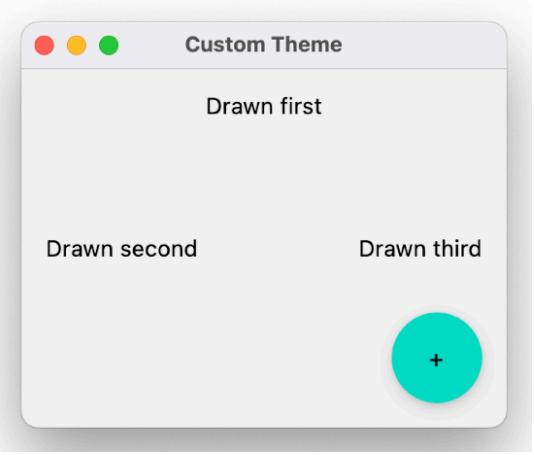
**Alignment**: Orthogonal to the arrangement.

# Graphical Output

## 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("+")
      }
  }
}
```



# Graphical Output

## 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")
    }
}
```

# Application State

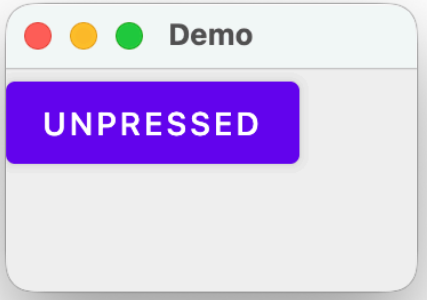
## Composable State

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.

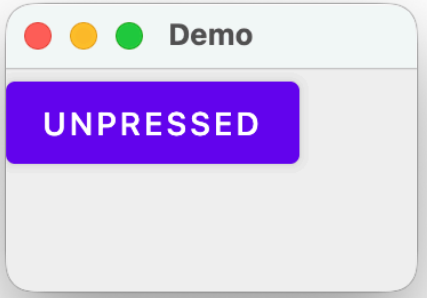
```
fun main() = application {
  Window(title = "Demo", onCloseRequest = ::exitApplication) {
    ToggleButton("Unpressed")
  }
}

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



# Application State

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)
  }
}
```



# Application State

## 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.

# Application State

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.

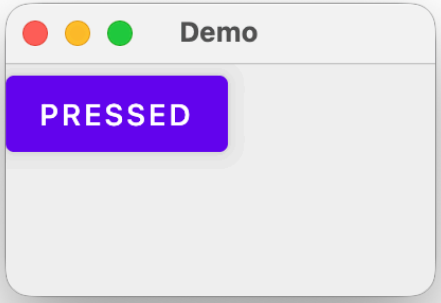
All we have to do is

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

# Application State

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)
  }
}
```



# Application State

## State Hoisting

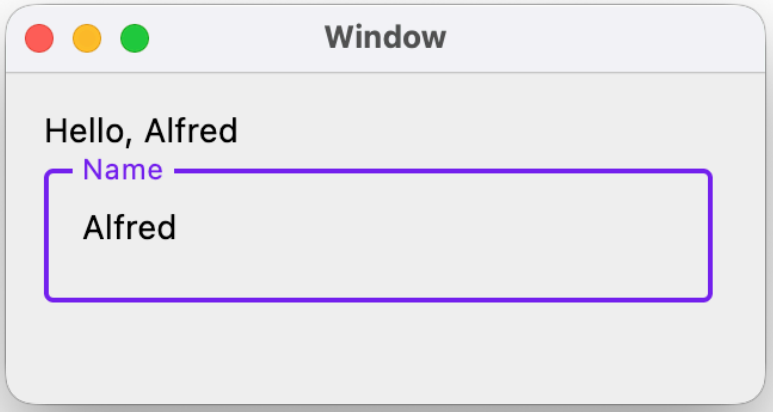
Storing state in a function can make it difficult to test and reuse. It's sometimes helpful to pull state out of a function into a higher-level, calling function. This process is called `state hoisting`.

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.




,

# Application State

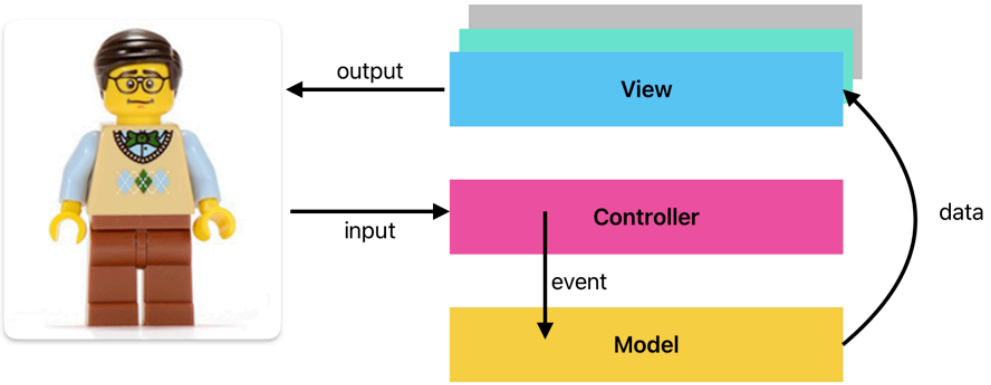
```
fun main() = application {
    Window( title = "Window", onCloseRequest = ::exitApplication ) {
        HelloScreen()
    }
}

