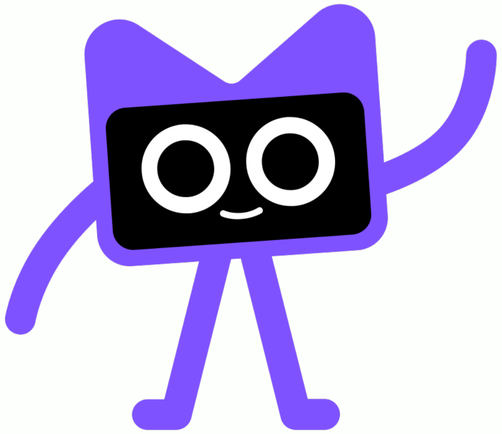
UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Listas e Material Design

QXD0276 - Desenvolvimento de Software para Dispositivos Móveis



Prof. Bruno Góis Mateus (brunomateus@ufc.br)

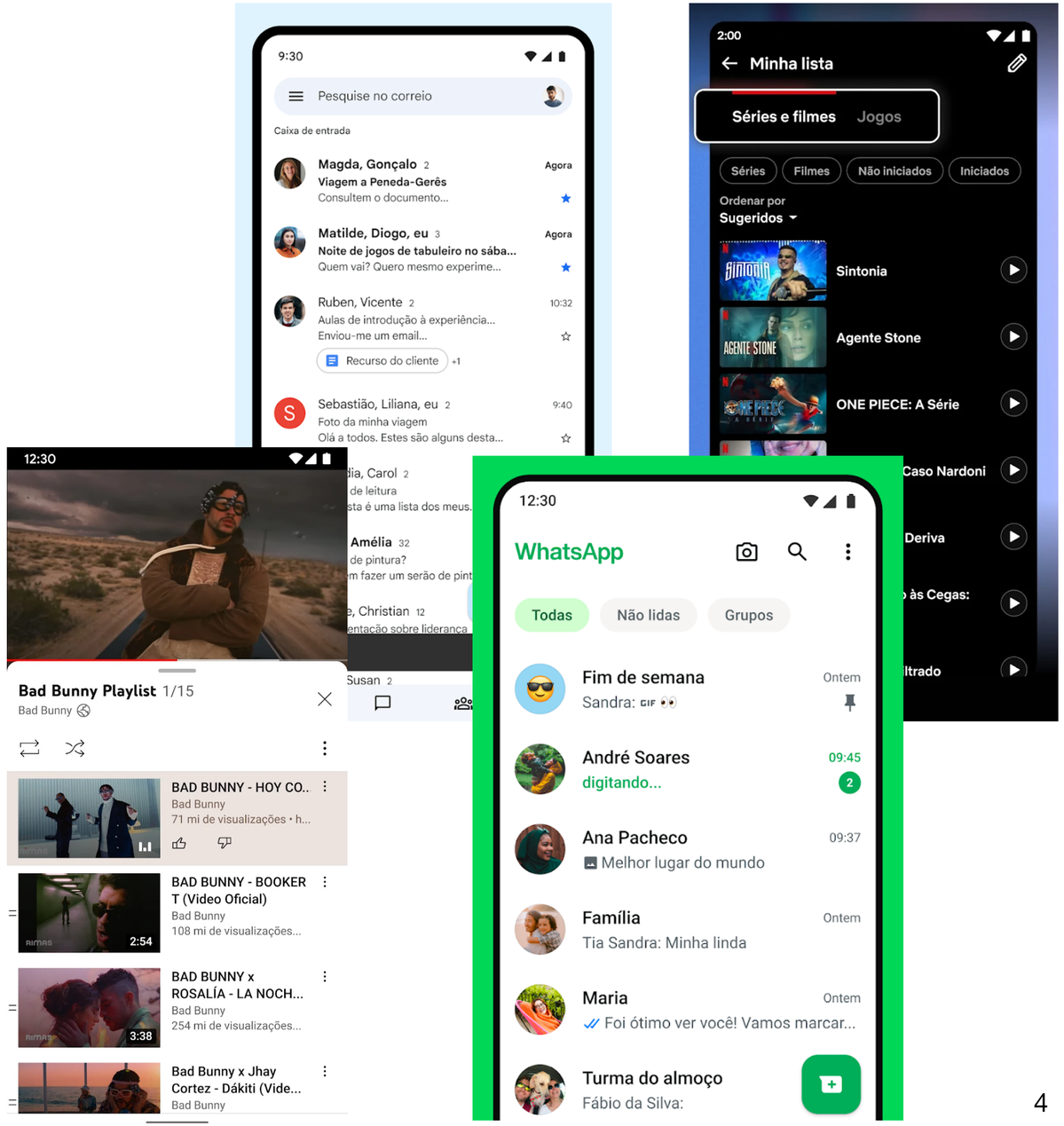
Conteúdo

- Introdução
- Um pouco mais de Kotlin
- Listas e Grades
- Introdução ao Material Design

