Android

Jetpack Compose

Mark Joselli

mark.joselli@pucpr.br

Jetpack compose

- A new toolkit for building native user interfaces on Android.
- Jetpack Compose is a modern toolkit developed by Google for building user interfaces (UI) on Android, based on a declarative model.
- Released to simplify UI development.
- It allows developers to create UI directly with Kotlin code, without the need for XML, simplifying the development process.

Why Jetpack Compose?

- Simplifying UI Development
- With Jetpack Compose, you no longer need to create XML layouts separate from your code. Instead, the entire UI is declared directly in the Kotlin code.
  - This reduces fragmentation between layout XML files and application logic, resulting in a cleaner, more efficient workflow.
- UI code is more concise and readable, eliminating the need to work with long imperative classes and methods to modify interface components.

Why Jetpack Compose?

- Declarative Model
  - Jetpack Compose uses a declarative model, in which you describe what the interface should display based on the current state of the application.
- This means that instead of manually manipulating the interface to reflect changes in state, Compose automatically reacts to data changes and updates the UI accordingly.
- This model improves code clarity, making it easier to manage complex and dynamic UIs.

Why Jetpack Compose?

- Less Boilerplate Code
- Jetpack Compose eliminates much of the repetitive code developers face when working with traditional Views and XML.
- Less code means less chance of errors, as well as easier-to-read and maintain interfaces.

Why Jetpack Compose?

- Better Productivity
- Jetpack Compose offers more fluid integration with Android Studio, including tools such as interactive preview, which allows you to view interface changes in real time without having to compile the project.
- Additionally, Compose supports hot-reload, which allows you to instantly see changes to the UI while the application is running, which speeds up the development cycle.

Why Jetpack Compose?

- Kotlin integration
- Compose is built on Kotlin, which allows developers to leverage the language's modern features, such as high-order functions and lambda expressions, to create more flexible and powerful UIs.
- Leveraging the power of Kotlin also reduces the need for extensive code and increases the expressiveness of the language.

Why Jetpack Compose?

- Reactive State Management
- Compose makes it easy to manage state within user interfaces using reactive APIs like State and remember.
- This makes it much simpler to create UIs that respond to real-time changes, such as displaying updated search results or lists of data.

Why Jetpack Compose?

- Ease of Adaptation and Integration with Existing Projects
- Compatibility with Traditional Views: Although Jetpack Compose is new, it can coexist with Android's traditional Views system. This facilitates gradual adoption into existing projects, allowing developers to migrate parts of the interface to Compose as needed.
- Flexibility: You can start using Compose in smaller parts of your application and gradually convert the entire interface to the new framework.

Why Jetpack Compose?

- Integrated Material Design
- Compose has seamless integration with Material Design, allowing you to quickly create consistent, modern interfaces. Material Design components are directly available and easy to customize.
- Improved Performance
- Compose's architecture is highly optimized for rendering complex interfaces and dynamic UI changes. It performs recompositions only on the parts of the interface that really need to be updated, reducing resource usage and improving the application's overall performance.

Imperative Model

- The imperative model is the traditional method of building UIs on Android, where the developer tells the system how to build the interface and what should be done at each step. This is generally done using XML to define layouts and Java/Kotlin code to modify the behavior of components at runtime.
- Features:
- Defines Step by Step:
- The developer needs to define the interface layout and then control the behavior of each element individually, specifying each action and response to events.
  - Example: When clicking a button, you need to capture the event and manually change the text of a TextView through code.

Imperative Model

- Separation of Logic and Interface:
- Typically, the UI is defined in XML files, while the logic is in Kotlin or Java code. This results in multiple points of interaction (between XML and code) to define and change the interface.
- Explicit State Maintenance:
- The developer is responsible for explicitly managing the state of the UI, which means they need to ensure that the UI updates correctly when the state changes.

