Lecture 7

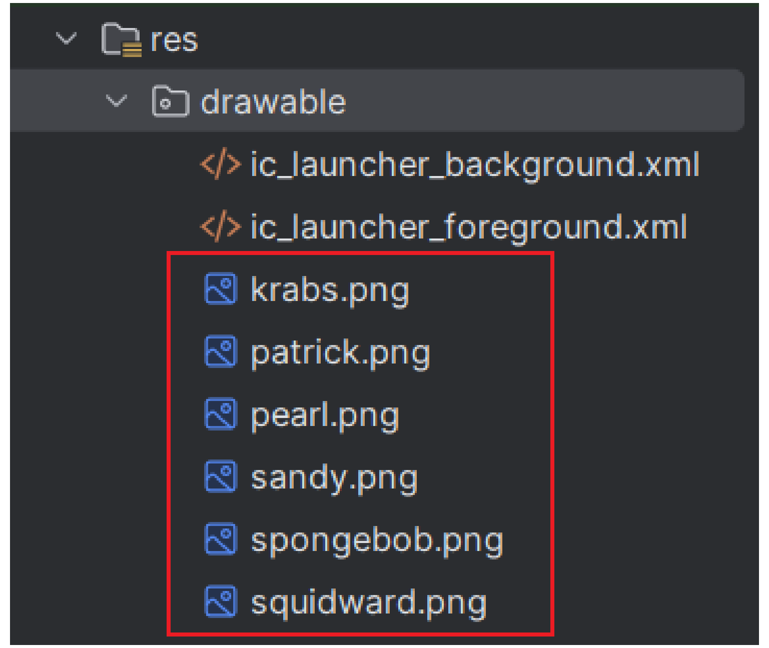
COMP 3717- Mobile Dev with Android Tech

第7讲

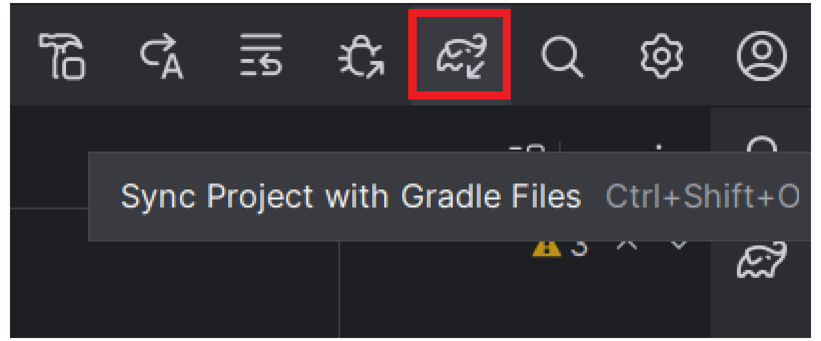
COMP 3717 - 使用Android技术进行移动开发

Displaying an Image

- To add an image to your app first drag one or more images over into your drawable folder
- After adding resources to your project, you should do a *Sync Project with Gradle Files*

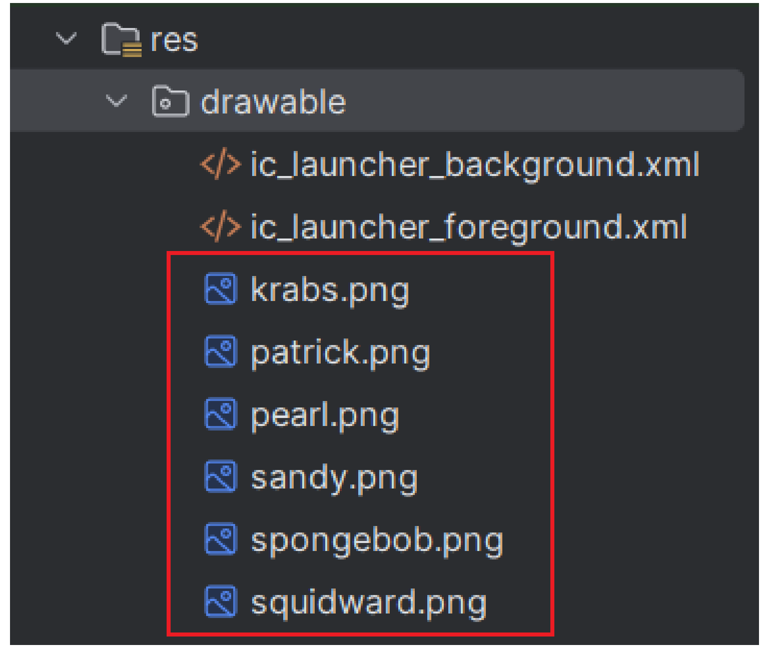


res
- drawable
  - ic_launcher_background.xml
  - ic_launcher_foreground.xml
  - krabs.png
  - patrick.png
  - pearl.png
  - sandy.png
  - spongebob.png
  - squidward.png

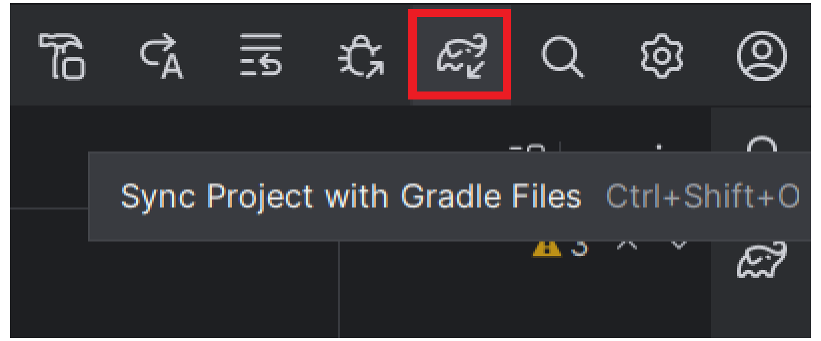


显示图像

- 要向应用中添加图像，请先将一个或多个图像拖入 drawable 文件夹
- 将资源添加到项目后，应执行 *Sync Project with GradleFiles*

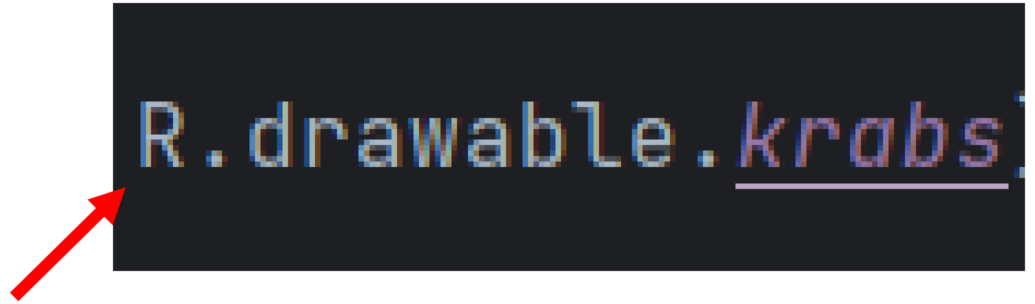


res
- drawable
  - ic_launcher_background.xml
  - ic_launcher_foreground.xml
  - krabs.png
  - patrick.png
  - pearl.png
  - sandy.png
  - spongebob.png
  - squidward.png



Displaying an Image (cont.)

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

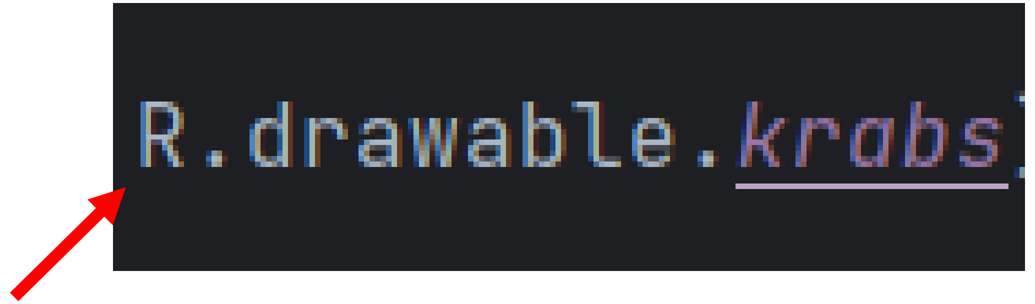


```
R.drawable.krabs
```

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

显示图像（续）

- R类使我们能够访问项目中的所有资源
  - 可绘制对象、字符串、字体、颜色、文件等。

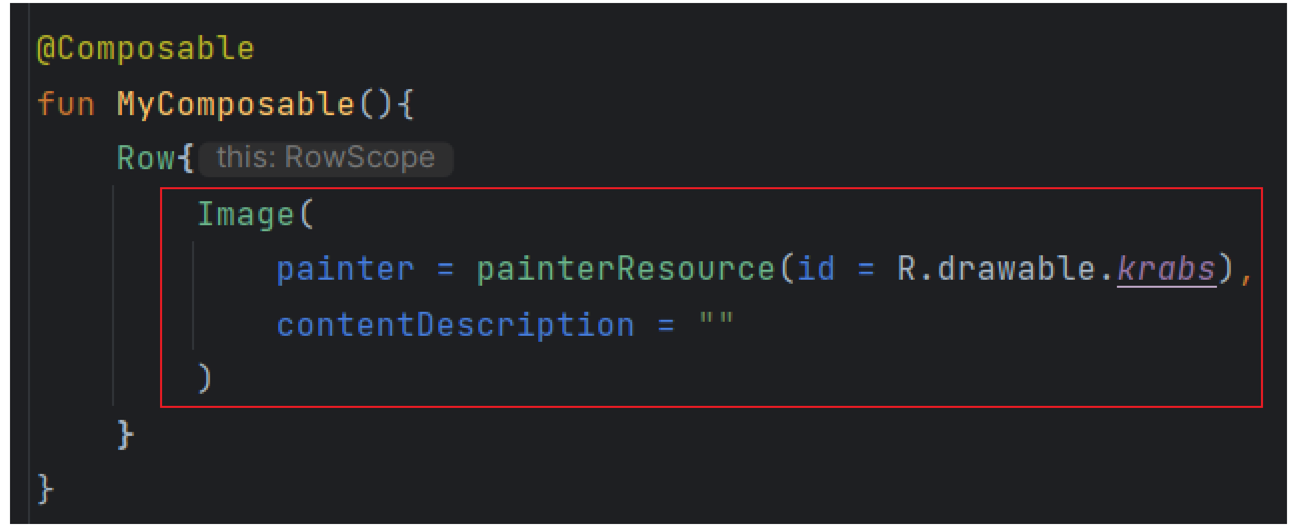


```
R.drawable.krabs
```

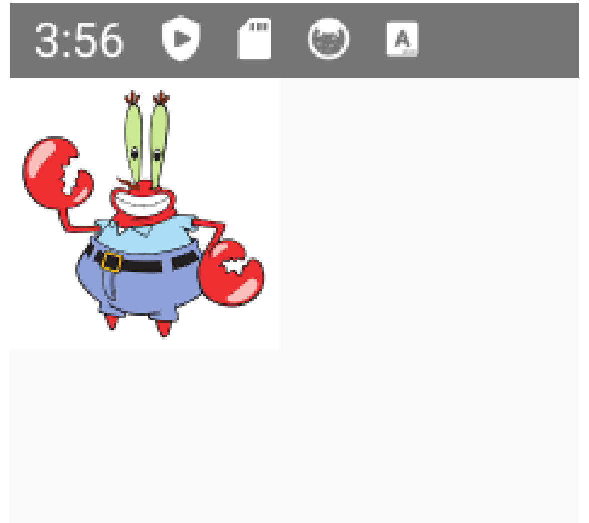
- 当我们通过R类访问资源时，它会返回一个作为整数的资源 ID

Displaying an Image (cont.)

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



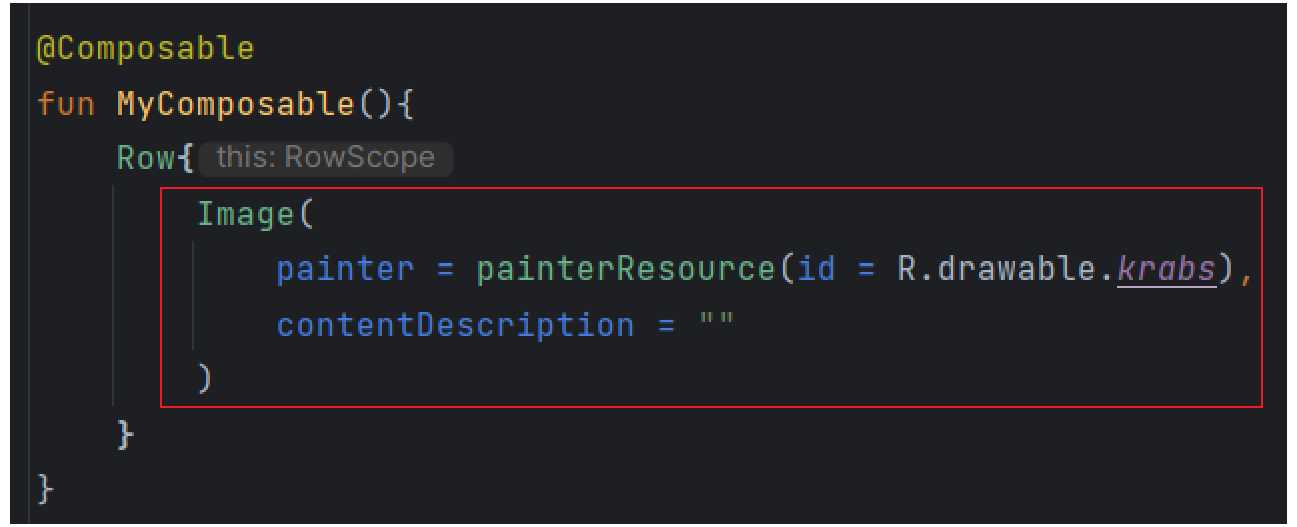
```
@Composable
fun MyComposable(){
    Row{ this: RowScope
        Image(
            painter = painterResource(id = R.drawable.krabs),
            contentDescription = ""
        )
    }
}
```



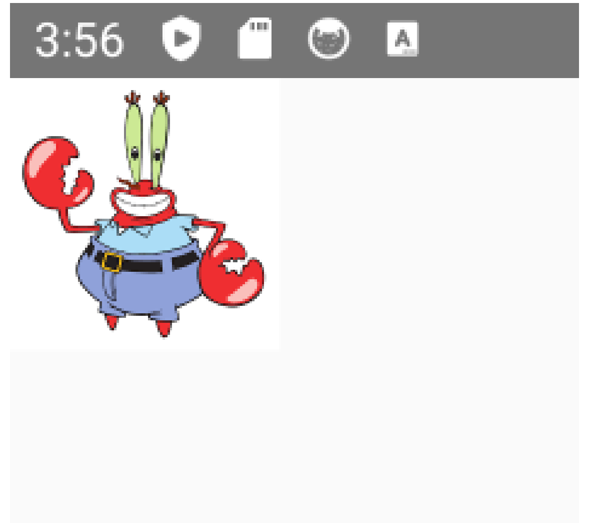
- The *painerResource* function takes in an id param as an Integer

显示图像（续）

- 使用两个必需参数创建一个 Image 可组合项
  - painter 和 contentDescription



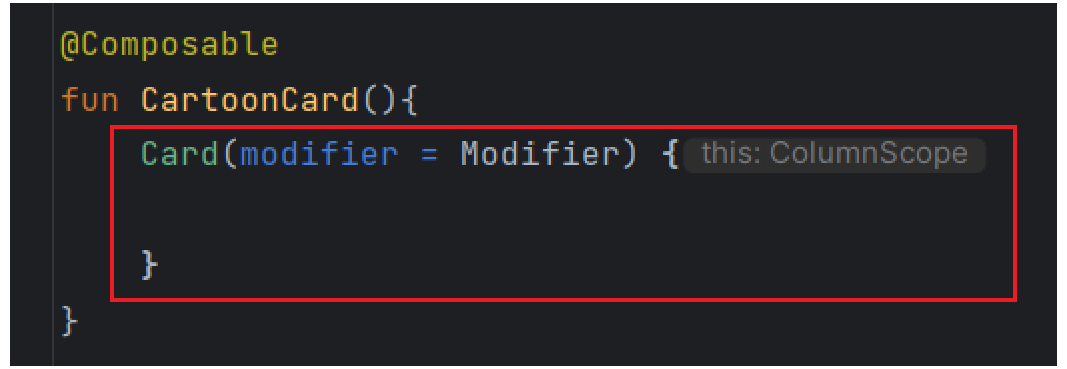
```
@Composable
fun MyComposable(){
    Row{ this: RowScope
        Image(
            painter = painterResource(id = R.drawable.krabs),
            contentDescription = ""
        )
    }
}
```



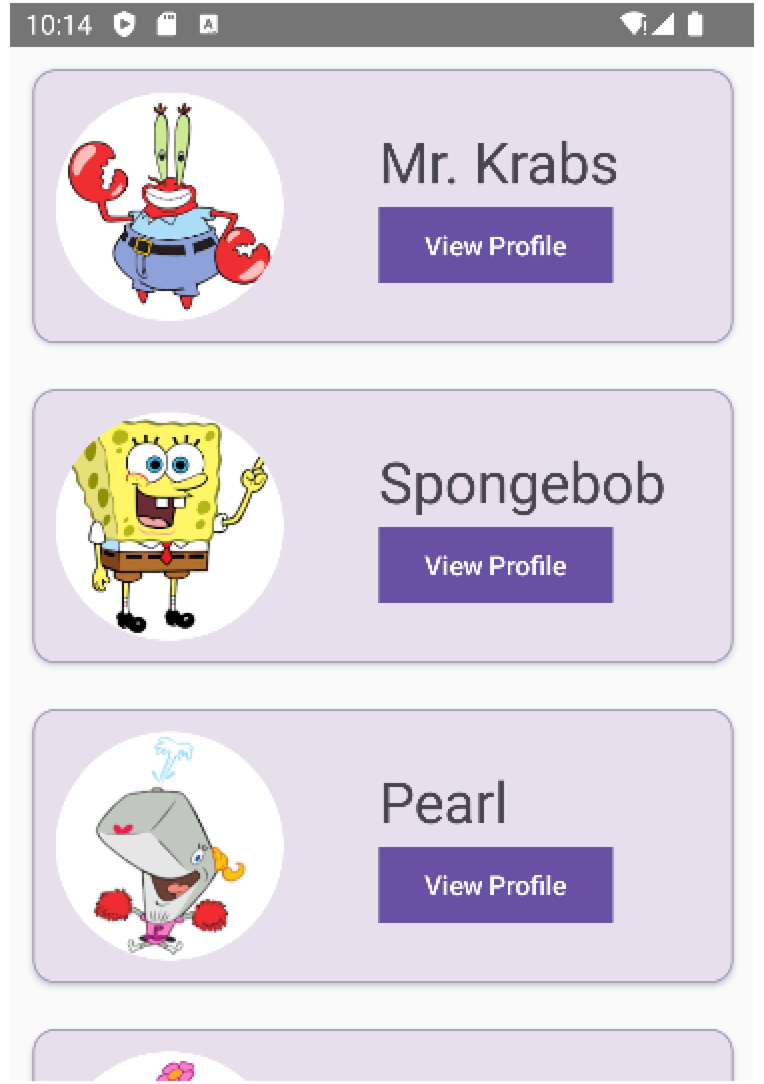
- *painerResource* 函数接收一个 Integer 类型的 id 参数

Card

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

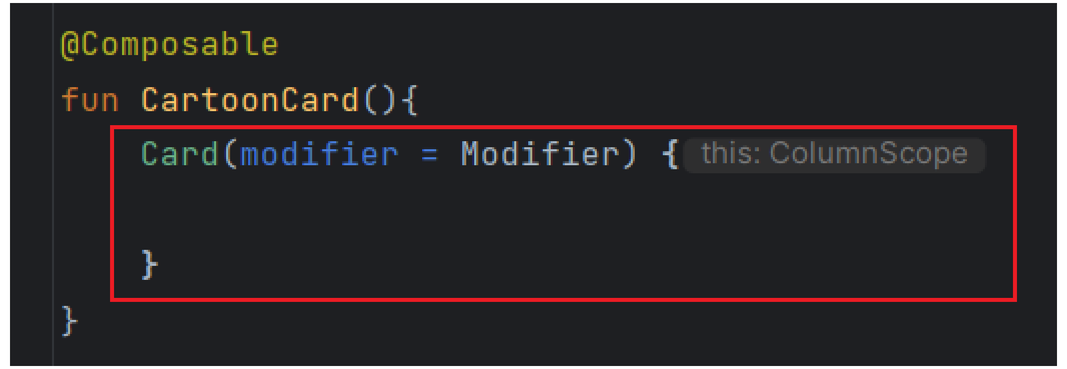


```
@Composable
fun CartoonCard(){
    Card(modifier = Modifier) { this: ColumnScope

    }
}
```

