STATE MANAGEMENT AND ESSENTIAL KOTLIN SYNTAX

State Management

Theory:

State represents data about a component at a specific time. Changes in state update the UI.

Syntax:

Mutable State:

```
val state = mutableStateOf(initialValue)
```

Immutable State:

```
val state = remember { mutableStateOf(initialValue) }
```

Examples:

Mutable State:

```
val countState = mutableStateOf(0)
countState.value = 5
```

Immutable State with Remember:

```
val countState = remember { mutableStateOf(0) }
countState.value = 5
```

State in Composable:

```
@Composable
fun Counter() {
val count = remember {
mutableStateOf(0) }
Button(onClick = { count.value++ }) {
Text("Clicked ${count.value} times")
}
}
```

Remember

Theory:

remember preserves state across recompositions.

Syntax:

```
val state = remember { calculationOrState }
```

Examples:

Simple Calculation:

```
@Composable
fun ExampleComposable() {
val calculatedValue = remember { 2 * 2
}
}
```

Mutable State:

```
@Composable
fun Counter() {
val count = remember {
mutableStateOf(0) }
Button(onClick = { count.value++ }) {
Text("Clicked ${count.value} times")
}
}
```

Custom Object:

```
@Composable
fun CustomDataComposable() {
val customData = remember { CustomData("John Doe",30) }
}
```

mutableStateOf

Theory:

mutableStateOf creates a mutable state variable that updates the UI when changed.

Syntax:

```
val variableName = mutableStateOf(initialValue)
```

Examples:

Color Picker:

```
@Composable
fun ColorPicker() {
val color = remember { mutableStateOf(Color.Red) }
ColorButton(color = Color.Red, onClick = { color.va
lue = Color.Red })
Box(modifier = Modifier.background(color.value)) {
/* Content */ }
}
```

Visibility Toggle:

```
@Composable
fun VisibilityToggle() {
val isVisible = remember { mutableStateOf(true) }
Button(onClick = { isVisible.value = !isVisible.val
ue }) {
Text(if (isVisible.value) "Hide" else "Show")
}
}
```

Rating System:

```
@Composable
fun RatingSystem() {
val rating = remember { mutableStateOf(0) }
Button(onClick = { if (rating.value < 5) rating.val
ue++ }) {
Text("Upvote")
}
Text("Rating: ${rating.value}")
}
```

Random.nextBoolean()

Theory:

Generates a random boolean value.

Syntax:

```
val randomBoolean = Random.nextBoolean()
```

Examples:

Show/Hide Message:

```
fun showMessage() {
if (Random.nextBoolean()) println("Hello, Kotlin!")
else println("The message is hidden.")
}
```

Choose Option:

```
fun chooseOption() {
val option = if (Random.nextBoolean()) "Option A" e
lse "Option B"
println("The chosen option is: $option")
}
```

Change Background Color:

```
fun getBackgroundColor(): String {
return if (Random.nextBoolean()) "Red" else "Blue"
}
```

by Keyword

Theory:

Used for delegation to simplify state access.

Syntax:

```
var state by mutableStateOf(initialValue)
```

Examples:

Counter State:

```
@Composable
fun Counter() {
var count by remember { mutableStateOf(0) }
Button(onClick = { count++ }) {
Text("Clicked $count times")
}
}
```

Text State:

```
@Composable
fun TextInput() {
var text by remember { mutableStateOf("") }
TextField(value = text, onValueChange = { text = it
}, label = { Text("Enter text") })
}
```

Toggle State:

```
@Composable
fun ToggleButton() {
var toggled by remember { mutableStateOf(false) }
Button(onClick = { toggled = !toggled }) {
Text(if (toggled) "On" else "Off")
}
}
```

Label

Theory:

Labels provide context or information to UI elements.

Examples:

TextField Label:

```
@Composable
fun LabeledTextField() {
var text by remember { mutableStateOf("") }
TextField(value = text, onValueChange = { text = it
}, label = { Text("Enter your name") })
}
```

Styled Label:

```
@Composable
fun StyledLabel() {
Text(text = "Bold Label", fontWeight = FontWeight.B
old)
}
```

Typography Label:

```
@Composable
fun TypographyLabel() {
Text(text = "Styled Label", style =
MaterialTheme.typography.h6)
}
```

Recomposition

Theory:

Recomposition updates the UI in response to state changes.

Examples:

Simple Counter:

```
@Composable
fun Counter() {
val count = remember { mutableStateOf(0) }
Button(onClick = { count.value++ }) {
Text("Clicked ${count.value} times")
}
}
```

Toggle Button:

```
@Composable
fun ToggleButton() {
val toggled = remember { mutableStateOf(false) }
Button(onClick = { toggled.value = !toggled.value
}) {
Text(if (toggled.value) "On" else "Off")
}
}
```

Text Input:

```
@Composable
fun TextInput() {
val text = remember { mutableStateOf("") }
TextField(value = text.value, onValueChange = { tex
t.value = it }, label = { Text("Enter text") })
}
```

.toDoubleOrNull()

Theory:

Converts a string to a double, returning null if invalid.

Syntax:

```
val doubleValue: Double? = stringValue.toDoubleOrNull()
```

Examples:

Valid String:

```
val stringValue = "123.45"
val doubleValue = stringValue.toDoubleOrNull()
println(doubleValue ?: "Invalid double.")
```

Invalid String:

```
val stringValue = "Hello"
val doubleValue = stringValue.toDoubleOrNull()
println(doubleValue ?: "Invalid double.")
```

String with Spaces:

```
val stringValue = " 123.45 "
val doubleValue = stringValue.toDoubleOrNull()
println(doubleValue ?: "Invalid double.")
```

tutorials.EU

Elvis Operator

Theory:

Provides default values for nullable expressions.

Syntax:

```
val result = nullableExpression ?: fallbackValue
```

Examples:

Nullable String:

```
val name: String? = null
val displayName = name ?: "Guest"
println(displayName)
```

Non-Null Value:

```
val age: Int? = 25
val displayAge = age ?: 0
println(displayAge)
```

Function Return:

```
fun getLength(str: String?): Int {
return str?.length ?: 0
}
println(getLength("Hello"))
println(getLength(null))
```