Mobile Applications
CSCI 448
Lecture 04

Event Listeners

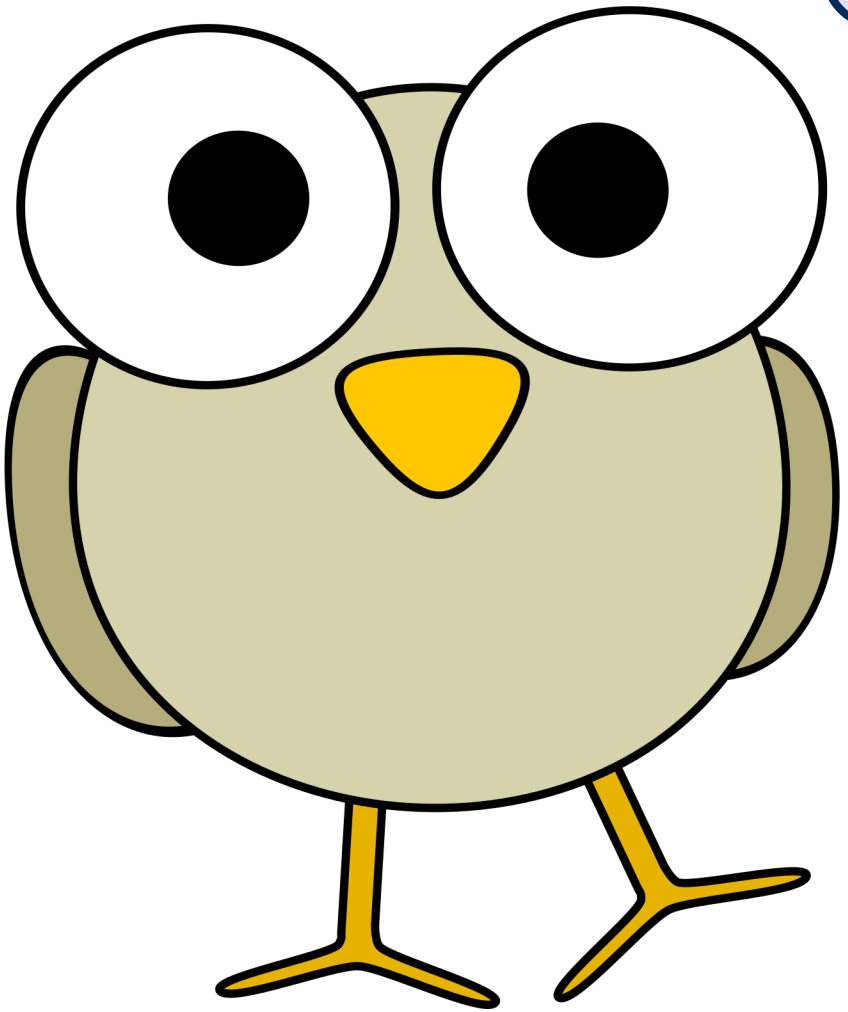
Recomposing

Download TempConverter Starter Code

Previously in CSCI 448

- **Model** – View – View Model
  - Represents and holds data
  - Specifies structure
  - Loads/persists data (to database or repository)

Questions?



Learning Outcomes For Today

- Access an application's context at runtime.
- Handle events in an app.
- Explain when a composable gets recomposed.
- Explain how Compose preserves unidirectional data flow.

On Tap For Today

- Event Listeners
- Recomposing
- Unidirectional Data Flow

On Tap For Today

- Event Listeners
- Recomposing
- Unidirectional Data Flow

Functional Programming

```
class MainActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
      MainActivityContent()
    }
  }

  @Composable
  private fun MainActivityContent() {
    ...
  }
}
```

ComponentActivity.setContent()

```
public fun ComponentActivity.setContent(
  content: @Composable -> Unit
) {
  ...
}
```

- Slot API
  - Can nest composable content

Functional Programming

```
class MainActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent(
      content = { MainActivityContent() } // pass function literal
                                          // as lambda expression
    )
  }

  @Composable
  private fun MainActivityContent() {
    ...
  }
}
```

Functional Programming

```
class MainActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {                    // if last parameter is a function
      MainActivityContent()         // then the lambda can be places outside ()
    }                               // this is called a trailing lambda
  }

  @Composable
  private fun MainActivityContent() {
    ...
  }
}
```

Compose Kotlin Style

- Default Activity

```
setContent {
    SamodelkinComposeTheme {
        // A surface container using the 'background' color from the theme
        Surface(
            modifier = Modifier.fillMaxSize(),
            color = MaterialTheme.colorScheme.background
        ) {
            Greeting("Android")
        }
    }
}


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

Compose Kotlin Style

- Do a function substitution