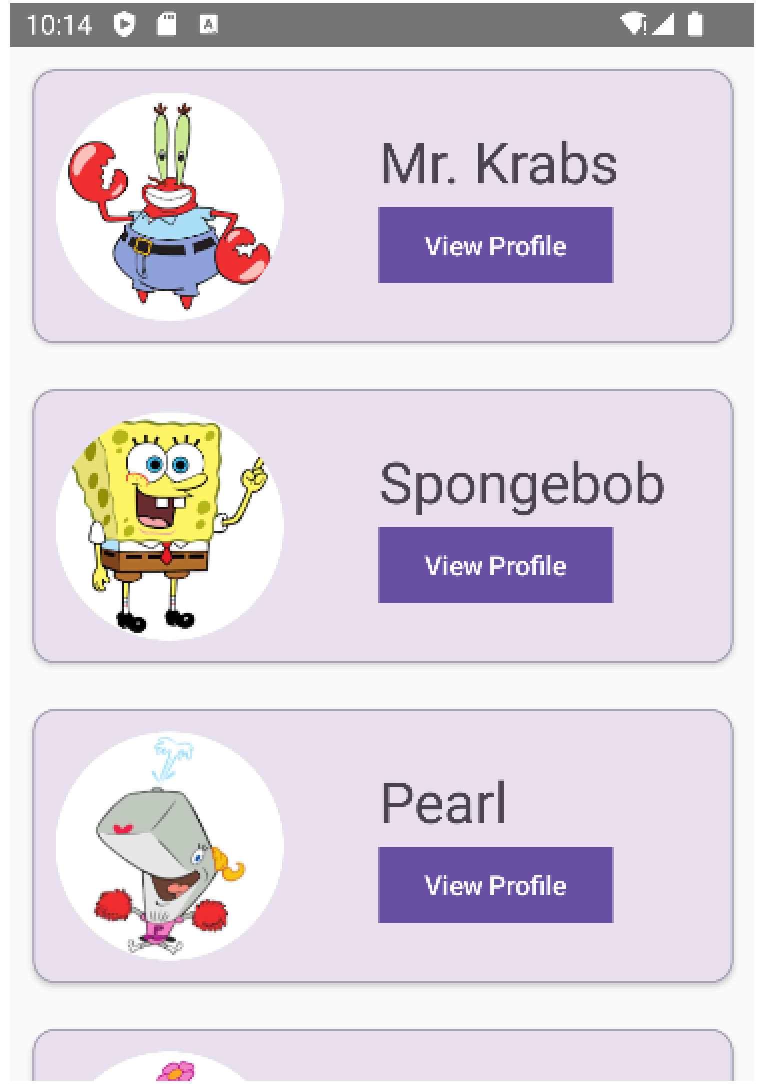


Card

- 卡片是一种小型容器，可向屏幕提供单个内容

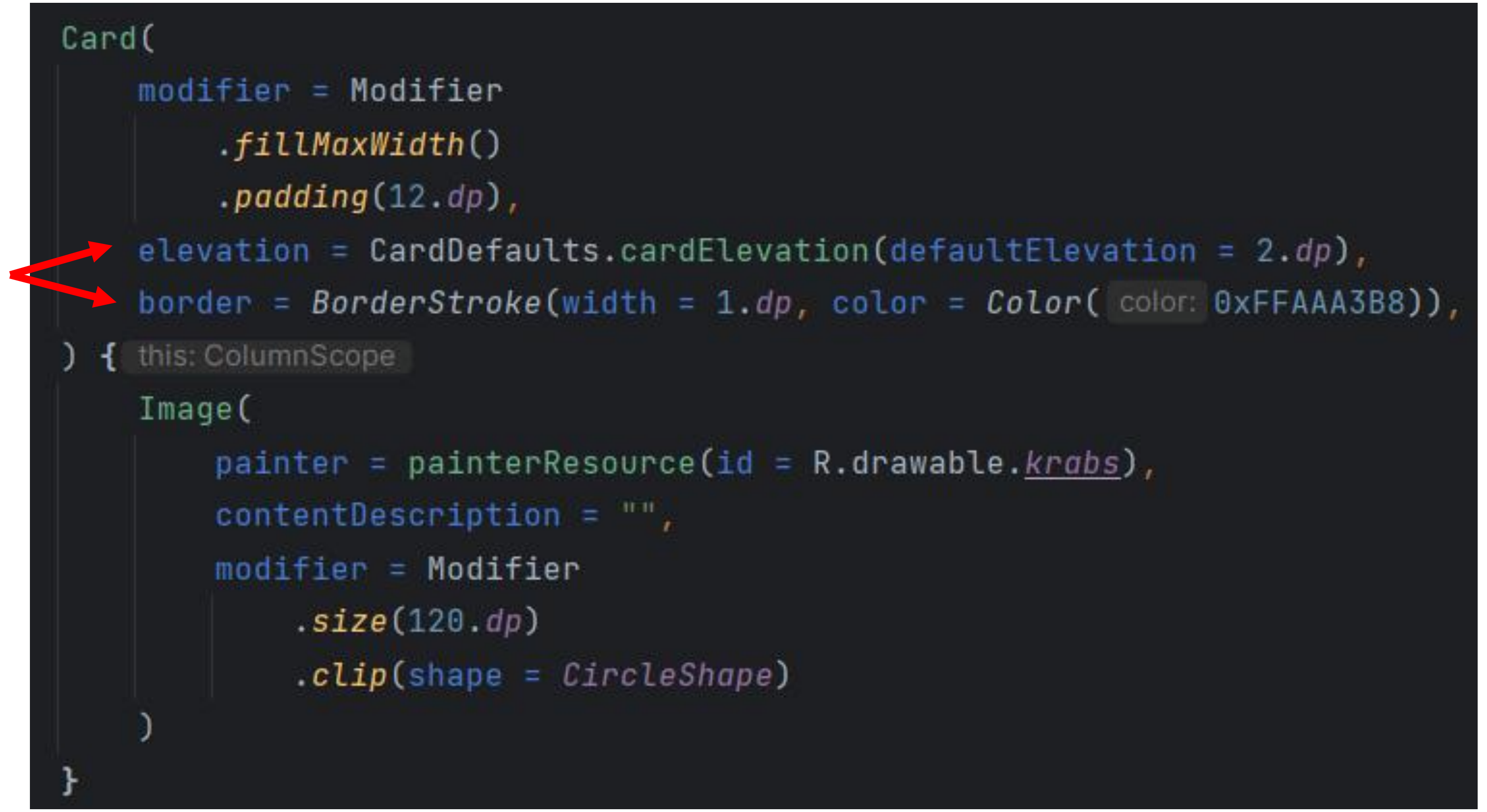


```
@Composable
fun CartoonCard(){
    Card(modifier = Modifier) { this: ColumnScope

    }
}
```

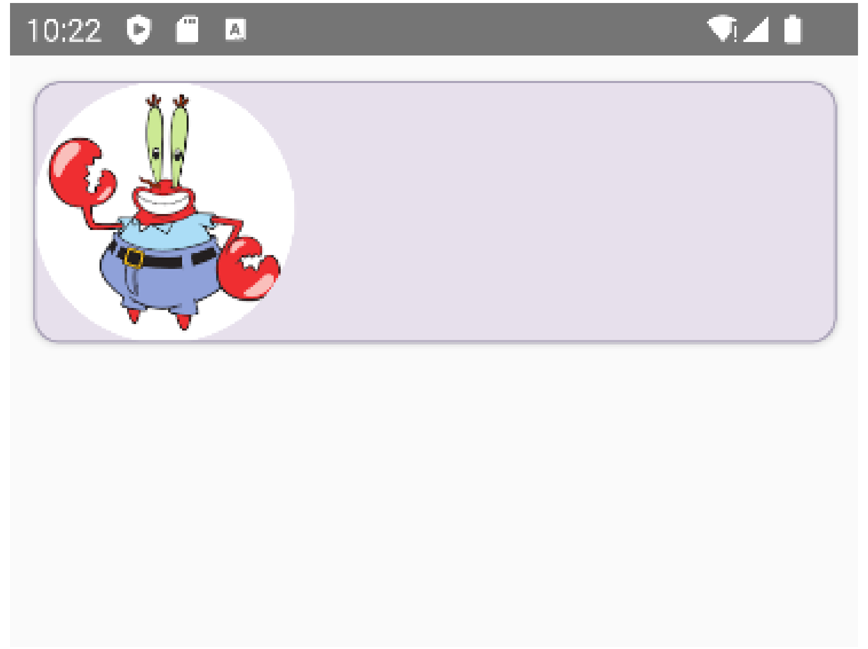


Card (cont.)

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

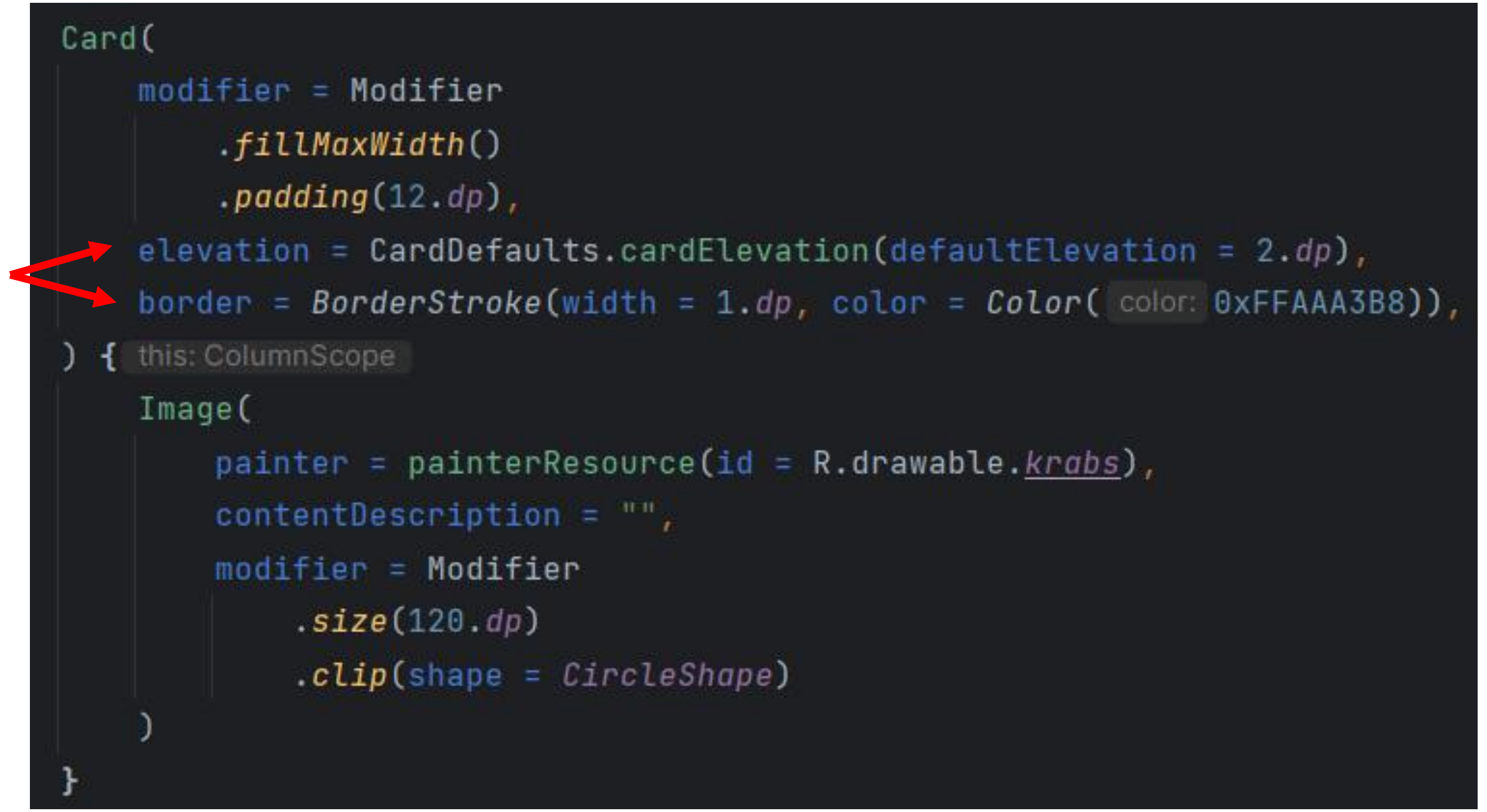


```
Card(
    modifier = Modifier
        .fillMaxWidth()
        .padding(12.dp),
    elevation = CardDefaults.cardElevation(defaultElevation = 2.dp),
    border = BorderStroke(width = 1.dp, color = Color( color: 0xFFAAA3B8)),
) { this: ColumnScope
    Image(
        painter = painterResource(id = R.drawable.krabs),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

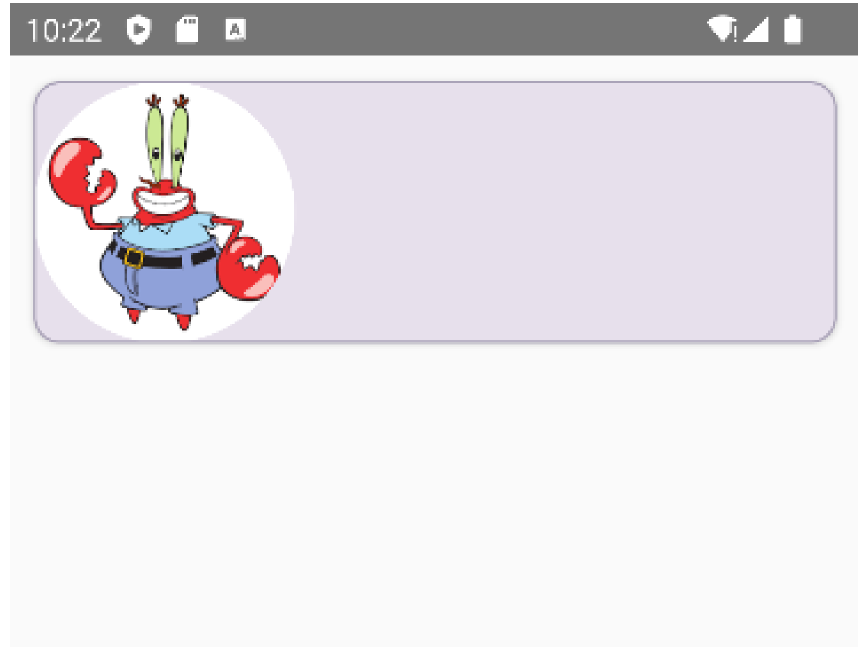


卡片（续）

- 卡片具有默认颜色，以及 高度和 边框 参数



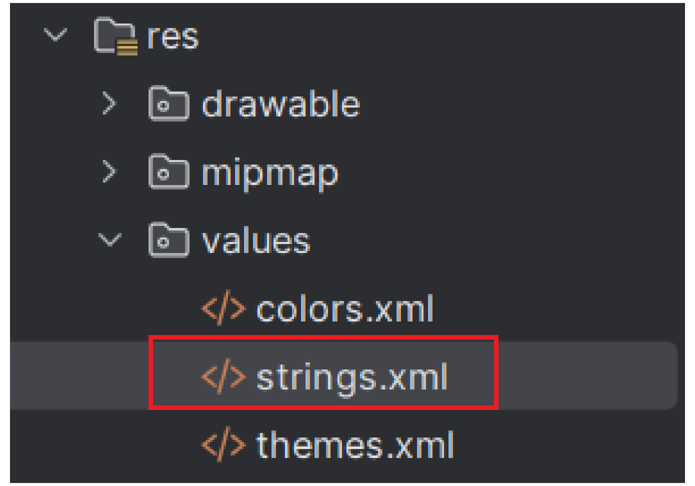
```
Card(
    modifier = Modifier
        .fillMaxWidth()
        .padding(12.dp),
    elevation = CardDefaults.cardElevation(defaultElevation = 2.dp),
    border = BorderStroke(width = 1.dp, color = Color( color: 0xFFAAA3B8)),
) { this: ColumnScope
    Image(
        painter = painterResource(id = R.drawable.krabs),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



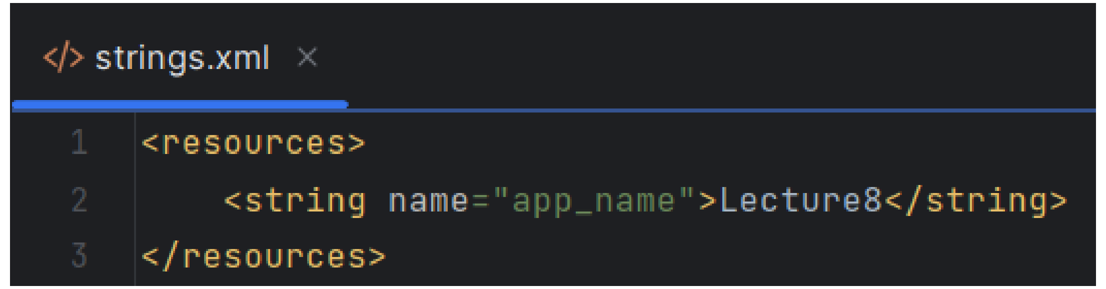
String resources

- An xml resource that provides text strings for your application

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



```
res
  drawable
  mipmap
  values
    colors.xml
    strings.xml
    themes.xml
```



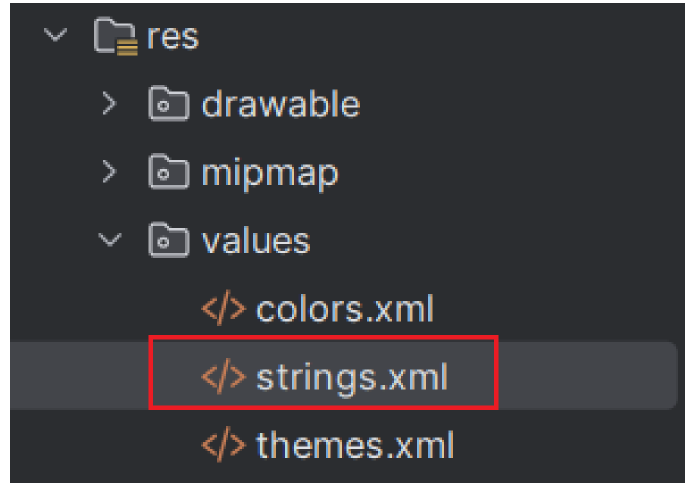
strings.xml

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

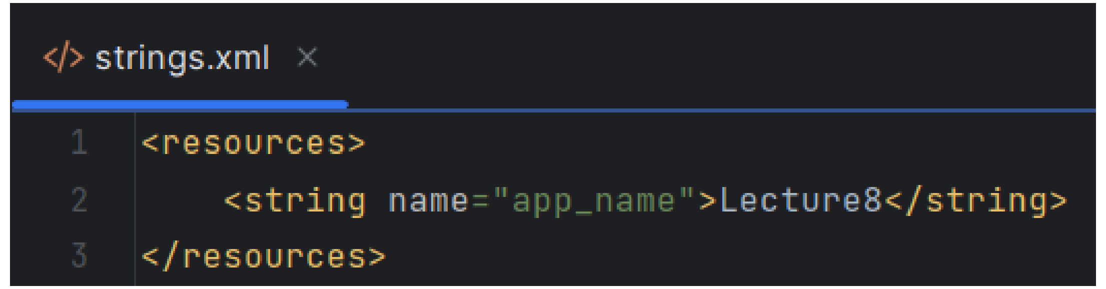
字符串资源

- 一个为您的应用程序提供文本字符串的 XML 资源

- 可以存储单个字符串或字符串数组



```
res
  drawable
  mipmap
  values
    colors.xml
    strings.xml
    themes.xml
```

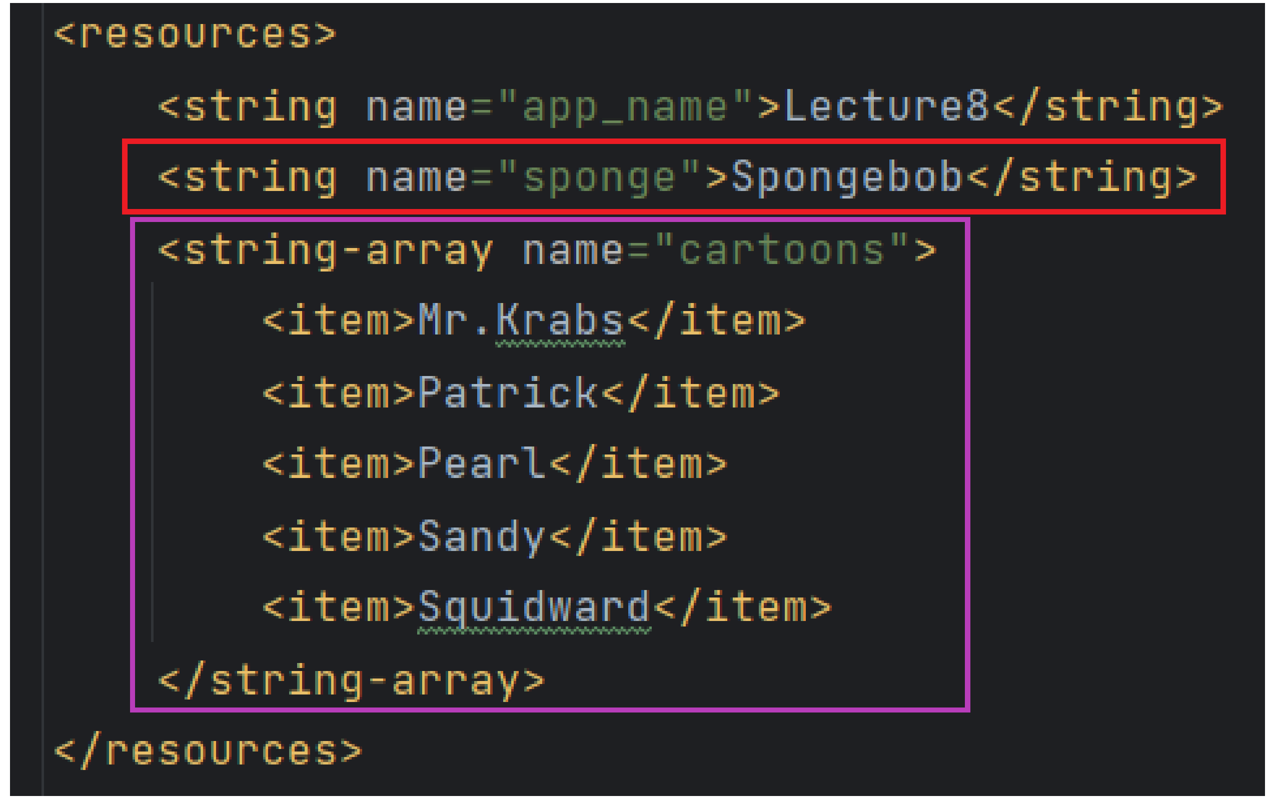


strings.xml

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

String resources (cont.)

