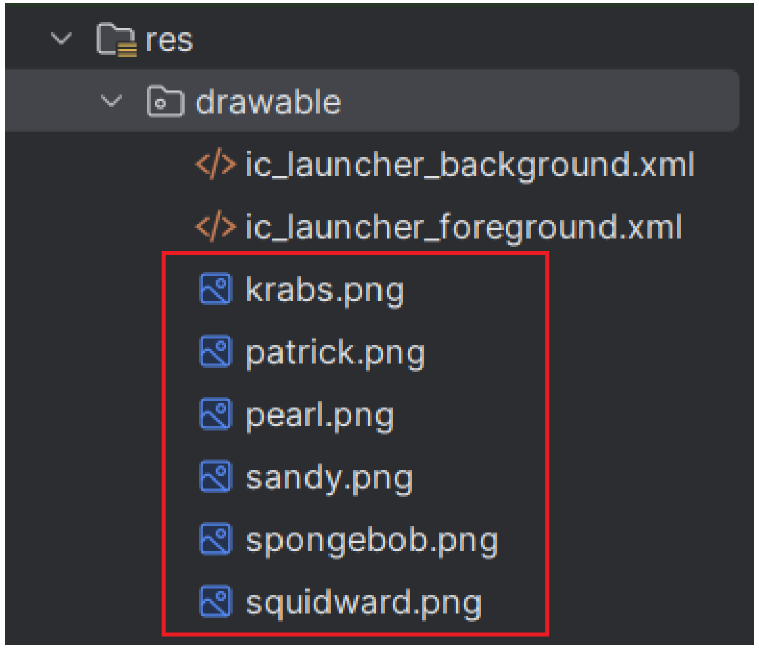
Lecture 7

COMP 3717- Mobile Dev with Android Tech

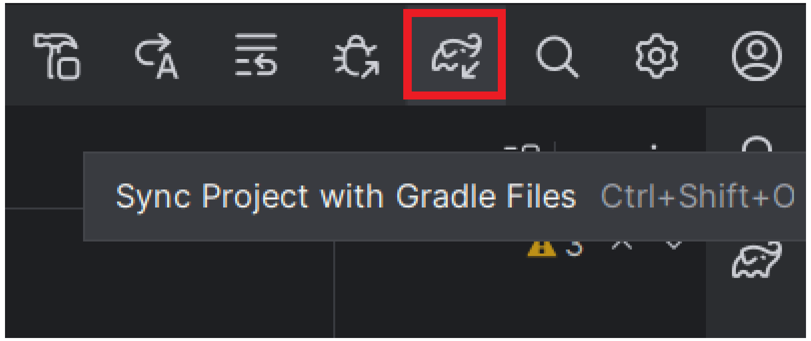
Displaying an Image

- To add an image to your app first drag one or more images over into your drawable folder



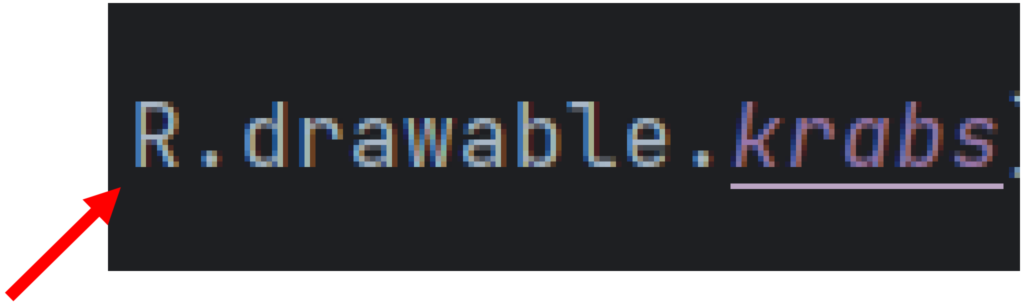
```
res
  drawable
    ic_launcher_background.xml
    ic_launcher_foreground.xml
    krabs.png
    patrick.png
    pearl.png
    sandy.png
    spongebob.png
    squidward.png
```

- After adding resources to your project, you should do a *Sync Project with Gradle Files*



Displaying an Image (cont.)

- The R class gives us access to all the resources in our project
  - drawables, strings, fonts, colors, files, etc.

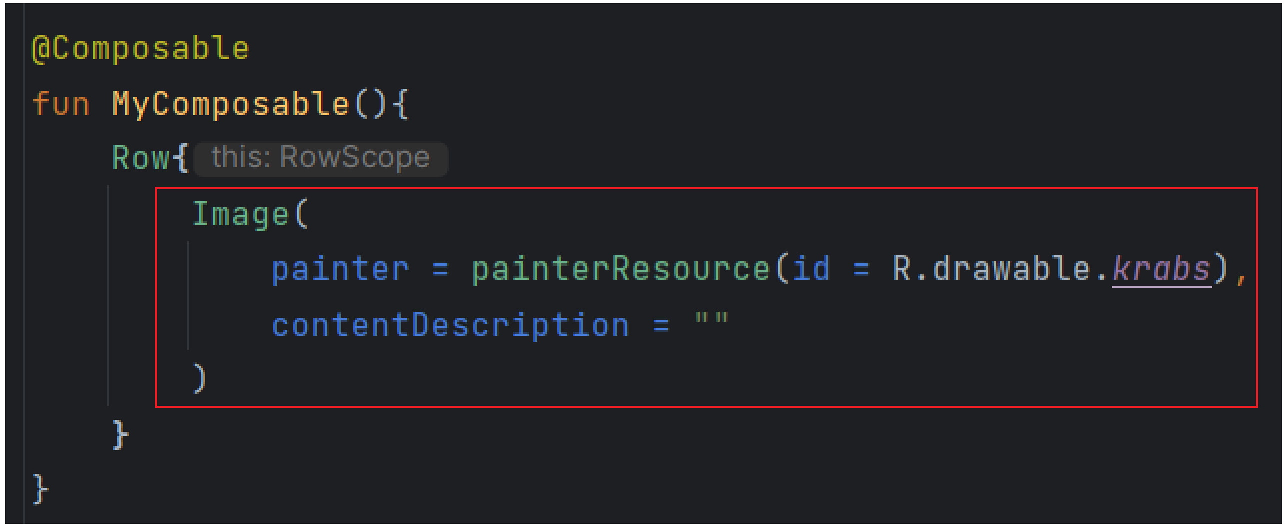


```
R.drawable.krabs
```

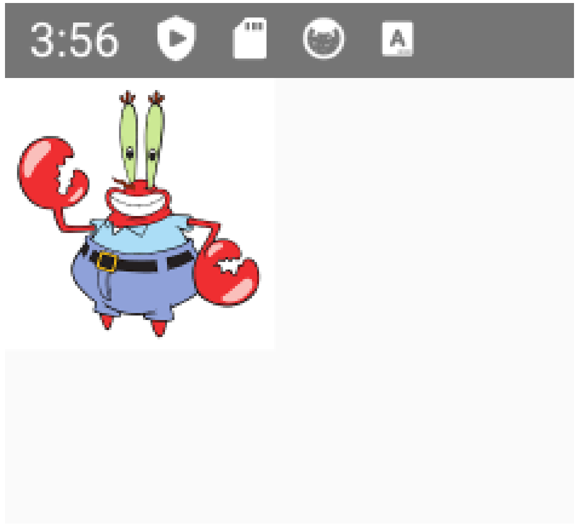
- When we access resources through the R class, it returns a resource id as an integer

Displaying an Image (cont.)

- Create an Image composable with the two required parameters
  - painter & contentDescription



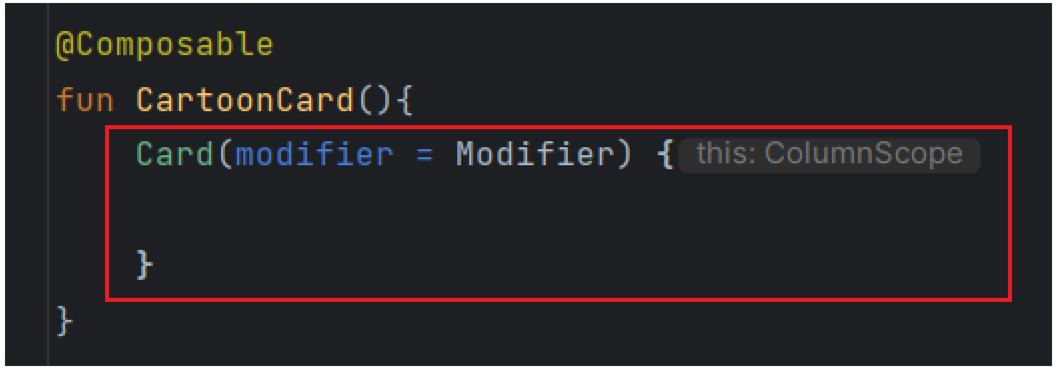
```
@Composable
fun MyComposable(){
    Row{ this: RowScope
        Image(
            painter = painterResource(id = R.drawable.krabs),
            contentDescription = ""
        )
    }
}
```



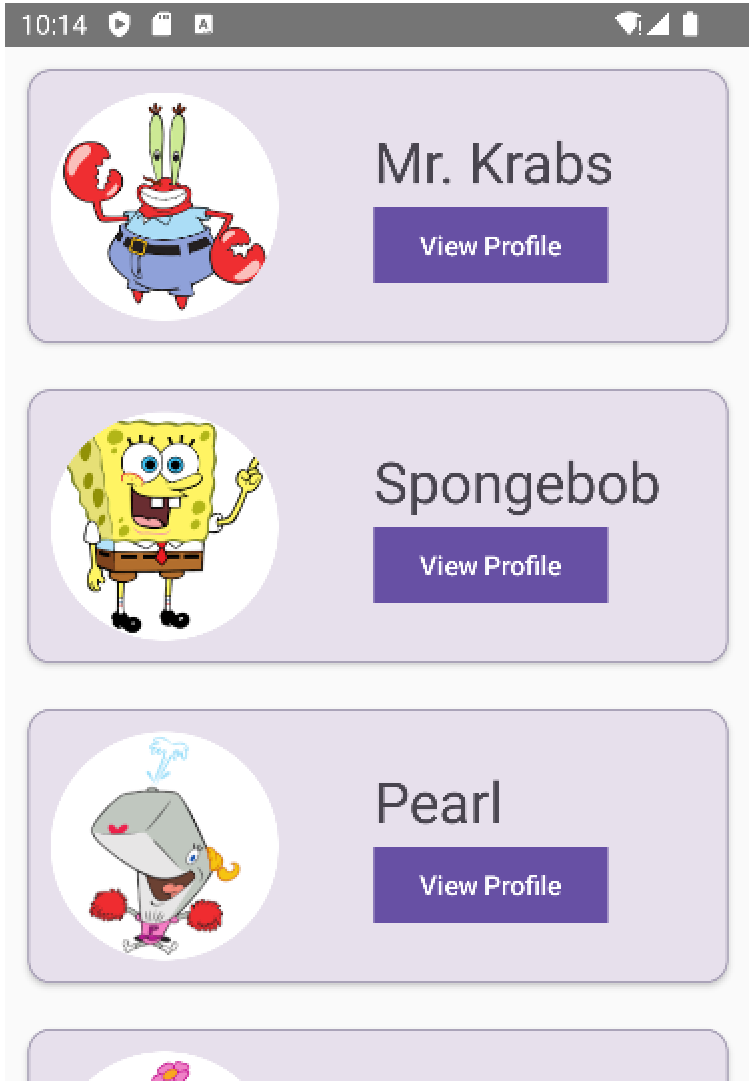
- The *painerResource* function takes in an id param as an Integer

Card

- A card is a small container that provides a single piece of content to the screen

