Leibniz Universität Hannover

Mobile Interaction

Android: State in Composables, State in View Models

Human-Computer Interaction Group

Prof. Dr. Michael Rohs
michael.rohs@hci.uni-hannover.de

remember

```
@Composable
fun MyComposable() {
    // new value on each recomposition
    var i: Int = Random.nextInt(0, 256)
    println("i = $i")

    // value computed once, then cached
    var j: Int = remember { Random.nextInt(0, 256) }
    println("j = $j")

    ...
}
```

```
@Composable
fun <T> remember(calculation: () -> T): T =
    currentComposer.cache(false, calculation)
```

mutable state

```
@Composable
fun MyComposable() {
    // observable state
    val k: MutableState<Int> =
            remember { mutableStateOf(0) }
    ...
}
```

constructor function

```
fun <T> mutableStateOf(value: T): MutableState<T>
```

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

val is read-write

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

val is read-only

mutable state: observable value

```
@Composable
fun MyComposable() {
    // observable state
    val k: MutableState<Int> =
            remember { mutableStateOf(0) }

    Button(onClick = {k.value++ }) {
        Text(text = "Click ${k.value}")
    }
}
```

write: inform subscribers

read: subscribe

```
fun <T> mutableStateOf(value: T): MutableState<T>

interface MutableState<T> : State<T> {
    override var value: T
}

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

mutable state: delegated properties

- get and set of a property are delegated to getValue and setValue extension methods
- when read from property, getValue is called
- when writing to property, setValue is called
- keyword "by":

```
@Composable
fun MyComposable() {
    var k: Int by
            remember { mutableStateOf(0) }
    Button(onClick = {k++ }) {
        Text(text = "Click ${k}")
    }
}
```

https://kotlinlang.org/docs/delegated-properties.html

```
fun <T> mutableStateOf(value: T): MutableState<T>

interface MutableState<T> : State<T> {
    override var value: T
}

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

operator fun <T> State<T>.getValue(obj: Any?,
    property: KProperty<*>): T = value

operator fun <T> MutableState<T>.setValue(obj: Any?,
    property: KProperty<*>, value: T) {
    this.value = value
}
```

Recompose Scopes

```
@Composable
fun MyComposable() {
    println("1 $currentRecomposeScope")        // scope A
    var i: Int by remember {
        mutableStateOf(0)
    }
    Column {
        println("2 $currentRecomposeScope")     // scope A
        Button(onClick = { i++; }) {     // write: inform subscribers
            println("3 $currentRecomposeScope") // scope B
            Text(text = "Click $i")          // read: subscribe
        }
        println("4 $currentRecomposeScope")     // scope A
        Text(text = "my text")
    }
}
```

Output on initial composition:

1 RecomposeScopeImpl@9722436

2 RecomposeScopeImpl@9722436

3 RecomposeScopeImpl@e1e92a5

4 RecomposeScopeImpl@9722436

Output on recomposition (onClick):

3 RecomposeScopeImpl@e1e92a5

Recompose Scopes

```
@Composable
fun MyComposable() {
    println("1 $currentRecomposeScope")        // scope A
    var i: Int by remember {
        mutableStateOf(0)
    }
    Column {
        println("2 $currentRecomposeScope")     // scope A
        Button(onClick = { i++; }) {     // write: inform subscribers
            println("3 $currentRecomposeScope") // scope B
            Text(text = "Click $i")          // read: subscribe
        }
        println("4 $currentRecomposeScope")     // scope A
        Text(text = "$i")                    // read: subscribe
    }
}
```

Output on initial composition:

1 RecomposeScopeImpl@9722436

2 RecomposeScopeImpl@9722436

3 RecomposeScopeImpl@e1e92a5

4 RecomposeScopeImpl@9722436

Output on recomposition (onClick):

1 RecomposeScopeImpl@9722436

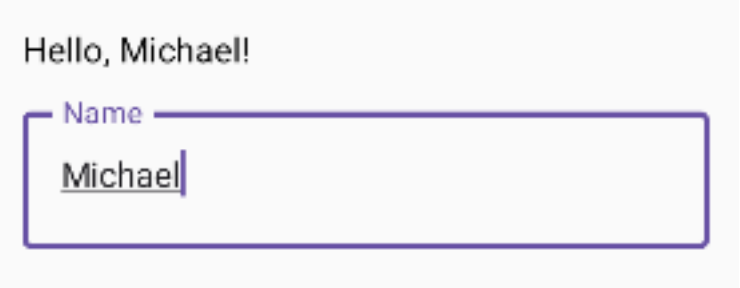
2 RecomposeScopeImpl@9722436

3 RecomposeScopeImpl@e1e92a5

4 RecomposeScopeImpl@9722436

Text Field Events, State in Composable