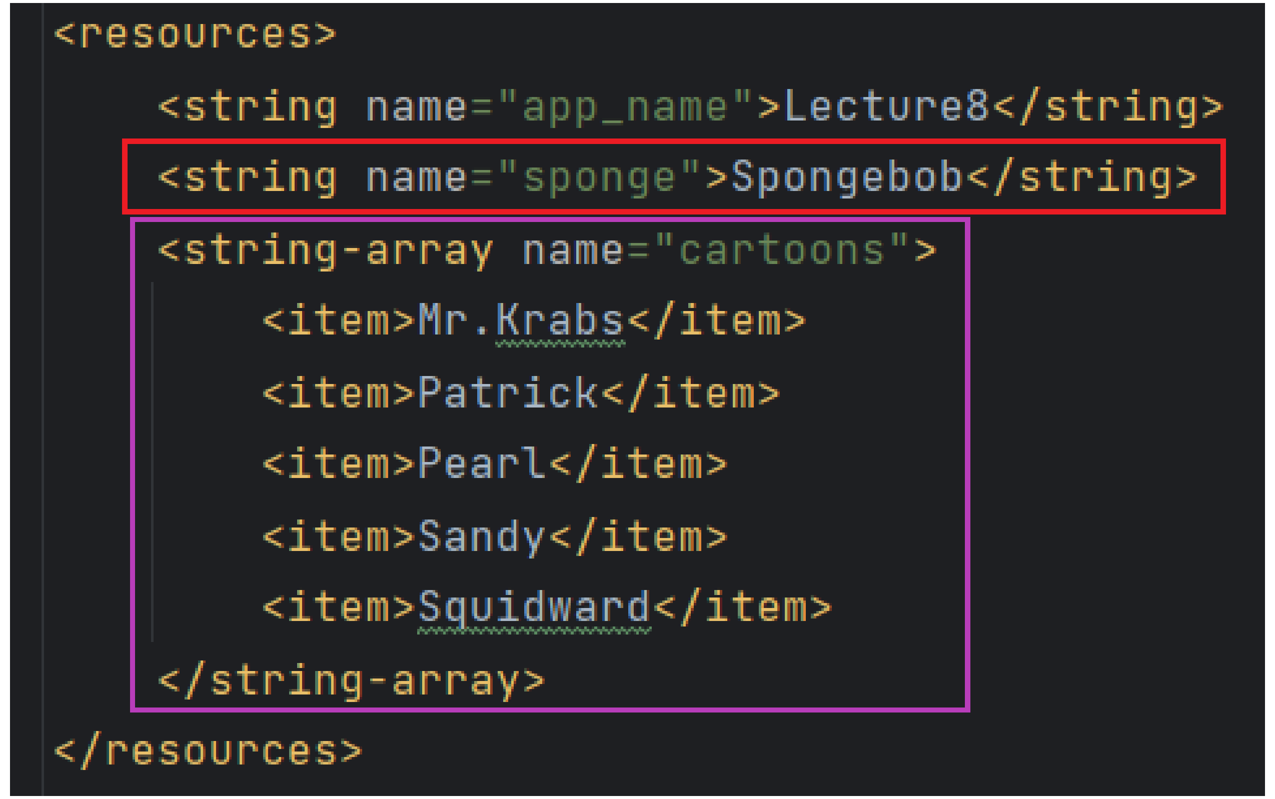
- Here I added a single string with the id sponge and a string array with the id cartoons



```
<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>
```

字符串资源（续）

- 在这里，我添加了一个 ID 为 sponge 的字符串和一个 ID 为 cartoons 的字符串数组

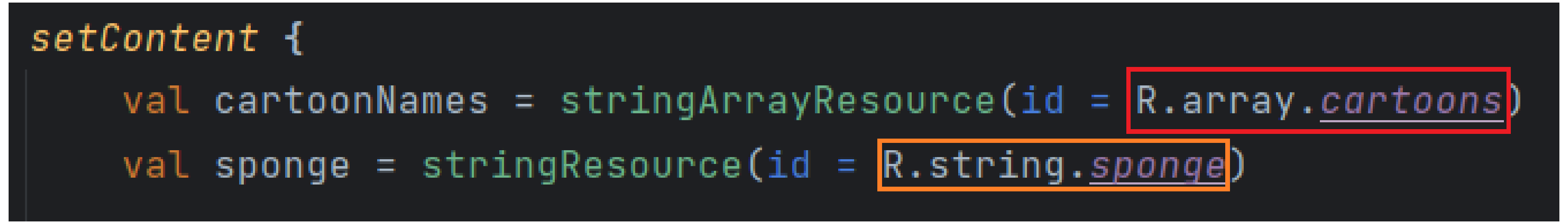


```
<resources>
    <string name="app_name">Lecture8</string>
    <string name="sponge">Spongebob</string>
    <string-array name="cartoons">
        <item>Mr.Krabs</item>
        <item>Patrick</item>
        <item>Pearl</item>
        <item>Sandy</item>
        <item>Squidward</item>
    </string-array>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

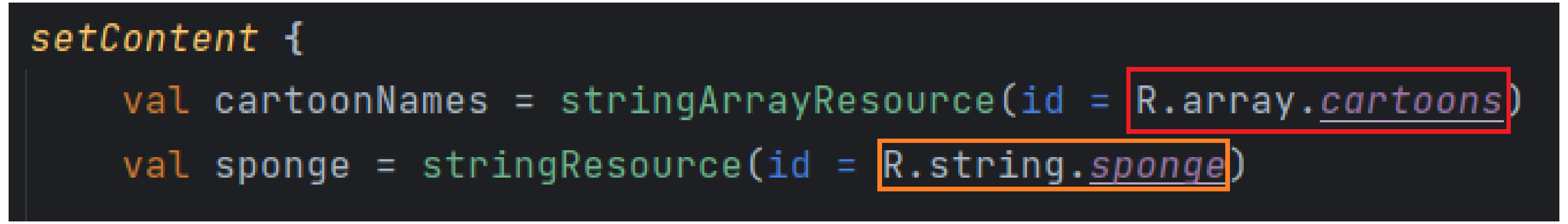


```
setContent {
    val cartoonNames = stringArrayResource(id = R.array.cartoons)
    val sponge = stringResource(id = R.string.sponge)
```

字符串资源（续）

- 要获取字符串资源，可以使用可组合项
  - stringArrayResource，或者
  - stringResource

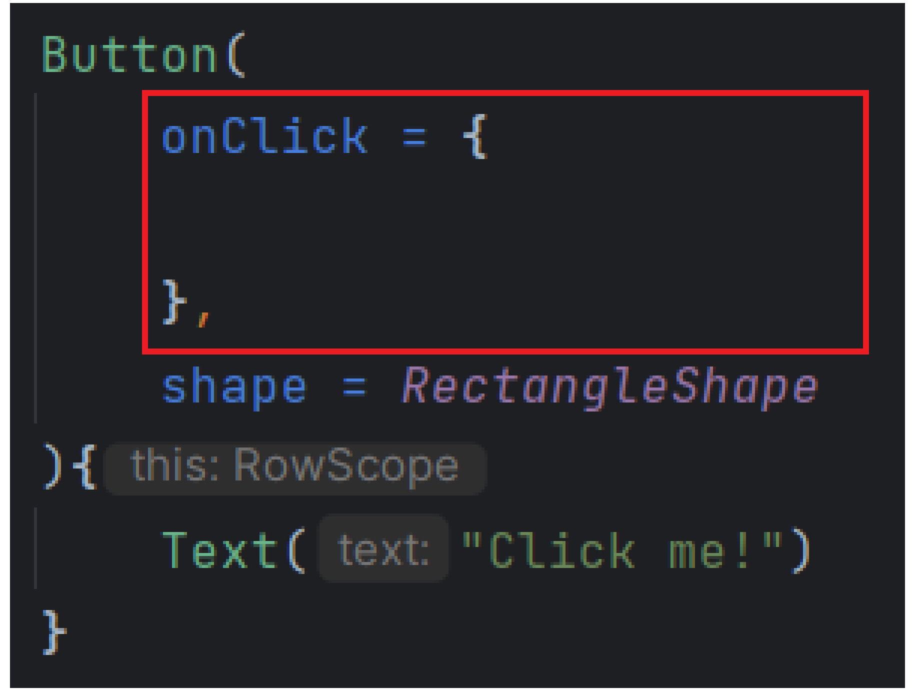
- 要查找特定的 ID，我们可以使用 R 类
  - R.array 用于字符串数组
  - R.string 用于单个字符串



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

Button

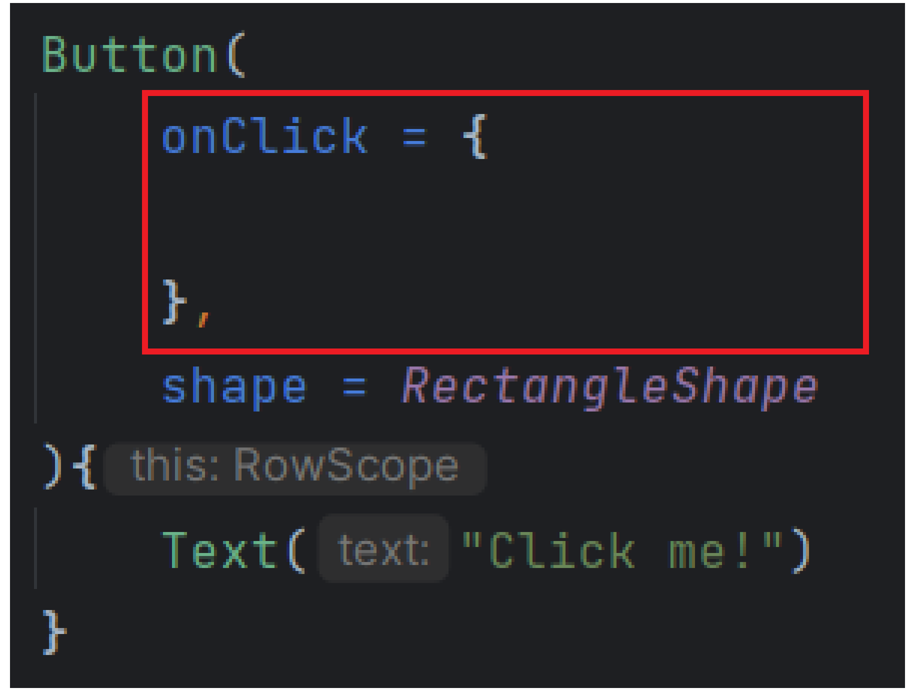
- A button has a *onClick* event callback
- What do we want to do when the button is clicked?



```
Button(
    onClick = {

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

按钮

- 一个按钮具有 *onClick* 事件回调
- 我们希望在
  按钮被点击时执行什么
  操作?



```
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

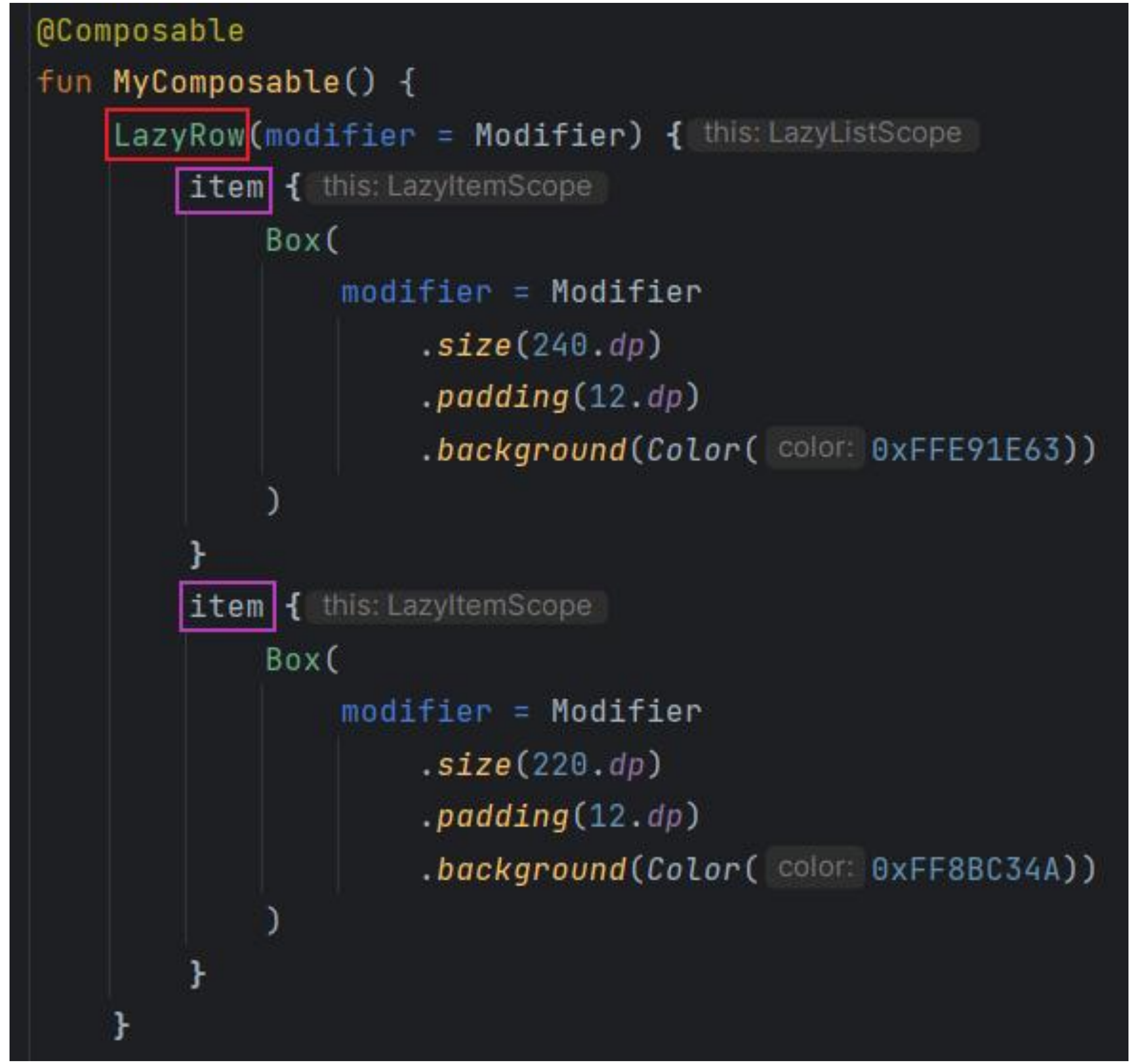
列表

- 我们经常需要滚动浏览元素
  - 也许我们无法将所有元素都放入想要的区域中

- 使用*LazyRow*和*LazyColumn* 可以处理大量数据的列表
  - 它们仅渲染屏幕上可见的元素，因此效率更高

Lists (cont.)

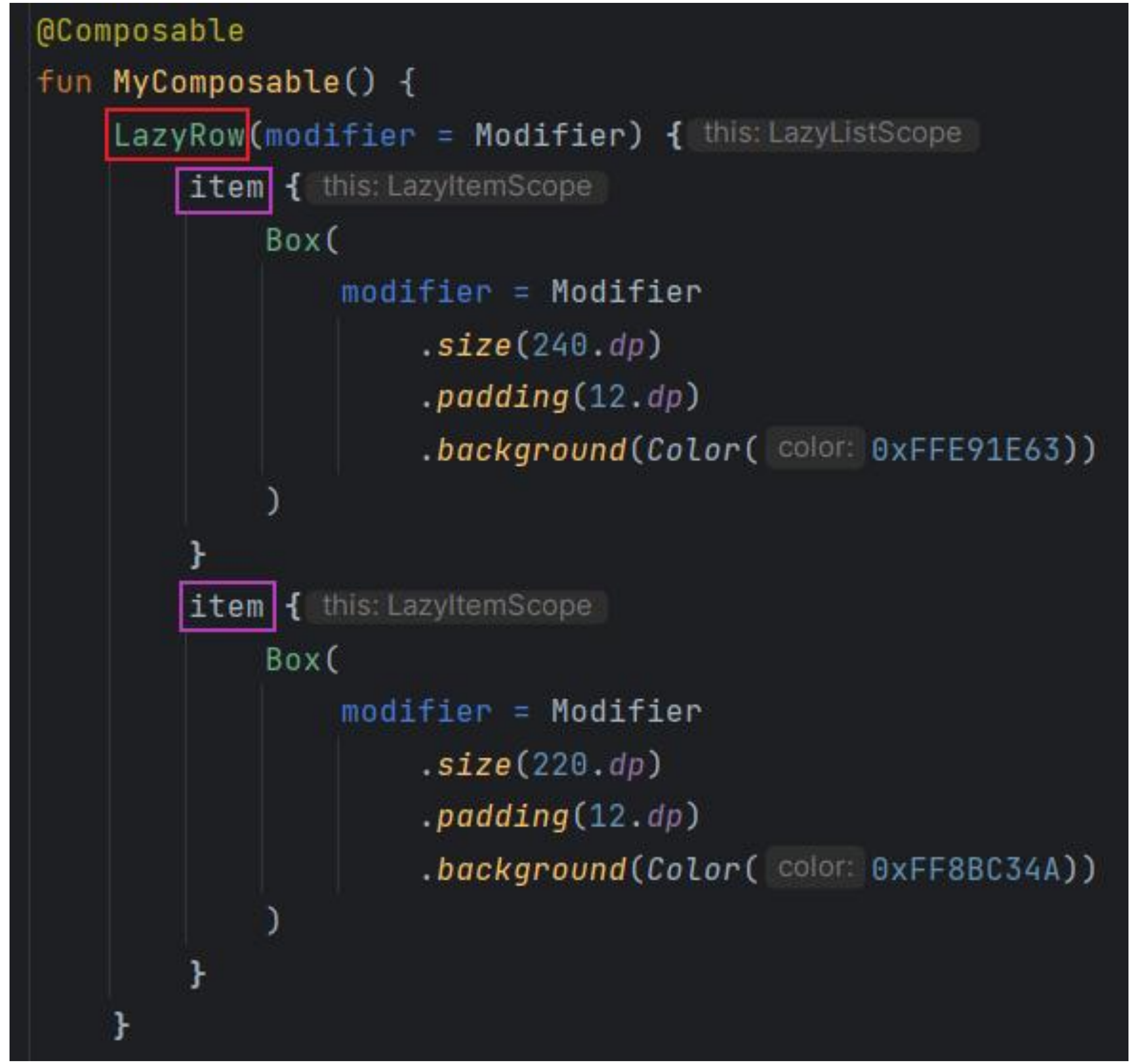
- When using a *LazyColumn* or *LazyRow*, just wrap the children with item



```
@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))
            )
        }
    }
```

列表（续）

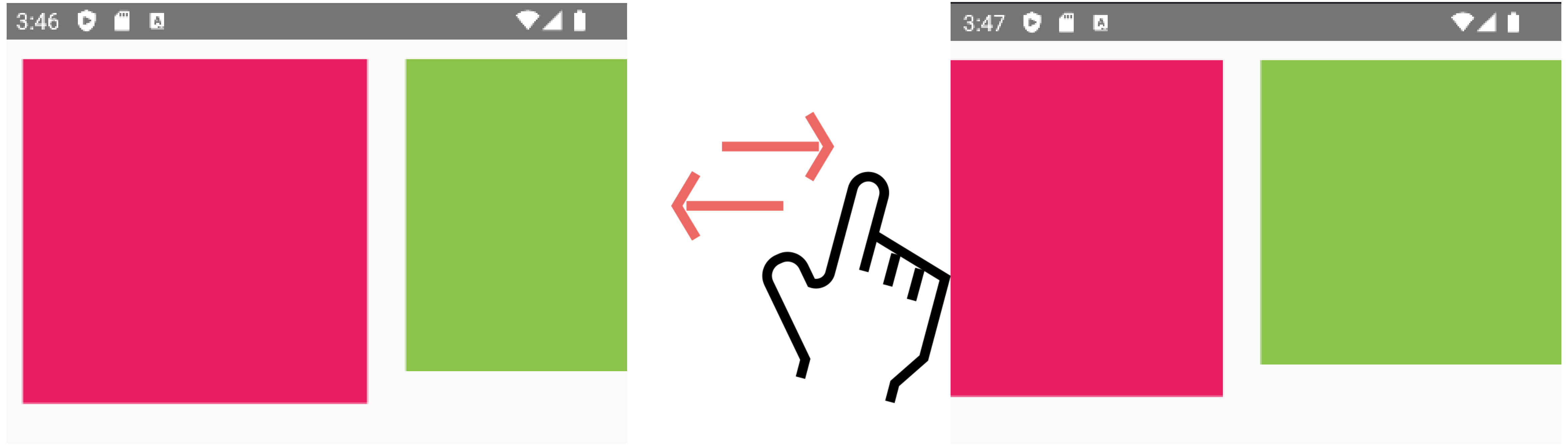
- 使用 *LazyColumn*或 *LazyRow*时，只需将子项用 item 包裹即可



