Mobile Application Development
Week 2: Managing Data and State

Along with being declarative Jetpack Compose is also data-driven. Compose handles creating the UI and keeps it up to date as the data, or state, changes.

State
Compose ensures that the user interface always reflects the latest data by using state.
State is any data used in an app can change over time. This sounds a lot like a variable but differs from a standard variable in two important ways:

1. A standard variable defined in a function would be re-initialized each time that function is called. The value assigned to a state variable in a composable function needs to be remembered. Each time a composable function containing state is called it must remember the state values from the last time it was invoked.
2. Any time a state value changes all composable functions that use that state value are executed and the UI is updated to reflect the new value.

Composition
Composition is a description of the UI built by Jetpack Compose when it executes composables.
Initial composition is the first time a composable is run and created.

Recomposition
Recomposition is when Compose re-executes composables that may have changed in response to data changes and then updates the Composition, or UI, to reflect those changes. Composables get recomposed when there are changes to state values the composable relies on such as state values passed to the composable parameters. This ensures that the UI is always showing up to date data.

Since Jetpack Compose is declarative a composable can only be updated when the state values it relies on change. Any time a state is updated all composables that use that state will be recomposed.
So composables have a live connection to any state it relies on so any changes to the UI state are immediately reflected in the UI.

State is updated in response to events. Events are inputs generated from outside or inside an application, such as:

- The user interacting with the UI by, for example, pressing a button.
- Other factors, such as sensors sending a new value, or network responses.

While the state of the app offers a description of what to display in the UI, events are the mechanism through which the state changes, resulting in changes to the UI.

- Event - An event is generated by the user or another part of the program.
- Update State - An event handler changes the state that is used by the UI.
- Display State – Composables are recomposed and the UI is updated to display the new state.

Recomposing the entire composable tree for a user interface each time a state value changes might sound like a highly inefficient approach to rendering and updating a user interface. Compose avoids this overhead using a technique called *intelligent recomposition* that involves only recomposing those composable functions directly affected by the state change. In other words, only composable functions that read the state value will be recomposed when the value changes.

Declaring State

Because Compose will call your entire composable function from the ground up every time it recomposes your UI, any local variables you declare inside a composable function will be lost between compositions. If you tried to store data inside a composable, it would reset each time the composable is invoked – which is not an ideal mechanism for storing UI state. And global variables should always be avoided if possible.

To address this issue, we use a function called `remember` to store an object in memory. `remember` takes in a lambda expression as its argument. A value computed by `remember` is stored in the Composition during initial composition. On subsequent compositions, `remember` immediately returns the value from the previous composition. So `remember` guards against recomposition resetting the value for that variable so the state data is preserved across compositions.

Composables can have any number of remembered values. Also, Compose keeps track of which instances of a composable have remembered which values. If you have several instances of the same composable, they will each remember their own values. If a value can be derived from the composable's inputs you don't need to store it as state, you can just derive it.

Mutable State

Along with the ability to remember data between compositions, the data also needs to be observable so Compose is notified when state changes.

The `State` and `MutableState` types in Compose can be used to make state in your app observable, or tracked, by Compose. The `State` type is immutable, so you can only read the value in it, while the type `MutableState` is mutable so that is what is used for state in Compose.

Whenever the value inside a `MutableState` object is changed, Compose is immediately notified of the change. Every composable that accesses the state object will then automatically recompose and the UI will be updated with the new value held in the state object.

You use the `mutableStateOf` function, which requires an initial value, to create an observable `MutableState` object. You must call `mutableStateOf` inside a composable by wrapping it in a `remember` block.

The value returned by the `mutableStateOf()` function:

- Holds state
- Is mutable, so the value can be changed.
- Is observable, so Compose observes any changes to the value and triggers a recomposition to update the UI.

```
val celsius = remember {mutableStateOf(0)}
```

This creates an object of type `MutableState`, sets its value to 0 and the `remember` function stores the value of celsius in the composable's memory during the initial composition.

You set a MutableState object's value by updating its value property.
```
celsius.value = 20
```

You can also use the Kotlin property delegate `by` keyword instead of the =. This saves you from typing `.value` every time.

```
val celsius by remember {mutableStateOf(0)}
celsius = 20
```

Each time the value of a state variable is changed any composables that use that state value are recomposed, keeping the UI up to date.
Compose is aware of changes to your UI state – but only when your state object itself is reassigned. So UI state classes should only contain `val` properties and make copies using "copy".

Recomposition in Compose only works with observable types. There are other observable types supported by Compose that we'll look at later in the semester.