Introdução

Introdução

- Listas e grades são layout que permitem a visualização de múltiplos itens
- Estão presentes em boa parte das aplicações que utilizamos no dia a dia
- Veremos com implementar diversos tipos de listas seguindo as boas práticas



Um pouco mais de Kotlin

Um pouco mais de Kotlin

Classes em Kotlin

```
class Pessoa(val nome: String, var idade: Int)
```

- A palavra-chave class define uma nova classe
- O construtor primário é declarado na própria linha da classe
- As propriedades val e var são criadas automaticamente
- 🟢 val: somente leitura (equivalente a final em Java)
  🟠 var: mutável

Um pouco mais de Kotlin

Construtores e inicialização

```
class Pessoa(val nome: String) {
    var idade: Int = 0

    init {
        println("Pessoa criada: $nome")
    }

    constructor(nome: String, idade: Int) : this(nome) {
        this.idade = idade
    }
}
```

É executado logo após a criação do objeto

Não se usa val ou var

Construtores secundários usam this( ) para chamar o primário.

Um pouco mais de Kotlin

Criando e usando objetos

```
val p1 = Pessoa("Maria", 25)
println(p1.nome) // Maria
p1.idade = 26
```

Em Kotlin, não há palavra-chave new

Um pouco mais de Kotlin

Métodos

```
class Calculadora {
    fun somar(a: Int, b: Int): Int {
        return a + b
    }

    fun subtrair(a: Int, b: Int) = a - b
}
```

Métodos são definidos com fun

É possível usar *function expression*

Um pouco mais de Kotlin

Objetos e Singletons

```
object Config {
    const val API_URL = "https://api.exemplo.com"
}
```

Cria um singleton (uma única instância global).

Ideal para guardar configurações, utilitários, ou dependências comuns.

Um pouco mais de Kotlin

Propriedades e acesso

```
class Conta {
    var saldo: Double = 0.0
        get() = field
        set(value) {
            if (value >= 0) field = value
        }
}
```

representa o backing field (armazenamento real da variável)

Podemos customizar get e set

Um pouco mais de Kotlin

Propriedades somente leitura

```
class Retangulo(val largura: Int, val altura: Int) {
    val area: Int
        get() = largura * altura
}
```

A propriedade não é armazenada, apenas calculada quando acessada.

Um pouco mais de Kotlin

Propriedades de classe

```
class Util {
    companion object {
        fun saudacao() = "Olá!"
    }
}
```

Funciona como o static do Java.

Um pouco mais de Kotlin

Modificadores de acesso

- public, private, protected, internal
  - Se não é definido, o modificador utilizado é o public
  - internal é muito útil em módulos Android (visível apenas dentro do módulo).

Um pouco mais de Kotlin

Modificadores de acesso

- Funções, propriedades, classes, objetos e interfaces podem ser declaro no nível mais alto dentro do um pacote
- Ao usar o modificador private, o elemento ficará visível apenas dentro do arquivo que a contém
- Ao usar o modificador internal, o elemento ficará visível em todo módulo
- O modificador `protected` não está disponível para declarações de nível mais alto

Um pouco mais de Kotlin

Modificadores de acesso

```
// file name: example.kt
package foo

private fun foo() { ... } // visible dentro do example.kt

public var bar: Int = 5 // propriedade visível em todo canto
    private set         // setter visível apenas no example.kt

internal val baz = 6    // visível dentro do mesmo módulo
```

Um pouco mais de Kotlin

Data classes

- Bastante utilizadas para representa dados
- O compilador Kotlin gera uma série de funções adicionais
  - equals, hashCode, toString, copy, componentN

```
data class Pessoa(val nome: String, val idade: Int)

val (nome, idade) = Pessoa("Joao", 20)
```

Um pouco mais de Kotlin

Herança

- Em Kotlin toda classe e seus membros são final por padrão
- Deve usar open para permitir herança

```
open class Comida(val preco: Double)

class Hamburger(preco: Double) : Comida(preco)
```

