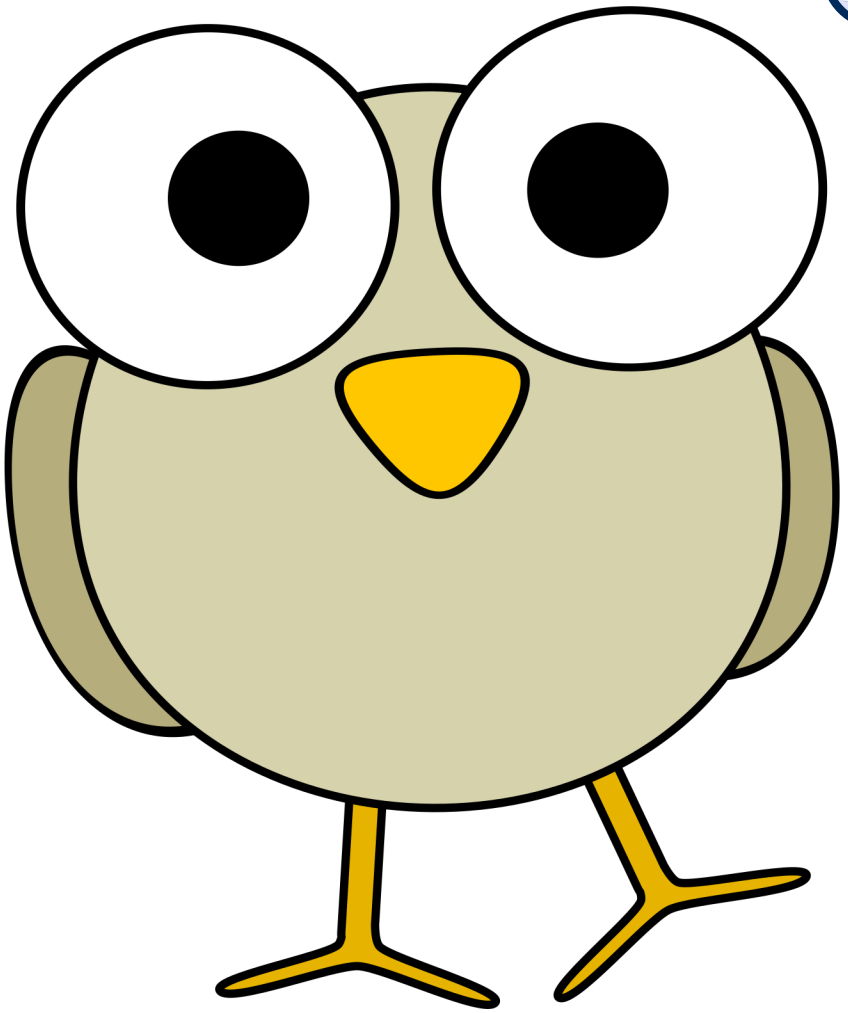
Mobile Applications Development
CSCI 448
Lecture 02

Jetpack Compose

Previously in CSCI 448

- MVVM architecture
  - Transition away from MVC
- Compose framework replacing View framework
- Kotlin replacing Java

Questions?



Learning Outcomes For Today

- Define Composables and how they fit the Composite design pattern
- Define Imperative Programming & Declarative Programming and how each contributes towards inheritance and/or composition
- Discuss the principles of Separation of Concerns and Composition over Inheritance
- Recite XML structure and its use for Android resources

On Tap For Today

- Strings
- Compose
- Practice
  - Lab01A
  - Kotlin Quiz

On Tap For Today

- Strings
- Compose
- Practice
  - Lab01A
  - Kotlin Quiz

Resources – strings.xml

- Contains key-value pairs

```xml
<resources>
    <string name="app_name">GeoQuizV1</string>
    <string name="action_settings">Settings</string>
    <string name="question_text">Constantinople is the largest city in Turkey.</string>
    <string name="true_button">True</string>
    <string name="false_button">False</string>
    <string name="correct_toast">Correct!</string>
    <string name="incorrect_toast">Incorrect!</string>
</resources>
```