State hoisting
Composable functions are usually categorized in terms of state handling in two categories.
Stateful

- A stateful composable owns state that can change over time
- The caller composable, or parent composable, doesn't manage the state

Stateless

- A stateless composable is a composable that doesn't own any state
- Instead of holding state, a stateless composable has parameters so the state values it needs are passed into the composable function
- The caller composable, or parent composable, passes the state value, and often a callback function that is triggered by an event to update the state value.
- An easy way to achieve stateless is by using state hoisting.

State hoisting in Compose is a pattern of moving state to a composable's caller to make a composable stateless. The general pattern for state hoisting in Jetpack Compose is to replace the state variable with two parameters:

- **value: T** - the current value to display
- **onValueChange: (T) -> Unit** - an event that requests the value to change, where T is the proposed new value where this value represents any state that could be modified. This callback function is usually a lambda expression

When the composable is called by the parent, the parent passes in both of these as arguments.

State hoisting has many benefits:

- Single source of truth

    - By moving state to the parent composable, instead of duplicating it, we're ensuring there's only one source of truth. The source of truth belongs to the composable where the state is created and controlled. This helps avoid bugs.
- Encapsulated
    - Only stateful composables will be able to modify their state. This limits the number of composables that handle the state.
- Reusability
    - Hoisted state can be shared with multiple composables.
    - Composables that render data that's received as input are easier to reuse since you can easily pass it different data.
- Interceptable
    - Callers to the stateless composables can decide to ignore or modify events before changing the state.
- Decoupled
    - The state for the stateless composables may be stored anywhere.
    - Moving where state is stored doesn't affect the UI

When deciding where to define state, there are three rules to help you figure out where state should go:

1. State should be hoisted to at *least* the lowest common parent of all composables that use the state (read).
2. State should be hoisted to at *least* the highest level it may be changed (write).
3. If two states change in response to the same events they should be hoisted together.

The pattern where the state goes down the component hierarchy, and events go up is called a *unidirectional data flow*. By following unidirectional data flow, you can decouple composables that display state in the UI from the parts of your app that store and change state.

We'll talk about other approaches to managing state later in the semester.

Built-in Composables
https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#top-level-functions

Let's look at two other built-in composables.

TextField
https://material.io/components/text-fields
Text fields let users enter and edit text.
The TextField composable displays a text field for user input.

- The value parameter is the text string to be shown in the text field
    - mandatory parameter
- The onValueChange parameter takes the callback that is triggered when the text field value changes
    - mandatory parameter
- The label parameter is used to display text in the text field (optional)
- The placeholder parameter is used to display text in the text field when it's in focus and the input text is empty

The TextField doesn't update itself—it only updates when its value parameter changes.

Button
https://material.io/components/buttons
Buttons add interactivity as they enable users to trigger an action when the button is tapped

- The content parameter displays the content inside the button
  - Text or Image composables are often used
  - mandatory parameter
- The onClick parameter takes a callback function that is called when the button is clicked and the click event is fired
  - mandatory parameter

HelloAndroid (state)
Let's update our app to add some interactivity and handle state.

Button
Create a new custom composable function for our button and its functionality.
We will need a string resource for the text on the button so add one in strings.xml.

```
<string name="buttonHello">Say hi</string>
```

The Button composable has an onClick parameter that takes a callback function, which is a lambda function in Kotlin, that will be called when the click event fires. We make that a parameter of our custom composable function and then assign it to the onClick parameter.

```
@Composable
fun SayHi(clicked: () -> Unit){
    Button(onClick= clicked) {
        Text(stringResource(id = R.string.buttonHello))
    }
}
```

Notice that we're using the Text composable for the text on the button.

In order to add the button to the UI you have to call sayHi() from Greeting(). We want the button at the top so add it as the first item in the column. For now we'll pass in an empty lambda function as the argument.

```
SayHi ({})
```

When the button is clicked we want it to update the text in the Text composable. We need to create a state variable to store the name. Remove the name parameter from Greeting() and the calls to Greeting() in onCreate() and DefaultPreview().

```
@Composable
fun Greeting() {
    val name = remember { mutableStateOf("") }
…
```

Now we need to create a separate custom composable function for our Text composable that has a parameter for the name.

```
@Composable
fun MessageText(newName: String){
    if(newName.isNotEmpty()) {
        Text(
            stringResource(R.string.greeting) + " " + newName,
            color = Color.Red,
            fontSize = 24.sp,
            textAlign = TextAlign.Center
        )
    }
}
```

Move the Text composable out of Greeting and instead call your new function, passing in the value property of name.

```
MessageText(newName = name.value)
```

Update the call to SayHi() to pass a lambda function that will be executed when the button is clicked.

```
SayHi ({ name.value = "Scotty"})
```

Now when you run the app MainActivity launches, its onCreate() method runs, setContent() is called which calls Greeting().