```
@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))
            )
        }
    }
```

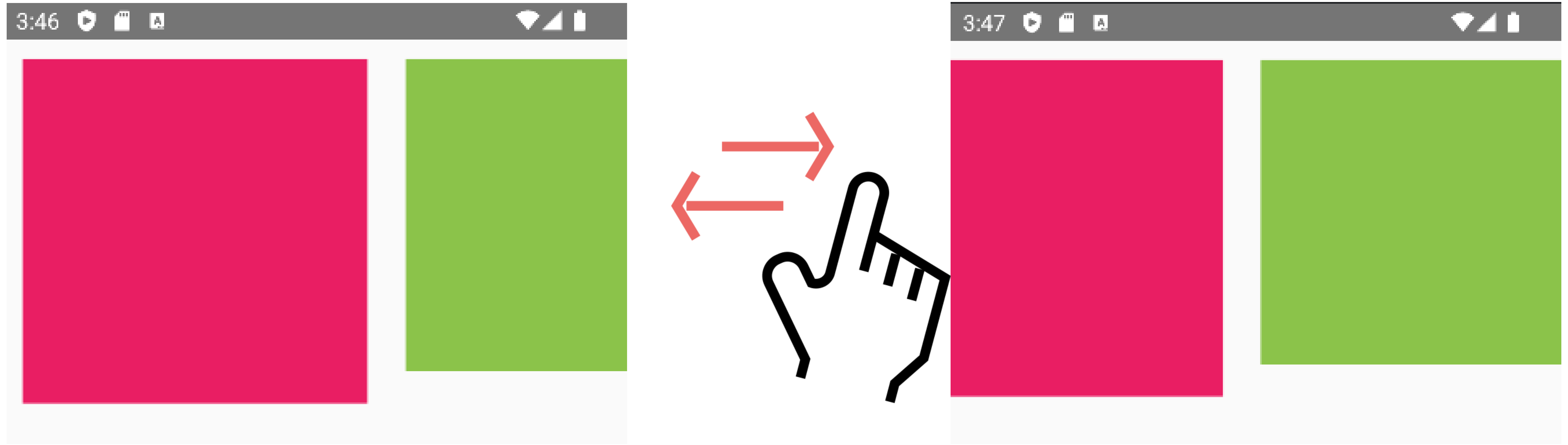
Lists (cont.)

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



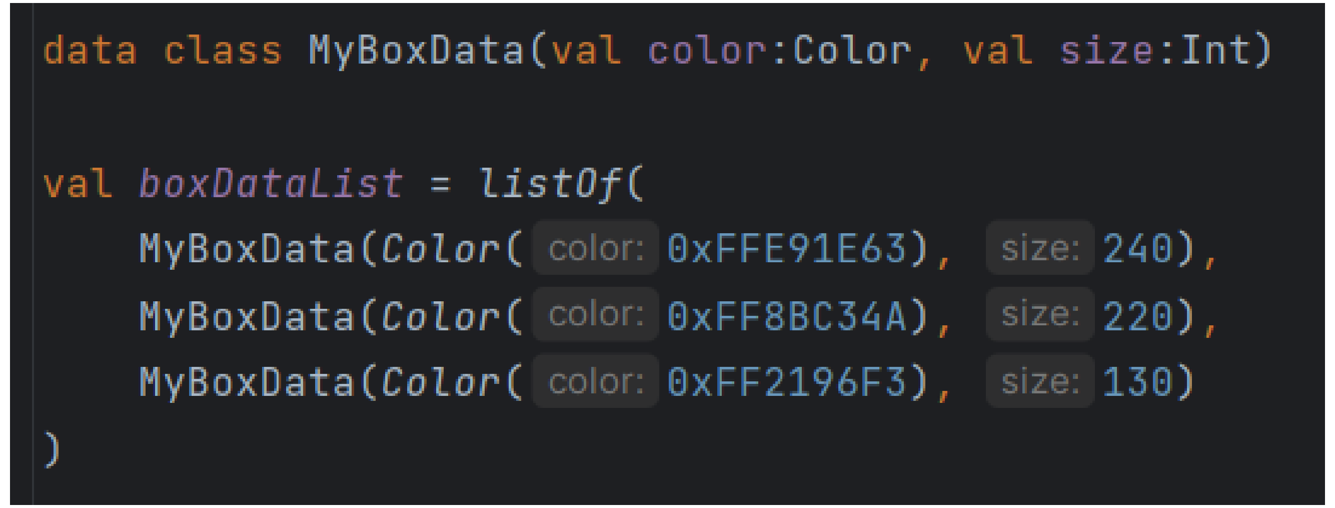
列表（续）

- 现在我可以在我的 LazyRow 中水平滚动这两个元素



Lists (cont.)

- Usually, you are working with lists of data

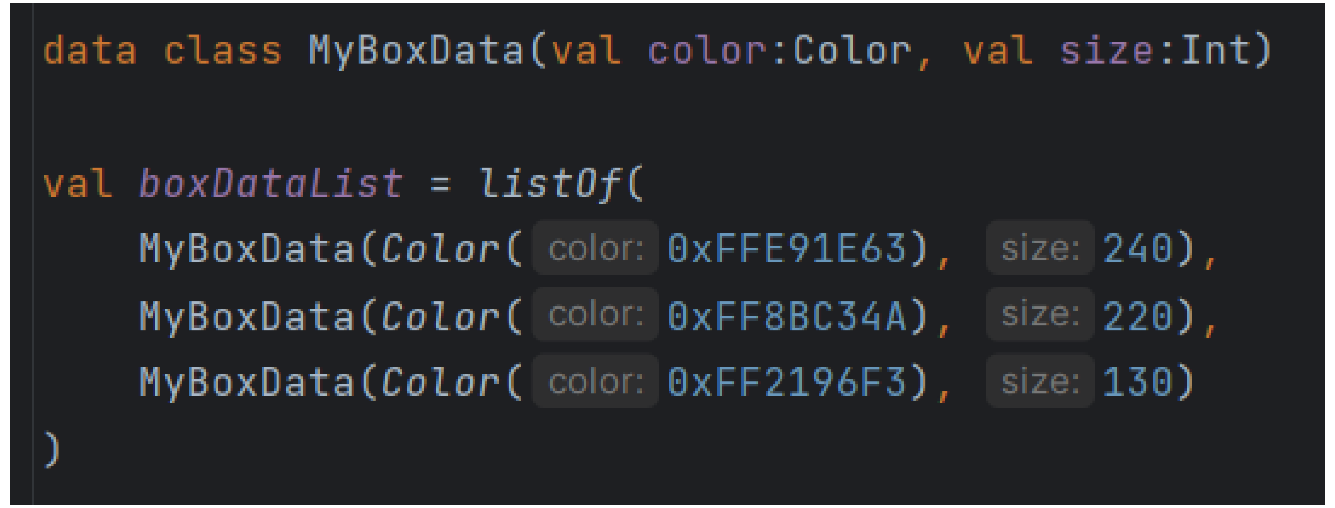


```
data class MyBoxData(val color:Color, val size:Int)

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

列表（续）

- 通常，你正在处理的是数据列表

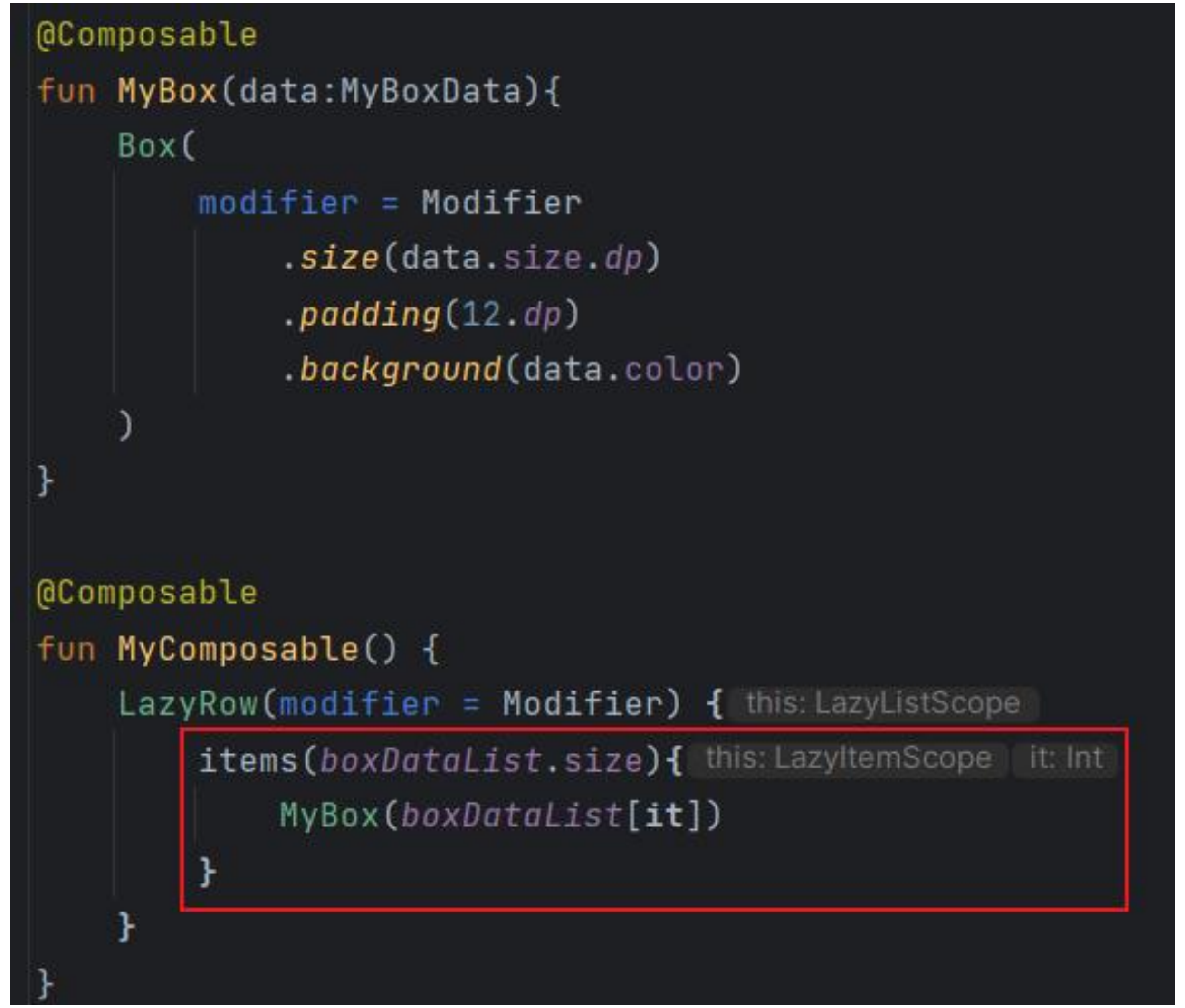


```
data class MyBoxData(val color:Color, val size:Int)

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

- 例如，此类数据类包含一个颜色和一个整数

Lists (cont.)

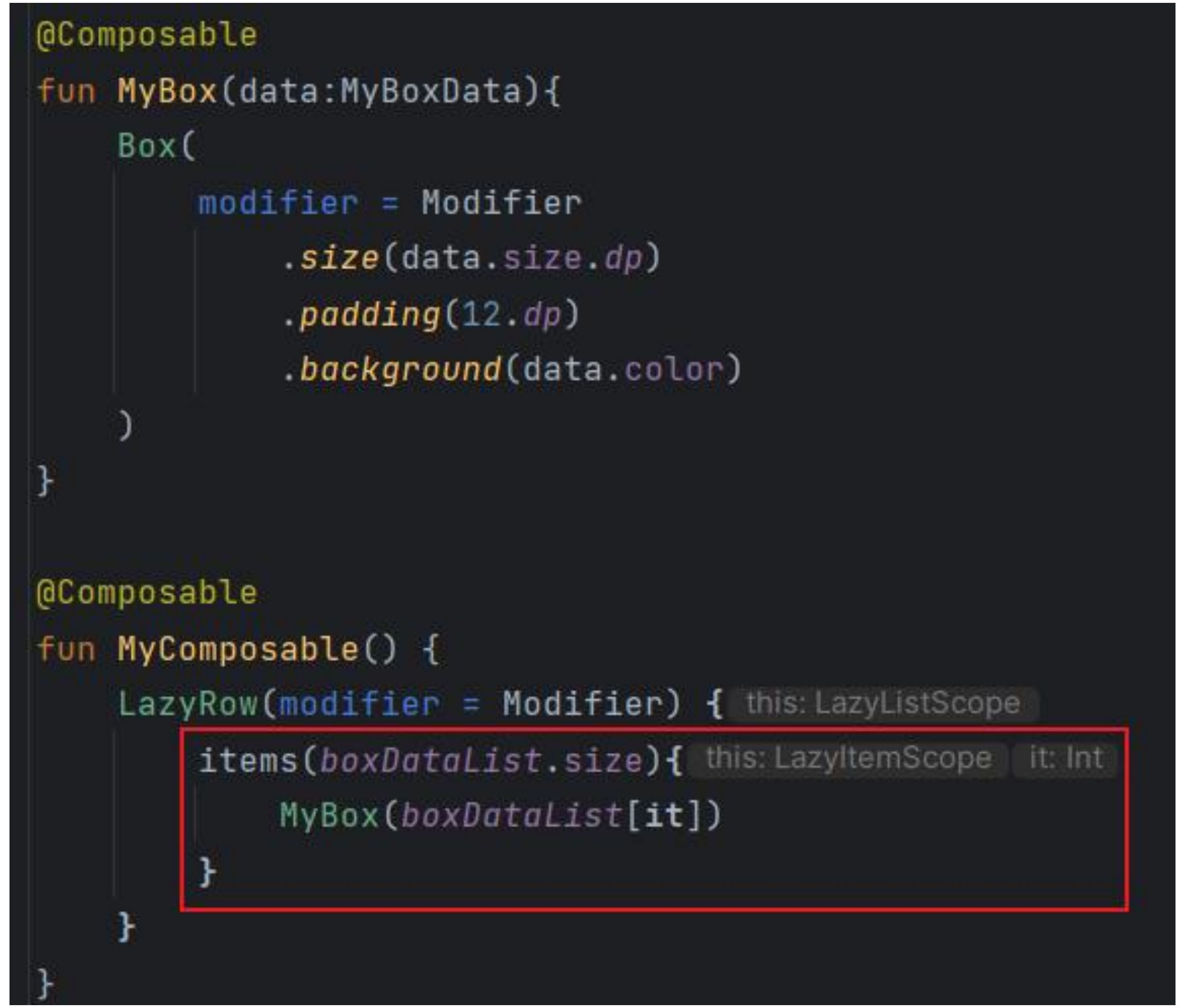
- Instead of repeating item we can use *items*, and use our list of data



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

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

列表（续）

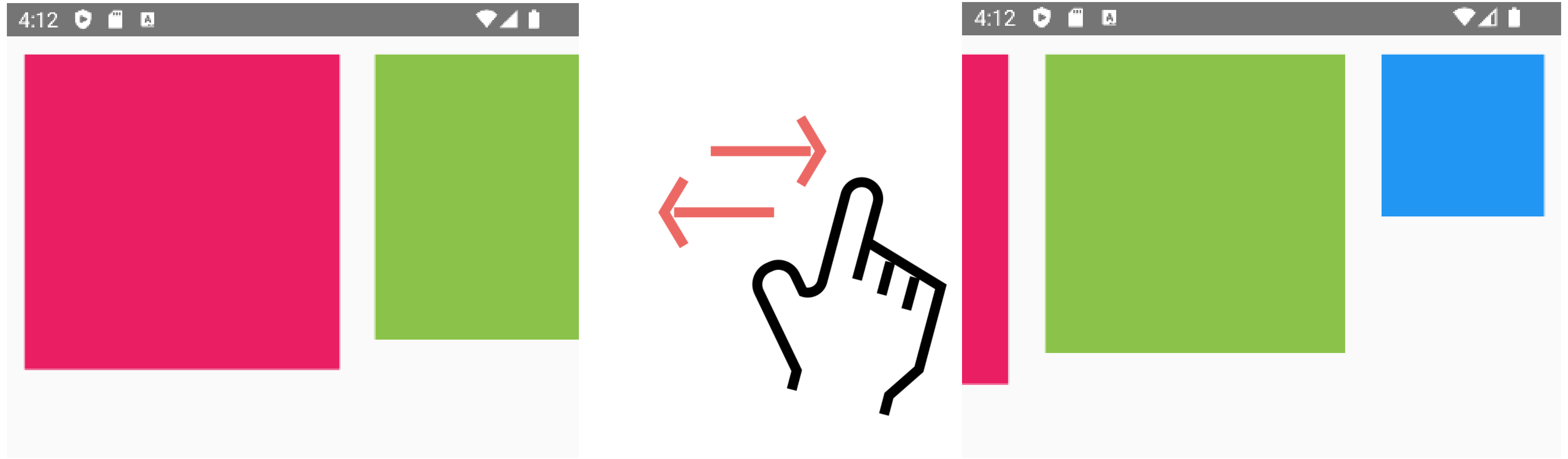
- 我们可以不用重复项
  而使用*items*，并使用我们的数据列表



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

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

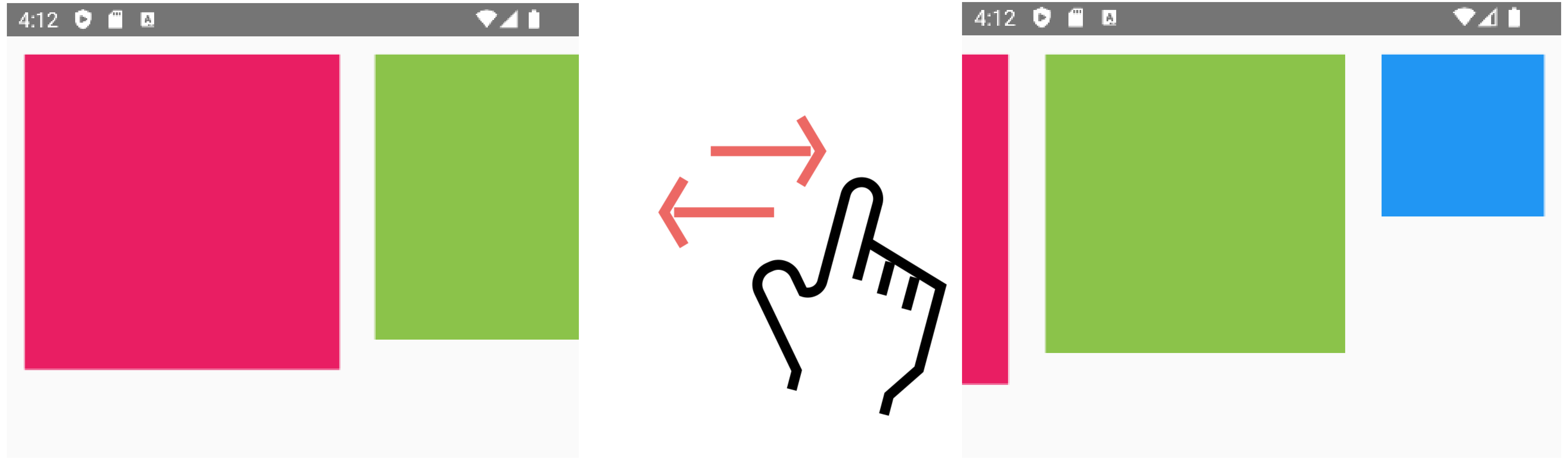
Lists (cont.)

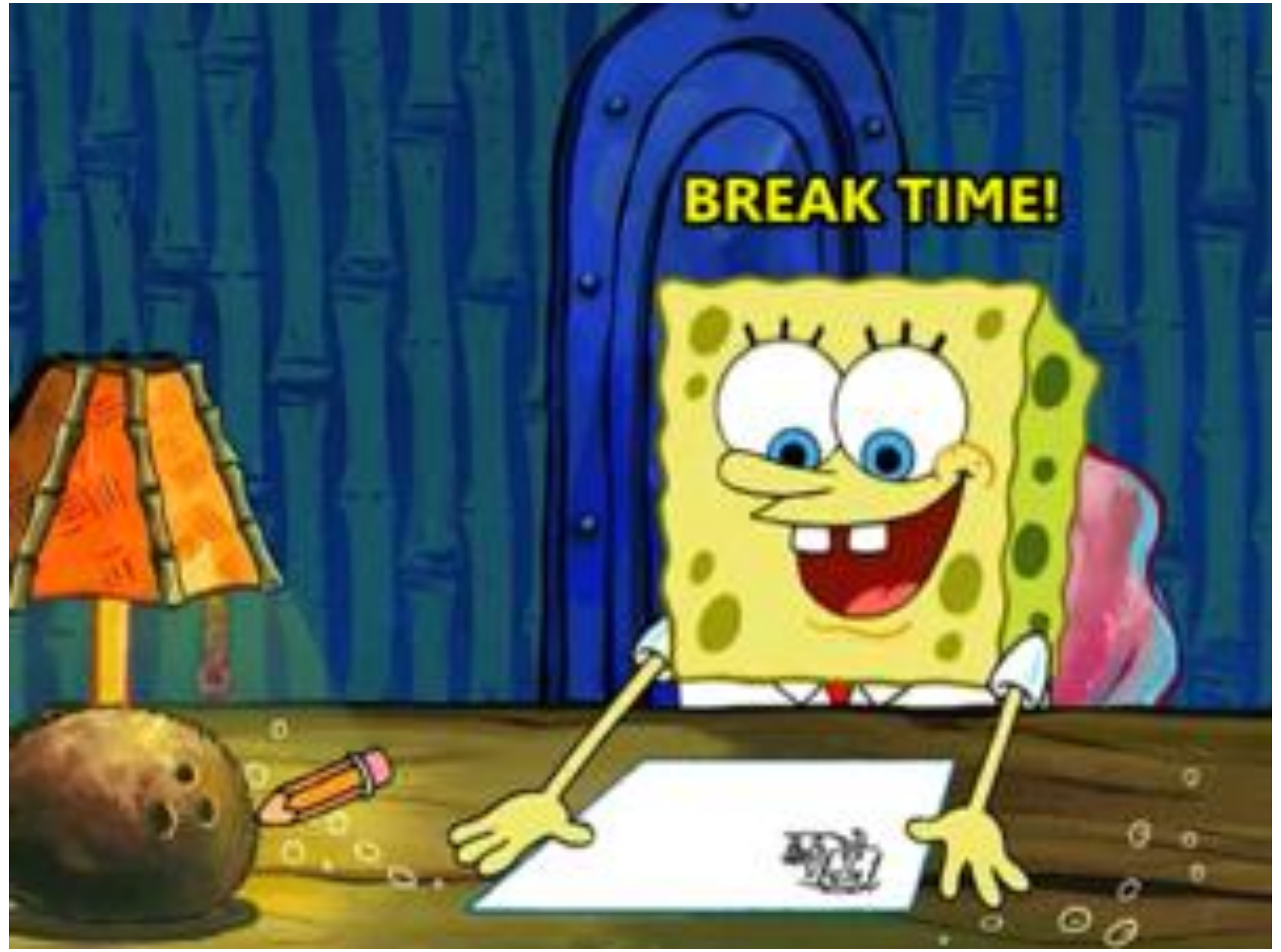
- Now I can scroll through all our data

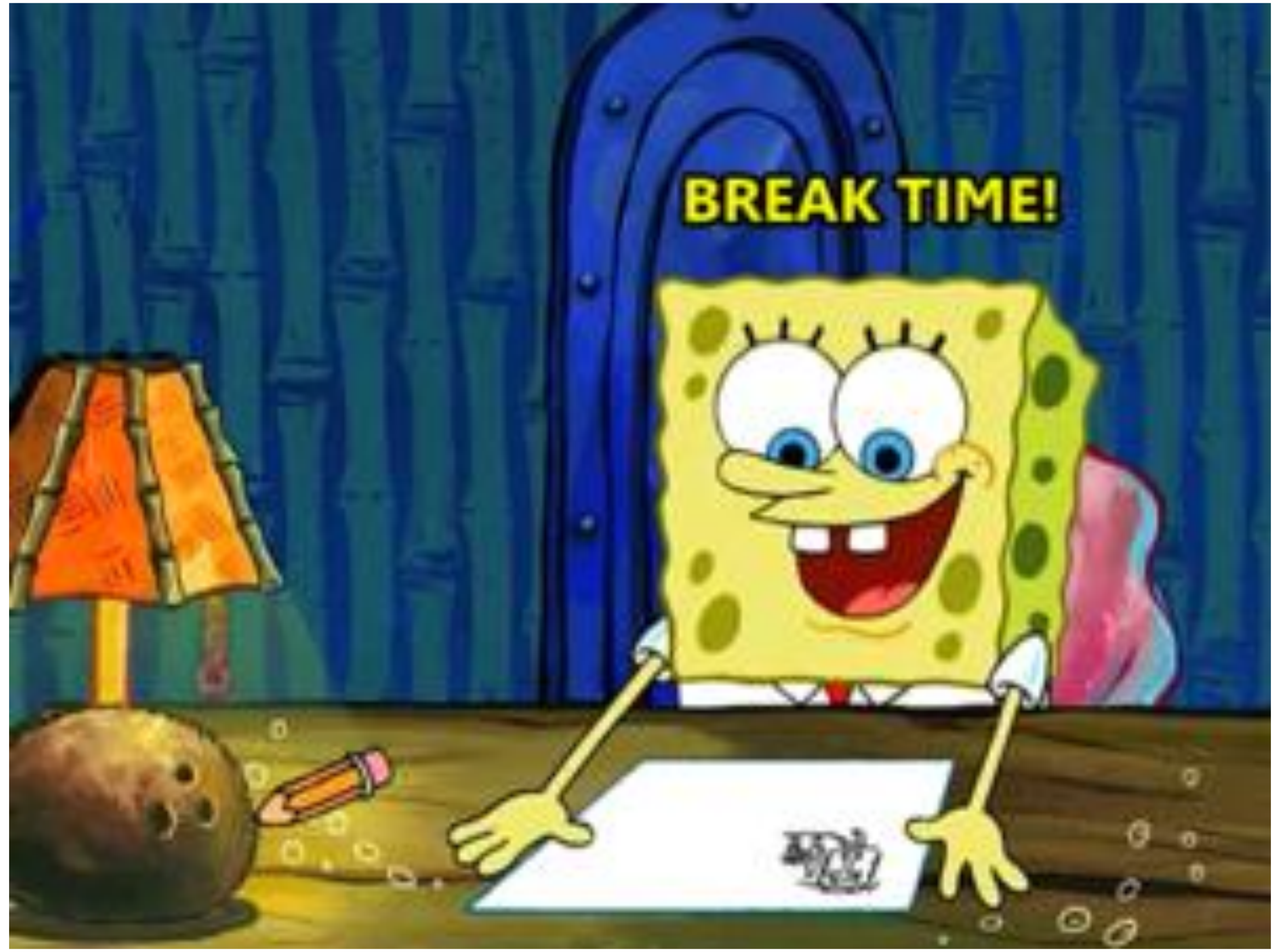


列表（续）

- 现在我可以滚动浏览我们所有的数据




BREAK TIME!


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*

可组合项的生命周期

- 当 Jetpack Compose 执行可组合项时，它会进入 Composition

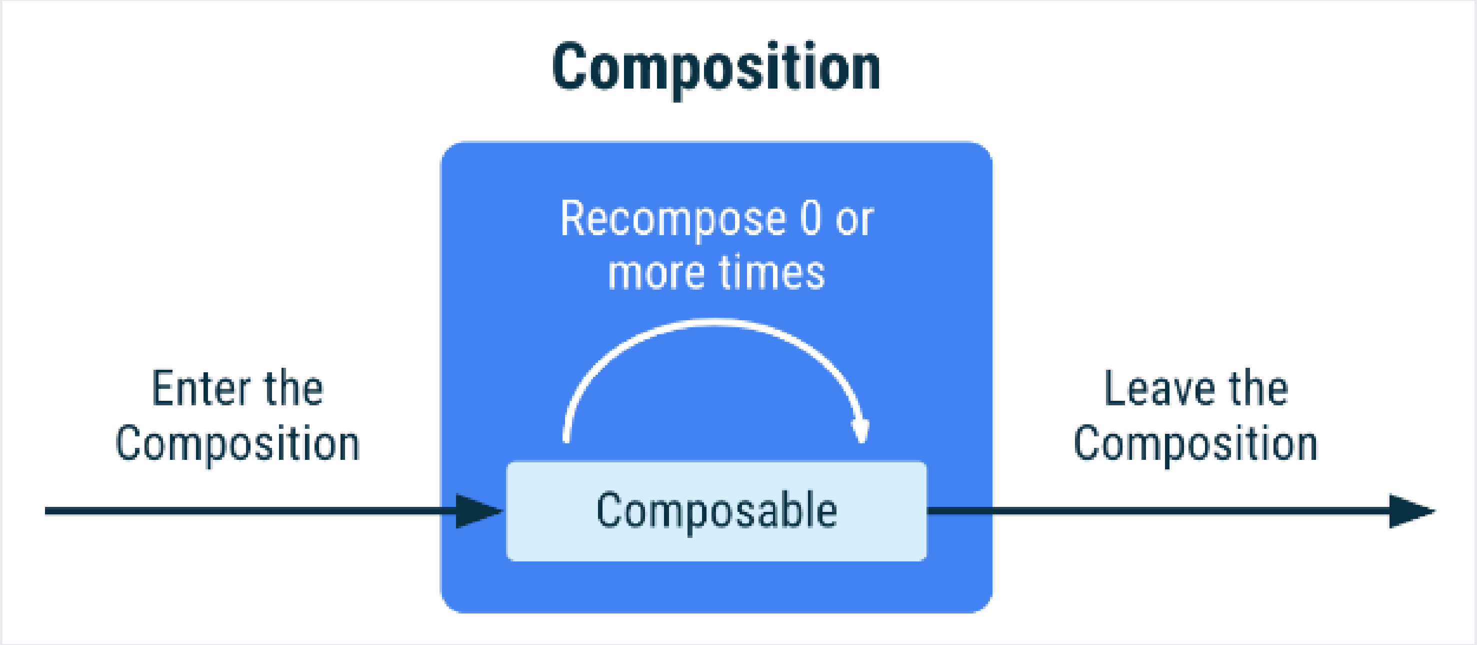
- 进入 *Composition* 有两种方式

  1. 当你首次运行可组合项时，它会经历 初始组合

  2. 当你的可组合项读取的 状态 发生变化时，它会经历重组

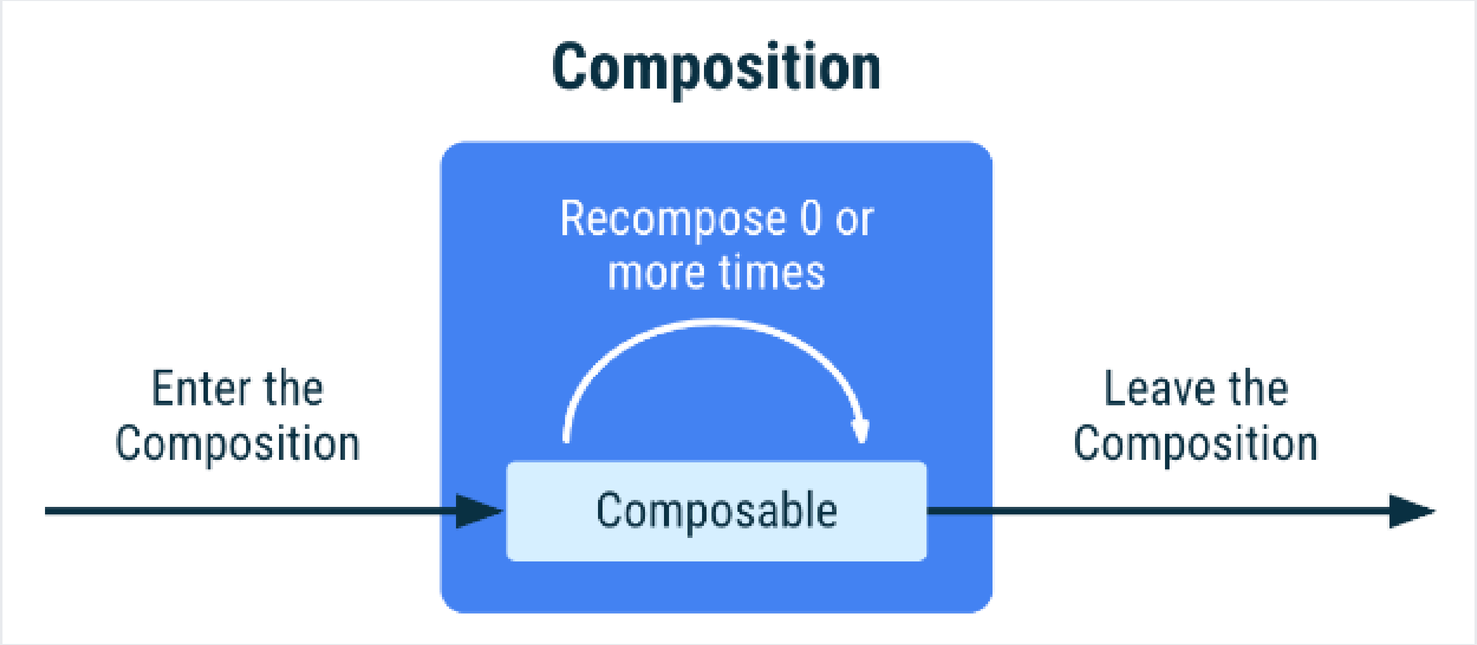
Composable lifecycle (cont.)

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



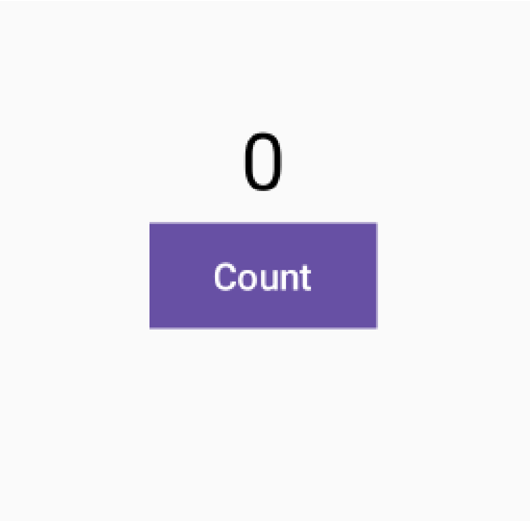
可组合项的生命周期（续）

- 将 *Composition* 理解为可组合项显示在 UI 上时的过程
- 当可组合项不再显示时，它就离开了 *Composition*

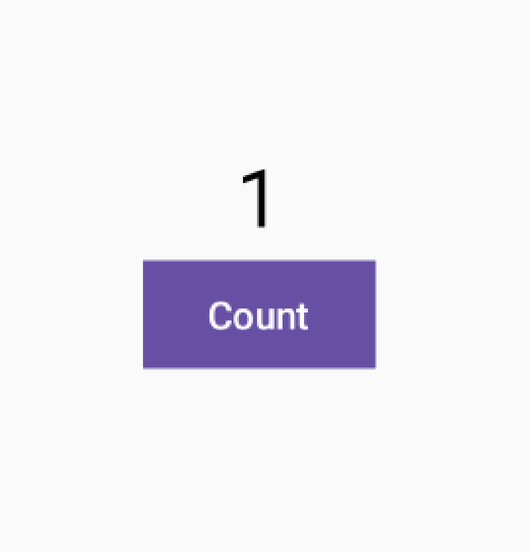


State

- Specific data that changes overtime within a composable
  - e.g., A Text composable could display multiple values over its lifetime



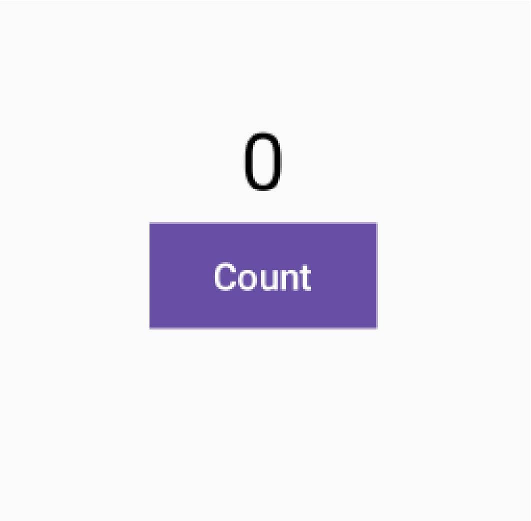
- To change state and trigger a recomposition, an event needs to occur
  - e.g., Pressing a button



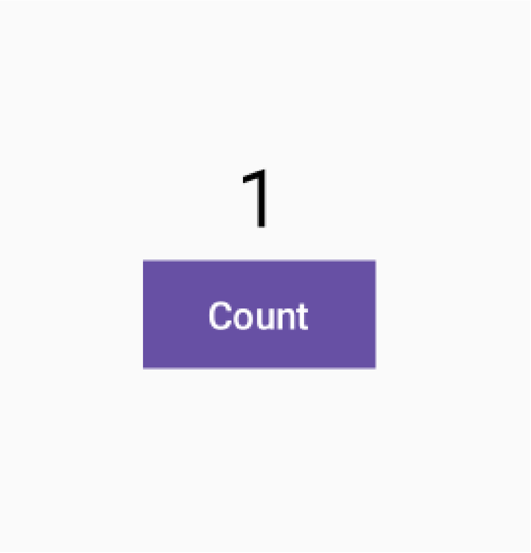
状态

- 在可组合项的生命周期内会随时间变化的特定数据

  可组合项
  - 例如，一个 Text 可组合项可以在其存在期间显示多个值 在其生命周期内

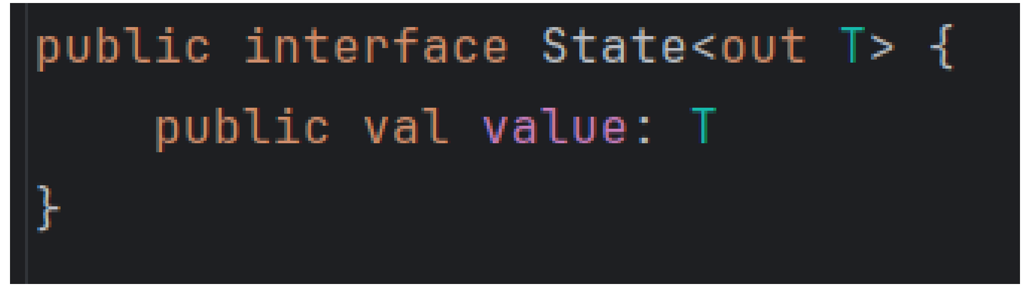


- 要更改状态并触发重组，需使用一个事件需要发生
  - 例如，按下按钮



State (cont.)

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

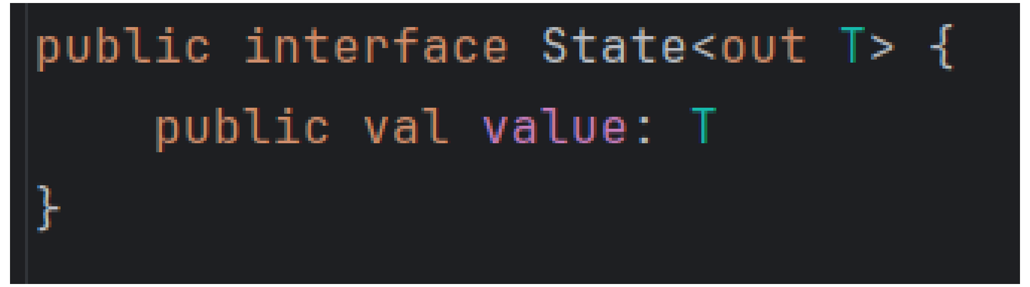