```
@Composable
fun MyTextField() {
    Column(modifier = Modifier.padding(16.dp)) {
        var name by remember { mutableStateOf("") }
        if (name.isNotEmpty()) {
            Text(text = "Hello, $name!")
        }
        OutlinedTextField(
            value = name,                          // read: subscribe
            onValueChange = { name = it },         // write: inform subscribers
            label = { Text("Name") }
        )
    }
}
```



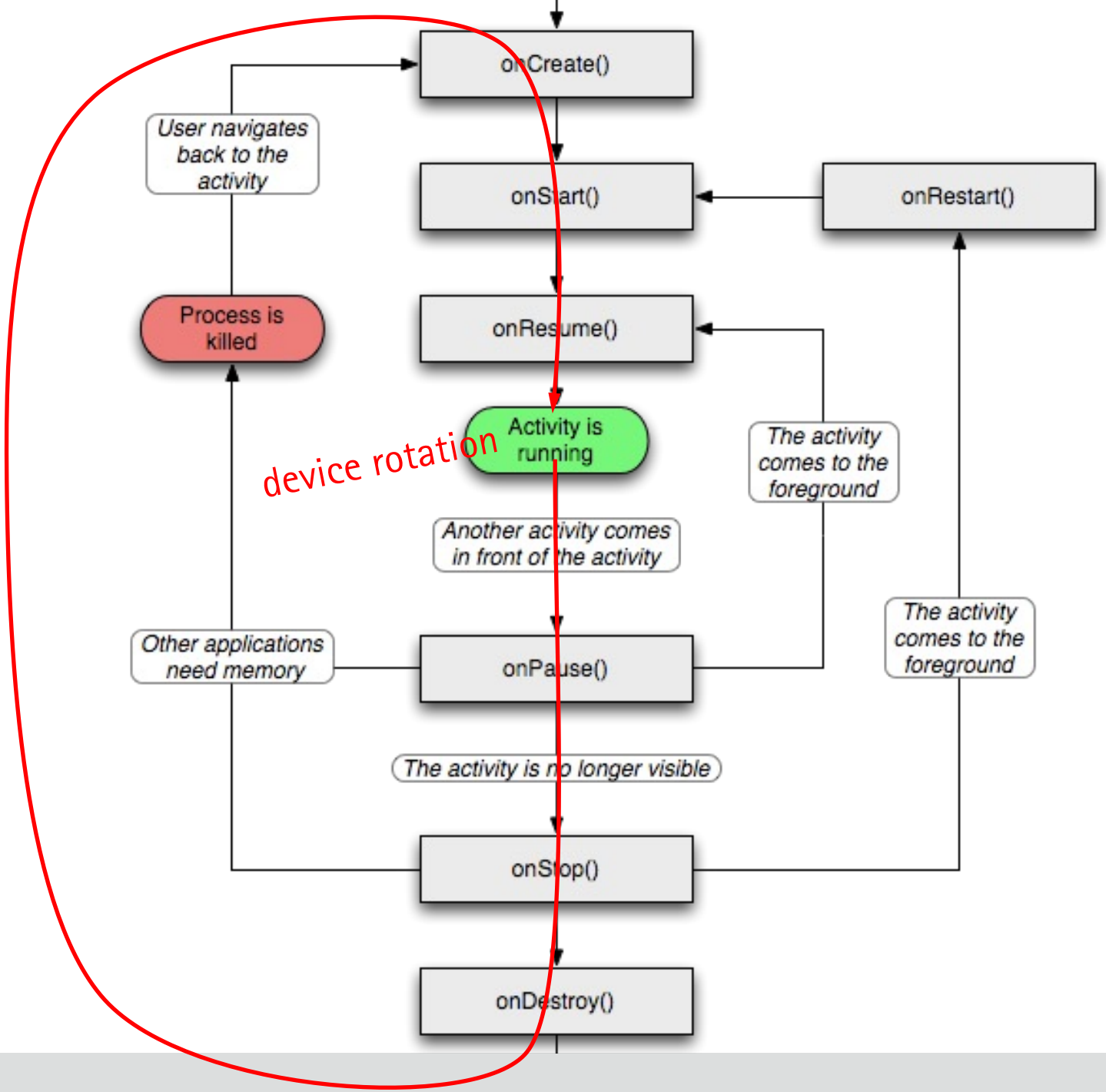
Text Field Events, State in View Model