1. Greeting() creates a MutableState<String> state variable called name, sets its value to an empty string, and stores it in memory.
2. Greeting() calls SayHi() and a lambda function is passed that assigns the String "Scotty" to the value of our name state variable. SayHi() calls the Button composable using buttonHello as the string resource for its content. Button is composed and added to the UI.
3. Greeting() then calls Image() and it's composed and added to the UI.
4. Greeting() then calls MessageText() and the name state variable value is passed into the newname parameter. The Text composable is composed for the first time using the greeting string resource and since name.value is an empty string you will just see Hello.
5. When the user taps the button the click event fires, which will update the value of our name state variable to Scotty.
6. Since MessageText() takes the name state variable as a parameter the composable is automatically recomposed and the Text composable will show Hello Scotty.

TextField

Let's add a text field so the user can enter their name and we can use their name in the message when the button is tapped.

We will need a string resource for the text field label so add one in strings.xml.

```
<string name="enterName">Name</string>
```

The TextField composable has two required parameters: a String for the TextField's value, and a lambda for the changed parameter that specifies what should happen when the user enters a new value. This is needed because the TextField doesn't update itself as the user is entering text. It only updates when its value parameter changes. So the lambda passed to the changed parameter usually updates the value parameter with what the user is entering.

We'll need a state variable to remember the value in the text field.

```
val textFieldName = remember { mutableStateOf("") }
```

Then we create a custom composable function for our TextField.

```
@Composable
fun NameTextField(name: String, changed: (String) ->Unit){
    TextField(
        value = name,
        label = {Text(stringResource(id = R.string.enterName))},
        onValueChange = changed,
        modifier = Modifier
            .fillMaxWidth()
            .padding(top = 10.dp, bottom = 10.dp)
    )
}
```

Notice that we're using the Text composable for the label on the text field.

We want this to be above the button so call it as the first item in the column in Greeting.

```
NameTextField(name = textFieldName.value, changed = {
textFieldName.value = it })
```

The Kotlin keyword *it* refers to the current value of the text field.

And now that the user is entering their own name, let's update the lambda function we pass to SayHi() so the name entered in the text field is used instead of Scotty.

```
SayHi({ name.value = textFieldName.value })
```

Now when you run the app here's what we've added.

1. Greeting() creates a MutableState<String> state variable called textFieldName, sets its value to an empty string, and stores it in memory.
2. Greeting() calls NameTextField() passing the value of textFieldName into the name parameter and a lambda function for the changed parameter. The lambda function assigns the current value of the textfield, accessed by the Kotlin keyword *it*, to the textFieldName state variable.
3. So each time the user types a character into the text field the changed event fires and the lambda function is run, updating the textFieldName state value. Since textFieldName is passed into the NameTextField() function, every time the value of textFieldName is changed, the NameTextField composable is recomposed and the UI is updated. It's this process that updates the text field as the user types into it.
4. When the user taps the button the click event fires, which will update the value of our name state variable to the value of textFieldName. The MessageText() composable is automatically recomposed and the Text composable will show Hello and the name the user entered.

State variables can also be used as logic in statements to change when composables are displayed. So if we don't want any text written to our Text composable if no name has been entered, in MessageText() we can wrap the Text composable in an if statement.

```
if(newName.isNotEmpty()) {
    Text(
        ...
```

```
    )
}
```

Lastly, lets change our state variables to use the Kotlin property delegate by keyword as that's the most common syntax you'll see. This also allows us to define them using `var` instead of `val`.

```
var name by remember { mutableStateOf("") }
var textFieldName by remember { mutableStateOf("") }
```

This will also require imports for getValue and setValue.

```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
```

Once we do that we can access the state variables value without explicitly referring to the MutableState's value property every time.

Layout

We can add a modifier to make the text field the full width of the screen and add some padding above and below it.

```
TextField(
    value = name,
    label = {Text(stringResource(id = R.string.enterName))},
    onValueChange = changed,
    modifier = Modifier
        .fillMaxWidth()
        .padding(top = 10.dp, bottom = 10.dp)
)
```

Let's center everything in our column horizontally.

```
Column(horizontalAlignment = Alignment.CenterHorizontally)
```

This makes the alignment we specified for the image redundant and can be removed.