```
public interface State<out T> {
    public val value: T
}
```

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

状态（续）

- 我们用来触发重组的状态是*State<T>*

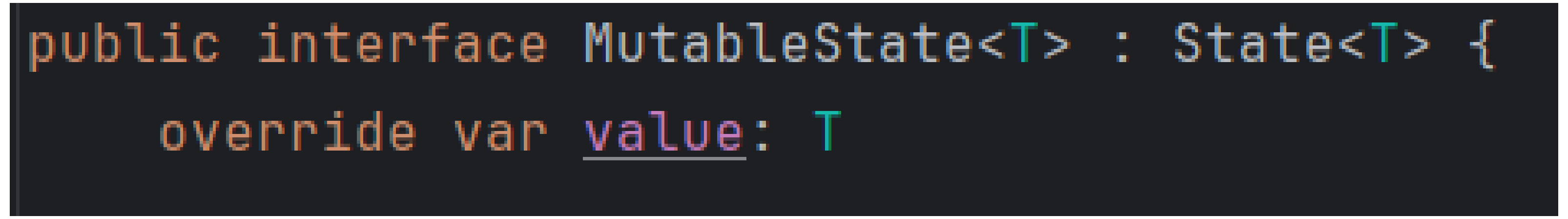


```
public interface State<out T> {
    public val value: T
}
```

- 状态*<T>* 是一个仅公开只读值的接口

State (cont.)

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

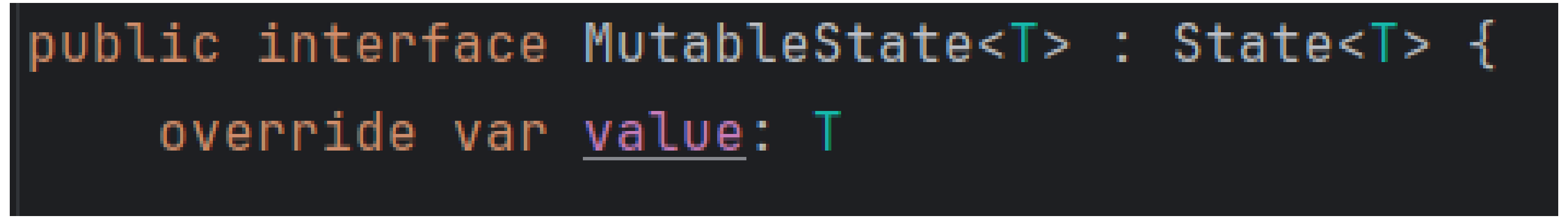


```
public interface MutableState<T> : State<T> {
    override var value: T
```

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

状态（续）

- *State<T>* 是只读的，因此更常见的类型是*MutableState<T>*

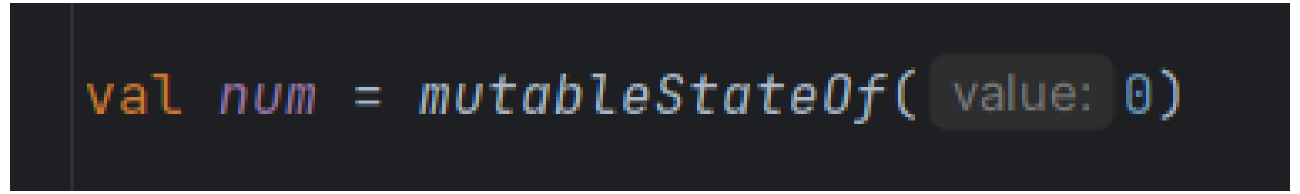


```
public interface MutableState<T> : State<T> {
    override var value: T
```

- Compose 会观察 *value* 属性，并在其发生变化时安排重组

State (cont.)

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

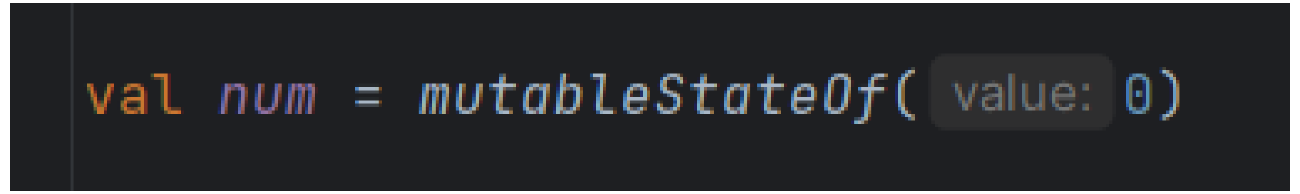


```
val num = mutableStateOf( value: 0)
```

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

状态（续）

- 创建*MutableState<T>* 最常见的方法是使用*mutableStateOf*函数

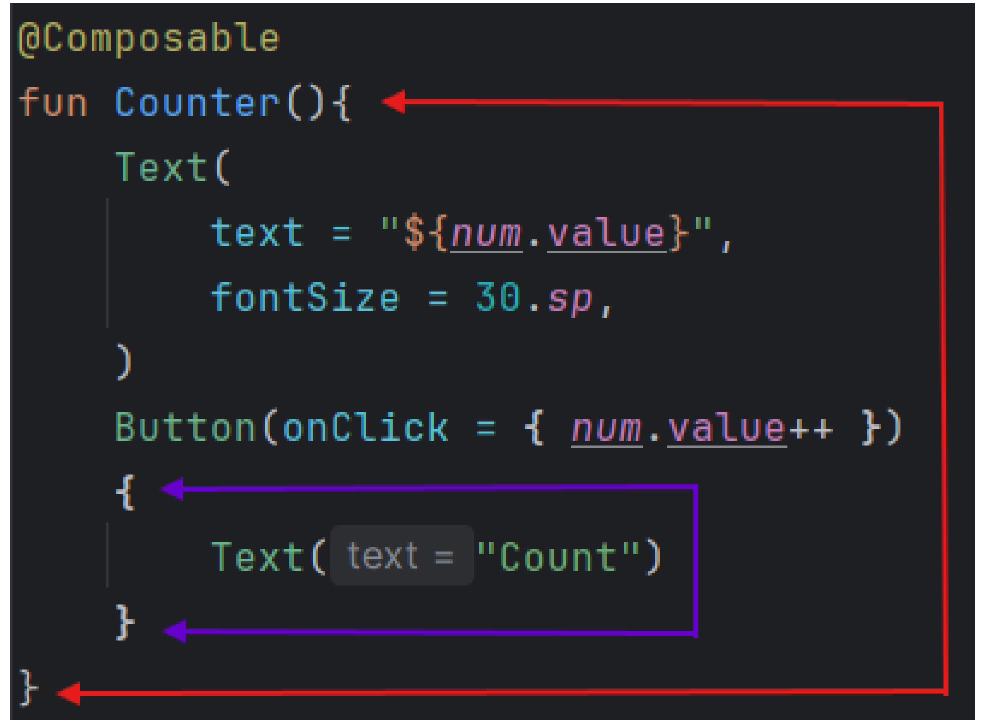