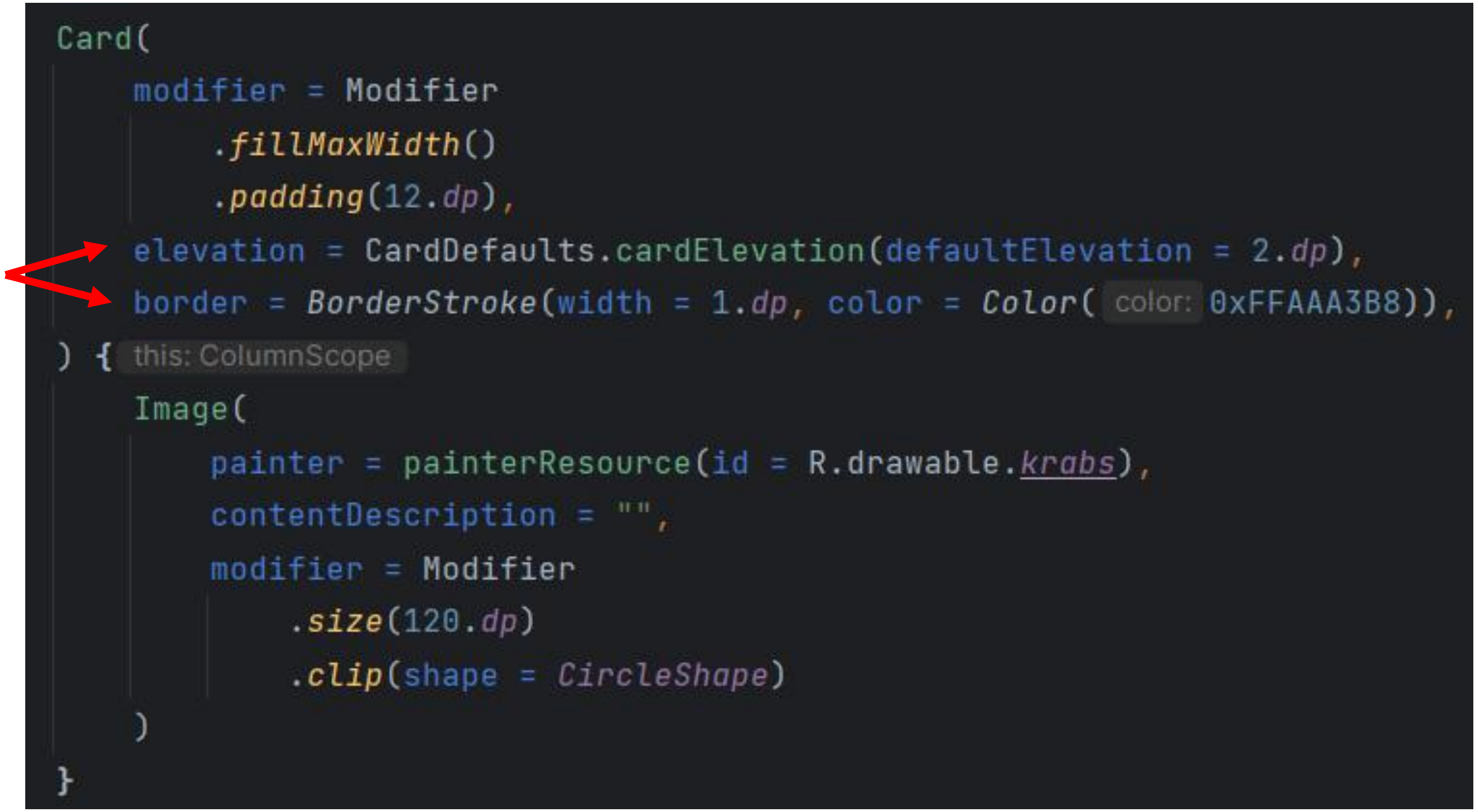


```
@Composable
fun CartoonCard(){
    Card(modifier = Modifier) { this: ColumnScope

    }
}
```

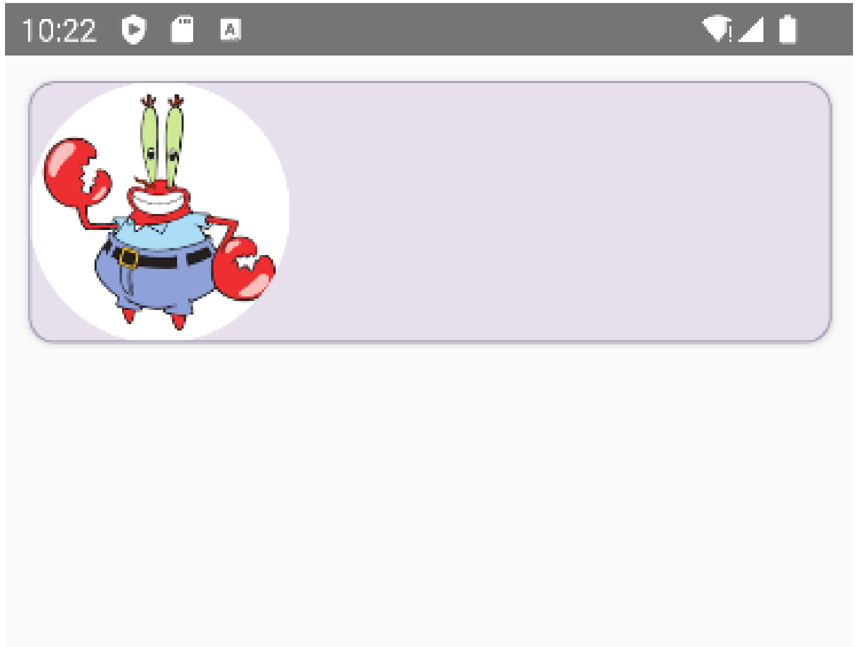


Card (cont.)

- A card has a default color with elevation and border params



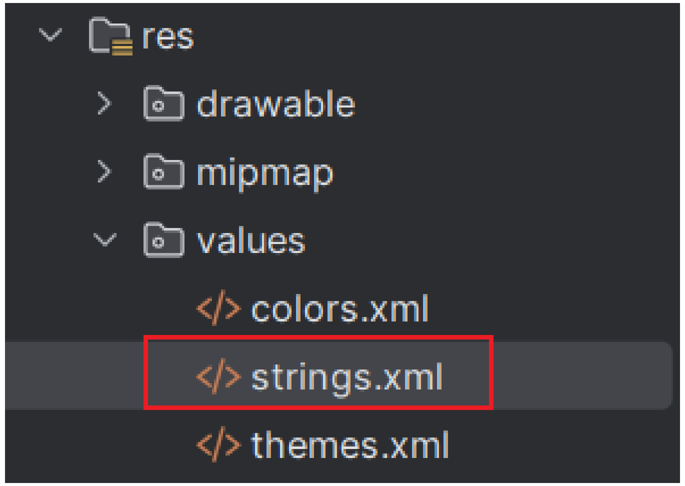
```
Card(
    modifier = Modifier
        .fillMaxWidth()
        .padding(12.dp),
    elevation = CardDefaults.cardElevation(defaultElevation = 2.dp),
    border = BorderStroke(width = 1.dp, color = Color( color: 0xFFAAA3B8)),
) { this: ColumnScope
    Image(
        painter = painterResource(id = R.drawable.krabs),
        contentDescription = "",
        modifier = Modifier
            .size(120.dp)
            .clip(shape = CircleShape)
    )
}
```

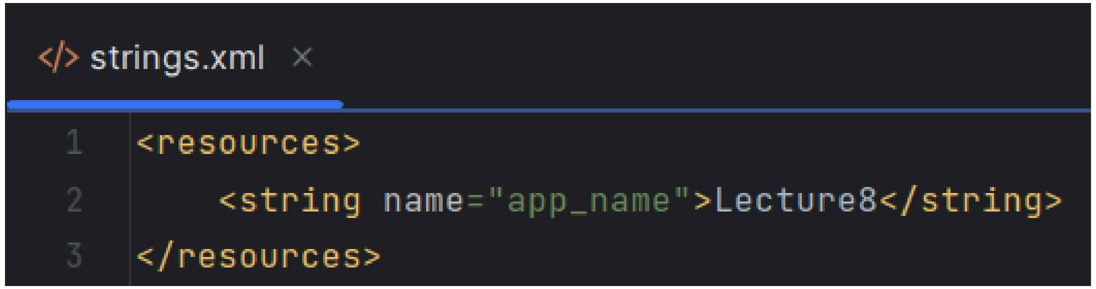


String resources

- An xml resource that provides text strings for your application

- You can store a single string or an array of strings

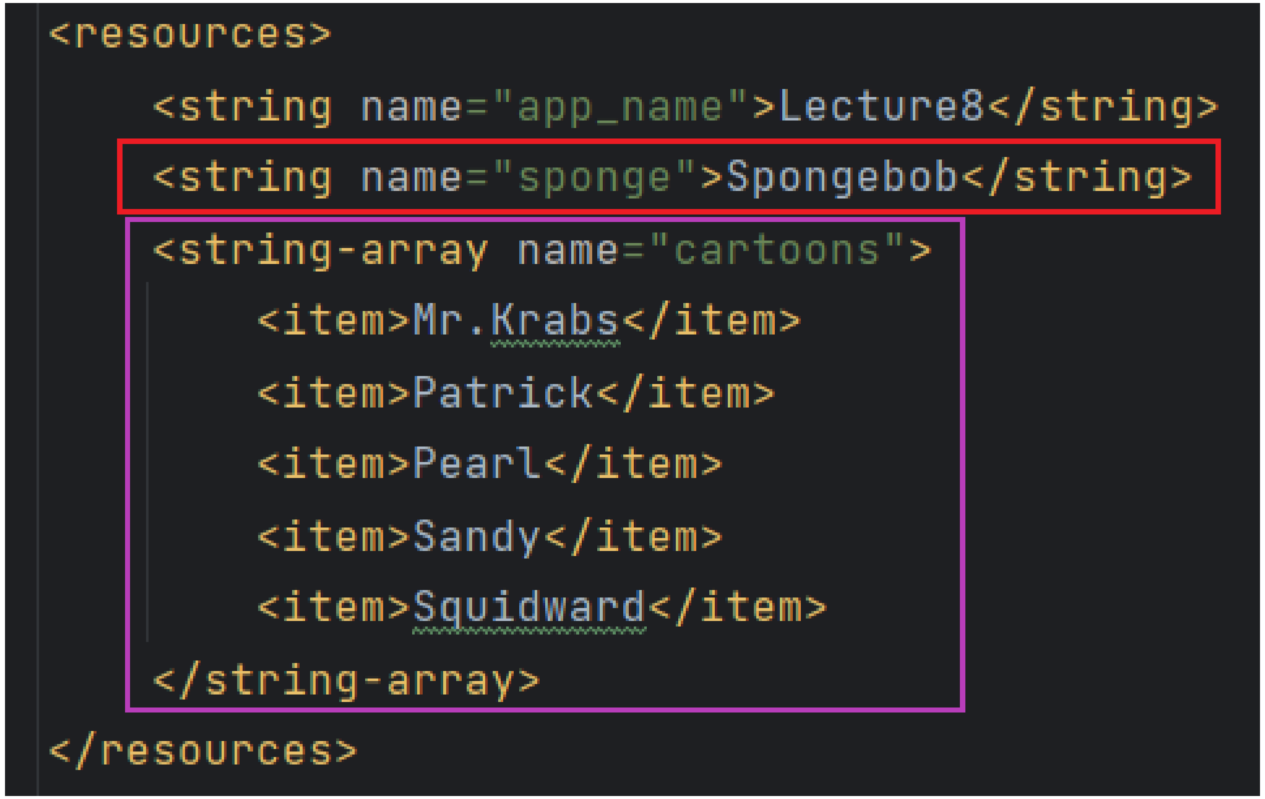




```
<resources>
    <string name="app_name">Lecture8</string>
</resources>
```

String resources (cont.)