- Resource files are configuration specific
  - Can have multiple strings.xml files for different languages
  - Based on device language, will load appropriate file at runtime

Design Principle #2: WORM

- Write Once Read Many Principle
  1. **Reusability**

- Applies to more than just persistent storage
  - `strings.xml` will be part of our "model"
  - Provides **Abstraction, Extendibility, Modularity**

On Tap For Today

- Strings
- Compose
- Practice
  - Lab01A
  - Kotlin Quiz

MVVM Architecture

- View Model prepares Model data for View
- V M decoupled from V
  - VM has no reference to V
  - V observes VM state

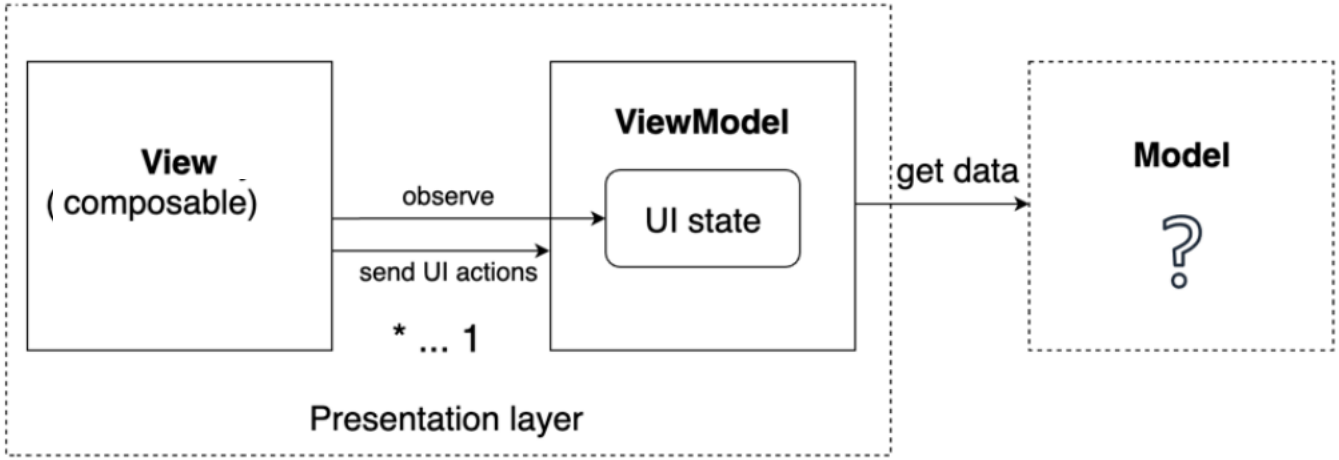
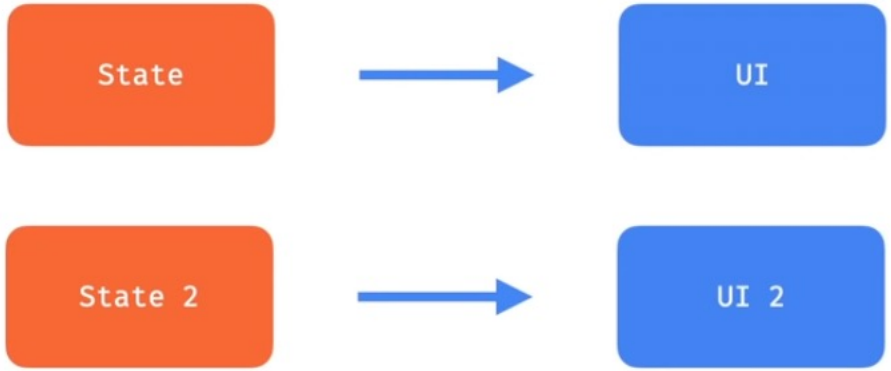


Figure 7.4 – Presentation layer in the MVVM pattern

Compose

- UI is immutable: there are no objects
  - But UI is dynamic
  - Different inputs → Different UI
- UI is **idempotent**