```
setContent {
    SamodelkinComposeTheme {
        // A surface container using the 'background' color from the theme
        Surface(
            modifier = Modifier.fillMaxSize(),
            color = MaterialTheme.colorScheme.background
        ) {
            Text(text = "Hello Android!")
        }
    }
}
```

Compose Kotlin Style

- Begin collapsing

```
setContent {
    SamodelkinComposeTheme {
        // A surface container using the 'background' color from the theme
        Surface(
            modifier = Modifier.fillMaxSize(),
            color = MaterialTheme.colorScheme.background,
            content = { Text(text = "Hello Android!") }
        )
    }
}
```

Compose Kotlin Style

- Continue collapsing

```
setContent {
    SamodelkinComposeTheme(
        content = {
            Surface(
                modifier = Modifier.fillMaxSize(),
                color = MaterialTheme.colorScheme.background,
                content = { Text(text = "Hello Android!") }
            )
        }
    )
}
```

Compose Kotlin Style

- Continue collapsing

```
setContent(
    content = {
        SamodelkinComposeTheme(
            content = {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background,
                    content = { Text(text = "Hello Android!") }
                )
            }
        )
    }
)
```

Compose Kotlin Style

- Remove whitespace

```
setContent(
    content = {
        SamodelkinComposeTheme(
            content = {
                Surface(modifier = Modifier.fillMaxSize(), color =
MaterialTheme.colorScheme.background, content = { Text(text = "Hello Android!")
}  )
            }
        )
    }
)
```

Compose Kotlin Style

- Remove whitespace

```
setContent(
    content = {
        SamodelkinComposeTheme( content = { Surface(modifier =
Modifier.fillMaxSize(), color = MaterialTheme.colorScheme.background, content =
{ Text(text = "Hello Android!") }  ) } )
    }
)
```

Compose Kotlin Style

- Remove whitespace

```
setContent( content = { SamodelkinComposeTheme( content = { Surface(modifier =
Modifier.fillMaxSize(), color = MaterialTheme.colorScheme.background, content =
{ Text(text = "Hello Android!") }  ) } ) } )
```

Compose Kotlin Style

- Remove named arguments

```
setContent( { SamodelkinComposeTheme( { Surface(Modifier.fillMaxSize(),
MaterialTheme.colorScheme.background, { Text("Hello Android!") }  ) } ) } )
```

Compose Kotlin Style

- Remove arguments

```
setContent( SamodelkinComposeTheme( Surface( Text() ) ) )
```

Yay Functional Programming

- Remove arguments

```
setContent( SamodelkinComposeTheme( Surface( Text() ) ) )
```

Yay Kotlin

- Leverage Kotlin notation, improve readability

```
setContent {
    SamodelkinComposeTheme {
        // A surface container using the 'background' color from the theme
        Surface(
            modifier = Modifier.fillMaxSize(),
            color = MaterialTheme.colorScheme.background
        ) {
            Greeting("Android")
        }
    }
}


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

Yay Kotlin

- Further code styling

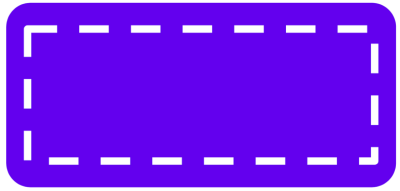
```
setContent {
    SamodelkinComposeTheme {
        // A surface container using the 'background' color from the theme
        Surface(
            modifier = Modifier
                .fillMaxSize()
                .padding(16.dp),
            color = MaterialTheme.colorScheme.background
        ) {
            Greeting("Android")
        }
    }
}


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

Slot API

```
Button {
    Text("Button")
}
```


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


```
@Composable
fun Button(
    onClick: () -> Unit,
    modifier: Modifier = Modifier,
    content: @Composable RowScope.() -> Unit
): Unit
```



Button

- Expects multiple function parameters