- Here I added a single string with the id sponge and a string array with the id cartoons

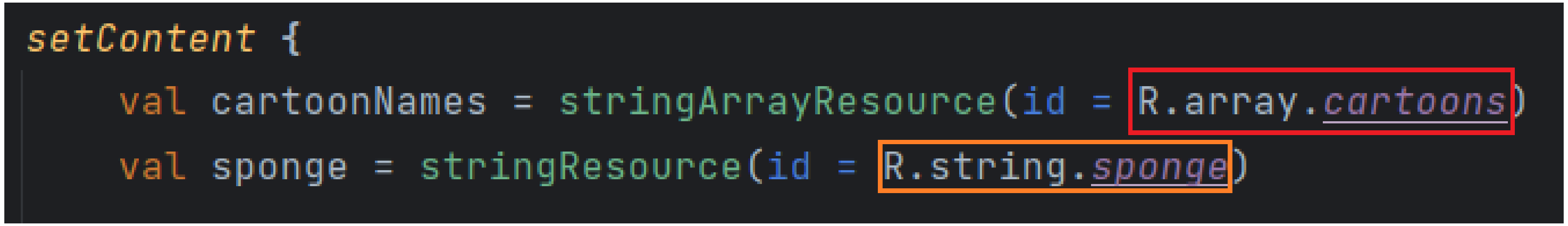


```
<resources>
    <string name="app_name">Lecture8</string>
    <string name="sponge">Spongebob</string>
    <string-array name="cartoons">
        <item>Mr.Krabs</item>
        <item>Patrick</item>
        <item>Pearl</item>
        <item>Sandy</item>
        <item>Squidward</item>
    </string-array>
</resources>
```

String resources (cont.)

- To get your string resources you can use the composable
  - stringArrayResource, or
  - stringResource

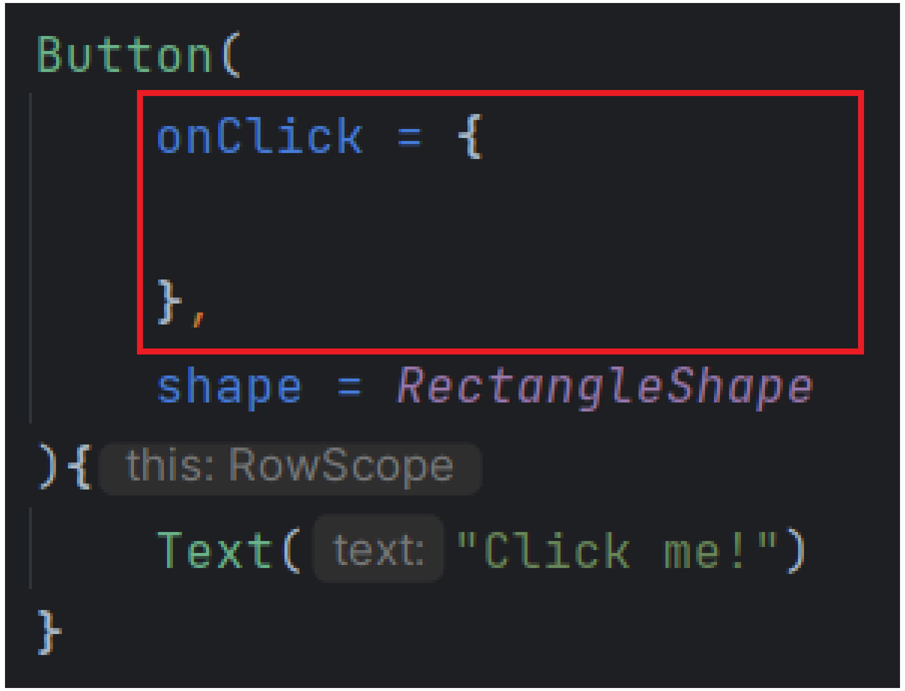
- To find the specific id, we use the R class
  - R.array for an array of strings
  - R.string for a single string



```
setContent {
    val cartoonNames = stringArrayResource(id = R.array.cartoons)
    val sponge = stringResource(id = R.string.sponge)
```

Button

- A button has a *onClick* event callback
- What do we want to do when the button is clicked?



```
Button(
    onClick = {


    },
    shape = RectangleShape
){ this: RowScope
    Text( text: "Click me!")
}
```

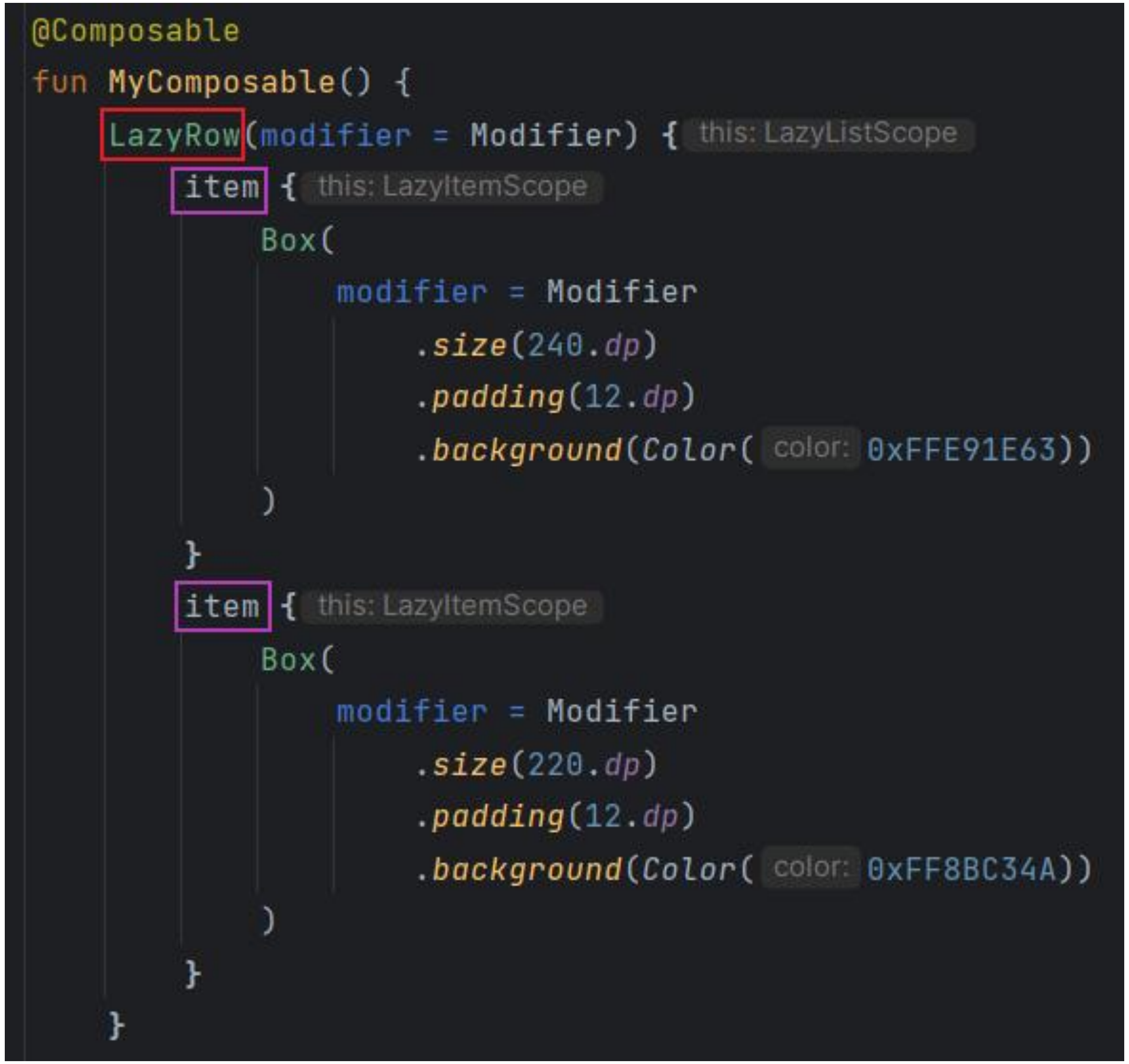
Lists

- Its often the case we want to scroll through our elements
  - Maybe we can't fit all our elements in the area we want

- A *LazyRow* and *LazyColumn* are designed for long lists of data
  - They are efficient by only rendering the elements that are on the screen

Lists (cont.)

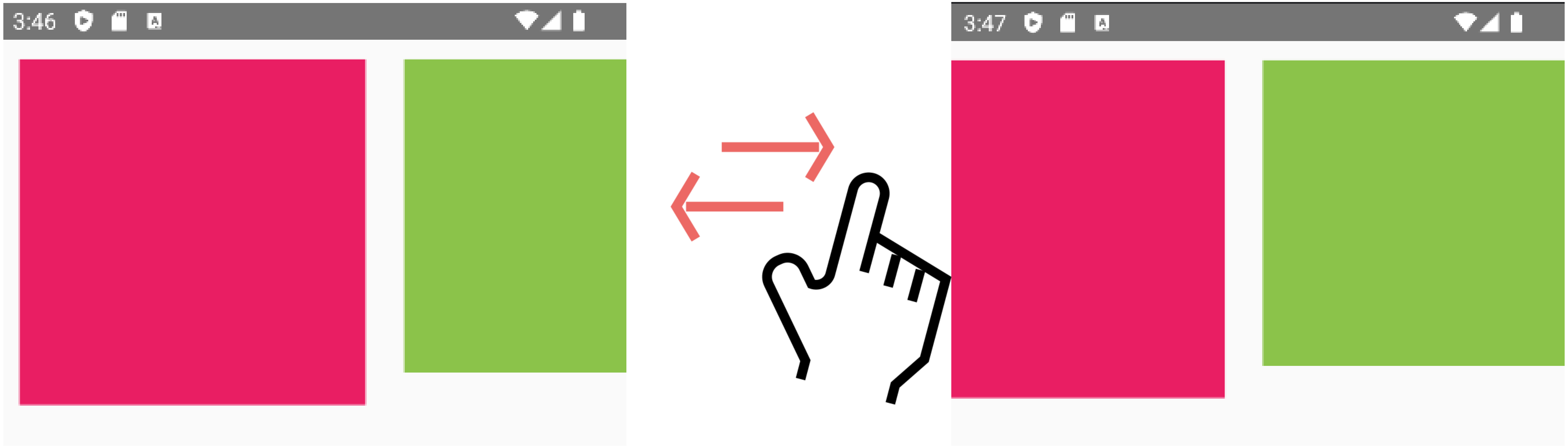
- When using a *LazyColumn* or *LazyRow*, just wrap the children with item



```
@Composable
fun MyComposable() {
    LazyRow(modifier = Modifier) { this: LazyListScope
        item { this: LazyItemScope
            Box(
                modifier = Modifier
                    .size(240.dp)
                    .padding(12.dp)
                    .background(Color( color: 0xFFE91E63))
            )
        }
        item { this: LazyItemScope
            Box(
                modifier = Modifier
                    .size(220.dp)
                    .padding(12.dp)
                    .background(Color( color: 0xFF8BC34A))
            )
        }
    }
```

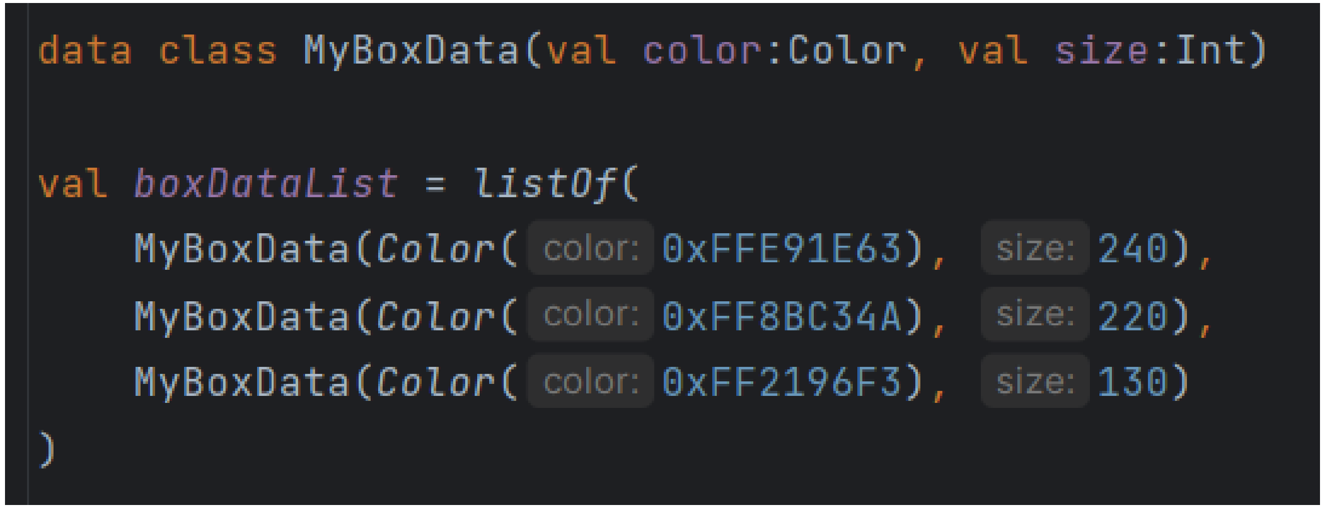
Lists (cont.)