```
val num = mutableStateOf( value: 0)
```

- 由于该值是一个整数，Kotlin 推断出*num* 是一个 *MutableState<Int>*

Recomposition Scope

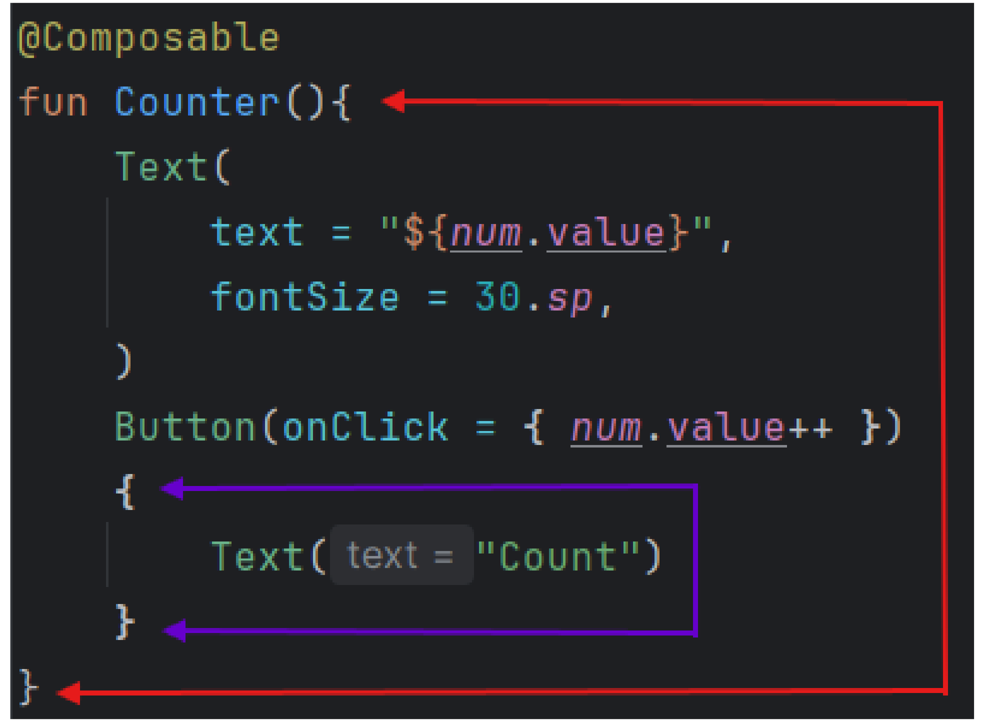
- In the example we have two recomposition scopes
  - Counter Scope and Button Scope
- Recomposition scope is usually marked by an opening and closing function bracket



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

重组作用域

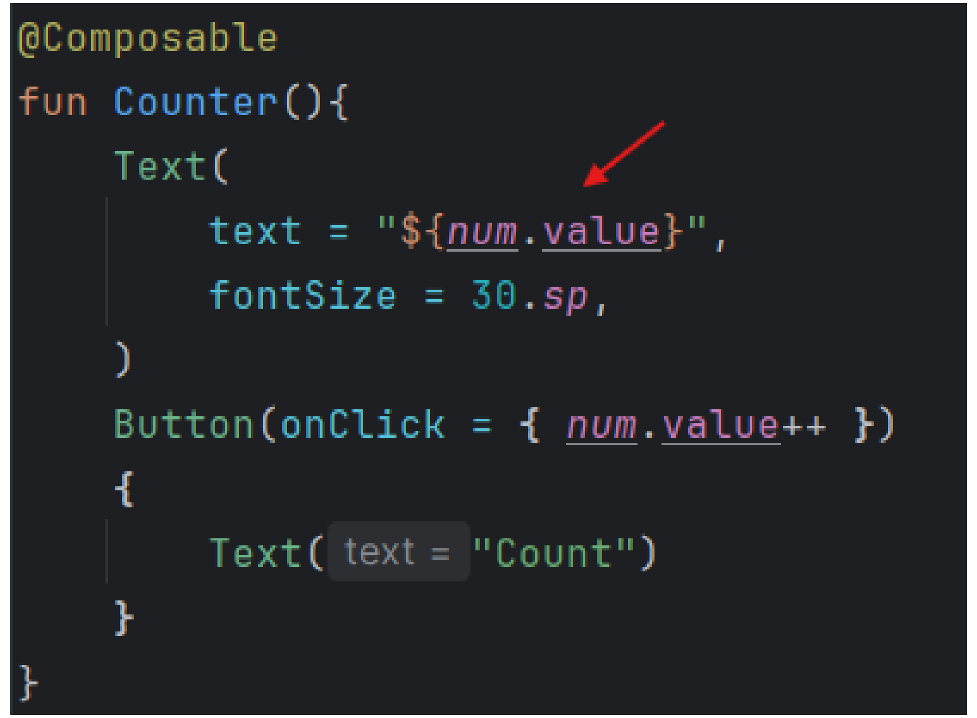
- 在该示例中，我们有两个重组作用域
  - 计数器作用域 和 按钮作用域
- 重组作用域是通常由一对函数的起始和结束括号标记



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

Recomposition Scope (cont.)

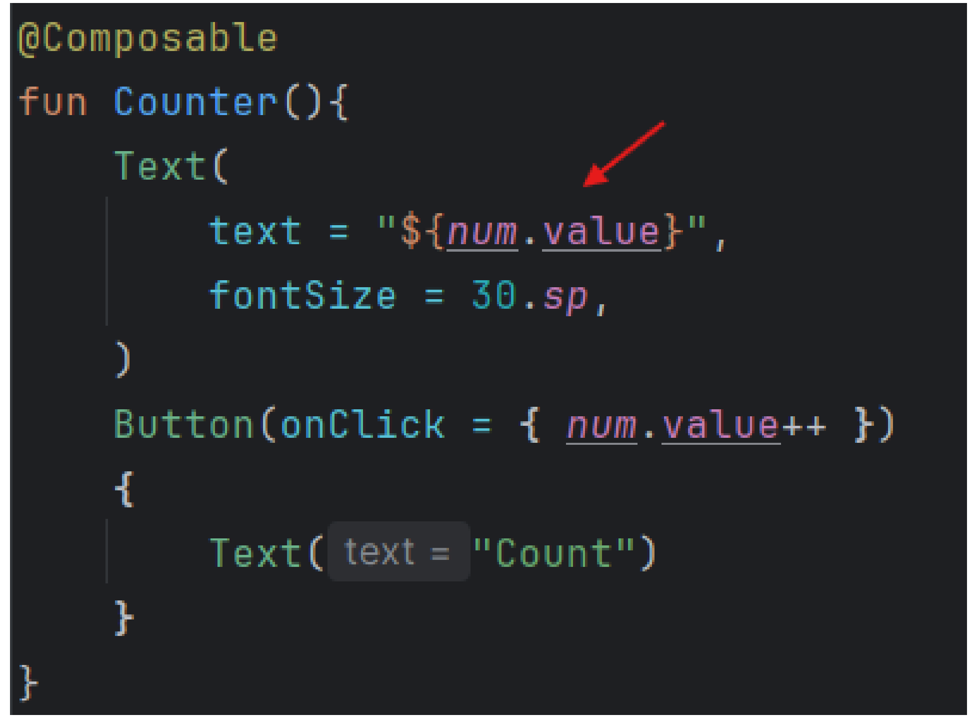
- The lowest recomposition scope to the state being read, is what will recompose (aka. Counter Scope)



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

重组作用域（续）

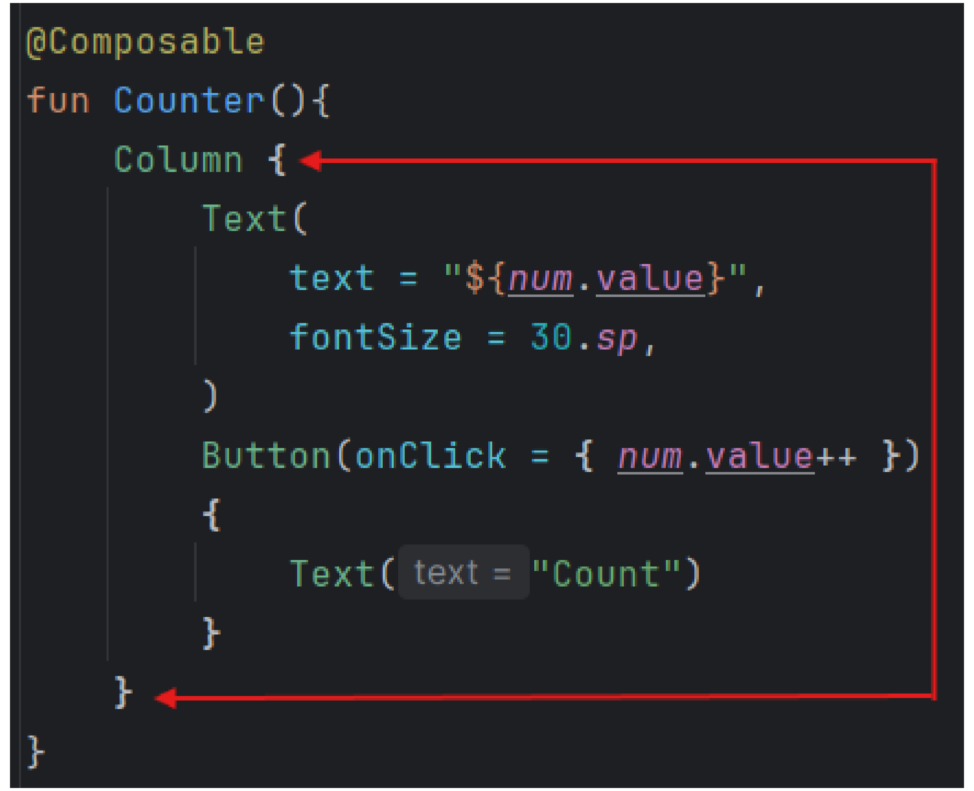
- 最低的重组作用域正在读取的状态，将决定重组（也称为计数器作用域）



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

Recomposition Scope (cont.)

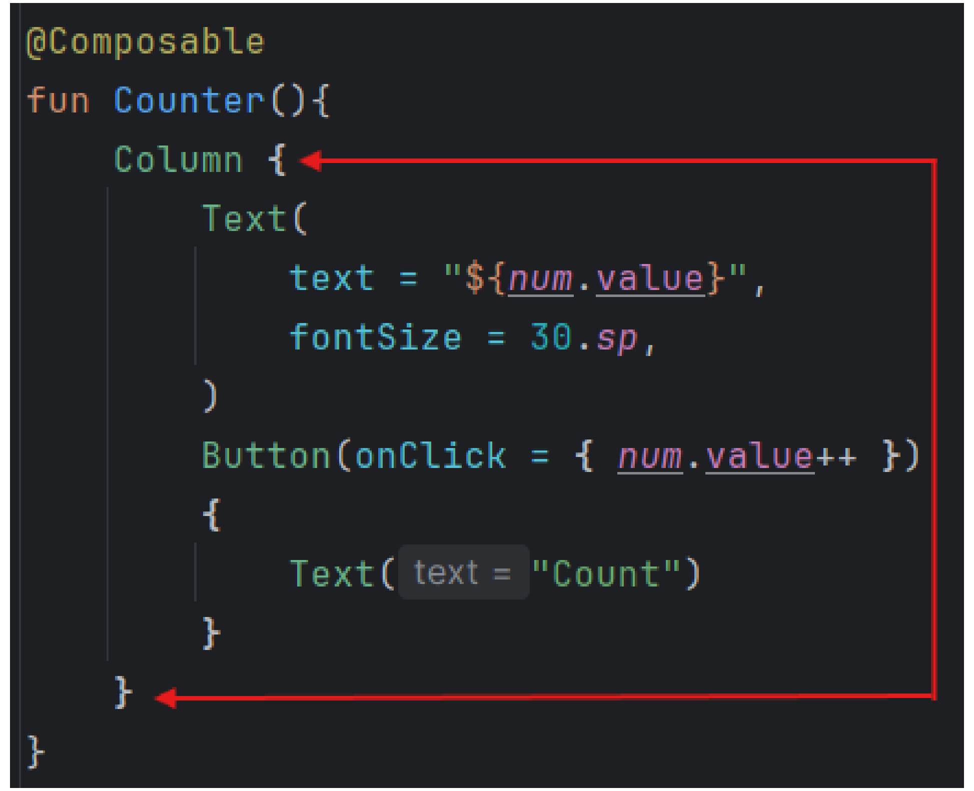
- In this situation, you might think the *Column Scope* is the lowest recomposition scope
- A *Column*, *Row* and *Box* are inline functions, and don't have a recomposition scope
- So *Counter Scope* is still the lowest recomposition scope



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

重组作用域（续）

- 在这种情况下，你可能会认为 Column 作用域 是最低的重组作用域
- *Column*、*Row* 和 *Box* 是内联函数，没有重组作用域
- 因此 Counter 作用域 仍然是最低的重组作用域



```
@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*

状态（续）

- 智能重组
  - 仅重组读取 *value* 的组件
  - 忽略未读取 *value* 的组件

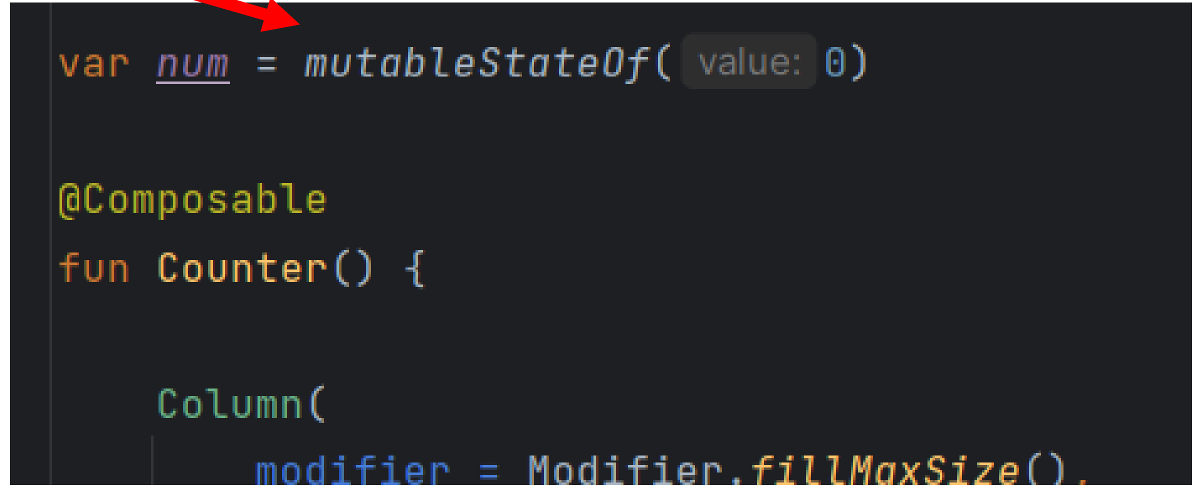
- 跳过（不在考试范围内）
  - 如果 compose 能确定数据未发生变化（稳定），则会跳过
  - 如果 Compose 无法确定数据是否已更改（不稳定），则会重新组合
  - https://developer.android.com/develop/ui/compose/performance/stability

- 这两者都可以通过 布局检查器 进行跟踪

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

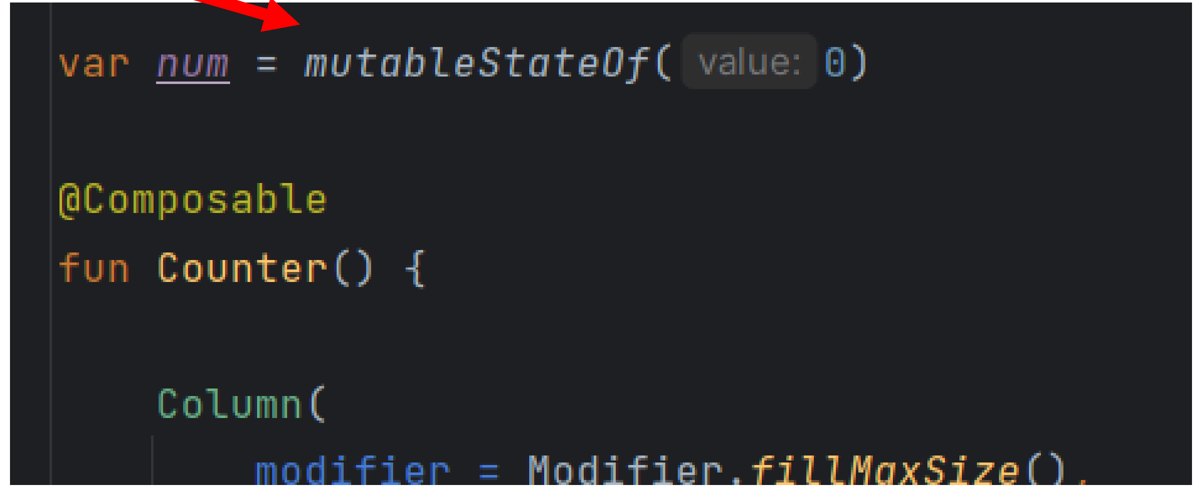


```
var num = mutableStateOf( value: 0)

@Composable
fun Counter() {

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

状态（续）

- 外部状态并非最佳实践

- 状态应为可组合项内部的（即局部的），**内部**，仅有少数情况例外而非外部状态
  - 外部状态 是在可组合项外部声明的

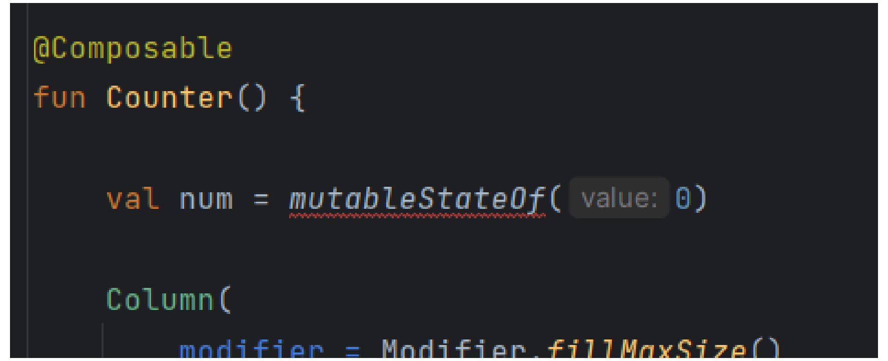


```
var num = mutableStateOf( value: 0)

@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”*

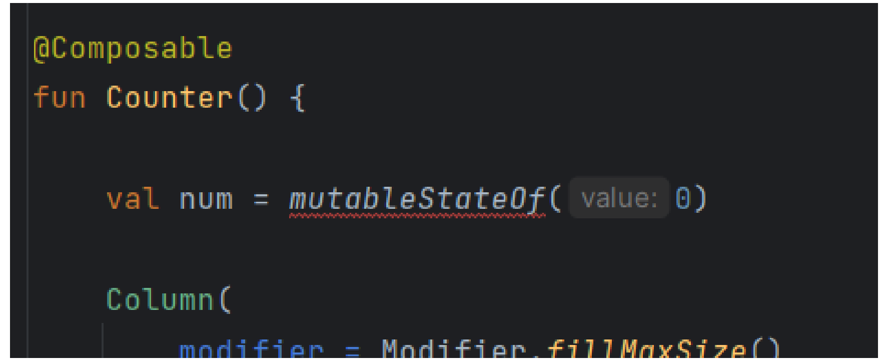


```
@Composable
fun Counter() {

    val num = mutableStateOf( value: 0)

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

状态（续）

- 当状态为内部状态时
  - 更易于测试
  - 提升封装性和模块化
  - 更优化的重组

- 一旦我们将状态移入可组合函数中，就会出现错误
  - “在组合过程中创建状态对象时未使用 remember ”

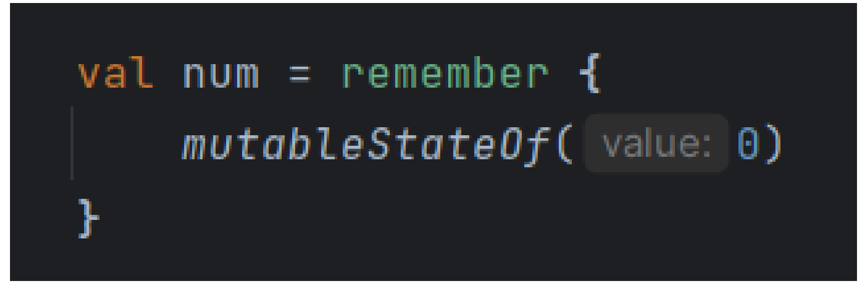


```
@Composable
fun Counter() {

    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*

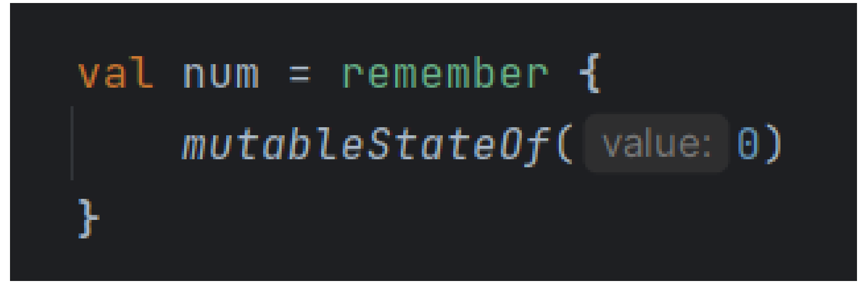


```
val num = remember {
    mutableStateOf( value: 0)
}
```

状态（续）

- 问题是每当数值变化时，Counter 都会重新组合（重新运行）
  - 而这反过来会导致每次都将数值重新初始化为 0

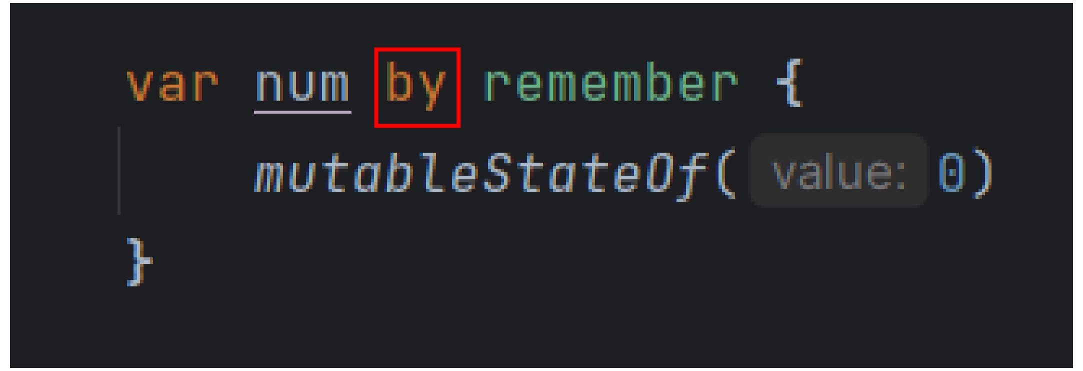
- 编译器提示我们应将其用 *remember* 可组合项包裹以避免此问题
  - 被 remember 包裹的值将存储在 *Composition* 中
  - 该存储的值会在多次 *recomposition* 中保持不变



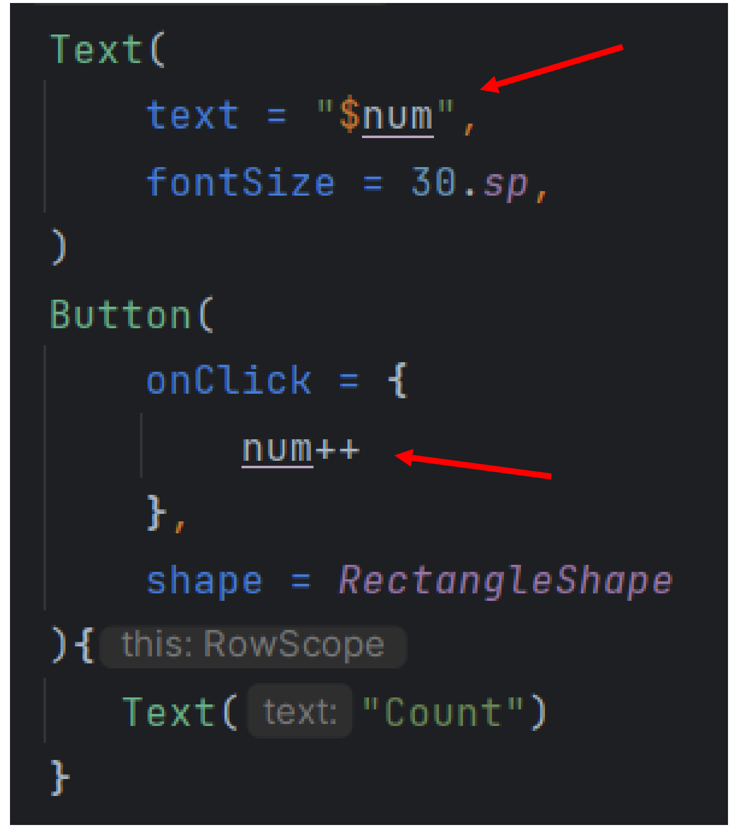
```
val num = remember {
    mutableStateOf( value: 0)
}
```

State (cont.)