```
@Composable
fun MyTextField(viewModel: MyViewModel) {
    Column(modifier = Modifier.padding(16.dp)) {

        if (viewModel.name.isNotEmpty()) {
            Text(text = "Hello, ${viewModel.name}!")
        }
        OutlinedTextField(
            value = viewModel.name,
            onValueChange = { viewModel.name = it },
            label = { Text("Name") }
        )
    }
}
```

```
class MyViewModel {
    var name by mutableStateOf("")
}

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val myViewModel = MyViewModel()
        setContent {
            MyTextField(myViewModel)
        }
    }
    ...
}
```

Problem: View model should not be created in activity, because activity may be recreated.

Activity Lifecycle



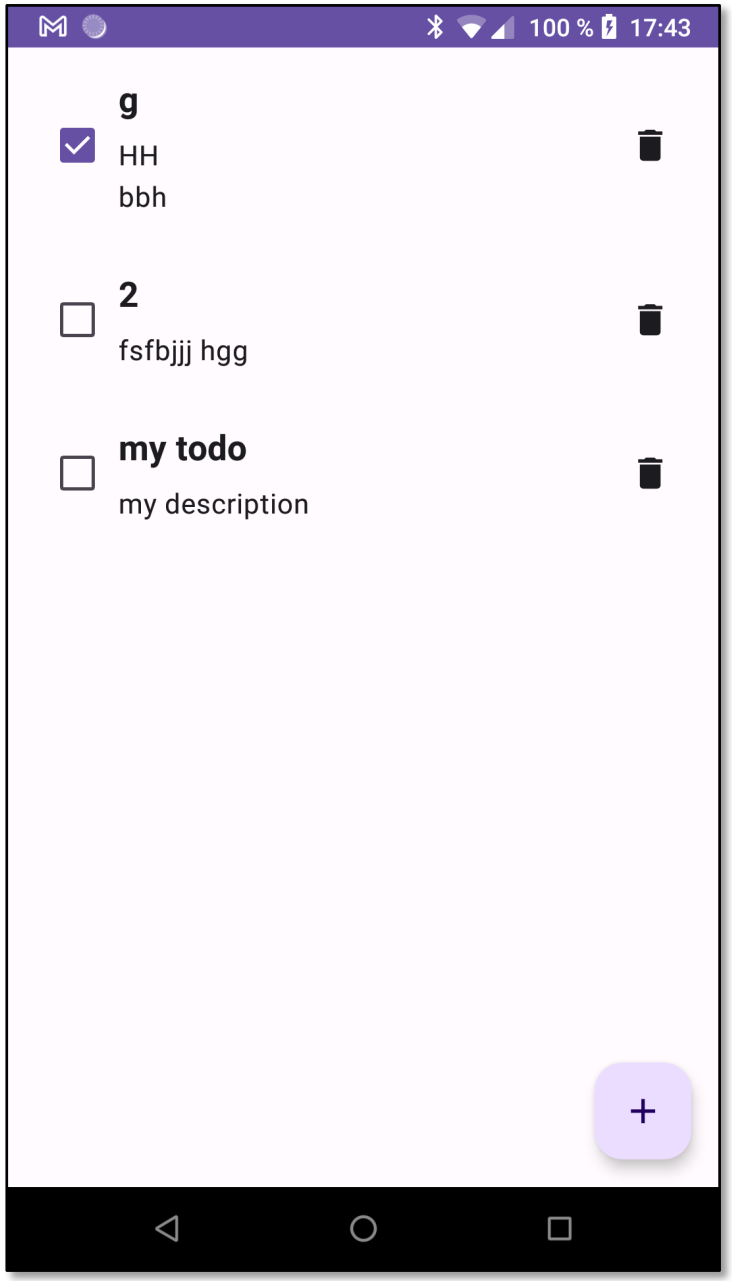
Text Field Events, State in View Model

```
@Composable
fun MyTextField(viewModel: MyViewModel) {
    Column(modifier = Modifier.padding(16.dp)) {

        if (viewModel.name.isNotEmpty()) {
            Text(text = "Hello, ${viewModel.name}!")
        }
        OutlinedTextField(
            value = viewModel.name,
            onValueChange = { viewModel.name = it },
            label = { Text("Name") }
        )
    }
}
```