Um pouco mais de Kotlin

Herança

- Em Kotlin toda classe e seus membro são final por padrão
- Deve usar open para permitir herança

```
open class Comida(val preco: Double)

class Hamburger(preco: Double) : Comida(preco)
```

Um pouco mais de Kotlin

Sobrescrita de propriedade

- É possível sobrescrever uma propriedade da superclasse utilizando a palavra reservada override

```
open class Comida(open val preco: Double)

class Hamburger(preco: Double) : Comida(preco)

abstract class Cobertura(preco: Double, private val comida: Comida): Comida(preco) {

    override val preco: Double = preco
        get() = field + comida.preco
}
```

Um pouco mais de Kotlin

Sobrescrita de propriedade

```
class Jalapeno(preco: Double, comida: Comida):Cobertura(preco, comida)
class Mushroom(preco: Double, comida: Comida):Cobertura(preco, comida)

val food: Food = Mushroom(
    .10,
    Jalapeno(
        .15,
        Hamburger(2.99)
    )
)
```

Um pouco mais de Kotlin

Sobrescrita de método

```
open class Comida(open val preco: Double) {

    open fun calcular(taxa: Double): Double {
        return preco * taxa
    }
}

class Hamburger(override val preco: Double, val desconto: Double = 0.0) : Comida(preco) {

    override fun calcular(taxa: Double): Double {
        return super.calcular(taxa) - this.desconto
    }
}
```

Um pouco mais de Kotlin

Sealed classes

- São classes fechadas para herança fora do mesmo arquivo
  - Usadas para representar estados limitados
  - Muito utilizadas com when

Um pouco mais de Kotlin

Sealed classes

```
sealed class Resultado
data class Sucesso(val dados: String) : Resultado()
data class Erro(val mensagem: String) : Resultado()
object Carregando : Resultado()

fun processar(resultado: Resultado) = when (resultado) {
    is Sucesso -> println("Sucesso: ${resultado.dados}")
    is Erro -> println("Erro: ${resultado.mensagem}")
    Carregando -> println("Carregando...")
}
```

Um pouco mais de Kotlin

Enum

```
enum class DiaSemana { SEG, TER, QUA, QUI, SEX }

enum class Status(val cor: Int) {
    SUCCESS(Color.GREEN), ERROR(Color.RED)
}

enum class Status(val descricao: String) {
    ATIVO("Em operação"),
    INATIVO("Desativado"),
    ERRO("Com falha");

    fun isAtivo() = this == ATIVO
}
```

Um pouco mais de Kotlin

Interfaces

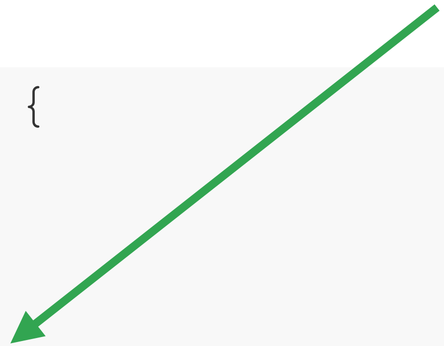
```
interface ClickListener {
    fun onClick()
}
```

Os métodos podem ter corpo (implementação)

Um pouco mais de Kotlin

Interfaces funcionais (Single Abstract Method - SAM)

- São interfaces com um único método abstrato
- Podem ser instanciadas usando uma função lambda



```kotlin
fun interface ClickListener {
    fun onClick()
}

button.setOnClickListener {
    Toast.makeText(context, "Clicou!", Toast.LENGTH_SHORT).show()
}
```

Um pouco mais de Kotlin

Delegated properties

```
import kotlin.reflect.KProperty

class Example {
    var p: String by Delegate()
}

class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return "$thisRef, thank you for delegating '${property.name}' to me!"
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String)
{
        println("$value has been assigned to '${property.name}' in $thisRef.")
    }
}
```

Um pouco mais de Kotlin

Delegated properties

- lazy
  - é uma função que recebe uma expressão lambda e retorna uma instância de Lazy<T>
  - A primeira chamada a get() executa a expressão lambda passada para e armazena o resultado
  - Chamadas subsequentes a get() simplesmente retornam o resultado armazenado.

