Add a Button to an App.

▼ Dice Roller App.

```
// Package declaration.
package com.example.diceroller

// Import necessary libraries and components from Android and
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.Image
import androidx.compose.foundation.layout.*
import androidx.compose.material3.Button
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.painterResource
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.example.diceroller.ui.theme.DiceRollerTheme

// MainActivity class which is the entry point of the app, in
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // Set the content of the UI.
        setContent {
            // Apply the custom Material Theme.
```

```
            DiceRollerTheme {
                // Surface container that fills the entire sc
                Surface(
                    modifier = Modifier.fillMaxSize(), // Fil
                    color = MaterialTheme.colorScheme.backgro
                ) {
                    // Call the DiceRollerApp composable func
                    DiceRollerApp()
                }
            }
        }
    }
}

// Preview annotation for Android Studio to render a preview
@Preview
@Composable
fun DiceRollerApp() {
    // DiceWithButtonAndImage composable function with center
    DiceWithButtonAndImage(
        modifier = Modifier
        .fillMaxSize() // Fills the maximum size of the paren
        .wrapContentSize(Alignment.Center) // Centers the con
    )
}

// Composable function to display a dice image and a button t
@Composable
fun DiceWithButtonAndImage(modifier: Modifier = Modifier) {
    // Remember a mutable state for the result of the dice ro
    var result by remember { mutableStateOf(1) } // Initial s

    // Choose the dice image based on the current result.
    val imageResource = when(result) {
        1 -> R.drawable.dice_1
        2 -> R.drawable.dice_2
```

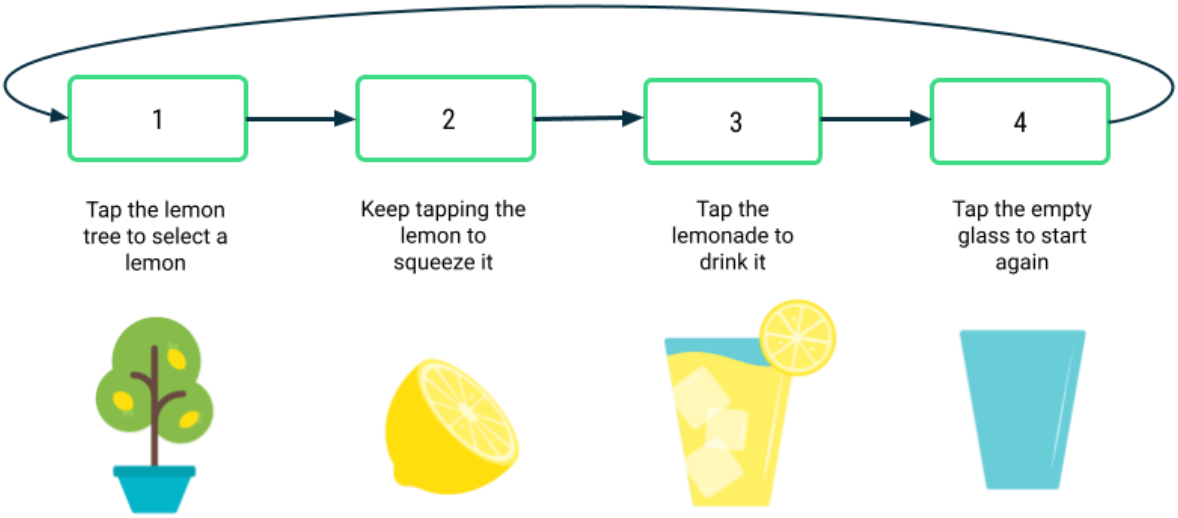
```
            3 -> R.drawable.dice_3
            4 -> R.drawable.dice_4
            5 -> R.drawable.dice_5
            else -> R.drawable.dice_6 // Default case for any oth
        }

        // Column layout to arrange the image and button vertical
        Column(modifier = modifier, horizontalAlignment = Alignme
            // Display the dice image.
            Image(painter = painterResource(id = imageResource),

            // Button to roll the dice.
            Button(
                // Set result to a random number between 1 and 6
                onClick = { result = (1..6).random() },
            ) {
                // Text inside the button.
                // Text to display and its font size.
                Text(text = stringResource(R.string.roll), fontSi
            }
        }
    }
```

▼ Practice: Click Behavior.

Making something clickable: Earlier in this pathway, you learned how to make a button clickable. In the case of the Lemonade app, there's no `Button` composable. However, you can make any composable, not just buttons, clickable when you specify the `clickable` modifier on it.



You may start to notice that there's repeated code in your app for each step of making lemonade. For the `when` statement in the previous code snippet, the code for case `1` is very similar to case `2` with small differences.

If it's helpful, create a new composable function, called `LemonTextAndImage()` for example, that displays text above an image in the UI. By creating a new composable function that takes some input parameters, you have a reusable function that's useful in multiple scenarios as long as you change the inputs that you pass in.

> 💡 Here's one additional hint: You can even pass in a lambda function to a composable. Be sure to use function type notation to specify what type of function should be passed in.

In the following example, a `WelcomeScreen()` composable is defined and accepts two input parameters: a `name` string and an `onStartClicked()` function of type `() -> Unit`. That means that the function takes no inputs (the empty parentheses before the arrow) and has no return value (the `Unit` following the arrow).

Any function that matches that function type `() -> Unit` can be used to set the `onClick` handler of this `Button`. When the button is clicked the `onStartClicked()` function is called.

```
@Composable
fun WelcomeScreen(name: String, onStartClicked: () -> Unit) {
    Column {
        Text(text = "Welcome $name!")
        Button(
            onClick = onStartClicked
        ) {
            Text("Start")
        }
    }
}
```

Passing in a lambda to a composable is a useful pattern because then the `WelcomeScreen()` composable can be reused in different scenarios. The user's name and the button's `onClick` behavior can be different each time because they're passed in as arguments.