Challenges of the Imperative Model

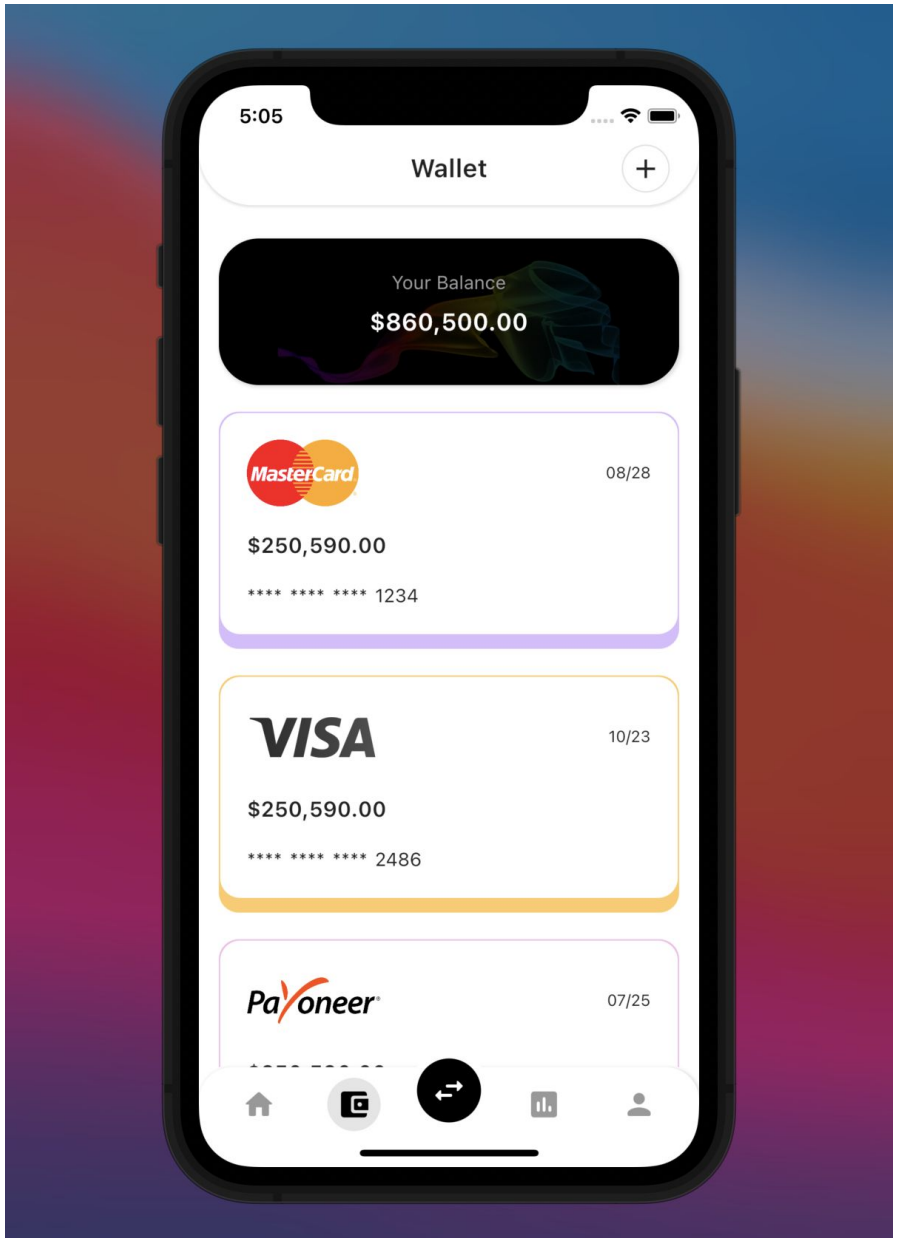
- Complexity:
- For dynamic interfaces, where the state changes frequently, the developer needs to manually ensure that the UI updates correctly with each change, which can lead to bugs and confusing code.
- Maintenance:
- Keeping track of what is being displayed in the UI can become difficult as the complexity of the application increases, as various parts of the code need to be updated.

Modelo Imperativo

```
<!-- Layout XML -->
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Click me"/>
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello"/>
```

```
val button = findViewById<Button>(R.id.button)
val textView = findViewById<TextView>(R.id.textView)

button.setOnClickListener {
    textView.text = "Button clicked!"
}
```


5:05
Wallet
Your Balance
$860,500.00
MasterCard
08/28
$250,590.00
**** **** **** 1234
VISA
10/23
$250,590.00
**** **** **** 2486
Payoneer
07/25

Declarative Model

- In the declarative model, like Jetpack Compose, the developer describes what the interface should show based on the current state, and the framework takes care of rendering that interface automatically, without having to explicitly define how each element should change over time.
- Features:
- Describes the UI State:
  - Instead of telling the system "how" to make changes, the developer simply defines "what" the interface should display. If the state changes, Compose takes care of updating the UI to reflect that change.