Um pouco mais de Kotlin

Delegated properties

```
val user by lazy { loadUserFromDB() }

val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}

fun main() {
    println(lazyValue)
    println(lazyValue)
}
```

Um pouco mais de Kotlin

Delegated properties

- observable
  - recebe dois argumentos: o valor inicial e um handler para as modificações
  - O handler é chamado sempre que você atribui um valor à propriedade
    - Possui três parâmetros: a propriedade à qual o valor está sendo atribuído, o valor antigo e o novo valor.

Um pouco mais de Kotlin

Delegated properties

```
import kotlin.properties.Delegates

class User {
    var name: String by Delegates.observable("<no name>") {
        prop, old, new ->
        println("$old -> $new")
    }
}

fun main() {
    val user = User()
    user.name = "first"
    user.name = "second"
}
```

Um pouco mais de Kotlin

Type aliases

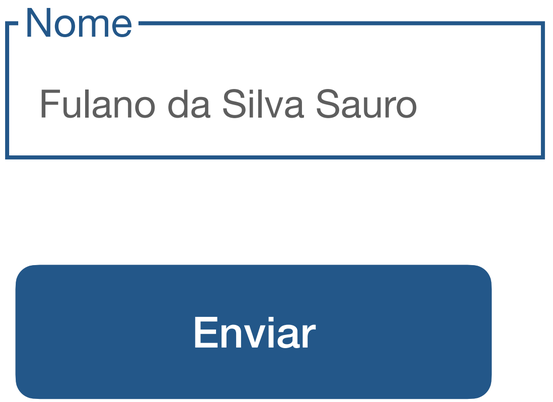
```
typealias ClickListener = (View) -> Unit
val onClick: (View) -> Unit
val onClick: ClickListener

typealias UserMap = Map<Int, Pair<String, String>>

val users: UserMap = mapOf(
    1 to ("Maria" to "maria@email.com"),
    2 to ("João" to "joao@email.com")
)
```

Introdução

- Em aplicativos reais, a interface muda conforme dados variam:
  - Texto de um campo
  - Itens de uma lista
  - Alternância de botões
- A UI reflete o estado atual dos dados
  - Quando o estado muda, a UI deve mudar junto



Introdução

O que é o “estado”?

- Estado é qualquer informação que pode mudar ao longo do tempo
- Exemplo:
  - Valor digitado em um TextField
  - Item selecionado em uma lista
  - Status de login de um usuário

Introdução

- No Compose, o estado controla como a tela é renderizada
- Estado = dados observáveis
  - Como o Compose é declarativo, a UI é uma função do estado:

    ⚙️ "UI = f(estado)"

  - Se o estado muda, a UI é recomposta *(recomposition)*

Problema: como garantir que o estado se mantenha correto entre recomposições e mudanças no app?

Listas e Grades

Listas e Grades

Introdução

- O que são listas dinâmicas?
  - Listas cujo conteúdo pode mudar durante a execução
  - Ex: adicionar, remover, atualizar itens
- Comuns em apps reais: listas de tarefas, contatos, produtos, mensagens, etc.

Listas e Grades

⚠️ O Problema da Abordagem Simples

- Se você usa um Column ou Row com Modifier.verticalScroll() , o processamento é eager (ansioso):

```
@Composable
fun MessageList(messages: List<Message>) {
    // PROBLEMA: TODOS os itens são processados e desenhados,
    // mesmo fora da tela.
    Column(Modifier.verticalScroll(rememberScrollState())) {
        messages.forEach { message ->
            MessageRow(message)
        }
    }
}
```

Em listas enormes, isso causa:

- Alto consumo de memória
- Queda na taxa de quadros (Jank/Lag)

Listas e Grades

⚡ A Solução "Lazy": Composição Sob Demanda

- Lazy (Preguiçoso)
  - Somente os composables que estão VISÍVEIS no viewport (área da tela) e são desenhados
  - Gerencia a reciclagem implicitamente
    - Elementos de UI de itens que saem da tela são descartadas e reutilizadas para itens que estão se tornando visíveis

Listas e Grades

Vantagens

- Performance Superior:
  - Reduz drasticamente a carga de trabalho inicial
- Uso de Memória:
  - Otimiza o consumo, pois só mantém na memória o que é necessário
- Simplicidade:
  - Elimina a necessidade de Adapters e View Holders manuais (maneira antiga usando XML)

Listas e Grades

🧩 A Sintaxe Lazy - O Bloco DSL

- Os componentes Lazy usam uma DSL (Domain-Specific Language), o bloco LazyListScope
- O DSL permite que você descreva o conteúdo da lista, em vez de emitir Composables diretamente

