Interact with UI and State (Building a Tip Calculator).

▼ Intro to State in Compose.

At its core, state in an app is any value that can change over time. This includes everything from a database to a variable in your app.

All Android apps display state to the user. A few examples include:

- A message that shows when a network connection can't be established.
- Forms, such as registration forms. The state can be filled and submitted.
- Tappable controls, such as buttons. The state could be *not tapped*, *being tapped* (display animation), or *tapped*(an `onClick` action).

Styles and themes are a collection of attributes that specifies the appearance for a single UI element. A style can specify attributes such as font color, font size, background color, and much more which can be applied for the entire app.

The `TextField` composable function lets the user enter text in an app.

```
@Composable
fun EditNumberField(
    @StringRes label: Int,
    @DrawableRes leadingIcon: Int,
    keyboardOptions: KeyboardOptions,
    value: String,
    onValueChanged: (String) -> Unit,
    modifier: Modifier = Modifier
) {
    TextField(
        value = value,
        singleLine = true,
        leadingIcon = { Icon(painter = painterResource(id = l
        modifier = modifier,
        onValueChange = onValueChanged,
        label = { Text(stringResource(label)) },
        keyboardOptions = keyboardOptions
    )
}
```

The Composition.

Compose is a declarative UI framework, meaning that you declare how the UI should look in your code.

The *Composition* is a description of the UI built by Compose when it executes composables. Compose apps call composable functions to transform data into UI. If a state change happens, Compose re-executes the affected composable functions with the new state, which creates an updated UI—this is called *recomposition*. Compose schedules a *recomposition* for you.

You use the `State` and `MutableState` types in Compose 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 `MutableState` type is mutable.

You can use the `mutableStateOf()` function to create an observable `MutableState`. It receives an initial value as a parameter that's wrapped in a `State` object, which then makes its `value` observable.

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

- Holds state, which is the bill amount.
- 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.

The `onValueChange` callback is triggered when the text box's input changes. In the lambda expression, the `it` variable contains the new value. When the user enters text in the text box, the `onValueChange` callback is called and the `amountInput` variable is updated with the new value.

The `amountInput` state is tracked by Compose, so the moment that its value changes, recomposition is scheduled and the `EditNumberField()` composable function is executed again. In that composable function, the `amountInput` variable is reset to its initial `0` value. Thus, the text box shows a `0` value.

However, you need a way to preserve the value of the `amountInput` variable across recompositions so that it's not reset to a `0` value each time that the `EditNumberField()` function recomposes.

Use Remember Function to Save State.

Composable methods can be called many times because of recomposition. The composable resets its state during recomposition if it's not saved. Composable functions can store an object across recompositions with `remember`.

A value computed by the `remember` function is stored in the Composition during initial composition and the stored value is returned during recomposition. Usually `remember` and `mutableStateOf` functions are used together in composable functions to have the state and its updates be reflected properly in the UI.

```
var amountInput by remember { mutableStateOf("") }
var tipInput by remember { mutableStateOf("") }
var roundUp by remember { mutableStateOf(false) }
```

💡 During initial composition, `value` in the `TextField` is set to the initial value, which is an empty string.

When the user enters text into the text field, the `onValueChange` lambda callback is called, the lambda executes, and the `amountInput.value` is set to the updated value entered in the text field.

The `amountInput` is the mutable state being tracked by the Compose, recomposition is scheduled. The `EditNumberField()` composable function is recomposed. Since you are using `remember` `{ },` the change survives the recomposition and that is why the state is not re-initialized to `""`.

The `value` of the text field is set to the remembered value of `amountInput`. The text field recomposes (redrawn on the screen with new value).

Modify the Appearance.

Every text box should have a label that lets users know what information they can enter.

In the `TextField()` composable function, add `singleLine` named parameter set to a `true` value. This condenses the text to a single, horizontally scrollable line from multiple lines.

Add the `keyboardOptions` parameter set to a `KeyboardOptions()`: Android provides an option to configure the keyboard displayed on the screen to enter digits, email addresses, URLs, and passwords, etc.

State Hoisting.

You should hoist the state when you need to:

- Share the state with multiple composable functions.
- Create a stateless composable that can be reused in your app.

When you extract state from a composable function, the resulting composable function is called stateless. That is, composable functions can be made stateless by extracting state from them.

💡 A *stateless* composable is a composable that doesn't have a state, meaning it doesn't hold, define, or modify a new state. On the other hand, a stateful composable is a composable that owns a piece of state that can change over time.