@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") })
    }
}
```

# Model-View-Controller

It's impossible to talk about building interactive systems without acknowledging the Model-View-Controller (MVC) pattern.

MVC is a user-interface specific pattern, which divides your program logic across three classes.

- The `model` handles all application state (single "source of truth").
- The `controller` accepts and handles input, sending updates to the model.
- The model publishes changes to the `view`, to be reflected as output.



# Model-View-Controller

MVC is typically implemented with a subscriber mechanism, like the Observer pattern [4].

- Subscriber/Publisher are interfaces.
- The `View` (Subscriber) registers with the `Model` (Publisher) when the application launches.
- The `Controller` accepts user input and updates the `Model`.
- The `Model` (Publisher) notifies all Subscribers of the state change.
- Each Subscriber decides how to act on the information, often by fetching updated data from the Publisher.

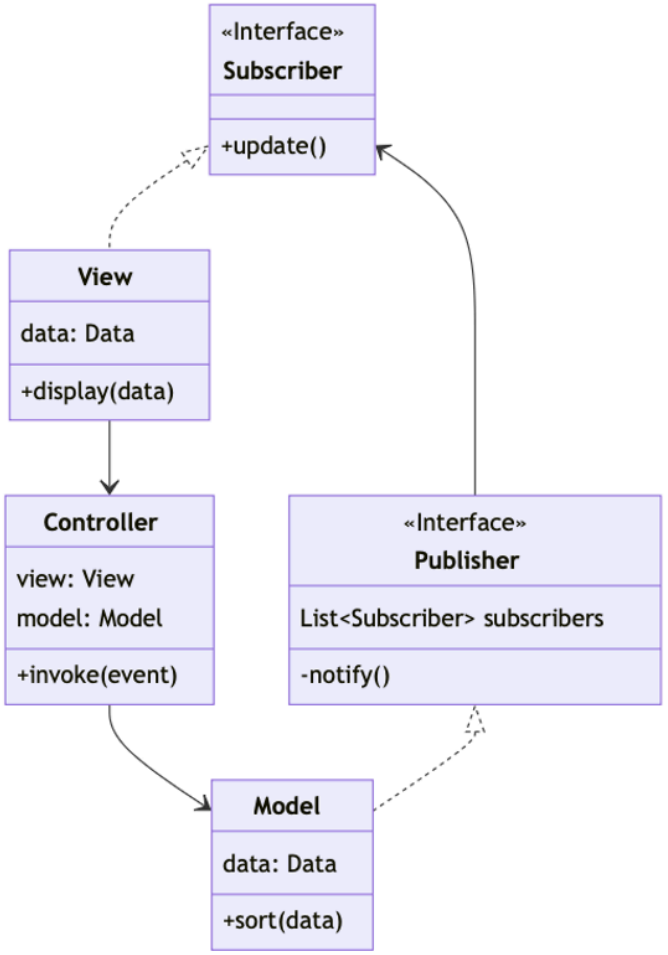
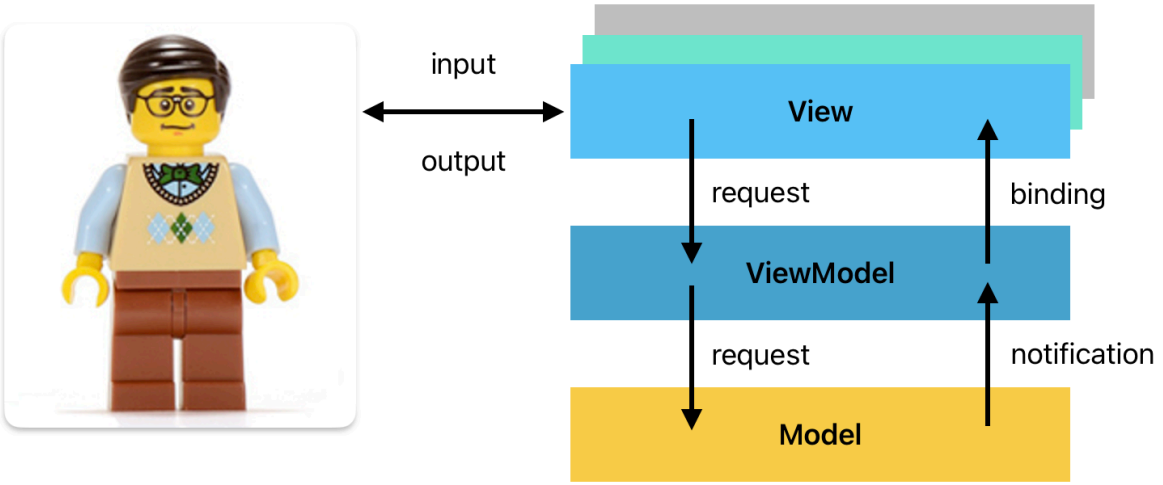



Figure 15: Subscribers register themselves with a Publisher, which agrees to send them updates when data changes.

# Model-View-ViewModel

Model-View-ViewModel (MVVM) was invented by Ken Cooper and Ted Peters in 2005. It was intended to simplify event-driven programming and user interfaces in C-sharp and .NET.