```
class MyApp : Application() {
    val myViewModel = MyViewModel()
}

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        setContent {
            MyTextField((application as MyApp).myViewModel)
        }
    }
    ...
}
```

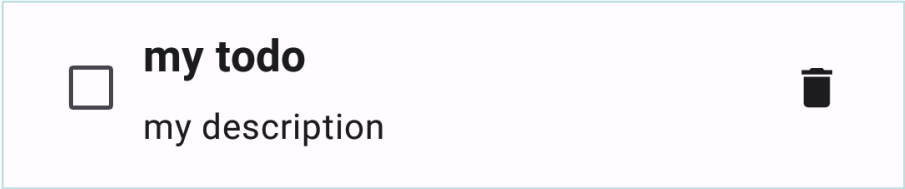
Problem: View model does not observe life cycle changes. → Use Compose factory to create and manage ViewModels

USE CASE: TODOAPP

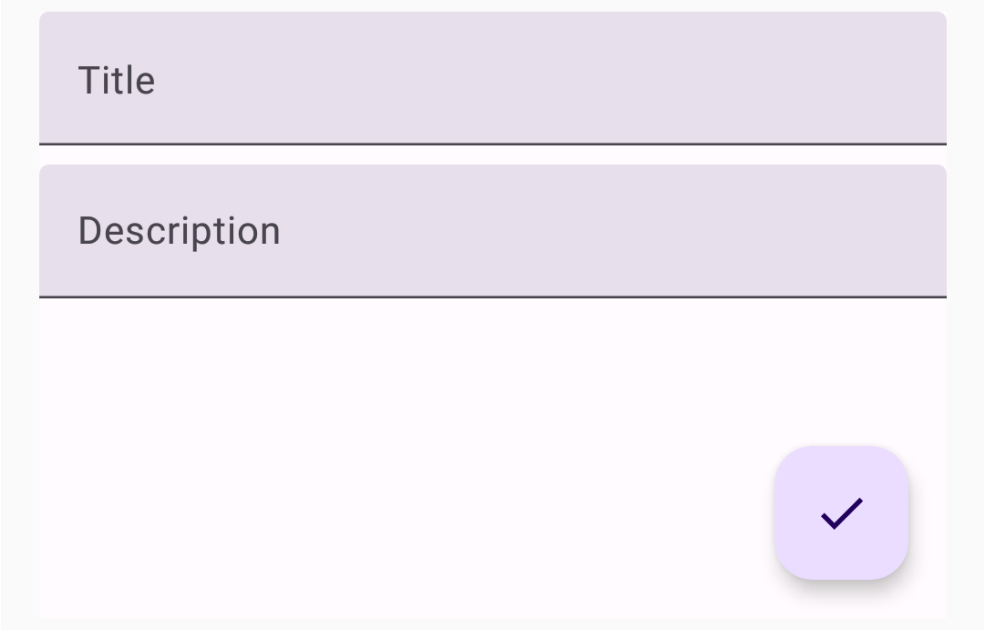


TodoApp Composables

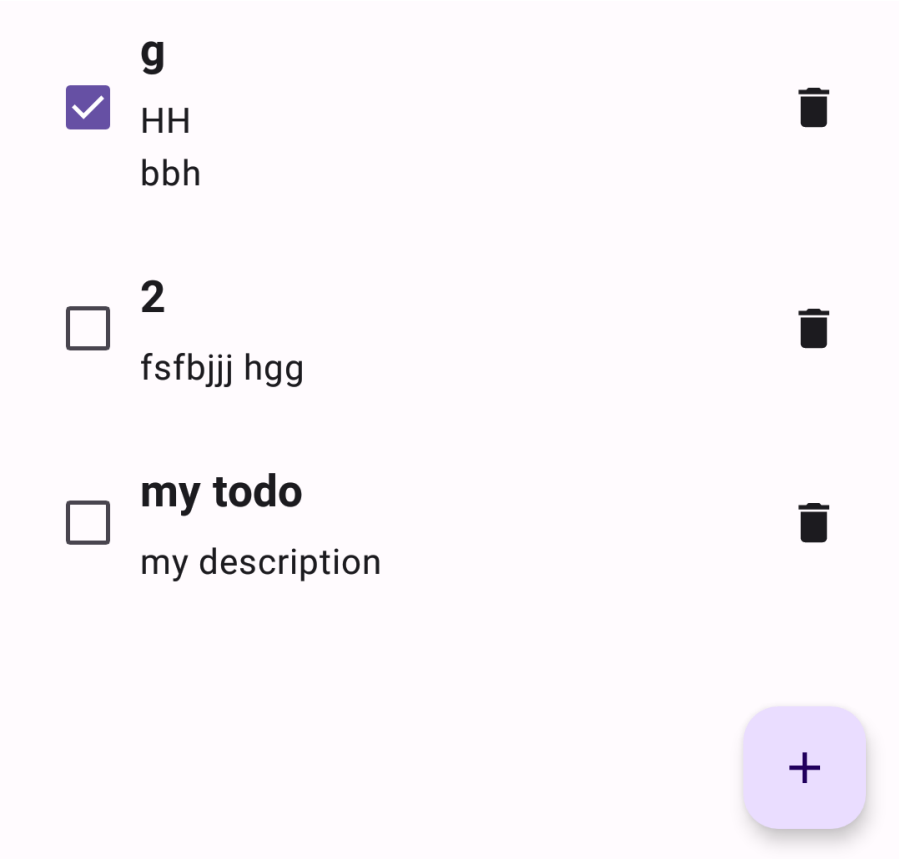
TodoItem:



AddEditTodoScreen:

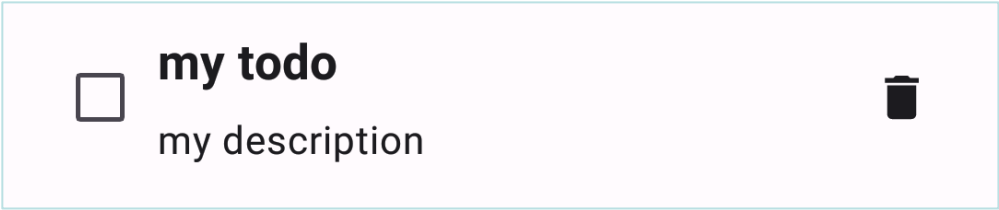


TodoListScreen:



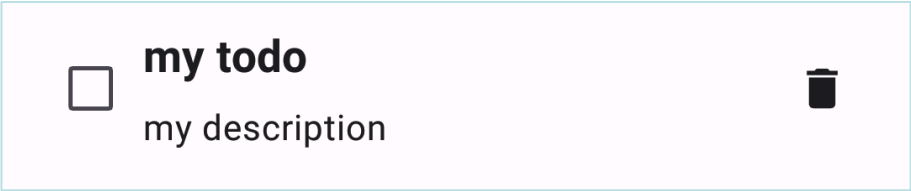
Todo Item