Função	Uso	Exemplo
item { ... }	Adiciona um único item (útil para cabeçalhos ou rodapés fixos).	`item { Text("Header") }`
items(list) { ... }	Adiciona itens a partir de uma coleção de dados.	`items(messages) { message -> ... }`

Listas e Grades

Exemplo

```
LazyColumn {
    // DSL de LazyListScope: Descrição de como a lista deve ser
    item {
        Header() // Um Composable fixo no topo
    }
    items(minhaListaDeDados) { item ->
        ItemRow(item) // Um Composable por item da lista
    }
}
```

Listas e Grades

Composables existentes

Nome	Função
LazyColumn	Rola verticalmente.
LazyRow	Rola horizontalmente.
LazyVerticalGrid	Grade de múltiplas colunas.
LazyHorizontalGrid	Grade de múltiplas linhas.
LazyVerticalStaggeredGrid	Grade de múltiplas colunas c/ items de tamanhos diferentes
LazyHorizontalStaggeredGrid	Grade de múltiplas linhas c/ items de tamanhos diferentes

Listas e Grades

Grids

- Usados para exibir itens em um padrão de grade
  - LazyVerticalGrid: Múltiplas colunas
  - LazyHorizontalGrid: Múltiplas linhas

Listas e Grades

📐 Configurando Grids: Fixed vs. Adaptive

- O parâmetro columns em LazyVerticalGrid e rows em LazyHorizontalGrid
- GridCells.Fixed
  - Define um número fixo de colunas
- GridCells.Adaptive (Recomendado para Responsividade)
  - Define uma largura mínima (minSize).
  - O sistema calcula quantas colunas cabem na tela

Listas e Grades

Fixed vs Adaptative

```
// Sempre 3 colunas, independente do tamanho da tela.
LazyVerticalGrid(columns = GridCells.Fixed(3)) { /* ... */ }

// Colunas com, no mínimo, 128.dp de largura.
LazyVerticalGrid(
    columns = GridCells.Adaptive(minSize = 128.dp)
) { /* ... */ }
```

Listas e Grades

🖼️ Listas Staggered (Grades Escalonadas)

- Usadas quando os itens possuem tamanhos diferentes, mas você ainda quer que eles se encaixem em um padrão de grade
  - Ex: diferentes alturas de imagem em uma galeria
  - LazyVerticalStaggeredGrid e LazyHorizontalStaggeredGrid

```
LazyVerticalStaggeredGrid(
    columns = StaggeredGridCells.Fixed(3),
    verticalItemSpacing = 4.dp,
    horizontalArrangement = Arrangement.spacedBy(4.dp)
) {
    items(randomSizedPhotos) { photo ->
        PhotoItem(photo)
    }
}
```

Listas e Grades

📏 Espaçamento e Distribuição

- Espaçamento entre Itens (Arrangement.spacedBy)
  - Adiciona espaço APENAS ENTRE os itens
  - Usado para o eixo principal da lista

```
// Exemplo em LazyColumn: 8.dp de espaço vertical entre cada item
LazyColumn(
    verticalArrangement = Arrangement.spacedBy(8.dp)
) {
    /* ... */
}
```

Listas e Grades

📏 Alinhamento e Distribuição em Grids

- Grids aceitam verticalArrangement e horizontalArrangement

```
// Exemplo em LazyVerticalGrid
LazyVerticalGrid(
    columns = GridCells.Fixed(2),
    verticalArrangement = Arrangement.spacedBy(16.dp),
    horizontalArrangement = Arrangement.spacedBy(16.dp)
) { /* ... */ }
```

Listas e Grades

↕️ Padding de Conteúdo - contentPadding

- Adiciona preenchimento ao redor do conteúdo da lista, dentro da área rolável

```
// 16.dp nas laterais (esquerda/direita) e 8.dp no topo/base
LazyColumn(
    contentPadding = PaddingValues(horizontal = 16.dp, vertical = 8.dp)
) {
    /* ... */
}
```

Listas e Grades

🤺 Desafios

- Manter o estado interno de cada item (ex: checkbox marcado)
- Evitar recomposição desnecessária
- Garantir bom desempenho em listas longas

Listas e Grades

🔁 Problema das mutações que recriam listas

- Quando alteramos uma lista imutável com operações como +, -, map(), o Compose entende que é uma nova instância e recompõe toda a lista

```
var usuarios by remember { mutableStateOf(listOf("Ana", "Bruno", "Carlos")) }

Button(onClick = {
    // ❌ Cria uma nova lista (nova referência)
    usuarios = usuarios + "Novo Usuário"
}) {
    Text("Adicionar")
}
```