`MVVM` adds a `ViewModel (VM)` that sits between the `View` and `Model`. The VM manages all data requests to-and-from the views.



# Model-View-ViewModel

If you compare MVC to MVVM, there are two significant differences:

1. MVVM has a `ViewModel` class, which stores view-specific state information e.g., currency stored in USD but displayed in a different format for that view.

2. In MVC, the `View` is responsible for “pulling” changes from the model. MVVM instead pushes data from the `Model` to the `ViewModel` to the `View`. In the case of Compose, we use a binding mechanism (i.e. `MutableState` objects) which handle this notification for us.

# Model-View-ViewModel

The recommended user-interface implementation involves:

- A Model class to handle all application state. It's part of the Domain layer, and responsible for managing requests to the Data layer.
- One or more screens, where each screen consists of a `View` class and a corresponding `ViewModel`. The `ViewModel` contains the `MutableState` objects that the View uses to populate the user interface.
- The Observer pattern can be used to push updates from the `Model` to `ViewModel`.

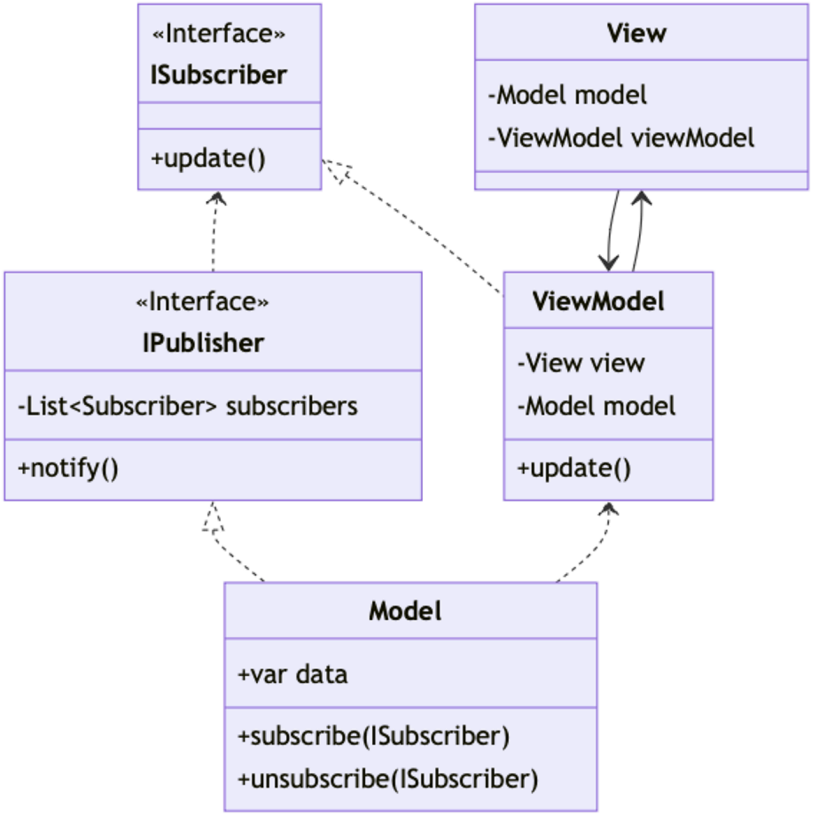



Figure 16: With MVVM, the ViewModel is the Subscriber, and coordinates updating the Views.

# Model-View-ViewModel

A simple example of a ViewModel and it's View class.

`Courses: ViewModel`