Declarative Model

- Integration with the State:
  - Jetpack Compose automatically manages the recomposition of parts of the UI when the state associated with it changes. This eliminates the need for manual UI updates.
- Life Cycle Controlled by the Framework:
  - The UI is reactive and, unlike the imperative model, the framework is responsible for reacting to changes in state and ensuring that the interface always reflects these states.

Declarative Model

```
@Composable
fun MyButton() {
    var clicked by remember { mutableStateOf(0) }

    Column {
        Button(onClick = { clicked++ }) {
            Text("Click me")
        }
        Text(text = "Button clicked $clicked times")
    }
}
```

Advantages of the Declarative Model

- **Simplicity:** The developer does not need to manually control each interface change. Instead, just set the UI based on the current state.
- **Readability:** The code is more concise and easier to understand, as the interface is defined as a pure function that reacts to state changes.
- **Fewer Errors:** As the framework takes care of ensuring that the UI is always synchronized with the state, there is less chance of errors when manipulating the interface.

Jetpack Compose Concepts - @Composable

- Functions annotated with @Composable are the fundamental building blocks of Jetpack Compose.
- They describe how the user interface should be rendered.
- Every function that draws something in the UI is marked with @Composable, indicating that this function is part of the Jetpack Compose composition system.
- These are functions that return UI instead of a value.
- They can be combined with each other to build complex interfaces.
- Each @Composable can call other composable functions.

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

Jetpack Compose Concepts - Status

- State management in Jetpack Compose is a core concept for creating dynamic user interfaces that automatically react to changes in data.
- State defines the data that the UI should display or respond to, and Compose takes care of ensuring that the interface is updated whenever that state changes.
- State refers to the data that controls what is being displayed in the user interface. When this data changes, the interface needs to be recomposed (re-rendered) to reflect the new information.
- In Jetpack Compose, state is managed reactively. This means that the UI automatically reacts to state changes without the developer having to manually update each component.

Jetpack Compose Concepts - Status

- remember is used to store values during composition.
- It preserves state between recompositions, ensuring that values persist as long as the composable function is recomposed.
- It is useful to avoid data loss when the interface is re-evaluated.
- In the example, remember preserves the value of count during recompositions, and mutableStateOf is used to define a mutable state that, when changed, triggers recomposition.

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }

    Button(onClick = { count++ }) {
        Text("Clicked $count times")
    }
}
```

Jetpack Compose Concepts - Status

- mutableStateOf creates a state object that Jetpack Compose observes. When the value of this state changes, Compose automatically recomposes the UI that depends on this value.
- It is generally used with by to simplify the syntax.
- In the example, when the button is clicked, the name state changes and the text is automatically updated.

```
@Composable
fun Greeting() {
    var name by remember { mutableStateOf("John") }
    Column {
        Text("Hello, $name!")
        Button(onClick = { name = "Jane" }) {
            Text("Change Name")
        }
    }
}
```

Jetpack Compose Concepts - State Hoisting

- State Hoisting is a common pattern in Jetpack Compose that facilitates state centralization and allows different composables to share and modify the same state.
- Rather than managing state locally within a composable function, the state is hoisted to a higher level, and passed to the composable functions via parameters.

Jetpack Compose Concepts - State Hoisting

In the example, the count state is maintained in the ParentComposable, and the Counter function receives this state as a parameter, as well as the onIncrement action to update the state.

```
@Composable
fun ParentComposable() {
    var count by remember { mutableStateOf(0) }
    Counter(count = count, onIncrement = { count++ })
}