- A lista inteira é redesenhada
- Estados internos (ex: campos de texto ou seleções) são perdidos
- Impacto negativo na performance

Listas e Grades

✅ Solução - Usar mutableStateListOf()

- Resulta em uma lista observável

```
@Composable
fun ListaUsuarios() {
    val usuarios = remember { mutableStateListOf("Ana", "Bruno", "Carlos") }
    Column {
        Button(onClick = { usuarios.add("Novo Usuário") }) {
            Text("Adicionar")
        }
        LazyColumn {
            items(usuarios) { usuario ->
                Text(usuario)
            }
        }
    }
}
```

- Evita recriação de lista.
- Compose detecta apenas as mudanças pontuais.
- Estados de cada item são preservados.

Listas e Grades

🧘 O Problema da Posição

- Por padrão, o Compose usa a posição do item como sua chave
  - Se a lista é reordenada, o Compose perde o rastro do item
- Ao fornecer uma chave única e estável, você garante:
  - Preservação de Estado: O estado de um item (remembered) é preservado, mesmo que ele mude de posição na lista.
  - Animações Corretas: Animações de reordenação funcionam perfeitamente

🧩 Sem key: Compose pode reconstruir widgets desnecessariamente.

🧱 Com key: apenas o item alterado é recomposido.

Listas e Grades

🔁 Evitando recriações indiretas

- Mesmo com listas mutáveis, operações dentro de @Composable podem recriar listas:

```
val filtrados = usuarios.filter { it.startsWith("A") }
// ❌ Esta linha roda a cada recomposição
```

Listas e Grades

✅ Solução - Use remember para evitar recalcular desnecessariamente

```
val filtrados = remember(usuarios) {
    usuarios.filter { it.startsWith("A") }
}
```

- Só será recalculado quando usuarios mudar

Listas e Grades

👀 Reagindo à Rolagem

- LazyListState
  - Permite ler e controlar o estado da lista
  - Deve ser criado com rememberLazyListState() e passado para o LazyColumn

```
// 1. Lembrar o estado
val listState = rememberLazyListState()

// 2. Passar para a lista
LazyColumn(state = listState) { /* ... */ }
```

Listas e Grades

🌟 Otimizando a Leitura de Estado (derivedStateOf)

- Ao ler o estado para atualizar a UI (ex: mostrar um botão), use derivedStateOf para evitar recomposições desnecessárias
  - É reativo e recalcula o valor apenas quando dependências mudam

```
// showButton só é reavaliado e recompõe se o índice mudar de 0 para > 0
val showButton by remember {
    derivedStateOf { listState.firstVisibleItemIndex > 0 }
}

AnimatedVisibility(visible = showButton) {
    ScrollToTopButton()
}
```

Explicação mais detalhada

Listas e Grades

Controle Programático da Rolagem

- As funções de controle de rolagem são suspending e devem ser chamadas dentro de uma Coroutine
  - Obter o Scope: Use rememberCoroutineScope()
  - Chamar a Função: Use launch { ... } para iniciar a rolagem

Função	Comportamento
animateScrollToItem(index)	`Rola suavemente (com animação) para o item`
scrollToItem(index)	`Rola instantaneamente (sem animação) para o item`

Listas e Grades

Controle Programático da Rolagem

```
val coroutineScope = rememberCoroutineScope()

ScrollToTopButton(
    onClick = {
        coroutineScope.launch {
            // Anima a rolagem para o topo (índice 0)
            listState.animateScrollToItem(index = 0)
        }
    }
)
```

Listas e Grades

🎬 Bonus: Animações de Itens

- Para aplicar transições suaves quando um item é adicionado, removido ou movido na lista chame o Modifier.animateItem()

```
LazyColumn {
    // ⚠️ O parâmetro 'key' é OBRIGATÓRIO para animações!
    items(books, key = { it.id }) { book ->
        Row(Modifier.animateItem()) {
            Text(book.title)
        }
    }
}
```

Material Design

Material Design

O que é?