```
class CoursesViewModel(private val model: Model) : Subscriber {
    var courses by mutableStateOf(listOf<Course>())
    var sections by mutableStateOf(listOf<Section>())

    init {
        model.add(this) // register subscriber
    }

    // notification from Model about new data
    // updating `courses` or `sections` will trigger recomposition
    override fun notify(message: Message) {
        when (message.event) {
            Event.COURSES_UPDATE → courses = model.courses
            Event.SECTIONS_UPDATE → sections = model.sections
        }
    }
```

# Model-View-ViewModel

Courses: View

```
@Composable
fun CoursesView(
    settings: Settings,
    viewModel: CoursesViewModel,
    onSettings: () → Unit,
) {
    // we rely on viewModel for state
    val courses = viewModel.courses

    if (courses.isNotEmpty()) {
        Column {
            Text("${courses.first().subject}")
        }
    //  ...
```

# 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.

Navigation refers to the ways that users move around your app. Users interact with UI elements, usually by tapping or clicking on them, and the app responds by displaying new content.

We typically want to support

- Advancing to a new screen based on user actions.
- Moving to a previously viewed screen by using a back gesture or back button.
- Jumping around between arbitrary screens based on user input e.g., moving from a list of items to a detail screen.
- Saving our navigation history so that the user can return to a session and still use navigation properly.

# Navigation

A convenient way of modeling this behavior is as a stack, representing the user's navigation history.

As the user navigates forward to new content, it is pushed on top of the stack. When they go back from that content, it is popped off the stack and the previous content is displayed. This stack is usually referred to as the back stack because it represents the content that the user can go back to.

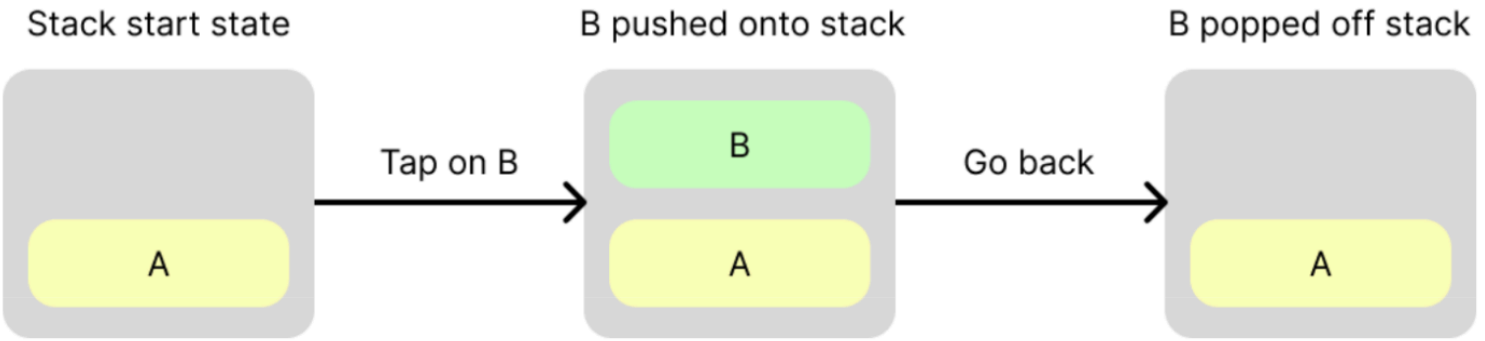



Figure 17: Diagram showing how the back-stack changes as a user navigates. Courtesy of Android developer documentation

# Navigation

## Option 1: Simple Navigation

What's the simplest way to move between screens?

The easiest solution:

- Create your screens as `Composables`. Each screen would be a View with a top-level `Composable` function.
- Have a `currentScreen` state variable to indicate which is the active screen.
- Based on user input, change the `currentScreen` to the desired screen. This will trigger recomposition and force the new screen to be drawn.
- You would want to use `state-hoisting` to keep the UI state outside of each composable.

Considerations

- This is very simple to implement.
- You would need to implement the back-stack navigation manually, including saving and restoring that state when the application launches.
- It doesn't work with OS features e.g., we don't get animations.

# Navigation