@Composable
fun Counter(count: Int, onIncrement: () -> Unit) {
    Button(onClick = onIncrement) {
        Text("Clicked $count times")
    }
}
```

Jetpack Compose Concepts - Recomposition

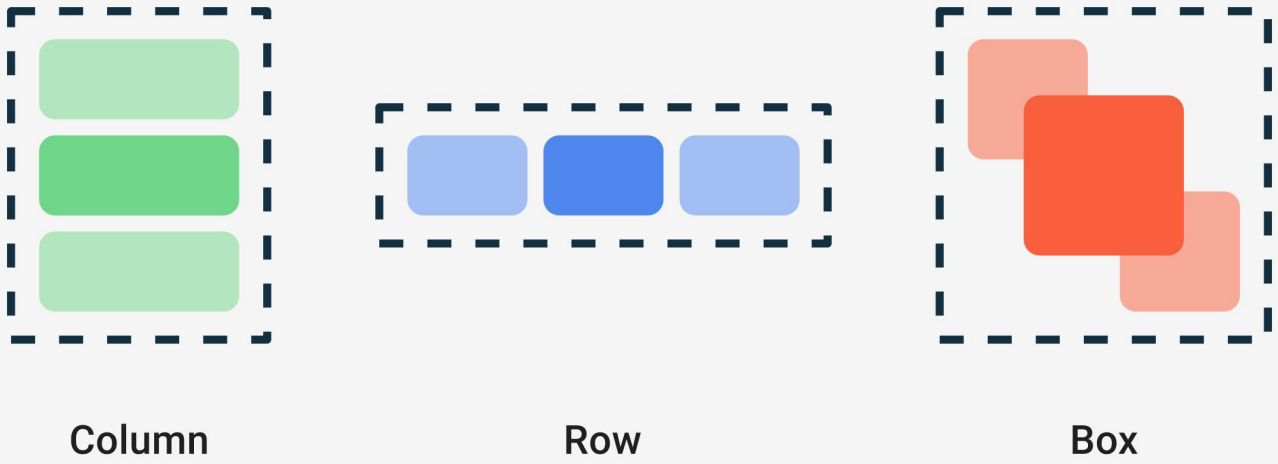
- Recomposition is the process in which Jetpack Compose re-executes @Composable functions to update the UI. This occurs when the state observed by these functions changes.
- Recomposition is automatic and efficient, occurring only in the parts of the interface that need to be updated.
- How recomposition works:
- Whenever a mutableStateOf is changed, all composables that depend on that state will be recomposed.
- However, Jetpack Compose optimizes recomposition, and only the parts of the UI tree that actually need to be updated will be recomposed, avoiding unnecessary rendering.
- Benefit: This eliminates the need for developers to manually update the UI when the state changes, as Compose takes care of the synchronization.

Jetpack Compose Concepts - Layouts

- Layouts are composable functions that organize visual elements on a screen.
- Unlike the Views system, layouts in Compose are more flexible and less verbose.
- Main Layouts:
  - Column: Arranges elements vertically.
  - Row: Arranges elements horizontally.
  - Box: Stacks elements on top of each other.

```
@Composable
fun MyScreen() {
    Column {
        Text(text = "Hello")
        Button(onClick = { /* ação */ }) {
            Text(text = "Click me")
        }
    }
}
```

Jetpack Compose Concepts - Layouts

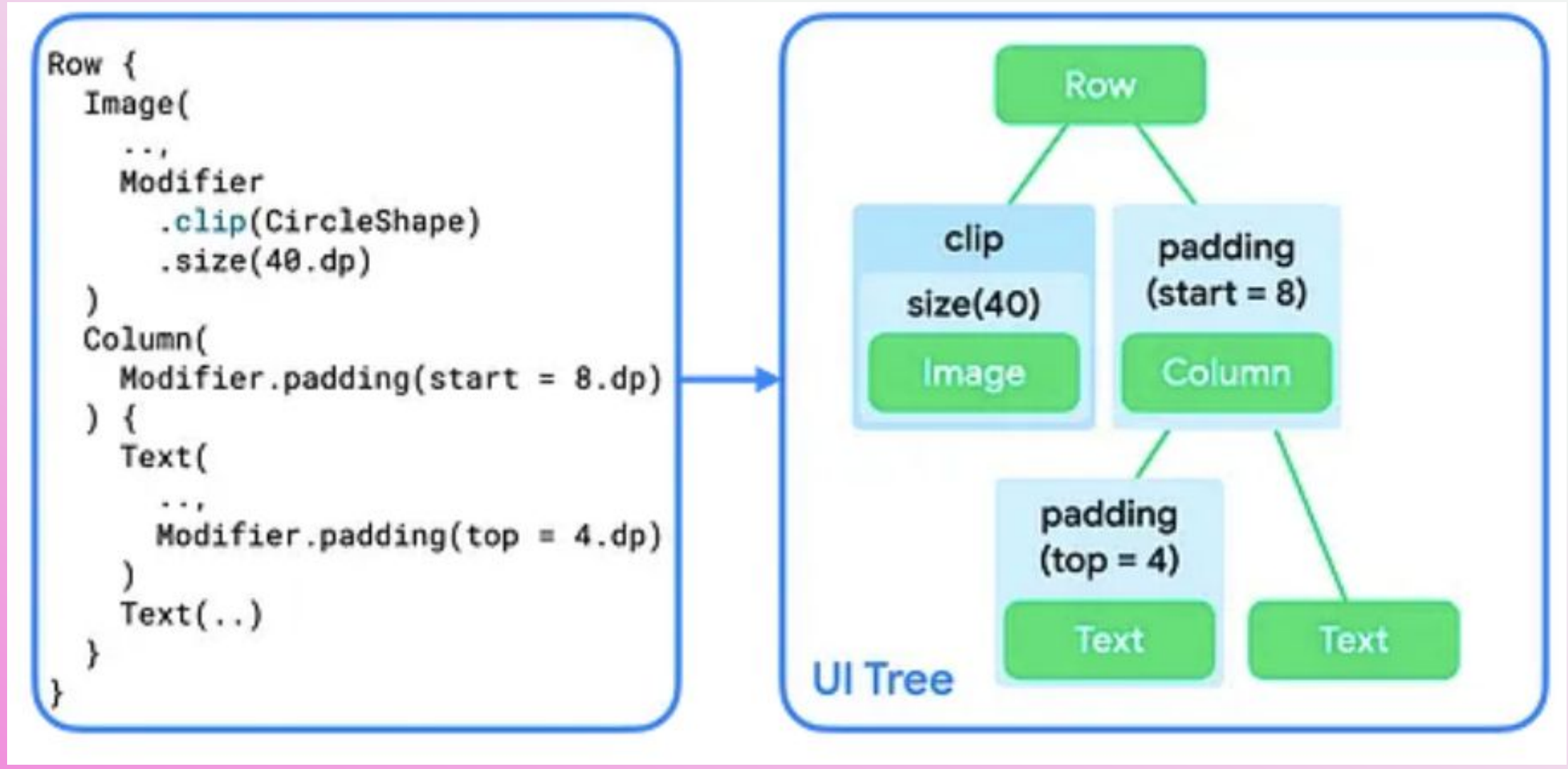