- Sistema de design criado pelo Google (2014)
- Unifica a experiência visual entre Android, Web e iOS
- Baseado em:
  - Design gráfico moderno
  - Movimento com propósito
  - Profundidade e hierarquia

Material Design

🎨 Material é o meio

- O MD é inspirado no mundo físico
- A interface se comporta como um material físico
- Os elementos (o "material") têm superfície, elevação e respondem ao toque de forma realista
- As animações refletem movimento natural e continuidade

Material Design

Princípios Fundamentais

- Material é o meio – tudo tem profundidade
- Movimento com propósito – animações guiam o olhar
- Tipografia clara – comunica hierarquia e significado
- Cor intencional – reforça ações e marca
- Componentes consistentes – previsibilidade = usabilidade

Material Design

A Evolução do Material Design

Versão	Ano	Destaques
Material Design 1	2014	Introdução de sombras, cartões e animações
Material Design 2	2018	Material Theming (cores e tipografia personalizáveis)
Material You (Material 3)	2021	Cores dinâmicas baseadas no papel de parede do usuário

Material Design

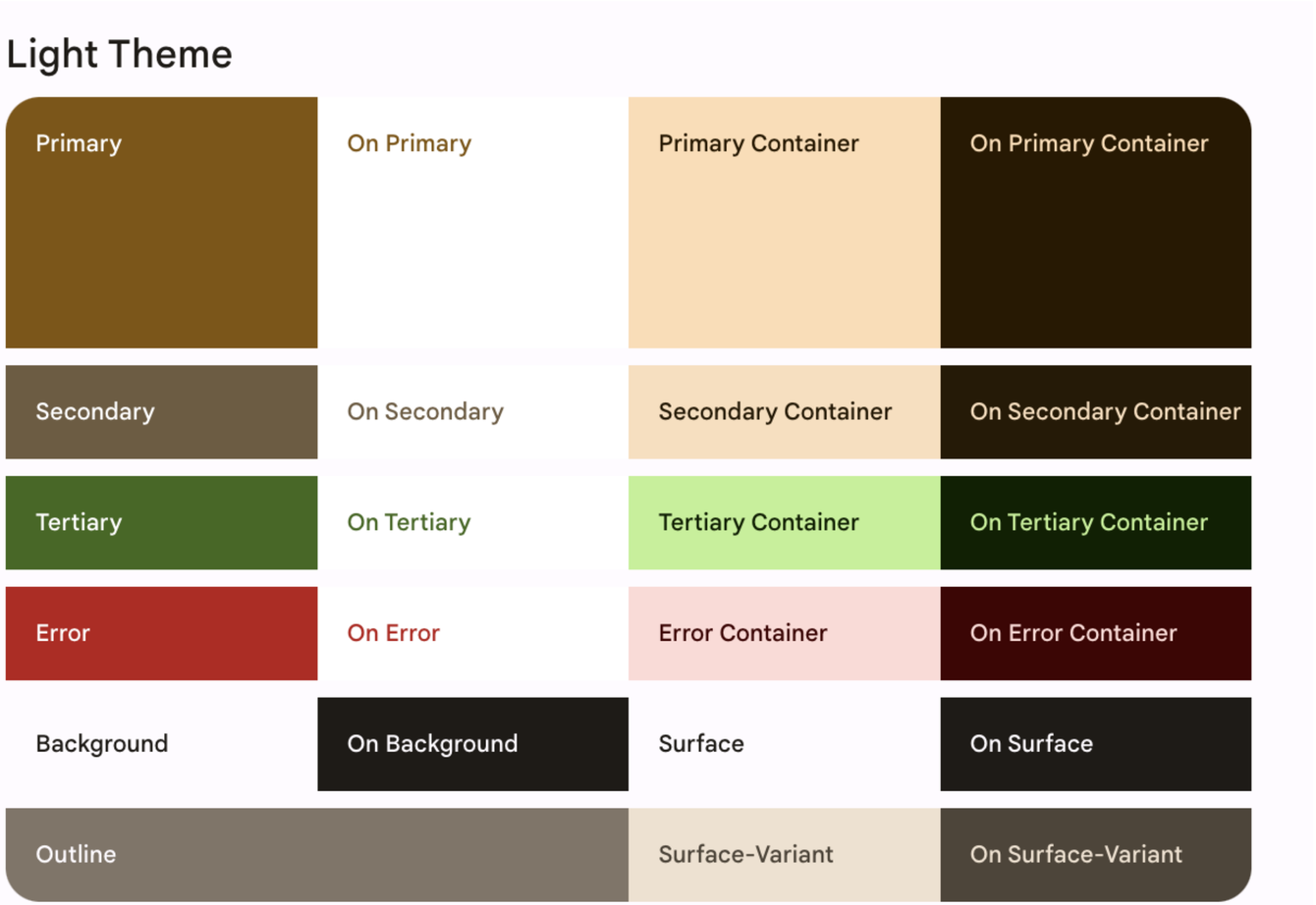
No Android

- Presente desde o Android Lollipop (API 21)
- Hoje, integrado ao Jetpack Compose Material 3
- Suporte nativo a:
  - Temas claros/escuros
  - Cores dinâmicas (Material You)
  - Acessibilidade