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



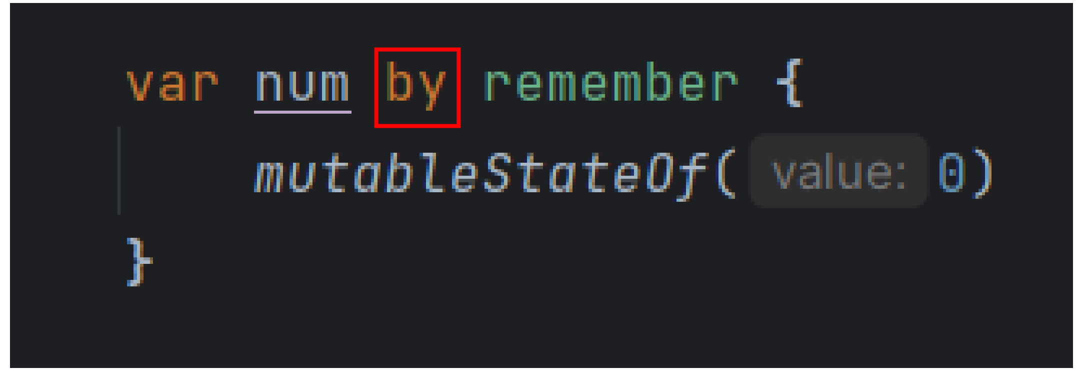
```
var num by remember {
    mutableStateOf( value: 0)
}
```



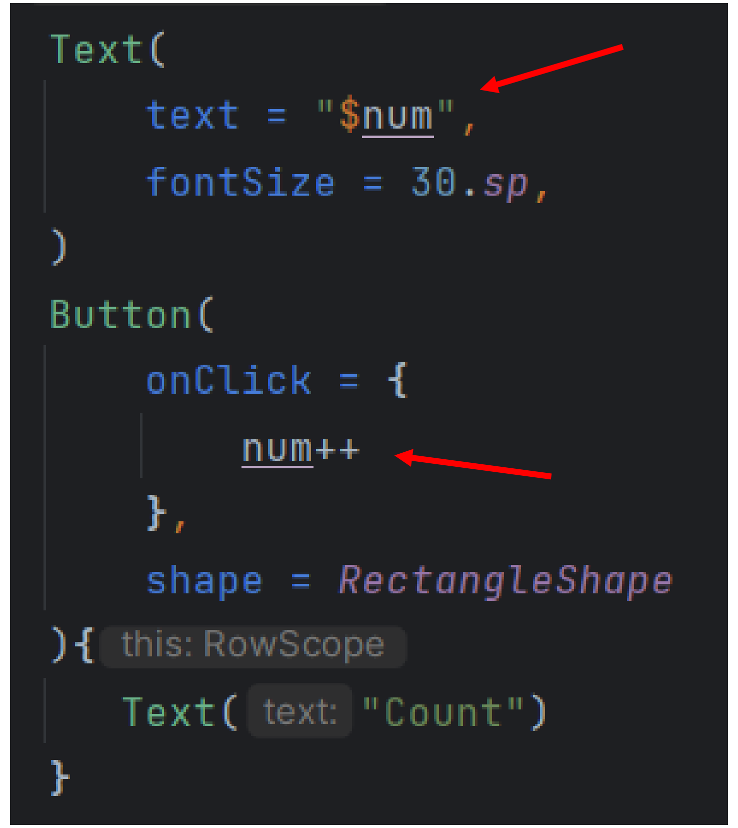
```
Text(
    text = "$num",
    fontSize = 30.sp,
)
Button(
    onClick = {
        num++
    },
    shape = RectangleShape
){ this: RowScope
   Text( text: "Count")
}
```

状态（续）

- 要省略 值，我们可以使用 委托属性



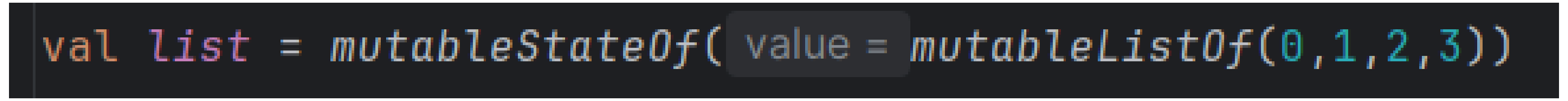
```
var num by remember {
    mutableStateOf( value: 0)
}
```



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

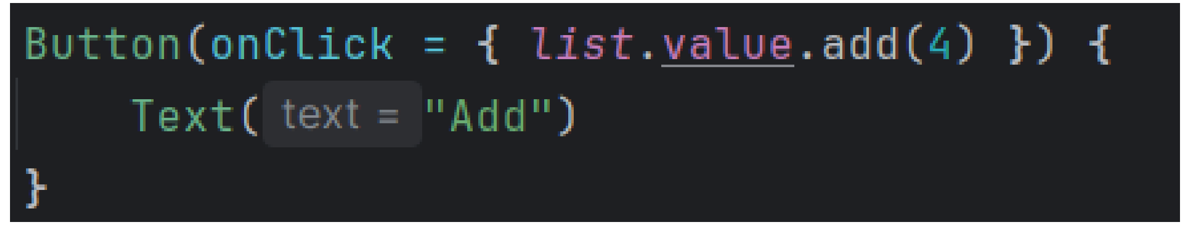
State (cont.)

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



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

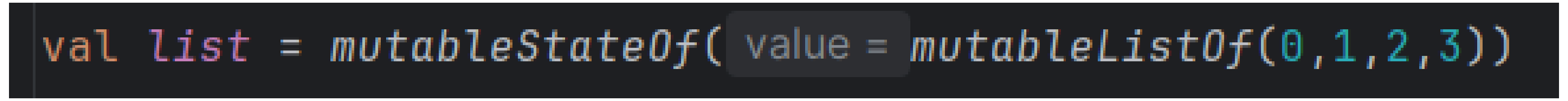
- This issue is that when we mutate the list, *value* is not being set, so no recomposition



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

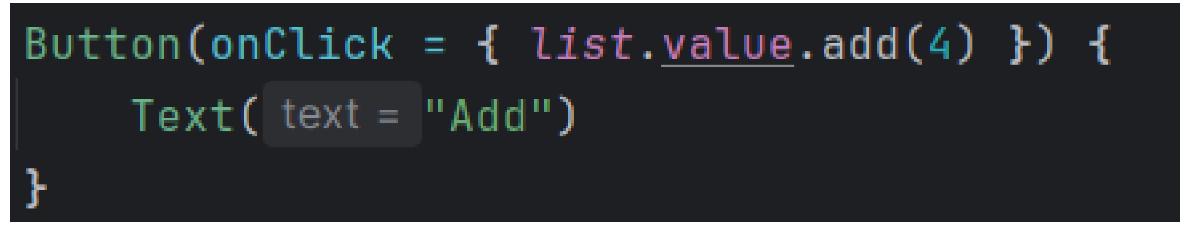
状态（续）

- 使用 *State<T>* 处理集合时，效果并不理想方法



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

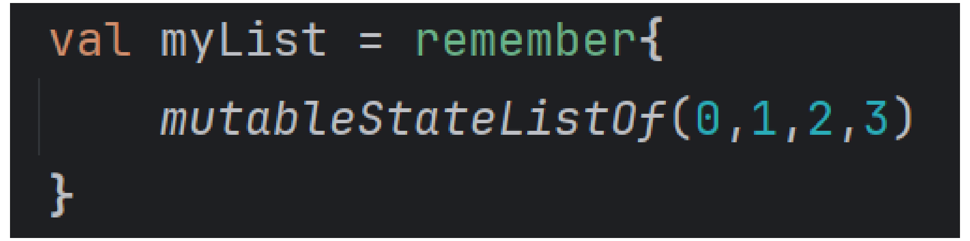
- 问题是当我们修改列表时，*value* 未被设置，因此不会触发重组



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

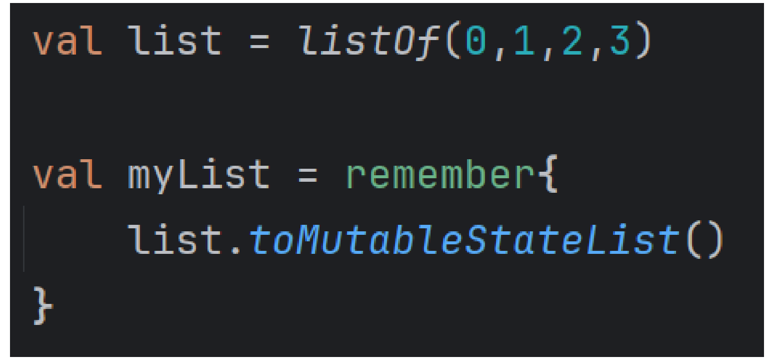
SnapshotStateList

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



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

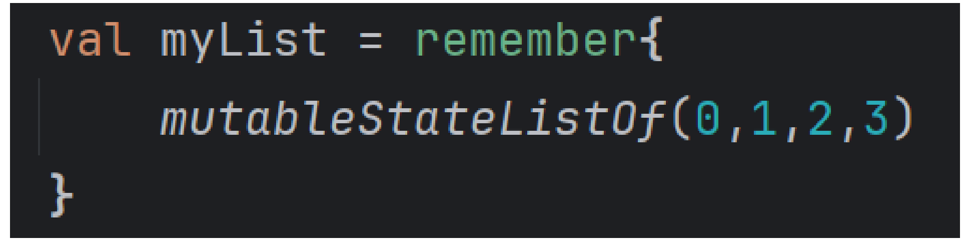
- We can also create a mutable state list from a regular list



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

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

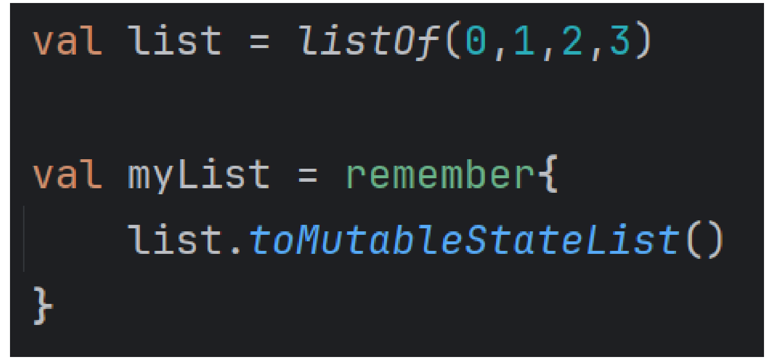
SnapshotStateList

- 最好通过函数 mutableStateListOf使用 SnapshotStateList<T> 来创建可变状态列表



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

- 我们还可以从普通列表创建一个可变状态列表



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

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

SnapshotStateList (cont.)

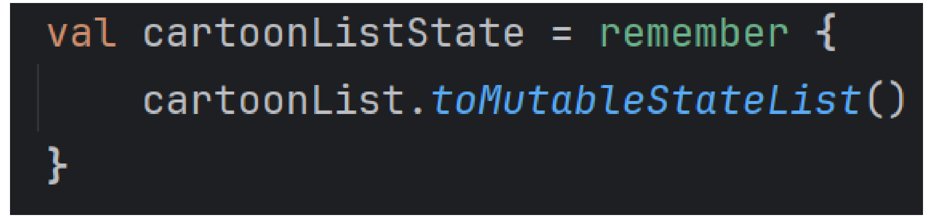
- A *SnapshotStateList<T>* uses a different mechanism to trigger a recomposition
  - Snapshotting
- Since it doesn’t use the *State<T>* interface, we don’t have a value property
  - Which means we also wouldn’t use the by keyword

SnapshotStateList （续）

- A *SnapshotStateList<T>* 使用一种不同的机制来触发重组
  - 快照
- 由于它不使用 *State<T>* 接口，因此我们没有 value 属性
  - 这意味着我们也不会使用 by 关键字

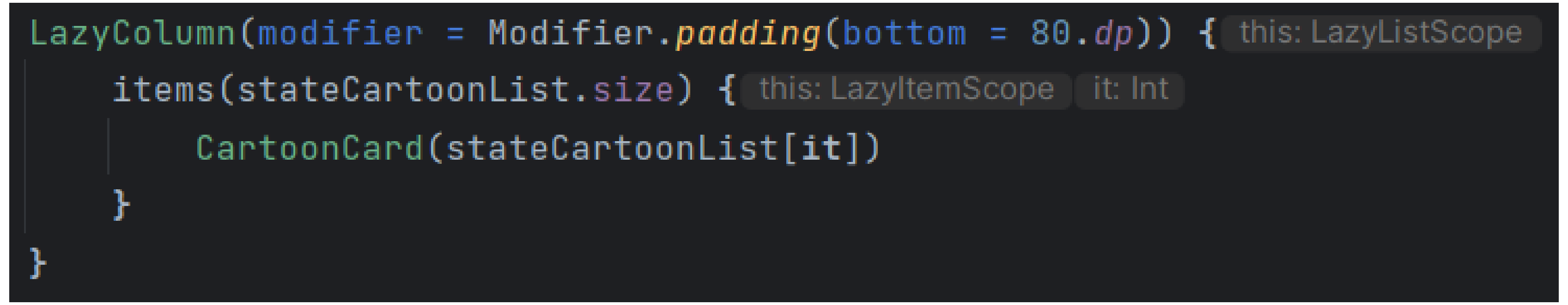
SnapshotStateList (cont.)

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



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

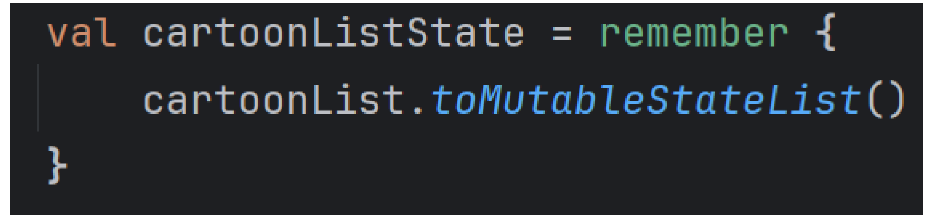
- Then using it in our *LazyColumn*



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

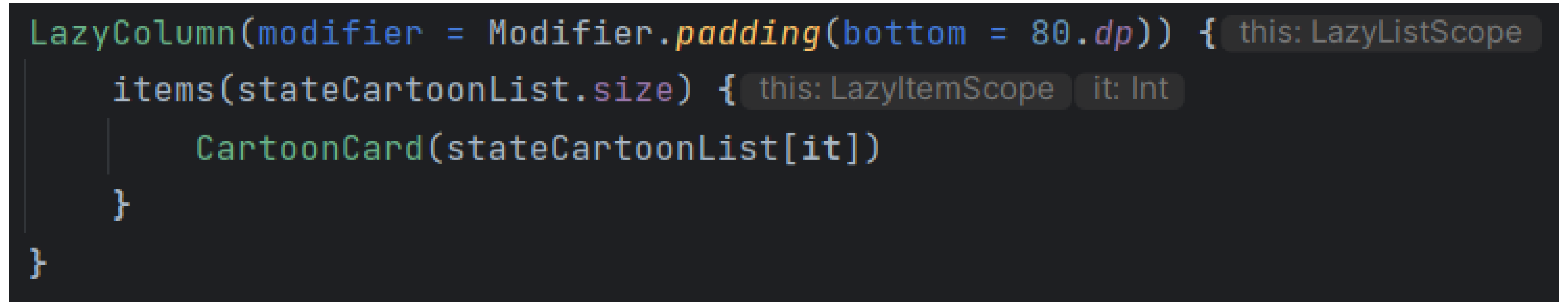
快照状态列表 （续）

- 我们正在从原始的卡通列表创建一个 *SnapshotStateList* 对象



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

- 然后在我们的 *LazyColumn* 中使用它



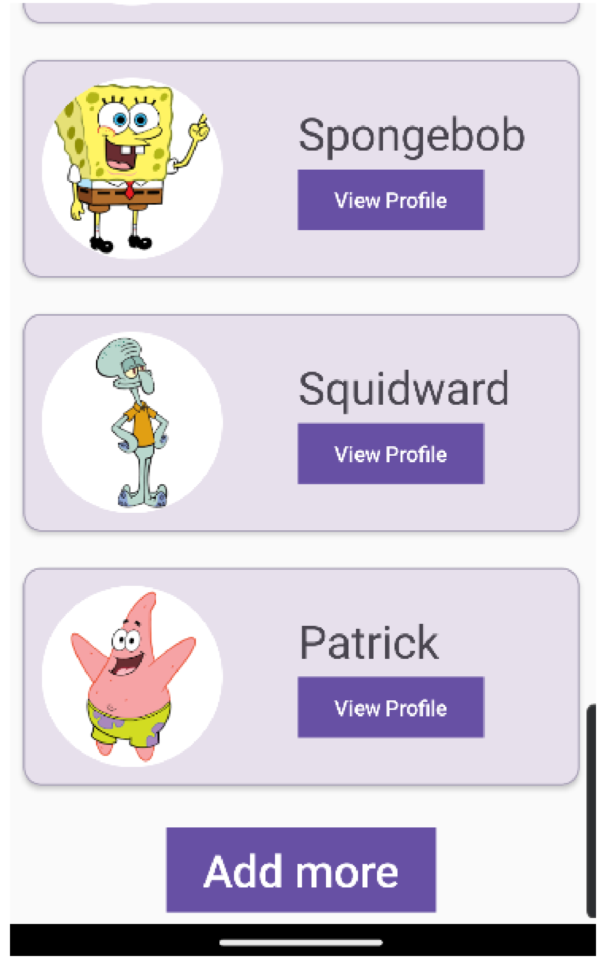
```
LazyColumn(modifier = Modifier.padding(bottom = 80.dp)) { this: LazyListScope
    items(stateCartoonList.size) { this: LazyItemScope  it: Int
        CartoonCard(stateCartoonList[it])
    }
}
```

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

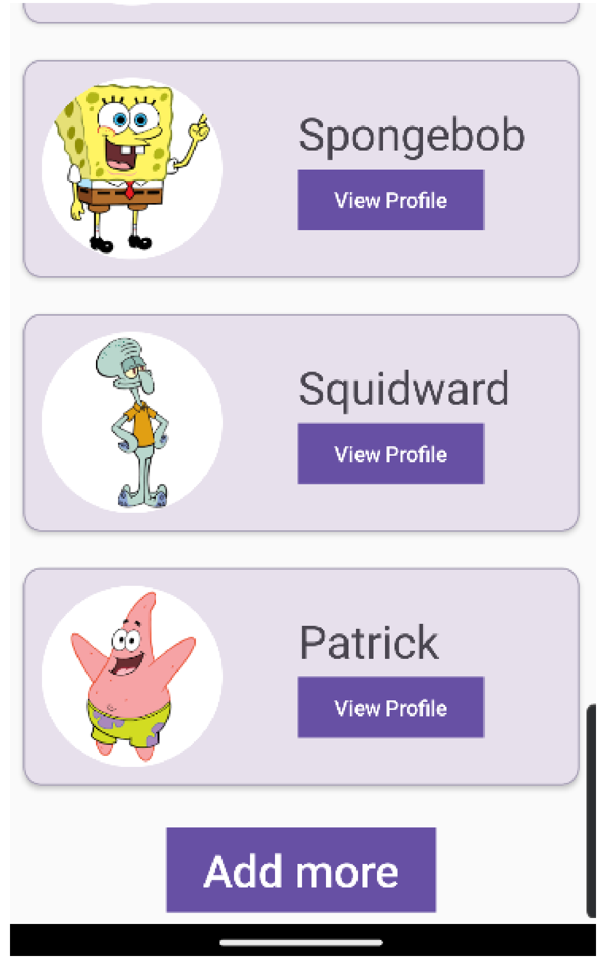


SnapshotStateList（续）

- 然后我们可以通过一个事件来修改该列表
  - 例如，按钮点击

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

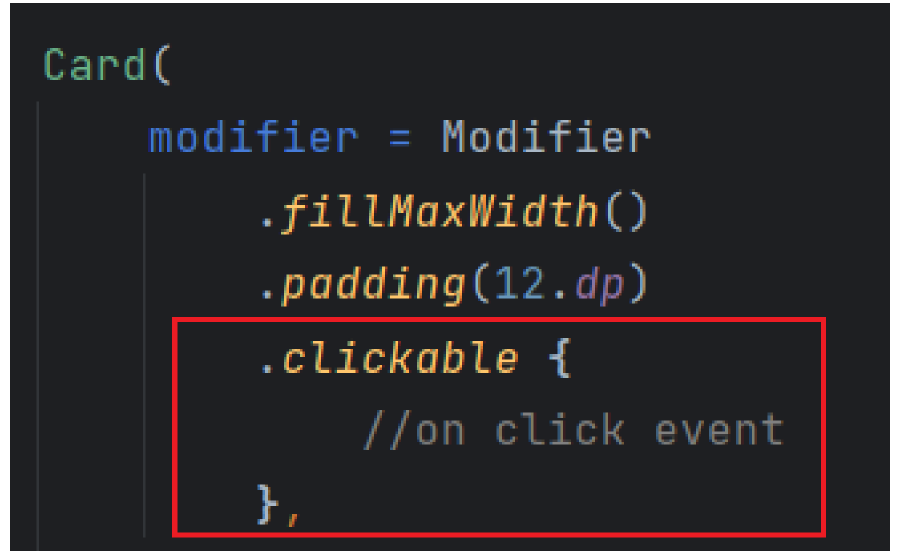
- 并且将成功触发一次重组，显示更新后的列表



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

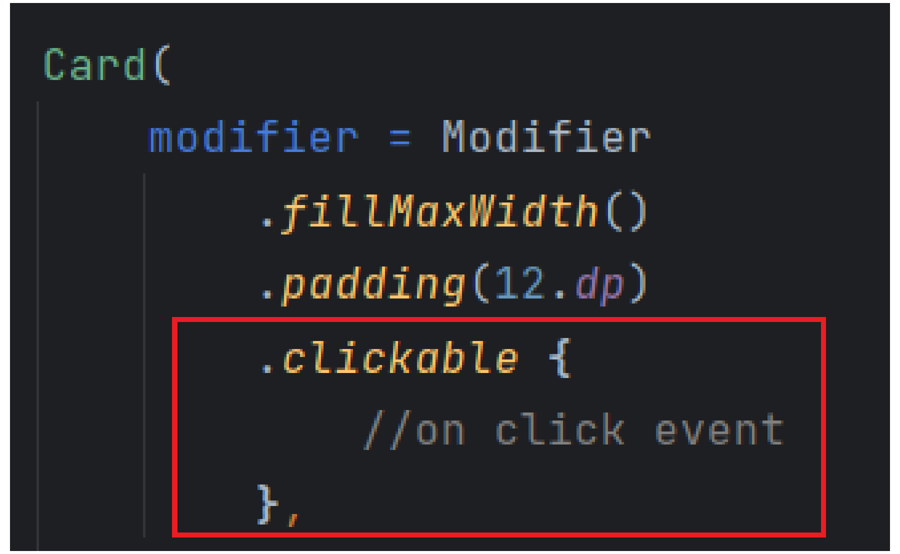


```
Card(
    modifier = Modifier
        .fillMaxWidth()
        .padding(12.dp)
        .clickable {
            //on click event
        },
```