```
@Composable
fun Button(
    onClick: () -> Unit,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    interactionSource: MutableInteractionSource = remember { MutableInteractionSource() },
    elevation: ButtonElevation? = ButtonDefaults.elevation(),
    shape: Shape = MaterialTheme.shapes.small,
    border: BorderStroke? = null,
    colors: ButtonColors = ButtonDefaults.buttonColors(),
    contentPadding: PaddingValues = ButtonDefaults.ContentPadding,
    content: @Composable RowScope.() -> Unit
): Unit
```

Button onClick

- Example of event handler
  - Click “event” fires
  - Function “handles” event
    - Executes associated function
- Let’s add to TempConverter project!

Design Pattern #1: Command

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queues or log requests, and support undoable operations.
- Participants:
  - `Command`: declares an interface for executing an operation
  - `ConcreteCommand`: defines binding between `Receiver` and an action, implements execution by invoking action on `Receiver`
  - `Client`: creates `ConcreteCommand` and sets its `Receiver`
  - `Invoker`: asks the command to carry out the request
  - `Receiver`: knows how to perform the operations associated with carrying out a request

Event Handling Callback Participants

- `Command` →
- `ConcreteCommand` →
- `Client` →
- `Invoker` →
- `Receiver` →

Android Design Patterns

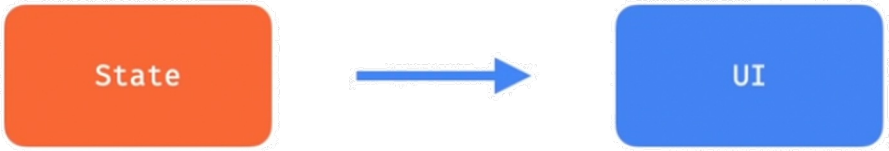
- Behavioral Patterns
  1. Command – UI Event Handling

On Tap For Today

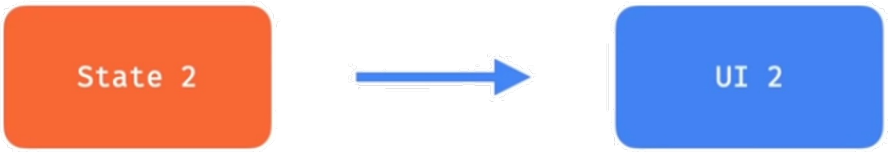
- Event Listeners
- Recomposing
- Unidirectional Data Flow

Composing

- Given state, composable emits corresponding UI

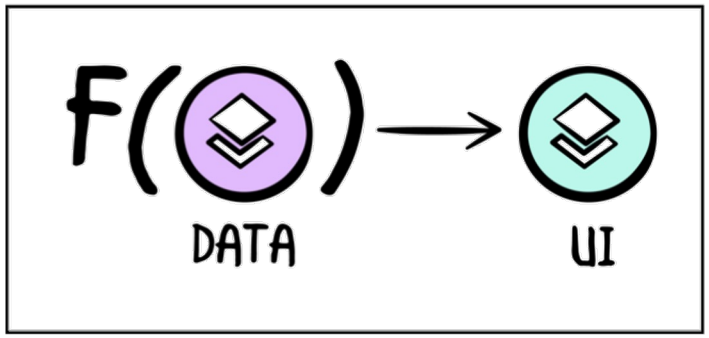


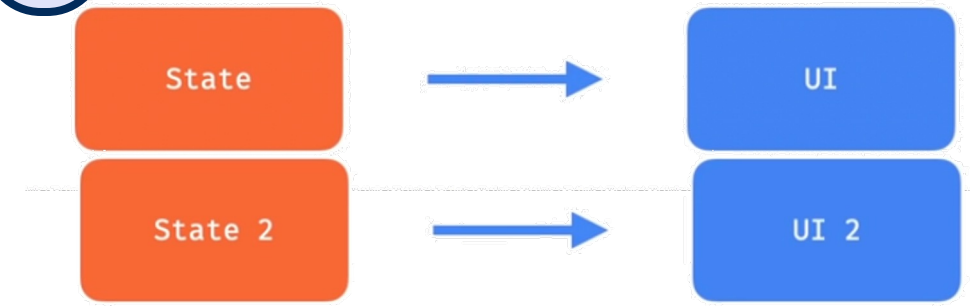
- UI is
  - Idempotent & immutable: there are no objects
  - Dynamic: different inputs → Different UI



- How to make different UI?

Different UI

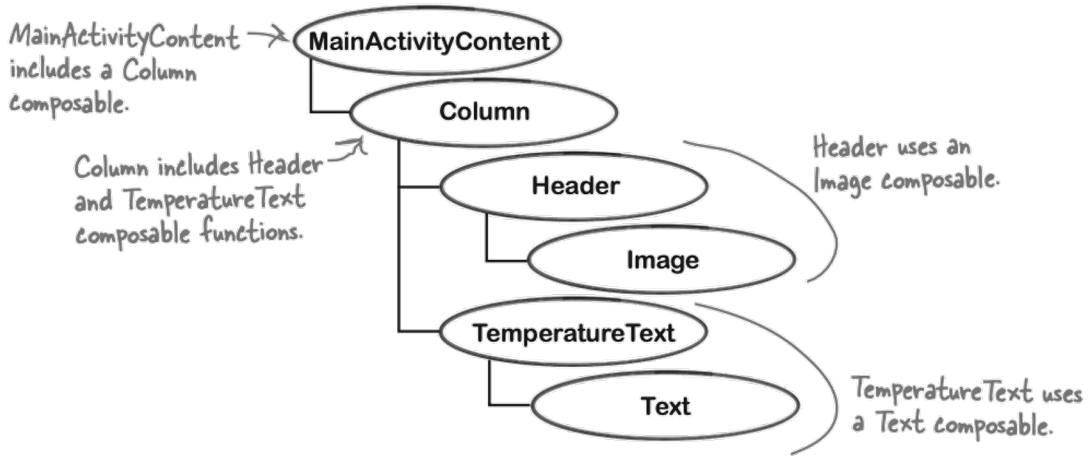




- Call it with different state!
- How?
  - Manually calling F(1) and F(2) would display both sets of UI

Composable Tree

- When first building UI, hierarchical tree of all composables created



- When the input to `TemperatureText` changes, we need to replace it with the new UI
  - This is known as **recomposing**

Recomposing

- Composable gets recomposed when the values it depends on get updated