```
@Composable
fun MyScreen() {
    Column {
        Text(text = "Hello")
        Button(onClick = { /* ação */ }) {
            Text(text = "Click me")
        }
    }
}
```

Jetpack Compose Concepts - Modifiers

- Modifiers are used to change the appearance or behavior of a composable element, such as defining dimensions, positioning, colors, borders, etc.
- Modifiers can be applied to any composable element and are highly chainable.

```
@Composable
fun ModifiedButton() {
    Button(
        onClick = { /* action */ },
        modifier = Modifier.padding(16.dp).fillMaxWidth()
    ) {
        Text("Click me")
    }
}
```


Row {
Image(
..,
Modifier
.clip(CircleShape)
.size(40.dp)
)
Column(
Modifier.padding(start = 8.dp)
) {
Text(
..,
Modifier.padding(top = 4.dp)
)
Text(..)
}
}
Row
clip
size(40)
Image
padding
(start = 8)
Column
padding
(top = 4)
Text
Text
UI Tree

Jetpack Compose Concepts - Side Effects

- These are operations that occur outside of the composition cycle, such as executing code once or interacting with APIs that are not directly part of the UI.
- Main Side Effects:
- LaunchedEffect: Executes a block of code when a specific key changes.
- rememberCoroutineScope: Used to create and manage coroutines within composables.

```
@Composable
fun MyComposable() {
    LaunchedEffect(Unit) {
        // Execute just on the first time the composable is
rendered
    }
}
```

Jetpack Compose Concepts - Preview

- Definition: The Preview feature in Android Studio allows you to preview a @Composable function without the need to run the application on an emulator or physical device.
- Preview is useful for seeing UI changes in real time as you develop.

```
@Preview
@Composable
fun PreviewGreeting() {
    Greeting(name = "Preview")
}
```

Conceitos Jetpack Compose - Navegação

- Jetpack Compose offers its own API for navigating between different screens within an application, replacing the Intents and Fragments system used in traditional Android.

```
val navController = rememberNavController()

NavHost(navController, startDestination = "home") {
    composable("home") { HomeScreen(navController) }
    composable("details") { DetailsScreen(navController) }
}
```

Jetpack Compose Concepts - Material Design

- Jetpack Compose has full integration with Material Design 3, allowing you to create modern and consistent interfaces.
- Components such as Button, Snackbar, Card, FloatingActionButton, among others, are directly available with customization support via MaterialTheme.