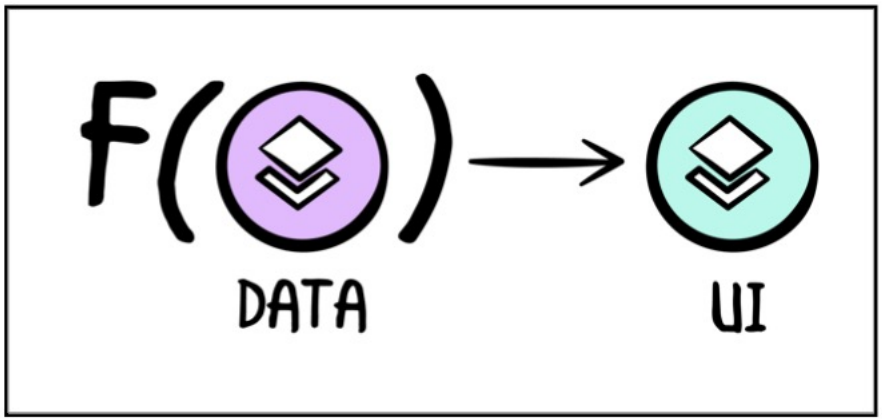


Composables

- Functions that take data (state) as parameters and emit UI
  - Composable is immutable

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

  - Function is idempotent without side effects (no global variables)
  - Functions run in parallel – need to be thread safe