```
@Composable
private fun TemperatureText(celsius: Double) {
    Log.d(LOG_TAG,  msg: "emitting TemperatureText")
    val fahrenheit = (celsius*9.0/5.0)+32.0
    Text( stringResource(id = "%1$.1f℃ is %2$.2f℉", celsius, fahrenheit) )
}
```

- How may it be called?
  - `TemperatureText(0.0)`
  - `var celsiusTemp = 0.0`
    `TemperatureText(celsiusTemp)`

Recomposing

- Composable gets recomposed when the values it depends on get updated

```
@Composable
private fun TemperatureText(celsius: Double) {
    Log.d(LOG_TAG,  msg: "emitting TemperatureText")
    val fahrenheit = (celsius*9.0/5.0)+32.0
    Text( stringResource(id = "%1$.1f℃ is %2$.2f℉", celsius, fahrenheit) )
}
```

- Compose skips as much as possible (only updates what has changed)
- Compose is optimistic (expects to finish before parameters change again)
- Compose can run frequently

Updating Temperature

- Concept:

```
@Composable
fun UI() {
  Column {
    var celsius = 0.0
    TemperatureText(celsius)
    Button(onClick = { celsius = 100.0 }) {
      Text("Boil!")
    }
  }
}

@Composable
fun TemperatureText(celsius: Double) {
  val fahrenheit = (celsius*9.0/5.0) + 32.0
  Text("$celsius C = $fahrenheit F")
}
```

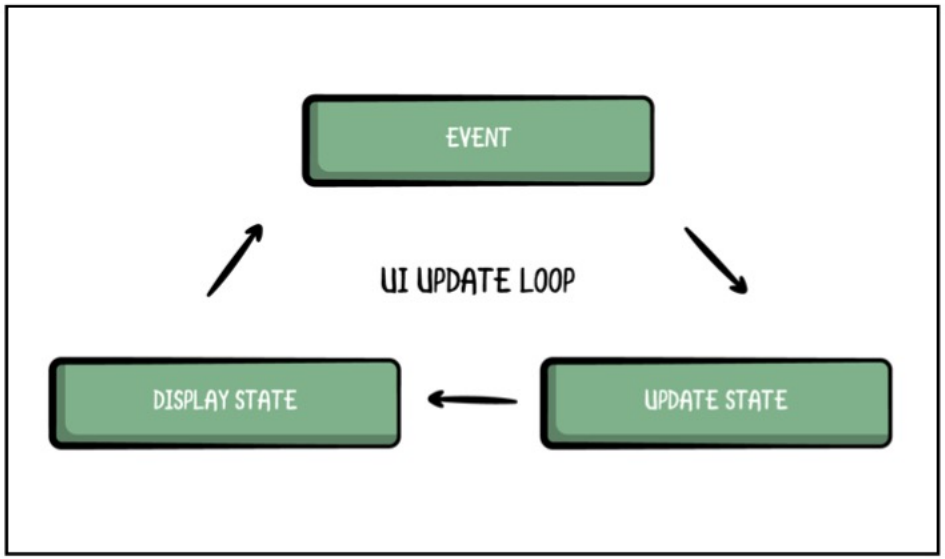
- But how does `TemperatureText` know `celsius` has changed?