可点击项

- 你可以让任意可组合项 可点击
  - clickable 修饰符提供了一个 *onClick* 事件回调

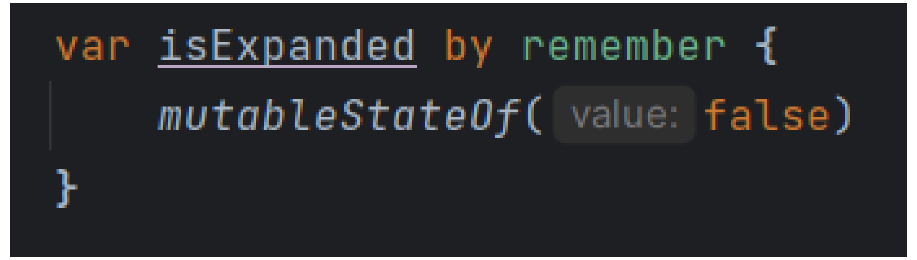
- 当你希望点击整个可组合项本身而不仅仅是一个按钮时，这非常有用



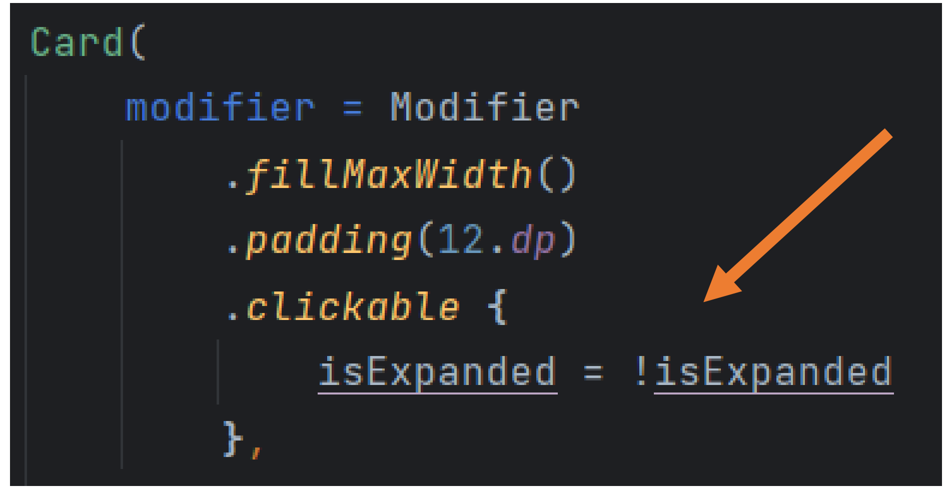
```
Card(
    modifier = Modifier
        .fillMaxWidth()
        .padding(12.dp)
        .clickable {
            //on click event
        },
```

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



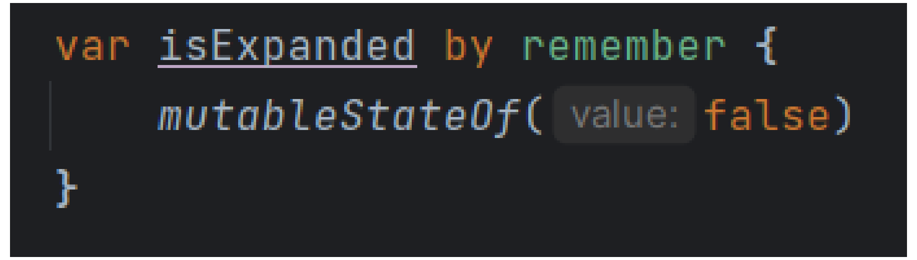
```
var isExpanded by remember {
    mutableStateOf( value: false)
}
```



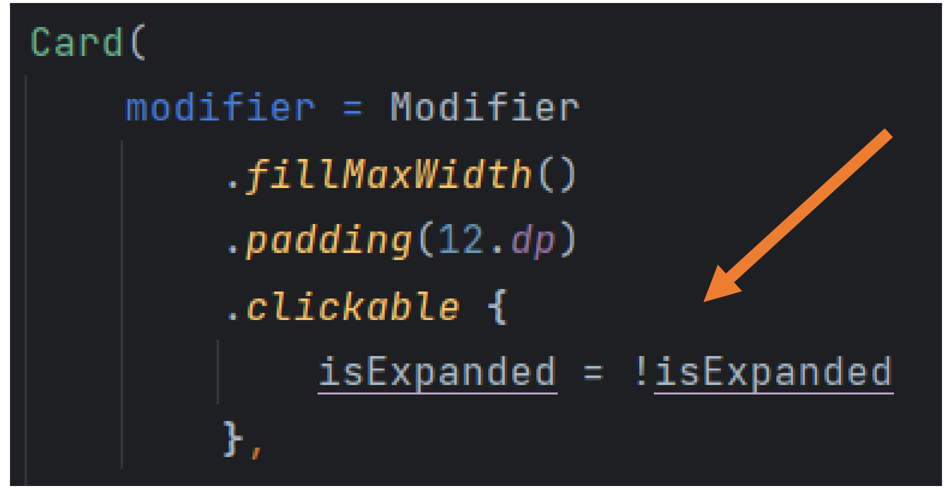
```
Card(
    modifier = Modifier
        .fillMaxWidth()
        .padding(12.dp)
        .clickable {
            isExpanded = !isExpanded
        },
```

可点击元素（续）

- 当点击卡片时，我们将其展开
- 为此，我们需要创建一个 *MutableState<Boolean>*
- 点击时，我们可以设置值为真或假



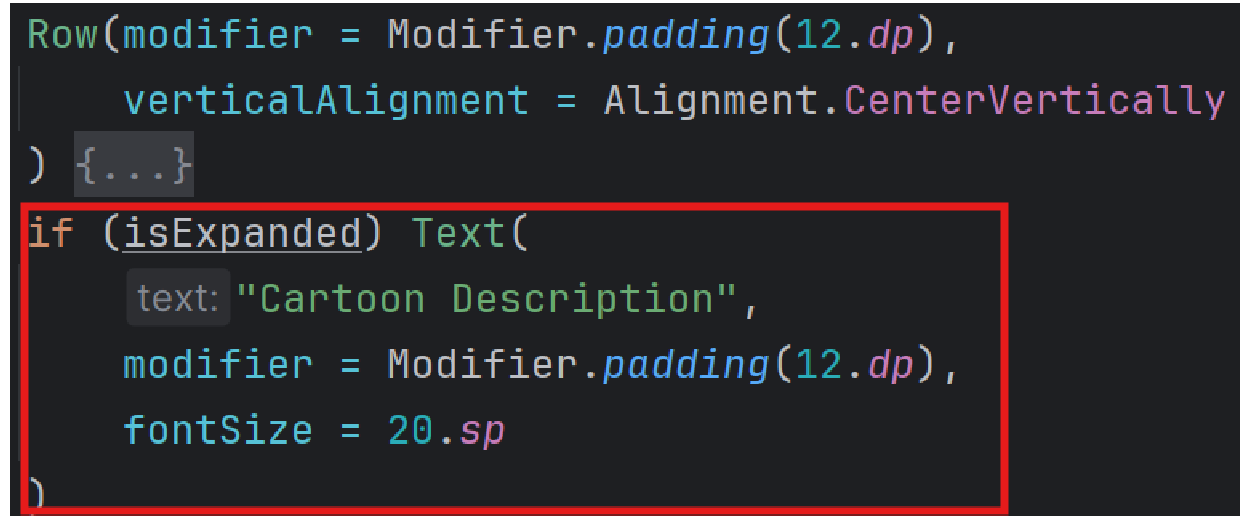
```
var isExpanded by remember {
    mutableStateOf( value: false)
}
```



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

Composable visibility

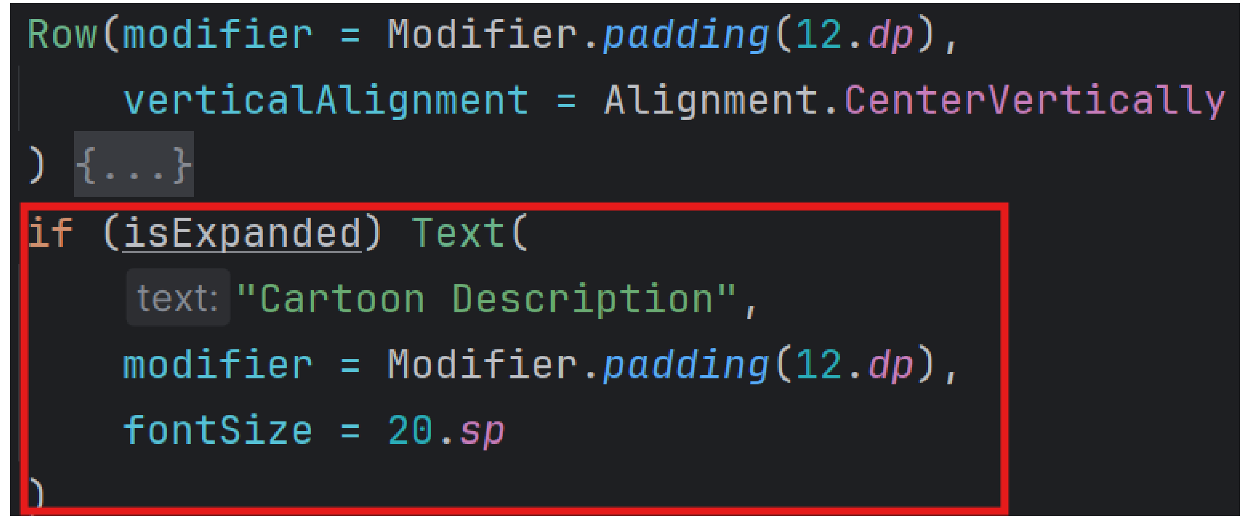
- Inside our Card, we check if *value* is true, then add a Text composable



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

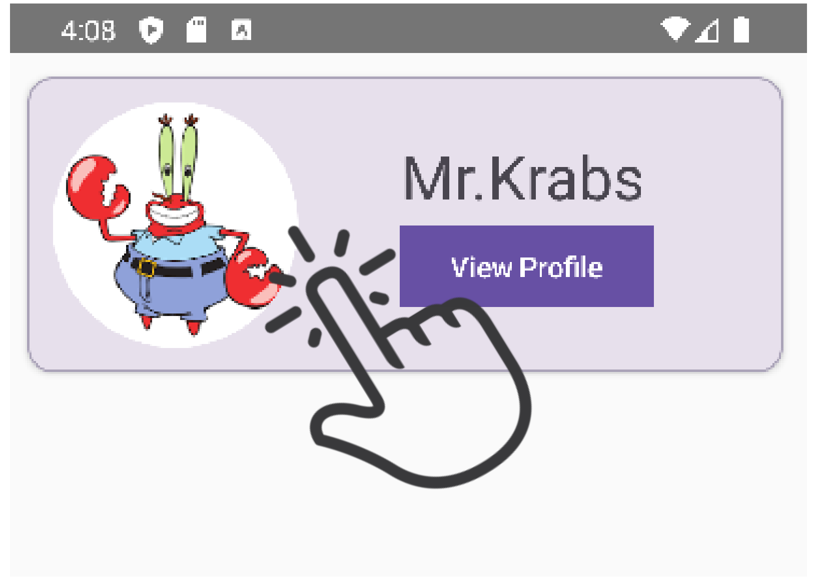
可组合的可见性

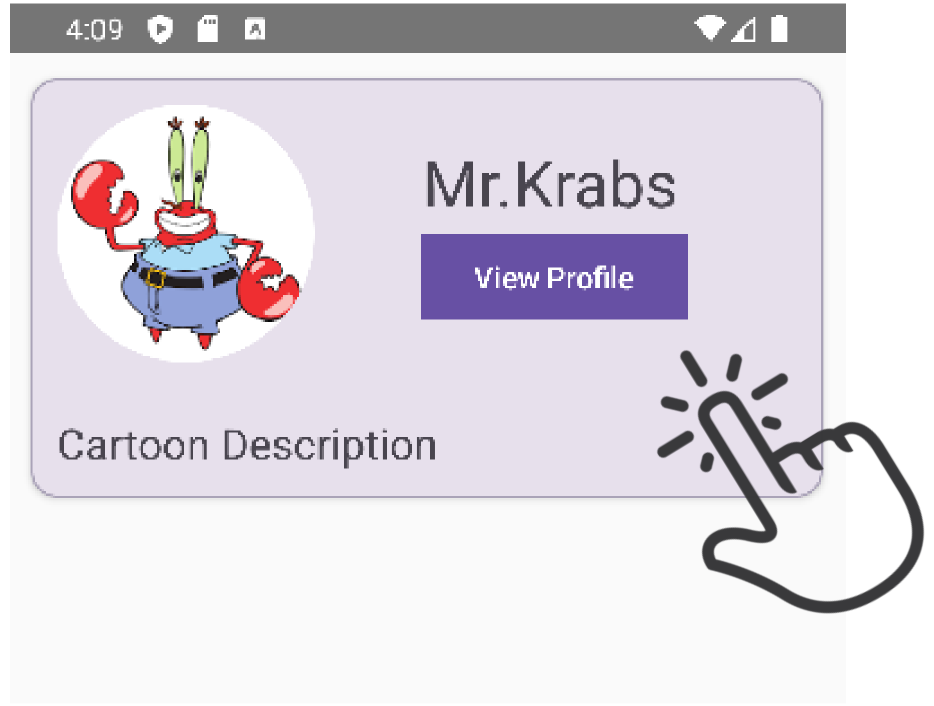
- 在我们的 Card 内部，我们检查 *value* 是否为 true，然后添加一个 Text 可组合项



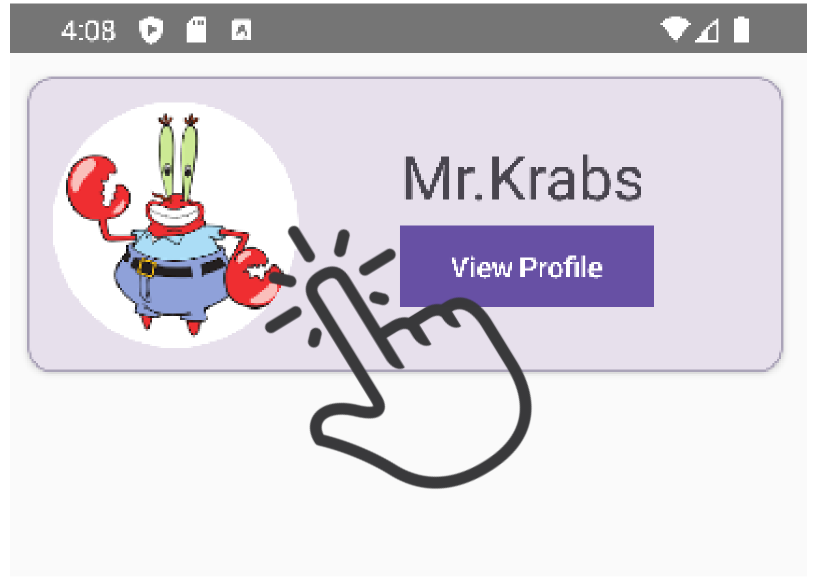
```
Row(modifier = Modifier.padding(12.dp),
    verticalAlignment = Alignment.CenterVertically
) {...}
if (isExpanded) Text(
    text: "Cartoon Description",
    modifier = Modifier.padding(12.dp),
    fontSize = 20.sp
)
```

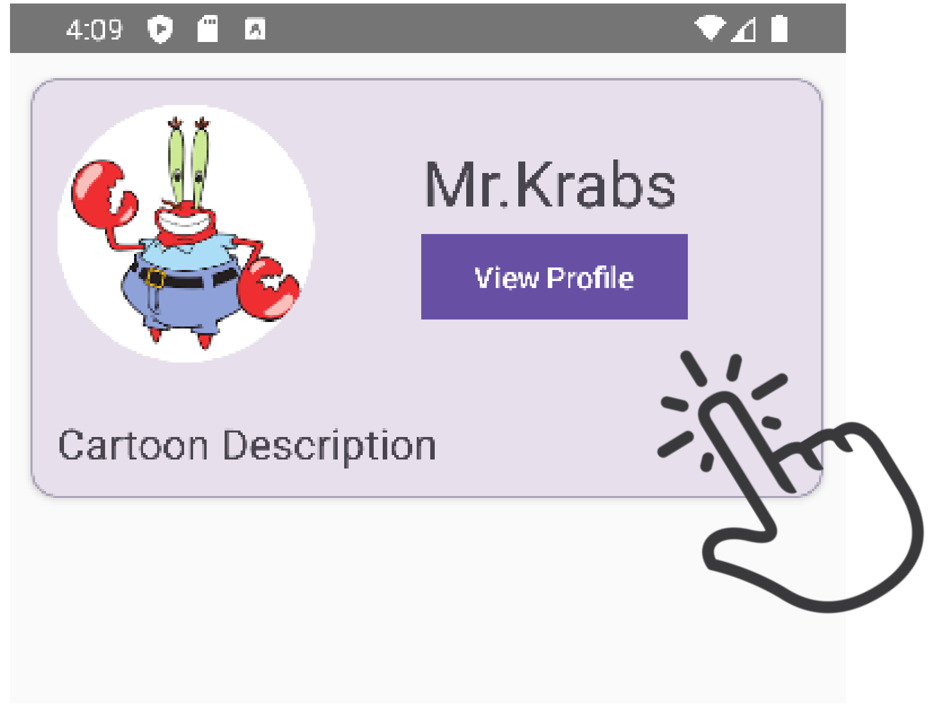
Composable visibility (cont.)





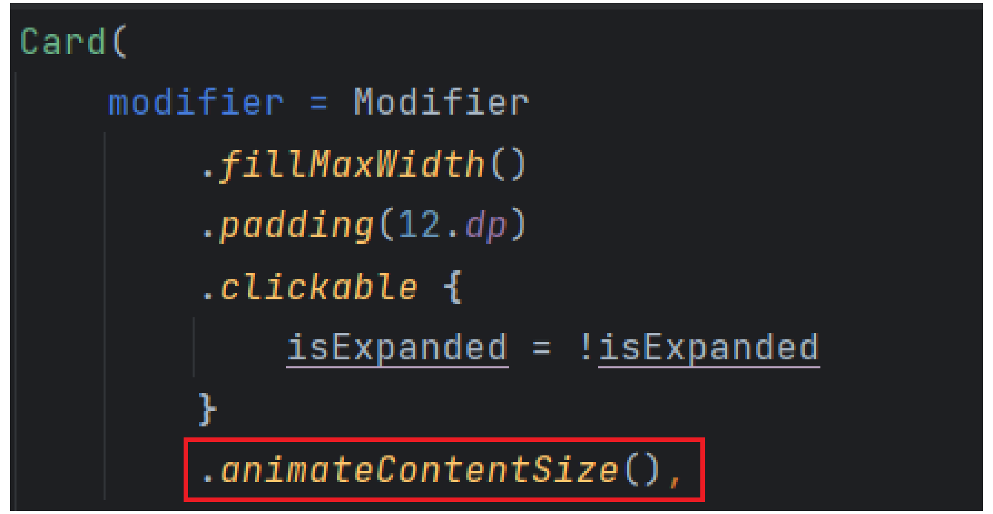
可组合的可见性（续）





Animations

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

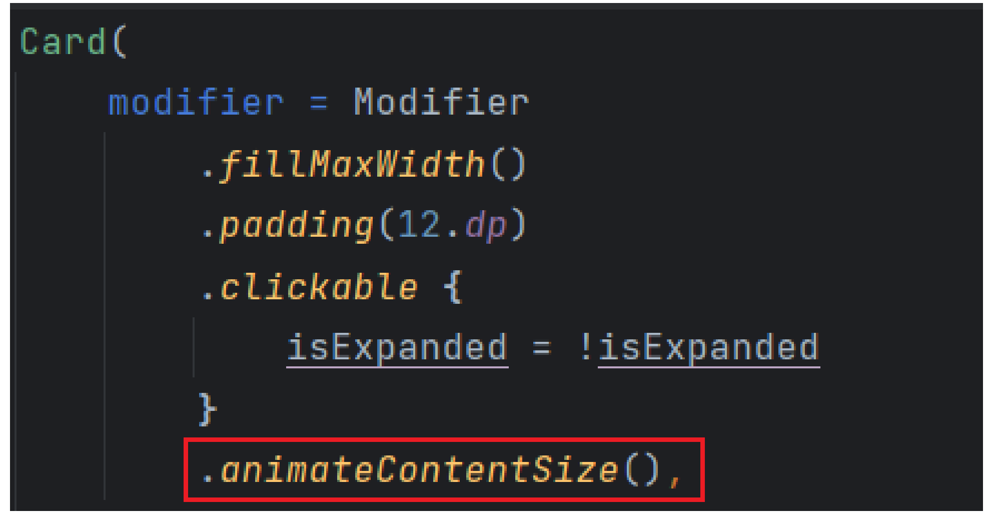


```
Card(
    modifier = Modifier
        .fillMaxWidth()
        .padding(12.dp)
        .clickable {
            isExpanded = !isExpanded
        }
        .animateContentSize(),
```

- Expanding your card will have a smoother transition now

动画

- Compose 提供了一些内置动画，例如 *animateContentSize*



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

- 现在，展开卡片时将具有更流畅的过渡效果