```
@Composable
fun ScreenA(onButtonPress: () → Unit) {
  Box(Modifier.fillMaxSize().background(Color.Yellow)) {
    Text("Screen A",
      fontSize = 24.sp,
      modifier = Modifier.align(Alignment.TopCenter)
    )
    Text(...)
    Button(
      modifier = Modifier.align(Alignment.BottomCenter),
      onClick = onButtonPress
    ) {
        Text("Change Screens")
    }
}

@Composable
fun ScreenB(onButtonPress: () → Unit) {
  Box(Modifier.fillMaxSize().background(Color.Green)) { ... }
}
```

# 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 shuffling composables isn't sufficient, we have richer libraries.

When 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 detail view (list of customers) to a summary (chart).
- You want to take advantage of system capabilities e.g., swipe to go back/forward through screens.
- You want animations, sound effects etc.

We have Navigation libraries that provide more advanced capabilities:

- Navigation 3 for KMP[1].
- Voyager for Compose & KMP.

[1] Make sure to use the latest version of Nav3. Earlier versions didn't work with KMP or desktop

# Navigation

## Navigation 3 (Nav3)

Navigation3 (aka `Nav3`) is Google's latest navigation framework. It's a KMP library, meaning it's designed to be used across Android, iOS and desktop targets and can easily be added as a dependency.

Nav3 works by associating a unique identifier to each screen. Your back stack is then a Mutable collection of keys, reflecting your visited screens. You are provided with a special view container that will display the current screen based on your backstack.

Here's the high-level steps to add Nav3 to your application:

- Define the content that users can navigate to, each with a unique key.
- Create a back stack to store the keys for visited screens.
- Use a NavDisplay composable to display your app's back stack. Whenever the back stack changes, it updates the UI to display relevant content.

# Navigation

## Step 1: Create keys

```
// Define keys that will identify content
data object ProductList
data class ProductDetail(val id: String)

@Composable
fun MyApp() {

    // Create a back stack, specifying the starting screen
    val backStack =
        remember { mutableStateListOf<Any>(ProductList) }

    // Supply your back stack to a NavDisplay (next slide)

    // Push a key, the navigation library will switch to it
    backStack.add(ProductDetail(id = "ABC"))

    // Pop a key, the navigation library will reflect that
    backStack.removeLastOrNull()
}
```

# Navigation

### Step 2: Resolve keys to content

Content is modeled in Navigation 3 using NavEntry, which is a class containing a composable function. It represents a destination - a single piece of content that the user can navigate forward to and back from.

To convert a key to a NavEntry, create an Entry Provider. This is a function that accepts a key and returns a NavEntry for that key. It is usually defined as a lambda parameter when creating a NavDisplay.

```
entryProvider = { key →
    when (key) {
        is ProductList → NavEntry(key) { Text("Product List") }
        is ProductDetail → NavEntry(key) { Text("${key.id} ") }
        else → {
            NavEntry(Unit) { Text(text = "Invalid Key: $it") }
        }
    }
}
```

# Navigation

### Step 3: Display the back stack

In Navigation 3, a NavDisplay observes your back stack and updates its UI accordingly.

Construct it with following parameters:

- Your back stack - this should be of type SnapshotStateList, where T is the type of your back stack keys. It is an observable List so that it triggers recomposition of NavDisplay when it changes.
- An entryProvider to convert the keys in your back stack to NavEntry objects.
- Optionally, supply a lambda to the onBack parameter. This is called when the user triggers a back event.

# Navigation

```
data object Home
data class Product(val id: String)

@Composable
fun NavExample() {

    val backStack = remember { mutableStateListOf<Any>(Home) }

    NavDisplay(
        backStack = backStack,
        onBack = { backStack.removeLastOrNull() },
        entryProvider = { key →
            when (key) {
                is Home → NavEntry(key) {
                    ContentGreen("Welcome to Nav3") {
                        Button(onClick = {
                            backStack.add(Product("123"))
                        }) {
                            Text("Click to navigate")
                        }
```