- Now I can scroll the two elements horizontally in my LazyRow



Lists (cont.)

- Usually, you are working with lists of data

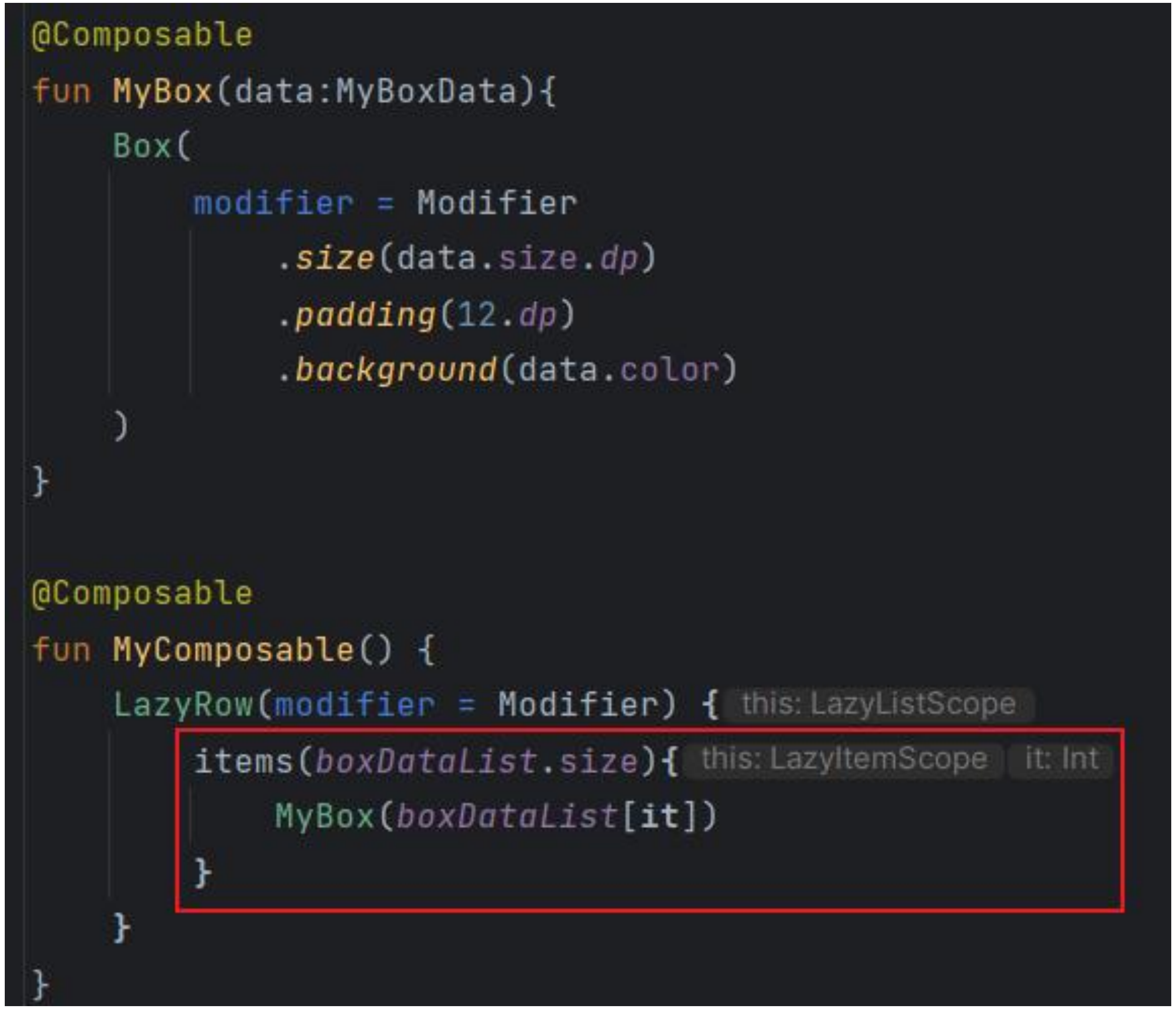


```
data class MyBoxData(val color:Color, val size:Int)

val boxDataList = listOf(
    MyBoxData(Color( color: 0xFFE91E63), size: 240),
    MyBoxData(Color( color: 0xFF8BC34A), size: 220),
    MyBoxData(Color( color: 0xFF2196F3), size: 130)
)
```

- For example, this data class holds a Color and an Int

Lists (cont.)

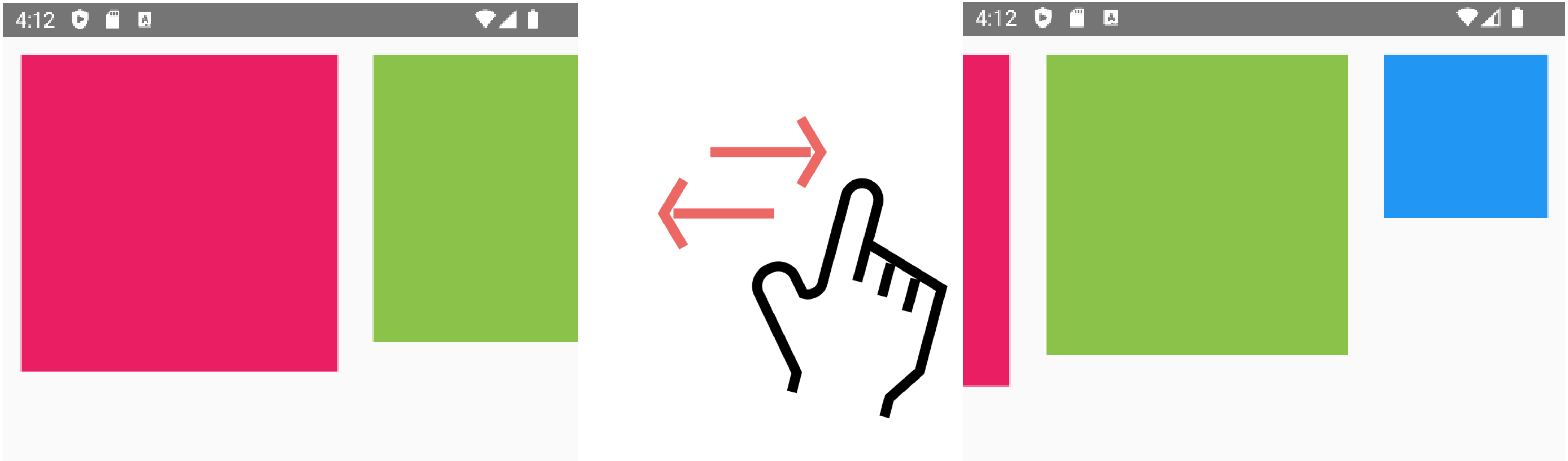
- Instead of repeating item we can use *items*, and use our list of data

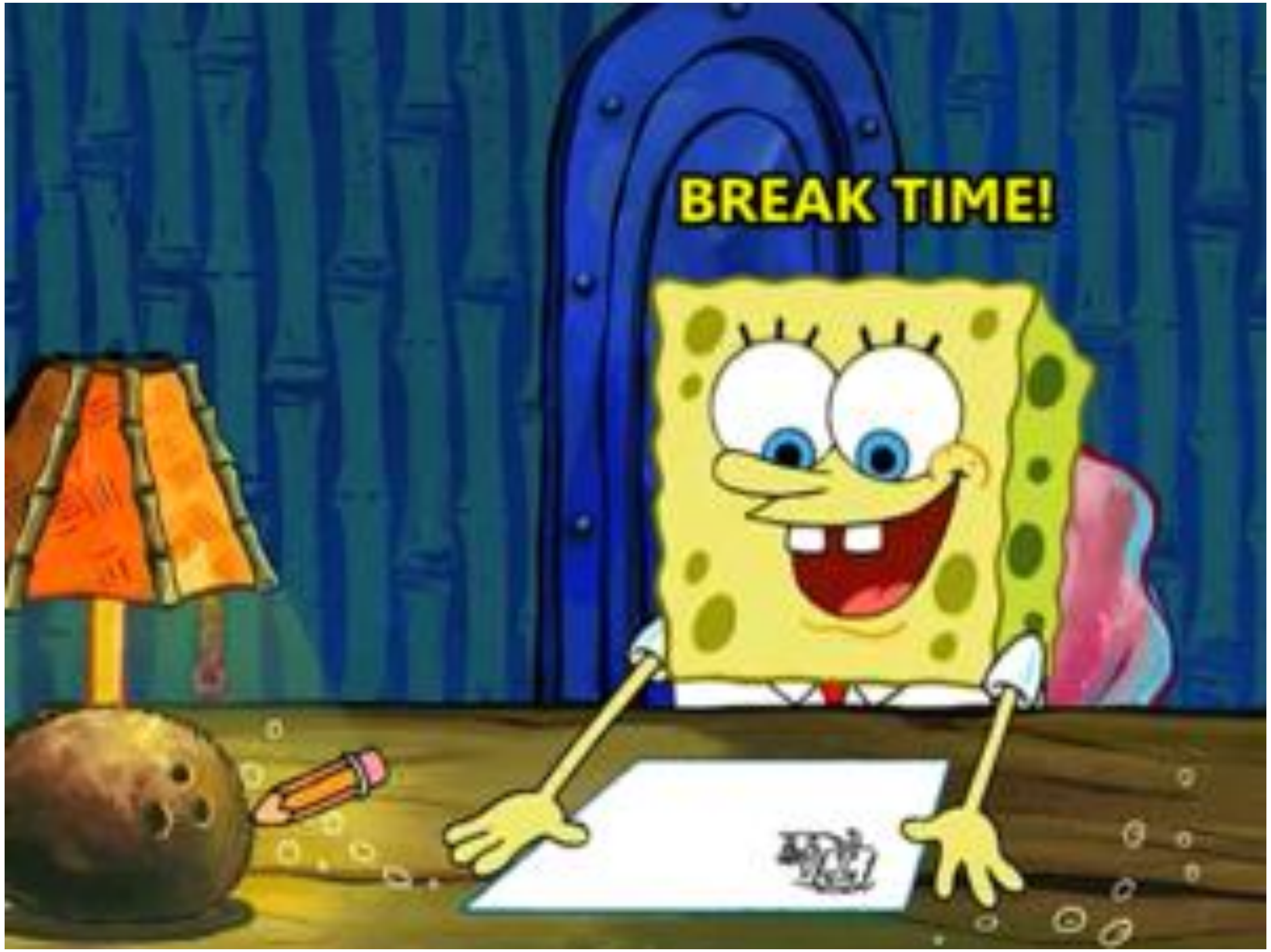


```
@Composable
fun MyBox(data:MyBoxData){
    Box(
        modifier = Modifier
            .size(data.size.dp)
            .padding(12.dp)
            .background(data.color)
    )
}

@Composable
fun MyComposable() {
    LazyRow(modifier = Modifier) {
        items(boxDataList.size){
            MyBox(boxDataList[it])
        }
    }
}
```

Lists (cont.)

- Now I can scroll through all our data




BREAK TIME!