```
data class Todo(
    val title: String,
    val description: String,
    val isDone: Boolean,
    val id: Int? = null // primary key for database
)
```




```
@Composable
fun TodoItem(
    todo: Todo,
    deleteTodo: (Todo) -> Unit,
    todoDone: (todo: Todo, isDone: Boolean) -> Unit,
    modifier: Modifier = Modifier
) {
    Row(
        modifier = modifier, ...
    ) {
        Checkbox(...)
        Column(...) {
            Text(text = todo.title, ...) ...
            Text(text = todo.description)
        } ...
        IconButton(...) {
            Icon(
                imageVector = Icons.Default.Delete,
                contentDescription = "Delete"
            )
} } }
```

```
@Composable
fun TodoItem(
    todo: Todo,
    deleteTodo: (Todo) -> Unit,
    todoDone: (todo: Todo, isDone: Boolean) -> Unit,
    modifier: Modifier = Modifier
) {
    Row(
        modifier = modifier,
        verticalAlignment = Alignment.CenterVertically
    ) {
        Checkbox(
            checked = todo.isDone,
            onCheckedChange = { isChecked ->
                todoDone(todo, isChecked)
            }
        )
        ...
    }
}
```




```
Column(
    modifier = Modifier.weight(1f),
    verticalArrangement = Arrangement.Center
) {
    Text(
        text = todo.title,
        fontSize = 20.sp,
        fontWeight = FontWeight.Bold
    )
    if (todo.description.isNotEmpty()) {
        Spacer(modifier = Modifier.height(8.dp))
        Text(text = todo.description)
    }
}
Spacer(modifier = Modifier.width(8.dp))
IconButton(onClick = {
    deleteTodo(todo)
}) {
    Icon(imageVector = Icons.Default.Delete,
        contentDescription = "Delete"
    )
}
```