On Tap For Today

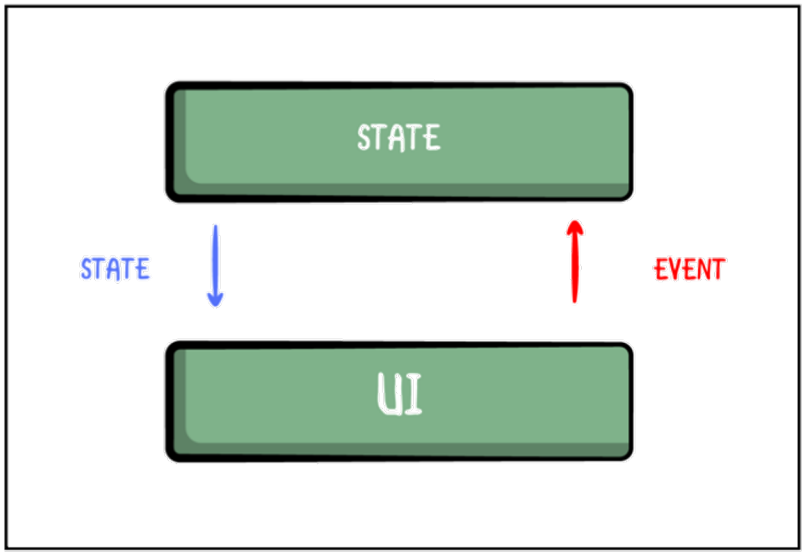
- Event Listeners
- Recomposing
- Unidirectional Data Flow

Unidirectional Data Flow

- External events trigger change in state



Single Source of Truth

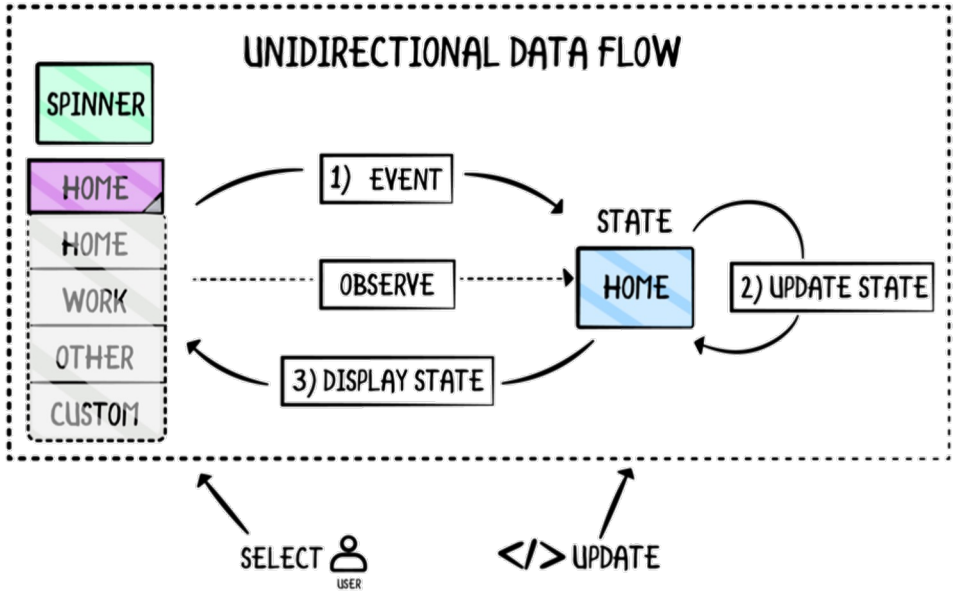


Unidirectional Data Flow

- Keep one state
- State “flows” down
- Events “flow” up
- UI “observes” the state

Flow in MVVM **Compose** Framework

1. Spinner fires event to alert state new value has been selected
2. State updates itself
3. Spinner sees state has changed, updates itself



Where to Store State???

- What is the single source of truth?
  1. Composable - In the composable itself
     - A **stateful** composable
     - Can change state itself
  2. ViewModel - “Hoist” the state to the caller of the composable
     - A **stateless** composable
     - Composable requires parameter and event
  3. StateHolder
     - Separate class that stores UI logic & UI element states

On Tap For Today

- Event Listeners
- Recomposing
- Unidirectional Data Flow

To Do For Next Time

- Due now:
  - Early Feedback survey
  - Access code: **toast**
- Due tonight: Kotlin Classes quiz
- Due tomorrow: Elevator Pitch feedback
- Next time:
  - Kotlin Collections Quiz
  - Stateful Composables
  - Lab01C