Composable lifecycle

- When Jetpack Compose executes a composable, it enters the *Composition*

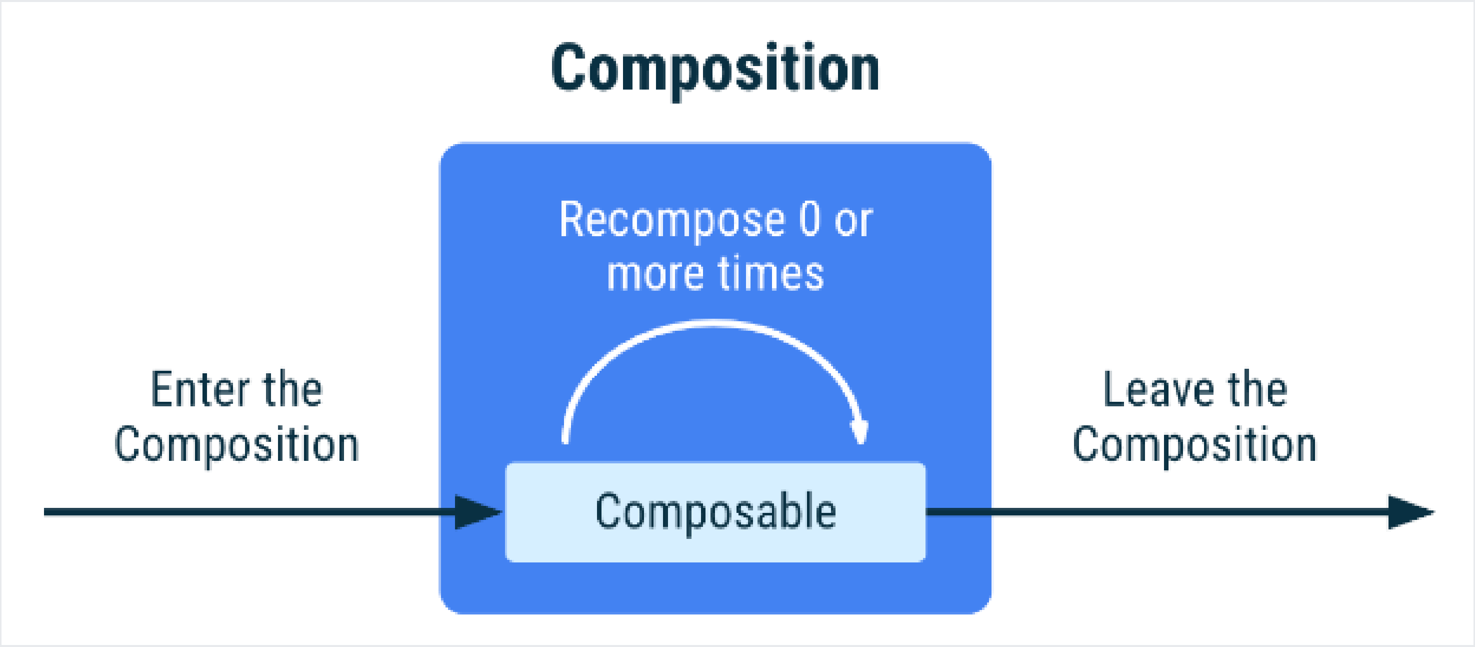
- There are two ways to enter the *Composition*

  1. The first time you run your composable it goes through *initial composition*

  2. When the *state* read by your composable changes, it goes through *recomposition*

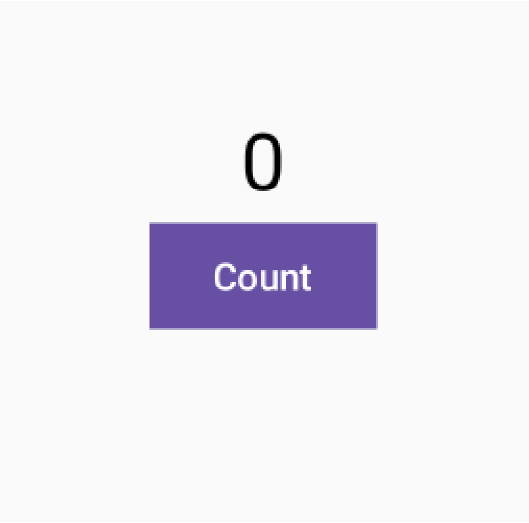
Composable lifecycle (cont.)

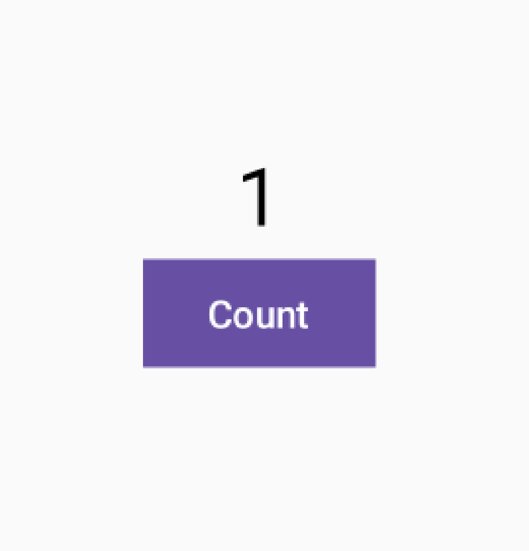
- Think of *Composition* as when a composable is being displayed to the UI
- It leaves *Composition* when it is not being displayed anymore



State

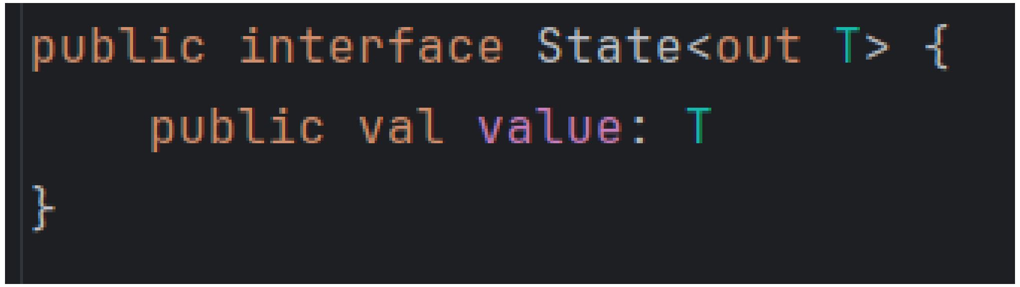
- Specific data that changes overtime within a composable
  - e.g., A Text composable could display multiple values over its lifetime
- To change state and trigger a recomposition, an event needs to occur
  - e.g., Pressing a button





State (cont.)

- The state we use to trigger recompositions is *State<T>*

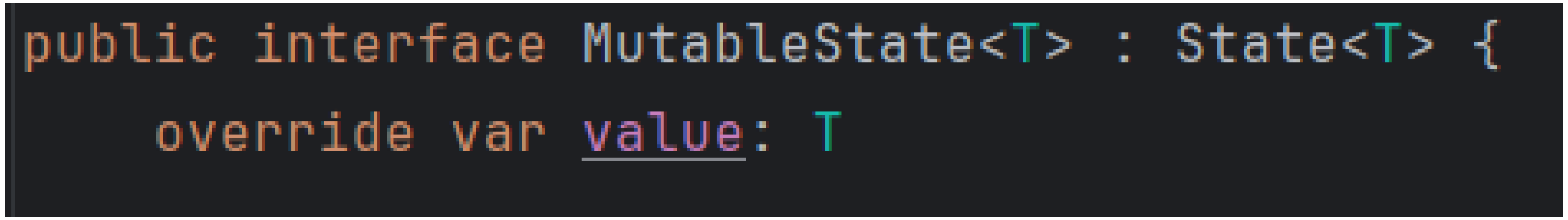


```
public interface State<out T> {
    public val value: T
}
```

- *State<T>* is an interface that simply exposes a read-only value

State (cont.)

- *State<T>* is read only, so the more common type is *MutableState<T>*

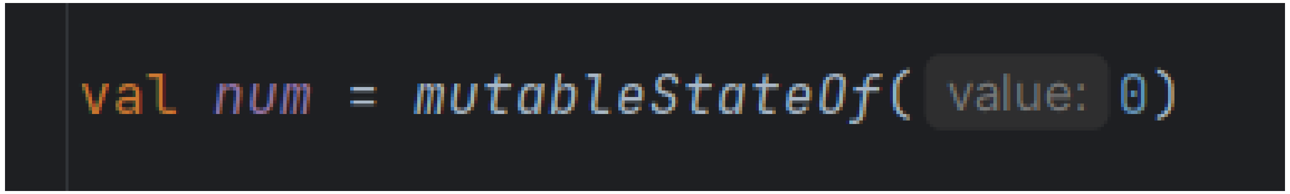


```
public interface MutableState<T> : State<T> {
    override var value: T
```

- Compose observes the *value* property and schedules recompositions when it changes

State (cont.)

- The most common way to create *MutableState<T>* is to use the *mutableStateOf* function



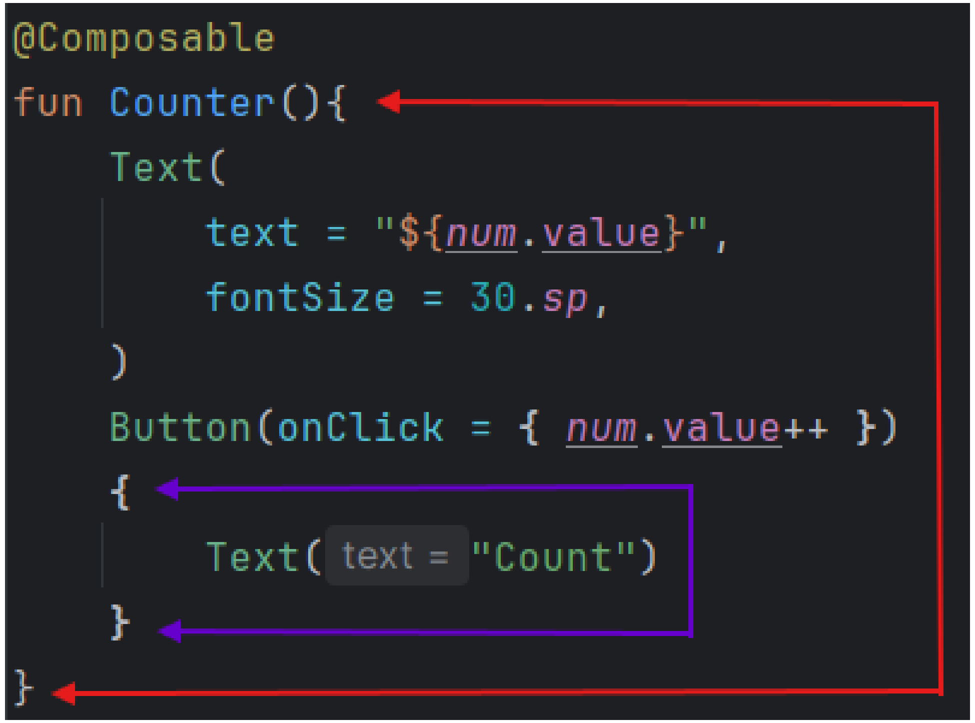
```
val num = mutableStateOf( value: 0)
```

- Kotlin infers *num* is a *MutableState<Int>* since the value is an integer

Recomposition Scope

- In the example we have two recomposition scopes
  - Counter Scope and Button Scope

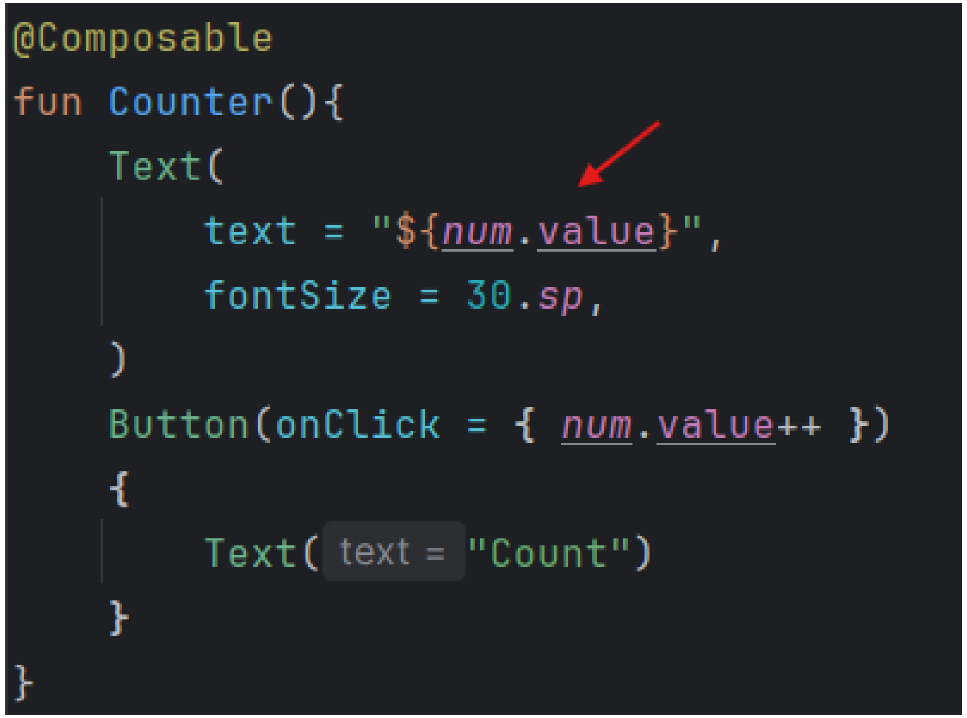
- Recomposition scope is usually marked by an opening and closing function bracket



```
@Composable
fun Counter(){
    Text(
        text = "${num.value}",
        fontSize = 30.sp,
    )
    Button(onClick = { num.value++ })
    {
        Text( text = "Count")
    }
}
```

Recomposition Scope (cont.)