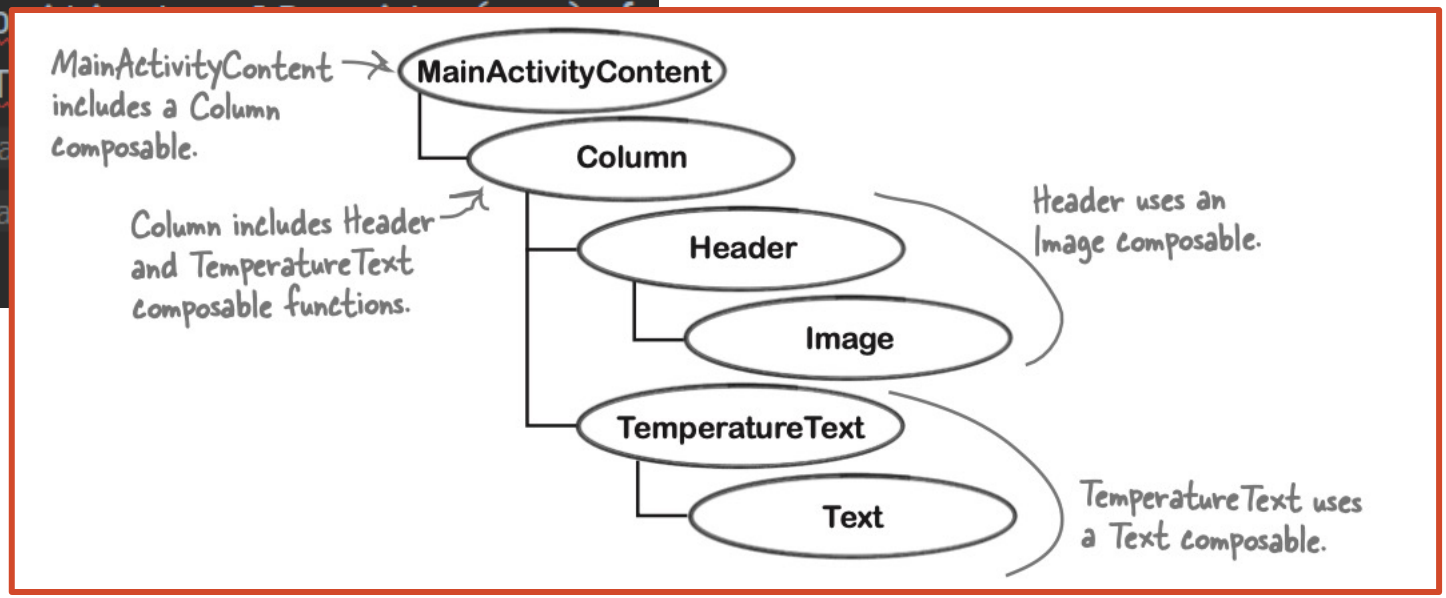


Style

```
@Composable
fun PhotographerCard(modifier: Modifier = Modifier) {
    Row(modifier = modifier
        .padding(8.dp)
        .clip(RoundedCornerShape(4.dp))
        .background(MaterialTheme.colors.surface)
        .clickable(onClick = {})
        .padding(16.dp)
    ) { this: RowScope
        Surface(
            modifier = Modifier.size(50.dp),
            shape = CircleShape,
            color = MaterialTheme.colors.onSurface.copy(alpha = 0.2f)
        ) {
            // Image goes here
        }
        Column(
            modifier = Modifier
                .padding(start = 8.dp)
                .align(Alignment.CenterVertically)
        ) { this: ColumnScope
            Text( text: "Alfred Sisley", fontWeight = FontWeight.Bold)
            CompositionLocalProvider( ...values: LocalContentAlpha provides ContentAlpha.medium) {
                Text( text: "3 minutes ago", style = MaterialTheme.typography.body2)
            }
        }
    }
}
```

Style Structure

```
Row(...) {
    Surface(...) {
        // Image goes here
    }
    Column(...) {
        Text(...)
        Compo
            T
        } ^la
    } ^lambda
}
```



Design Principle #1: Composition Over Inheritance

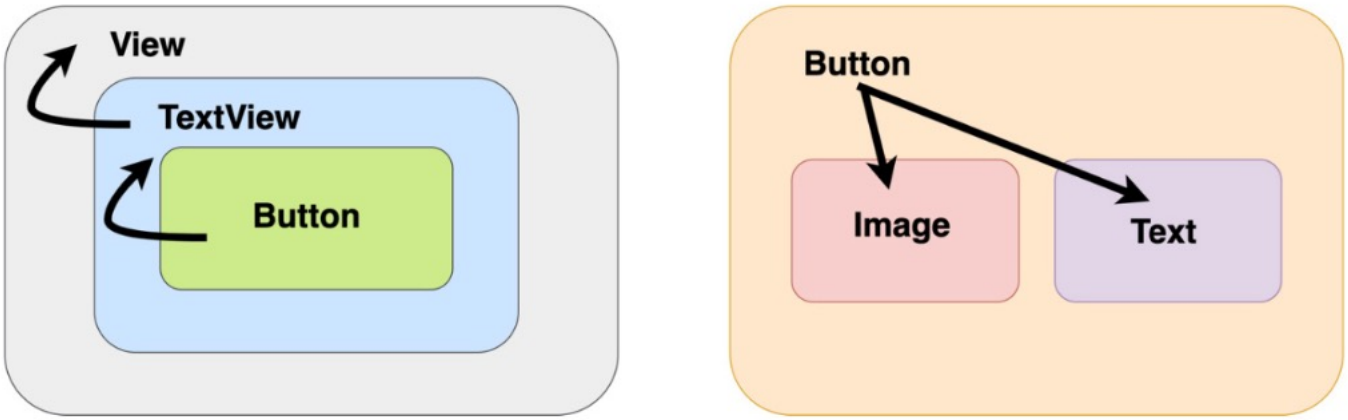


Figure 1.6 – Inheritance versus composition

- *Favor composition over Inheritance*: classes should achieve polymorphic behavior and code reuse by their composition (contain instances of other classes that contain the desired functionality) rather than inherit from a base class

Composition over Inheritance

- Jetpack Compose is a set of functions
  - Combine functions to compose more complex UI
- UI is immutable, there are no objects
- There are no types
- Nothing to inherit

- Jetpack Compose is not Object-Oriented!