State hoisting is a pattern of moving state to its caller to make a component stateless. When applied to composables, this often means introducing two parameters to the composable:

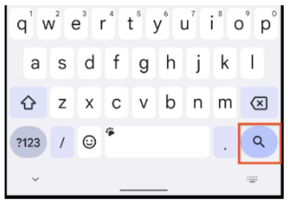
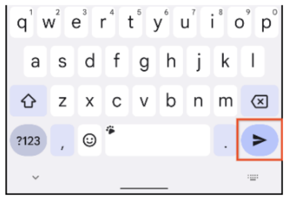
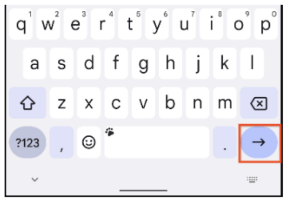
- A `value: T` parameter, which is the current value to display.
- An `onValueChange: (T) -> Unit` – callback lambda, which is triggered when the value changes so that the state can be updated elsewhere, such as when a user enters some text in the text box.

Update the `EditNumberField()` function definition, to hoist the state by adding the `value` and `onValueChange` parameters.

▼ Calculate a Custom Tip.

Set an Action Button.

We explore how to set the keyboard action button with `KeyboardOptions`. A keyboard action button is a button at the end of the keyboard.

Property	Action button on the keyboard
`ImeAction.Search` Used when the user wants to execute a search.	
`ImeAction.Send` Used when the user wants to send the text in the input field.	
`ImeAction.Go` Used when the user wants to navigate to the target of the text in the input.	

We set 2 different action buttons for the text boxes:

- A **Next** action button for the **Bill Amount** text box, which indicates that the user is done with the current input and wants to move to the next text box.
- A **Done** action button for the **Tip Percentage** text box, which indicates that the user finished providing input.

Add Support for Landscape Orientation.

To resolve this you will need a vertical scrollbar, that helps you scroll your app screen. Add `.verticalScroll(rememberScrollState())` to the modifier to enable the column to scroll vertically. The `rememberScrollState()` creates and automatically remembers the scroll state.

```
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.verticalScroll

@Composable
fun TipTimeLayout() {
    var amountInput by remember { mutableStateOf("") }
    var tipInput by remember { mutableStateOf("") }
    var roundUp by remember { mutableStateOf(false) }

    val amount = amountInput.toDoubleOrNull() ?: 0.0
    val tipPercent = tipInput.toDoubleOrNull() ?: 0.0
    val tip = calculateTip(amount, tipPercent, roundUp)

    Column(
        modifier = Modifier
            .statusBarsPadding()
            .padding(horizontal = 40.dp)
            .verticalScroll(rememberScrollState())
            .safeDrawingPadding(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) { ... }
```

▼ Write Automated Tests.

Automated Tests.

Testing is a structured method for checking your software to make sure it works as expected. Automated testing is code that checks to ensure that another piece of code that you wrote works correctly. Testing also provides a way to continuously check the existing code as changes are introduced. Creating a test every time you create a new feature in your app reduces your workload later as your app grows.

Automated tests are tests executed through software, as opposed to manual tests, which are carried out by a person who directly interacts with a device. Automated testing and manual testing play a critical role in ensuring that users of your product have a pleasant experience. However, automated tests can be more precise and they can be executed much faster than a manual test.

Types of Automated Tests.

1. *Local tests.*

Local tests directly test a small piece of code to ensure that it functions properly. With local tests, you can test functions, classes, and properties.

Local tests are executed on your workstation, which means they run in a development environment without the need for a device or emulator, i.e., they run on your computer. They also have very low overhead for computer resources, so they can run fast even with limited resources.

2. *Instrumentation tests.*

It's a UI test. It lets you test parts of an app that depend on the Android API, and its platform APIs and services. Unlike local tests, UI tests launch an app or part of an app, simulate user interactions, and check whether the app reacted appropriately. UI tests are run on a physics device or emulator.

When you run an instrumentation test on Android, the test code is actually built into its own Android Application Package (APK) like a regular Android app. An APK is a compressed file that contains all the code and necessary files to run the app on a device or emulator. The test APK is installed on the device or emulator along with the regular app APK. The test APK then runs its tests against the app APK.

Local tests directly test methods from the app code, so the methods to be tested must be available to the testing classes and methods.