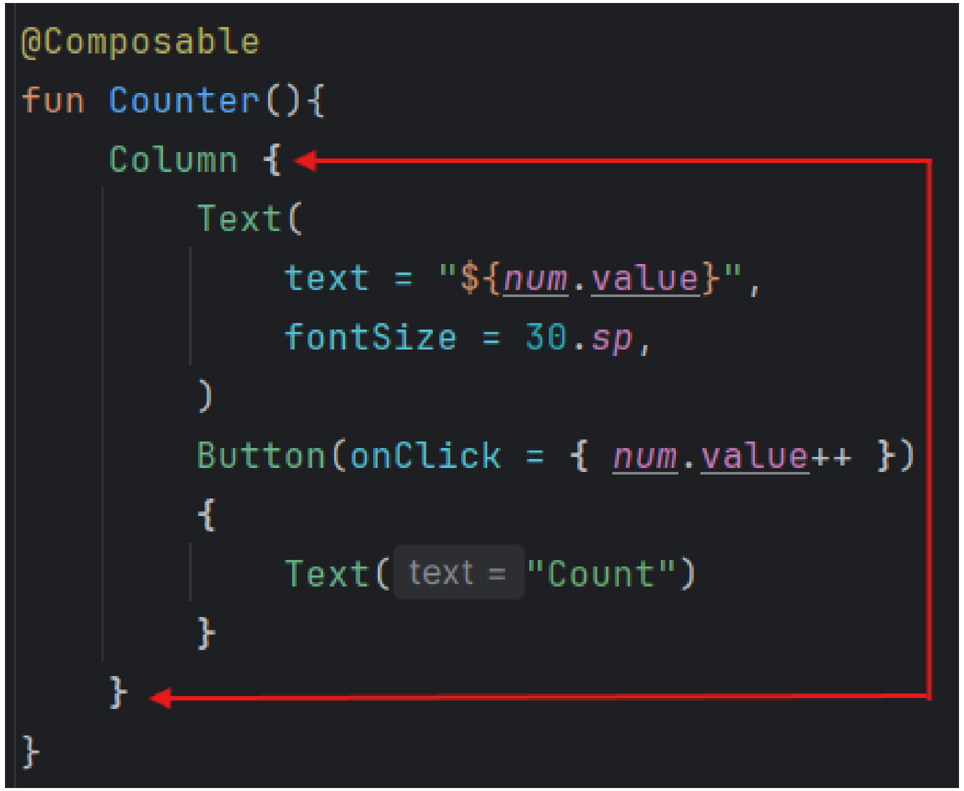
- The lowest recomposition scope to the state being read, is what will recompose (aka. Counter Scope)



```
@Composable
fun Counter(){
    Text(
        text = "${num.value}",
        fontSize = 30.sp,
    )
    Button(onClick = { num.value++ })
    {
        Text( text = "Count")
    }
}
```

Recomposition Scope (cont.)

- In this situation, you might think the *Column Scope* is the lowest recomposition scope
- A *Column*, *Row* and *Box* are inline functions, and don’t have a recomposition scope
- So *Counter Scope* is still the lowest recomposition scope



```
@Composable
fun Counter(){
    Column {
        Text(
            text = "${num.value}",
            fontSize = 30.sp,
        )
        Button(onClick = { num.value++ })
        {
            Text( text = "Count")
        }
    }
}
```

State (cont.)

- Intelligent recomposition
  - Recompose only the components that read *value*
  - Ignore the ones that don’t read *value*

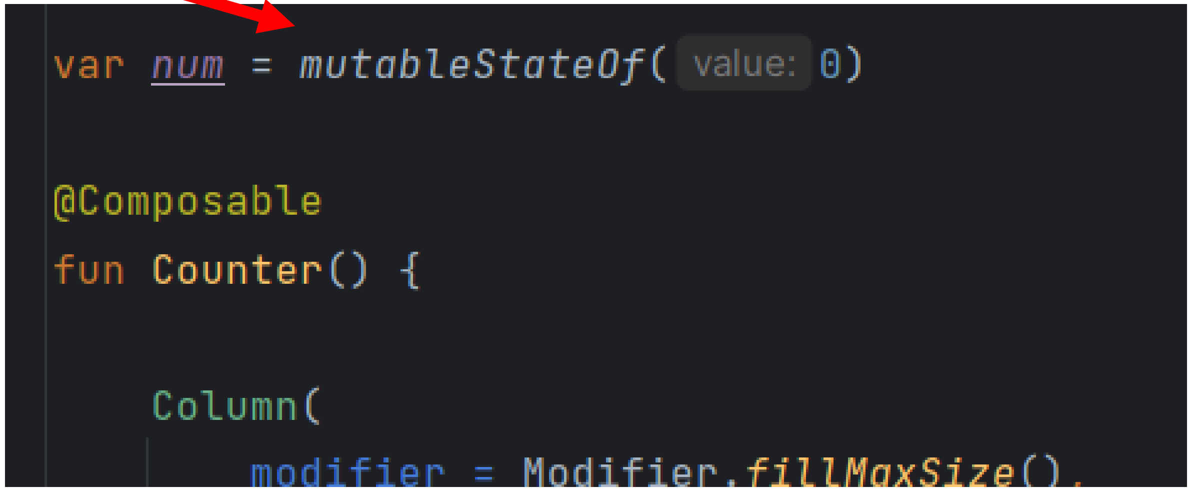
- Skipping (Not on exam)
  - If compose can determine data hasn’t changed (stable) it will be skipped
  - If compose can’t determine data has changed (unstable), it will be recomposed
  - https://developer.android.com/develop/ui/compose/performance/stability

- Both these can be tracked using the *Layout Inspector*

State (cont.)

- External state is not best practice

- State should be **internal** (aka. local) to the composable, with a few exceptions, not external
  - External state is declared outside the composable

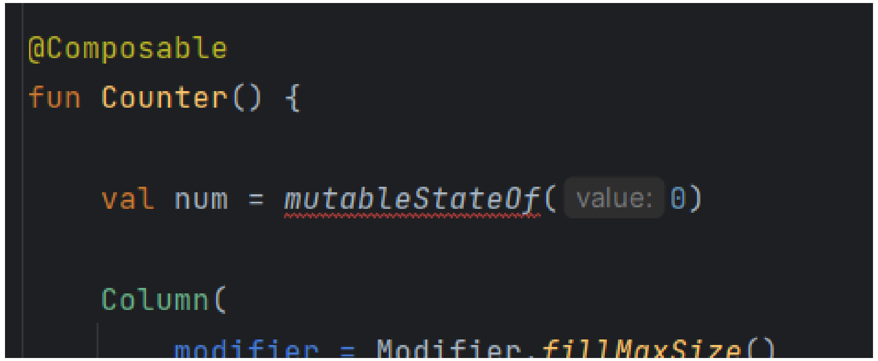


```
var num = mutableStateOf( value: 0)

@Composable
fun Counter() {

    Column(
        modifier = Modifier.fillMaxSize(),
```

State (cont.)

- When state is internal
  - Easier to test
  - Improves encapsulation and modularity
  - More optimized recomposition

- Once we move state inside the composable you will get an error
  - *“Creating a state object during composition without using remember”*

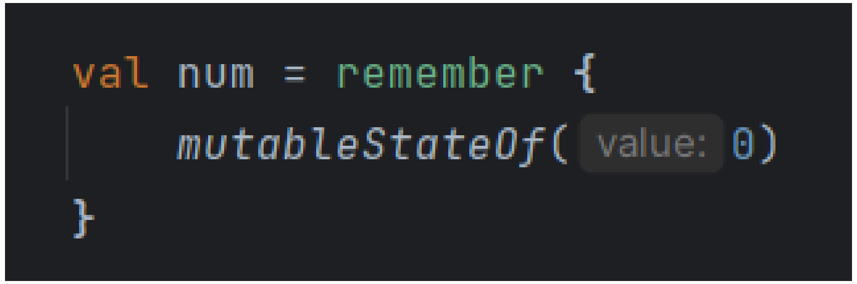


```
@Composable
fun Counter() {

    val num = mutableStateOf( value: 0)

    Column(
        modifier = Modifier.fillMaxSize()
```

State (cont.)

- The problem is Counter is recomposed (re-run) each time the value changes
  - Which in turn, re-initializes the value back to 0 each time

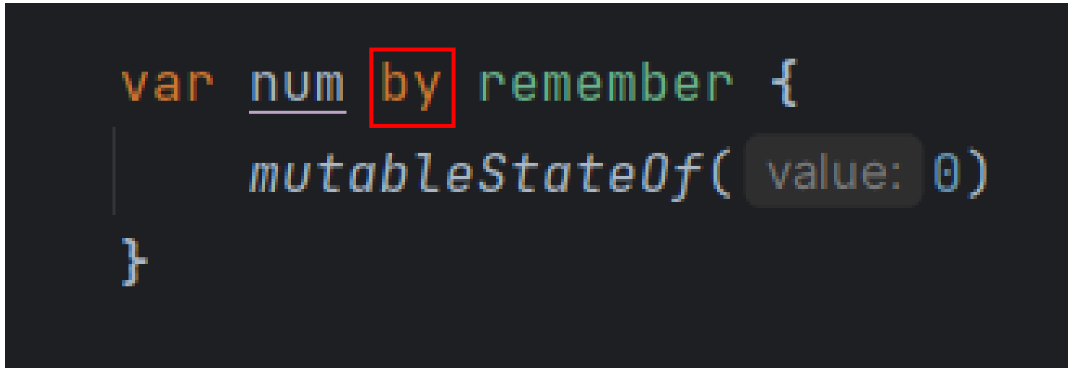
- The compiler tells us to wrap it in a *remember* composable to avoid this
  - A value wrapped in remember is stored in the *Composition*
  - This stored value is kept across *recomposition*



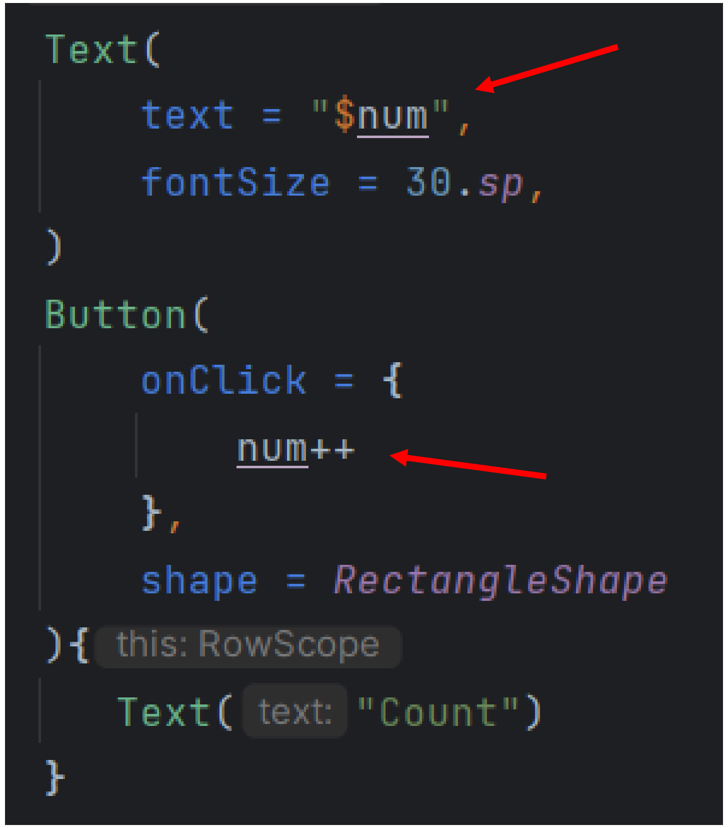
```
val num = remember {
    mutableStateOf( value: 0)
}
```

State (cont.)