Design Principles

1. ***Favor composition over inheritance***: Compose UI, Classes
2. ***Write Once Read Many***: resources (strings)

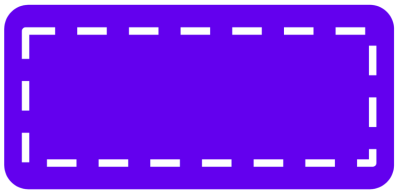
Slot API

```
Button(text = "Button")
```


```
Button(
    text = "Button",
    icon: Icon? = myIcon,
    textStyle = TextStyle(...),
    spacingBetweenIconAndText = 4.dp,
    ...
)
```


```
Button {
    Row {
        MyImage()
        Spacer(4.dp)
        Text("Button")
    }
}
```

```
@Composable
fun Button(
    modifier: Modifier = Modifier.None,
    onClick: (() -> Unit)? = null,
    ...
    content: @Composable () -> Unit
}
```



Making Buttons

1. Text Buttons

```
@Composable
fun TextButton(text: String) {
  Box(modifier = Modifier.clickable(onClick = { ... })) {
    Text(text = text)
  }
}
```

2. Image Buttons

```
@Composable
fun ImageButton() {
  Box(modifier = Modifier.clickable(onClick = { ... })) {
    Icon(painterResource(id = R.drawable.vector),
contentDescription = "")
  }
}
```

3. Text & Image Button

```
@Composable
fun TextImageButton(text: String) {
  Box(modifier = Modifier.clickable(onClick = { ... })) {
    Row(verticalAlignment = Alignment.CenterVertically) {
      Icon(painterResource(id = R.drawable.vector),
contentDescription = "")
      Text(text = text)
    }
  }
}
```

Layouts

- `Column`
  - Order items vertically
- `Row`
  - Order items horizontally
- `Box`
  - Layer items on top of eachother
- `LazyColumn`
  - Scrollable list
  - Draws only visible components
- `ScrollableColumn`
  - Srollable list
  - Draws all components at once

Material Design Components

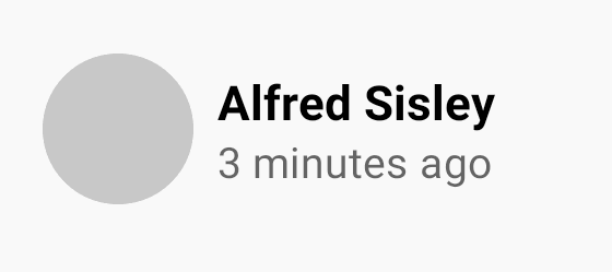
- `Text`
- `Button`
- `Card`
- `Image`
- And more

Modifiers

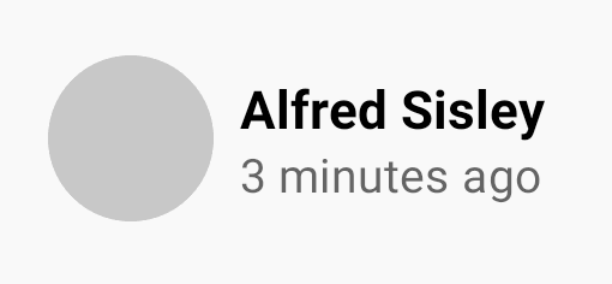
- Type safety of scope-specific modifiers
- Only some modifiers are available/applicable to certain layouts
- Only measures UI once to make nested layouts more efficient

Modifier Order Matters!

```
@Composable
fun PhotographerCard(modifier: Modifier = Modifier) {
    Row(modifier
        .padding(16.dp)
        .clickable(onClick = { /* Ignoring onClick */ })
    ) {
        ...
    }
}
```