```
                    }
                }

                is Product -> NavEntry(key) {
                    ContentBlue("Product ${key.id} ")
                }

                else -> NavEntry(Unit) { Text("Unknown route") }
            }
        }
    )
}
```

> **Tip**
>
> See the nav3-recipes page for many, many examples of how to use navigation effectively.

# 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

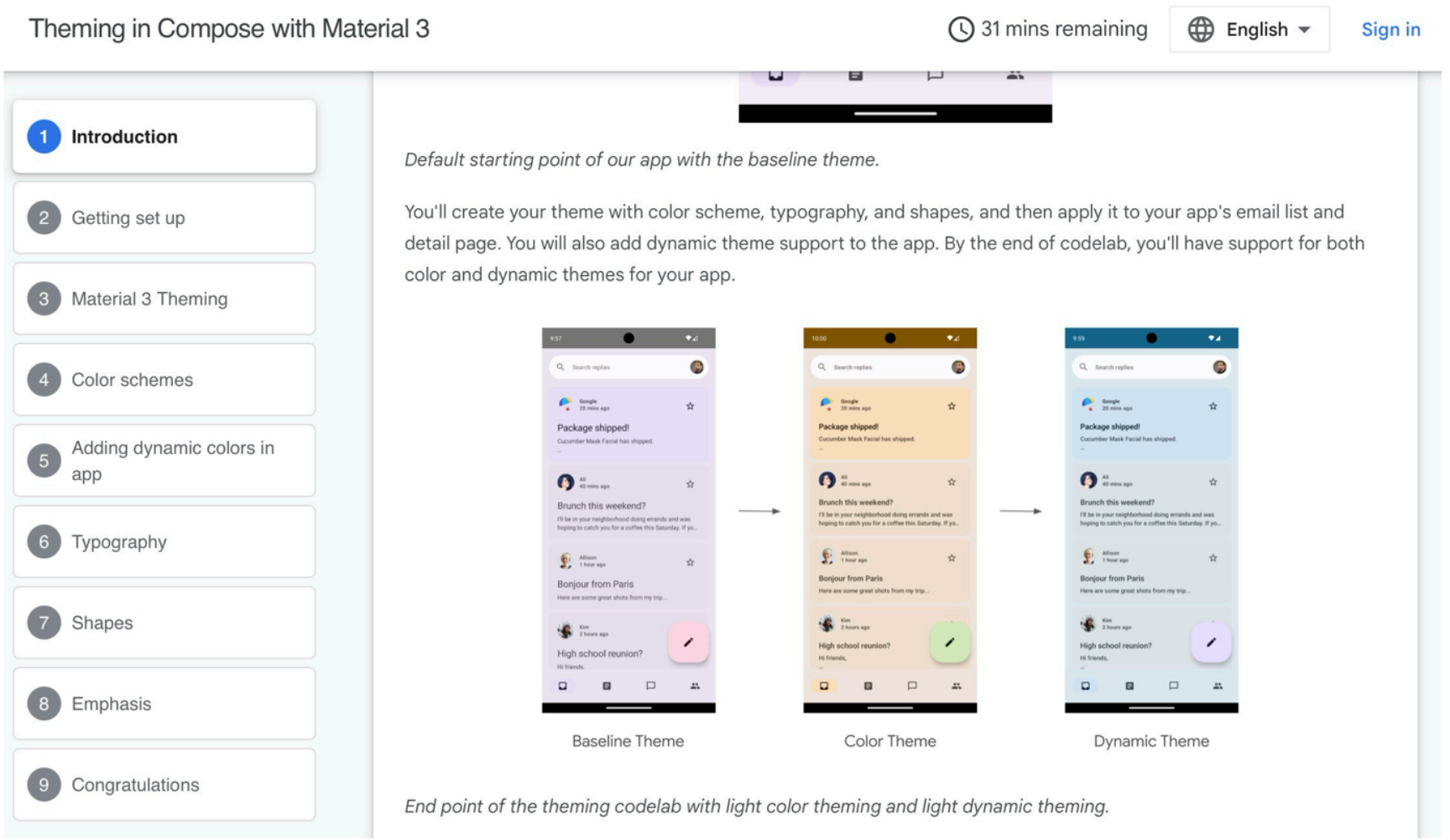



Figure 18: See the [Theming in Compose with Material 3] tutorial for examples of how to customize your application.

# Bibliography


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

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

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

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison-Wesley Professional, 1994.