- To omit *value*, we can use delegated properties



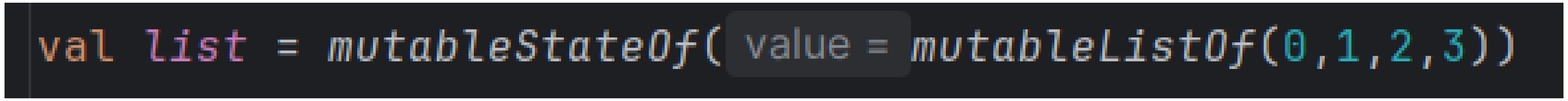
```
var num by remember {
    mutableStateOf( value: 0)
}
```



```
Text(
    text = "$num",
    fontSize = 30.sp,
)
Button(
    onClick = {
        num++
    },
    shape = RectangleShape
){ this: RowScope
   Text( text: "Count")
}
```

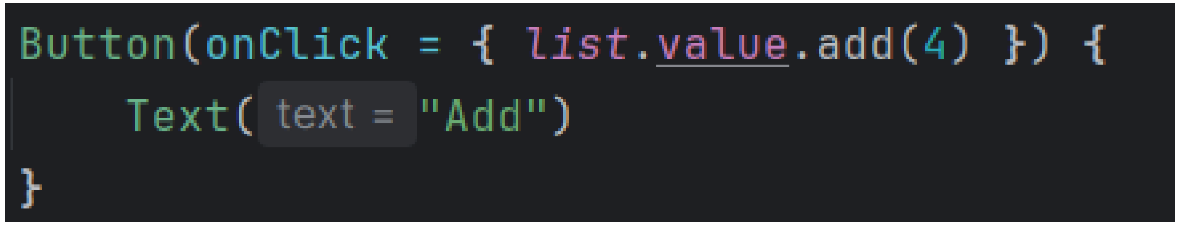
State (cont.)

- When working with collections using *State<T>* isn't the most desirable approach



```
val list = mutableStateOf( value = mutableListOf(0,1,2,3))
```

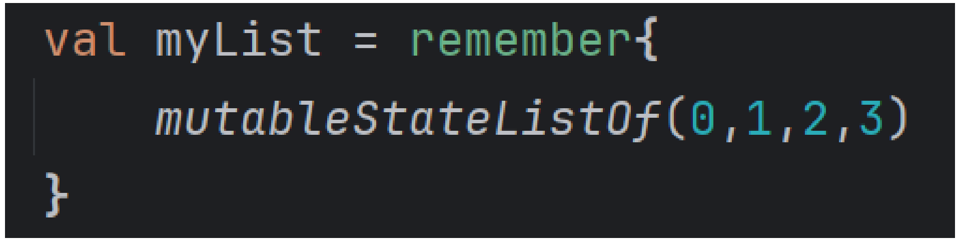
- This issue is that when we mutate the list, *value* is not being set, so no recomposition



```
Button(onClick = { list.value.add(4) }) {
    Text( text = "Add")
}
```

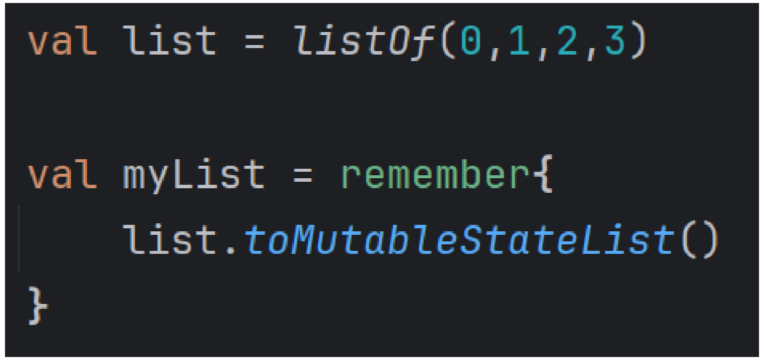
SnapshotStateList

- It’s better to use a *SnapshotStateList<T>* through the *mutableStateListOf* function



```
val myList = remember{
    mutableStateListOf(0,1,2,3)
}
```

- We can also create a mutable state list from a regular list



```
val list = listOf(0,1,2,3)

val myList = remember{
    list.toMutableStateList()
}
```

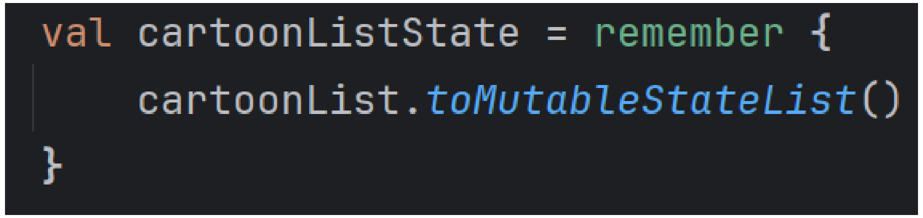
SnapshotStateList (cont.)

- A *SnapshotStateList<T>* uses a different mechanism to trigger a recomposition
  - Snapshotting

- Since it doesn’t use the *State<T>* interface, we don’t have a value property
  - Which means we also wouldn’t use the by keyword

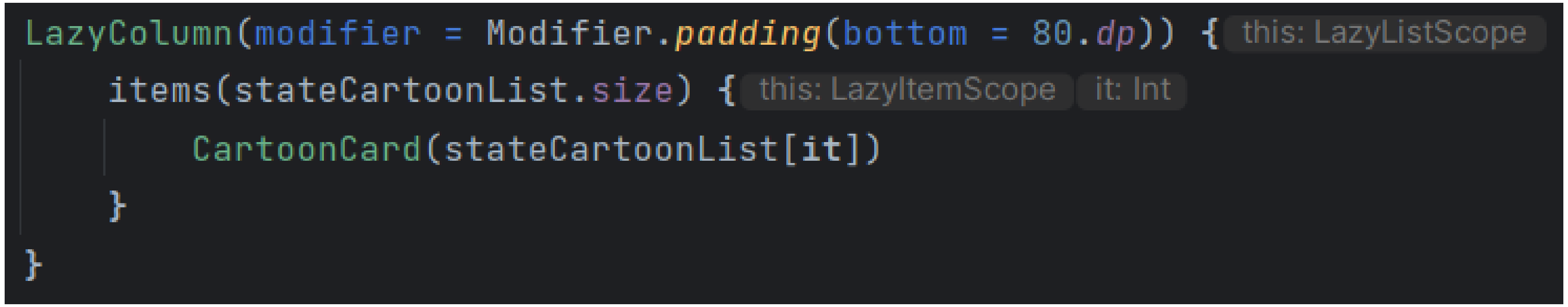
SnapshotStateList (cont.)

- Here we are creating a *SnapshotStateList* from our original cartoon list



```
val cartoonListState = remember {
    cartoonList.toMutableStateList()
}
```

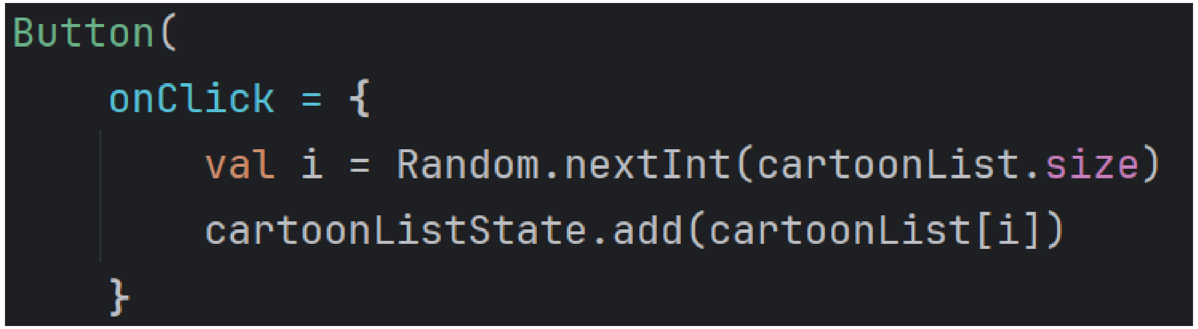
- Then using it in our *LazyColumn*



```
LazyColumn(modifier = Modifier.padding(bottom = 80.dp)) { this: LazyListScope
    items(stateCartoonList.size) { this: LazyItemScope  it: Int
        CartoonCard(stateCartoonList[it])
    }
}
```

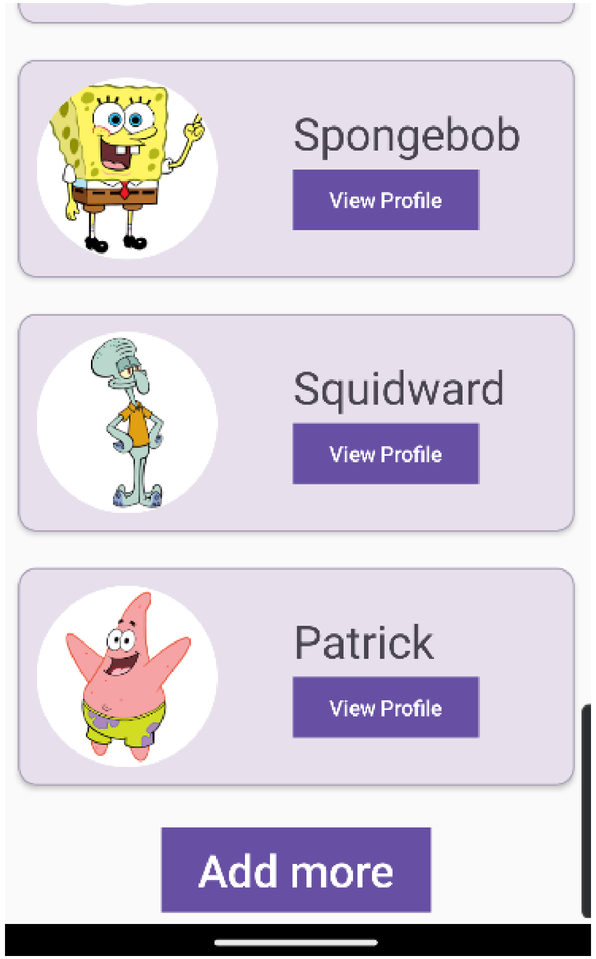
SnapshotStateList (cont.)