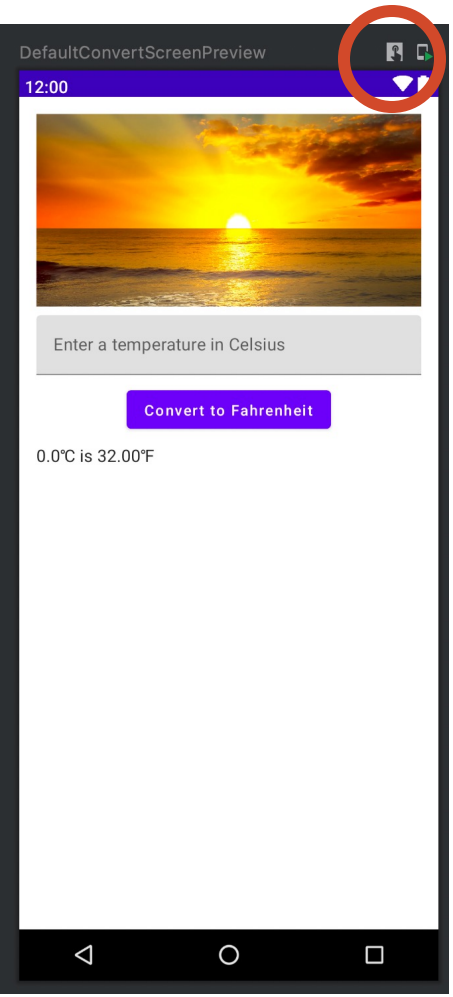

```
@Composable
fun PhotographerProfile(modifier: Modifier = Modifier) {
    Row(modifier
        .clickable(onClick = { /* Ignoring onClick */ })
        .padding(16.dp)
    ) {
        ...
    }
}
```



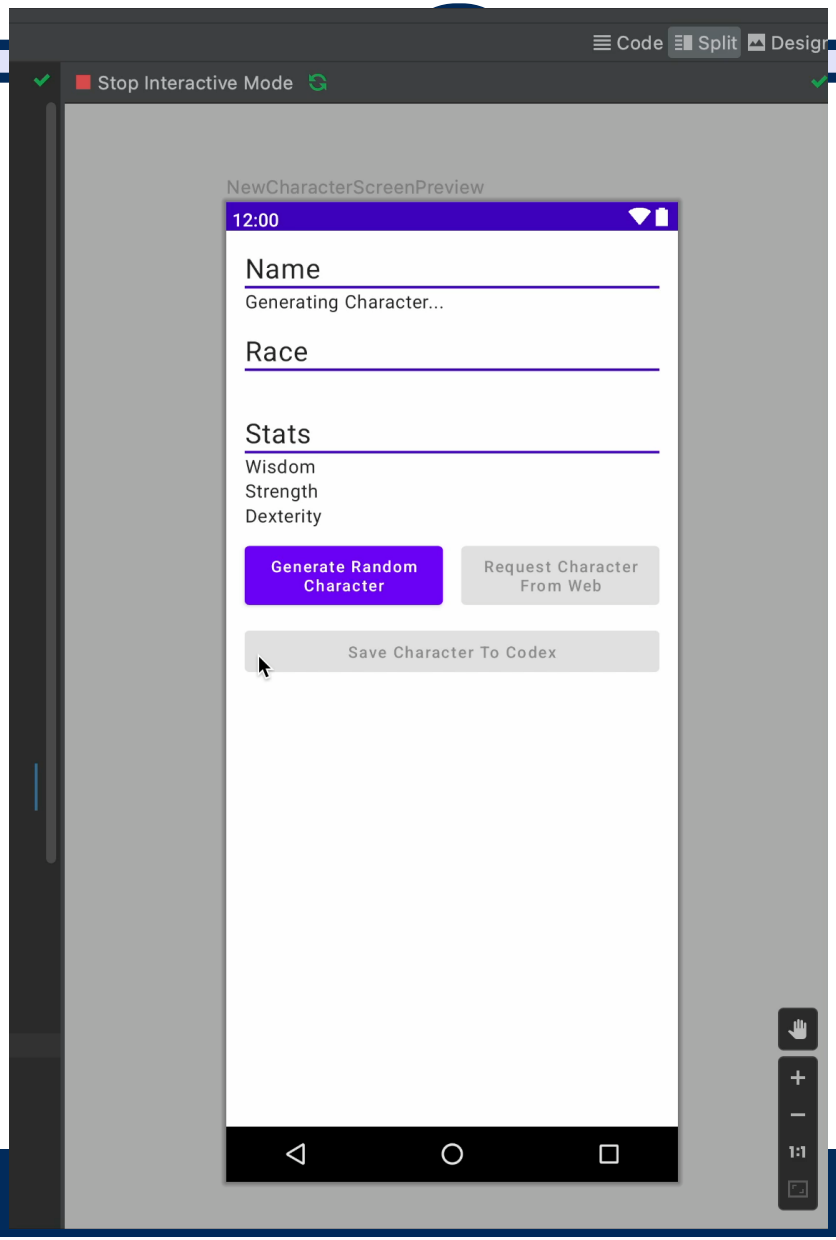
Android Studio Preview Tool

Interactive Mode

```
22  private const val LOG_TAG = "448.ConvertScreen"
23
24  @Composable
25  fun ConvertScreen() {
26      val celsius = remember { mutableStateOf( value: 0.0) }
27      val newCelsiusString = remember { mutableStateOf( value: "") }
28      Column(
29          modifier = Modifier
30              .padding(16.dp)
31              .fillMaxWidth()
32      ) {  this: ColumnScope
33          Log.d(LOG_TAG,  msg: "emitting ConvertScreen")
34          Header(R.drawable.sunrise, stringResource(R.string.sunrise_image_description))
35          EnterTemperature(newCelsiusString.value) { enteredText ->
36              newCelsiusString.value = enteredText
37          }
38          Row(
39              modifier = Modifier.fillMaxWidth(),
40              horizontalArrangement = Arrangement.Center
41          ) {  this: RowScope
42              ConvertButton {
43                  newCelsiusString.value.toDoubleOrNull()?.let { newCelsiusDouble ->
44                      celsius.value = newCelsiusDouble
45                  }
46              }
47          }
48          TemperatureText(celsius.value)
49      }
50  }
51
52  @Composable
53  private fun Header(@DrawableRes image: Int, description: String) {
54      Image(
55          painter = painterResource(image),
56          contentDescription = description,
57          modifier = Modifier
58              .height(180.dp)
59              .fillMaxHeight(),
60          contentScale = ContentScale.Crop
61      )
62  }
63
64  @Composable
```



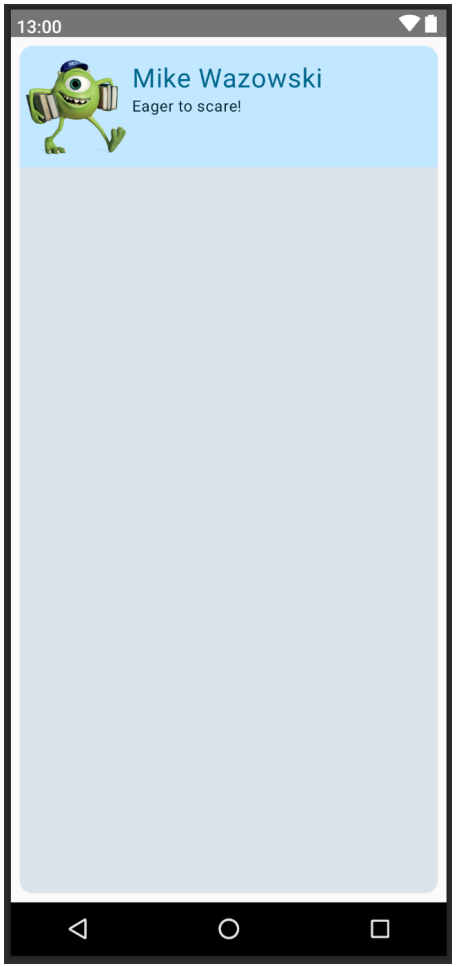
Interactive Mode



On Tap For Today

- Strings
- Compose
- Practice
  - Lab01A
  - Kotlin Quiz

Lab01A: MonsterCard



On Tap For Today

- Strings
- Compose
- Practice
  - Lab01A
  - Kotlin Quiz

Tasks for Today

- Take Kotlin Basics quiz in Canvas by Fri 2p
  - Covers Kotlin videos 1-4A
  - Open notes
  - Code: `rockies`