- We can then mutate the list through an event
  - E.g., Button Click



```
Button(
    onClick = {
        val i = Random.nextInt(cartoonList.size)
        cartoonListState.add(cartoonList[i])
    }
```

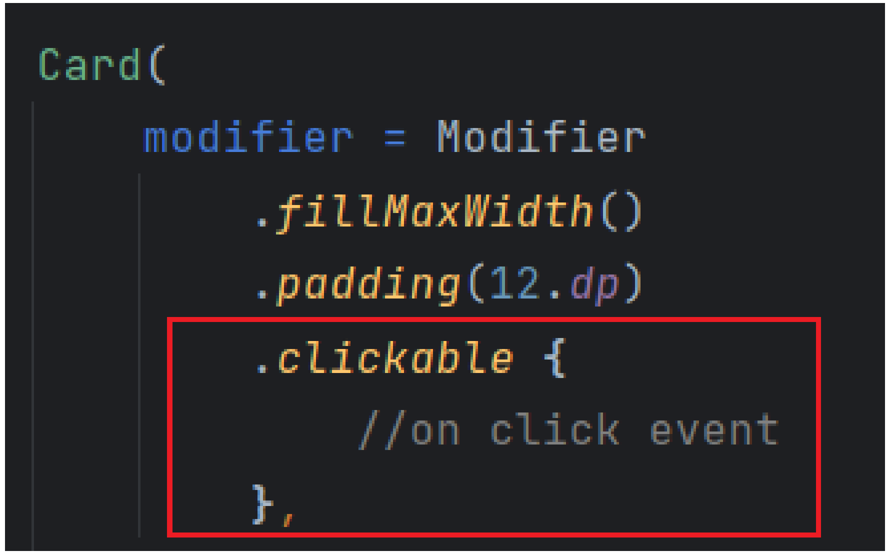
- And a recomposition will successfully occur, displaying the updated list



Clickables

- You can make any composable clickable
    - The clickable modifier provides an *onClick* event callback

- This is useful when you want to click a whole composable itself rather than just a button



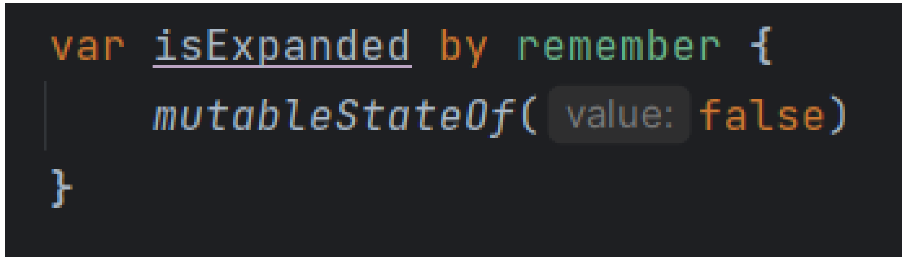
```
Card(
    modifier = Modifier
        .fillMaxWidth()
        .padding(12.dp)
        .clickable {
            //on click event
        },
```

Clickables (cont.)

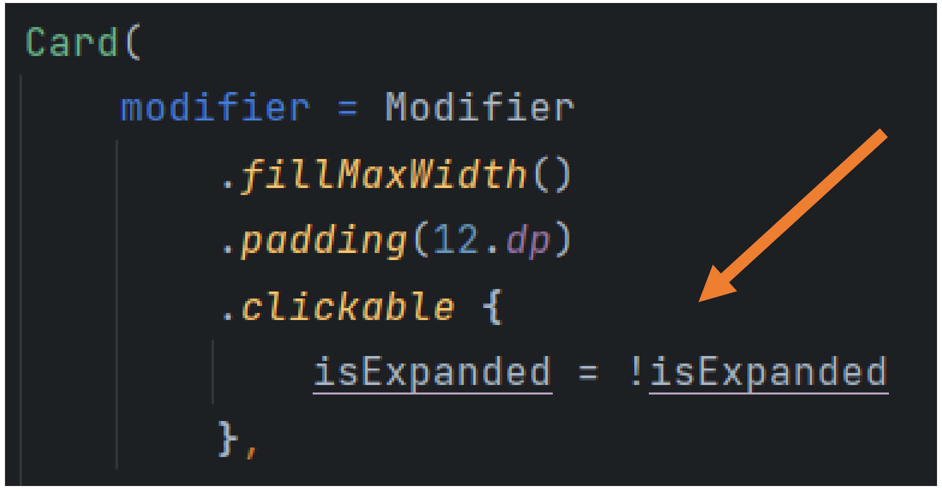
- Let's expand our card when it's clicked on

- For this we need to create a *MutableState<Boolean>*

- When it is clicked, we can set the value to true or false



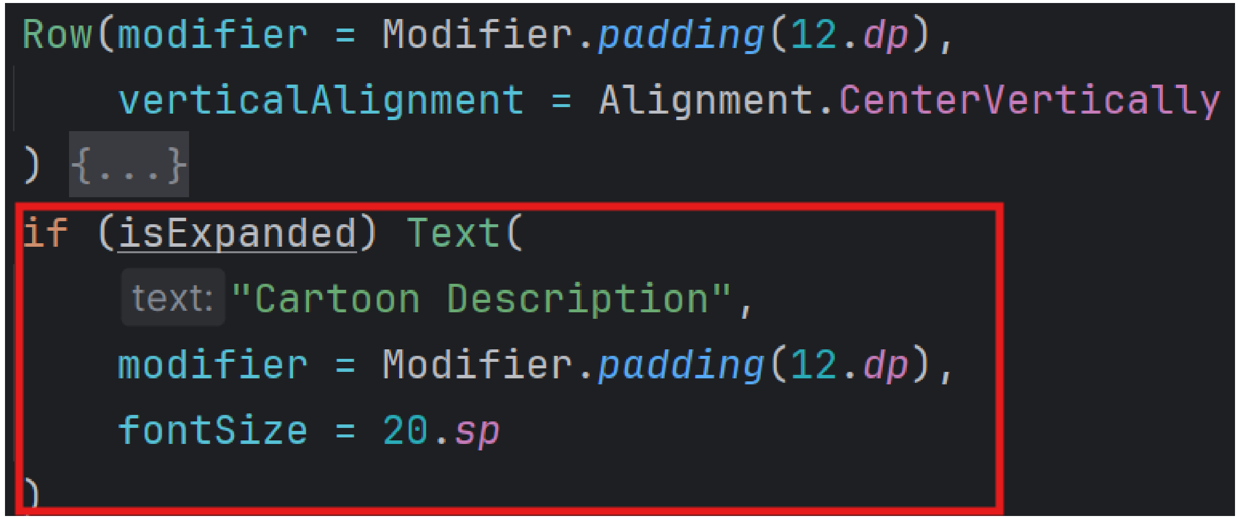
```
var isExpanded by remember {
    mutableStateOf( value: false)
}
```



```
Card(
    modifier = Modifier
        .fillMaxWidth()
        .padding(12.dp)
        .clickable {
            isExpanded = !isExpanded
        },
```

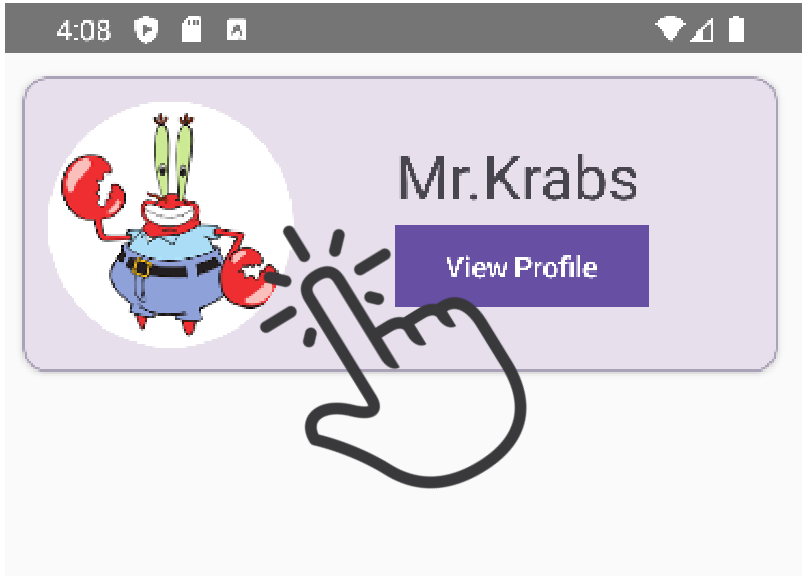
Composable visibility

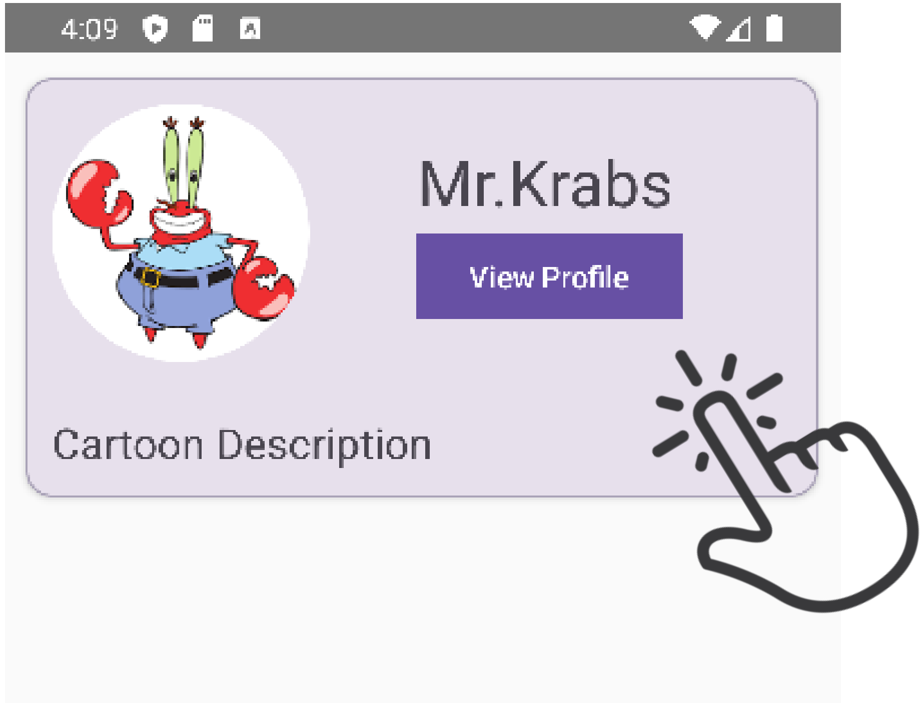
- Inside our Card, we check if *value* is true, then add a Text composable



```
Row(modifier = Modifier.padding(12.dp),
    verticalAlignment = Alignment.CenterVertically
) {...}
if (isExpanded) Text(
    text: "Cartoon Description",
    modifier = Modifier.padding(12.dp),
    fontSize = 20.sp
)
```

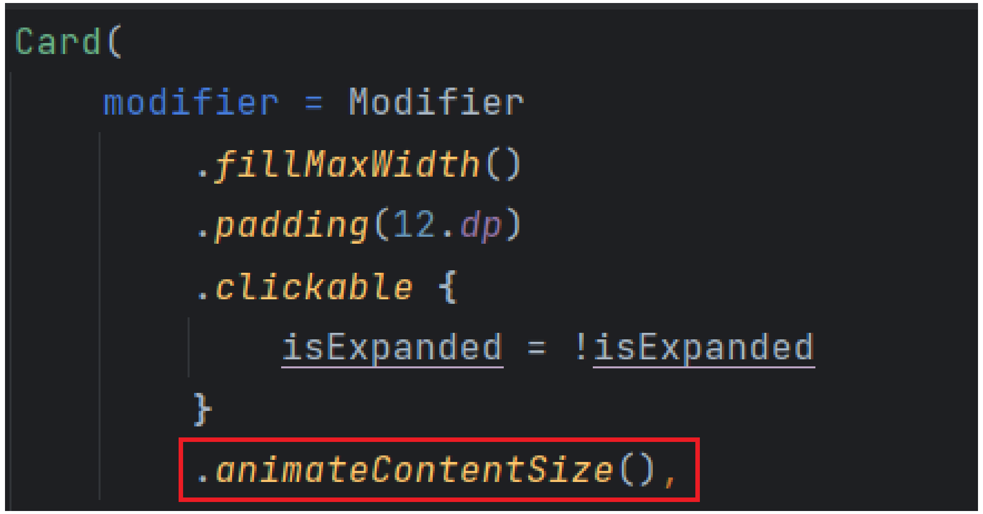
Composable visibility (cont.)





Animations

- Compose has some built in animations such as *animateContentSize*



```
Card(
    modifier = Modifier
        .fillMaxWidth()
        .padding(12.dp)
        .clickable {
            isExpanded = !isExpanded
        }
        .animateContentSize(),
```

- Expanding your card will have a smoother transition now