Material Design

🎨 Sistema de cores

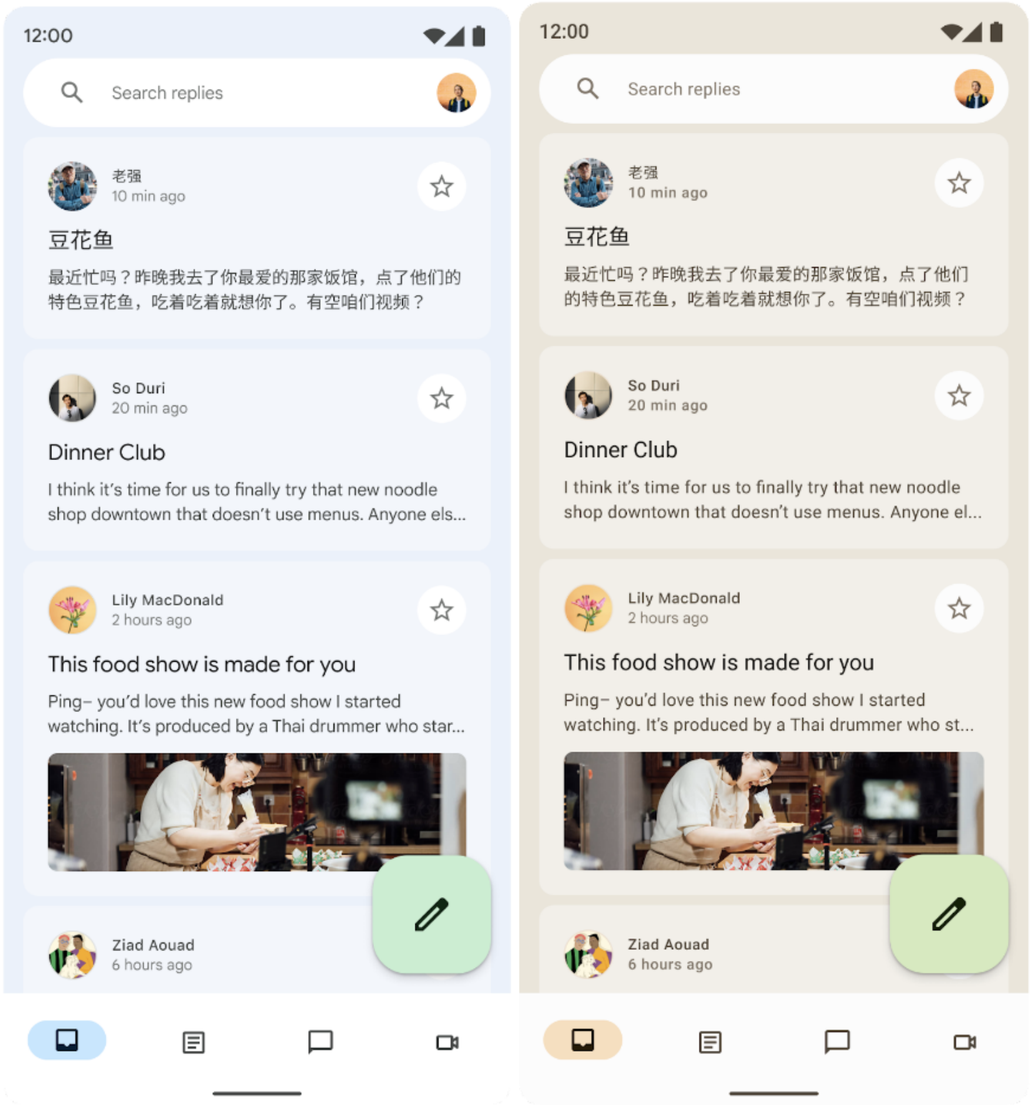
- A base de um esquema de cores é o conjunto de cinco cores principais
- Cada uma dessas cores se relaciona a uma paleta tonal de 13 tons, que são usados pelos componentes do Material 3



Material Design

🎨 Dynamic color

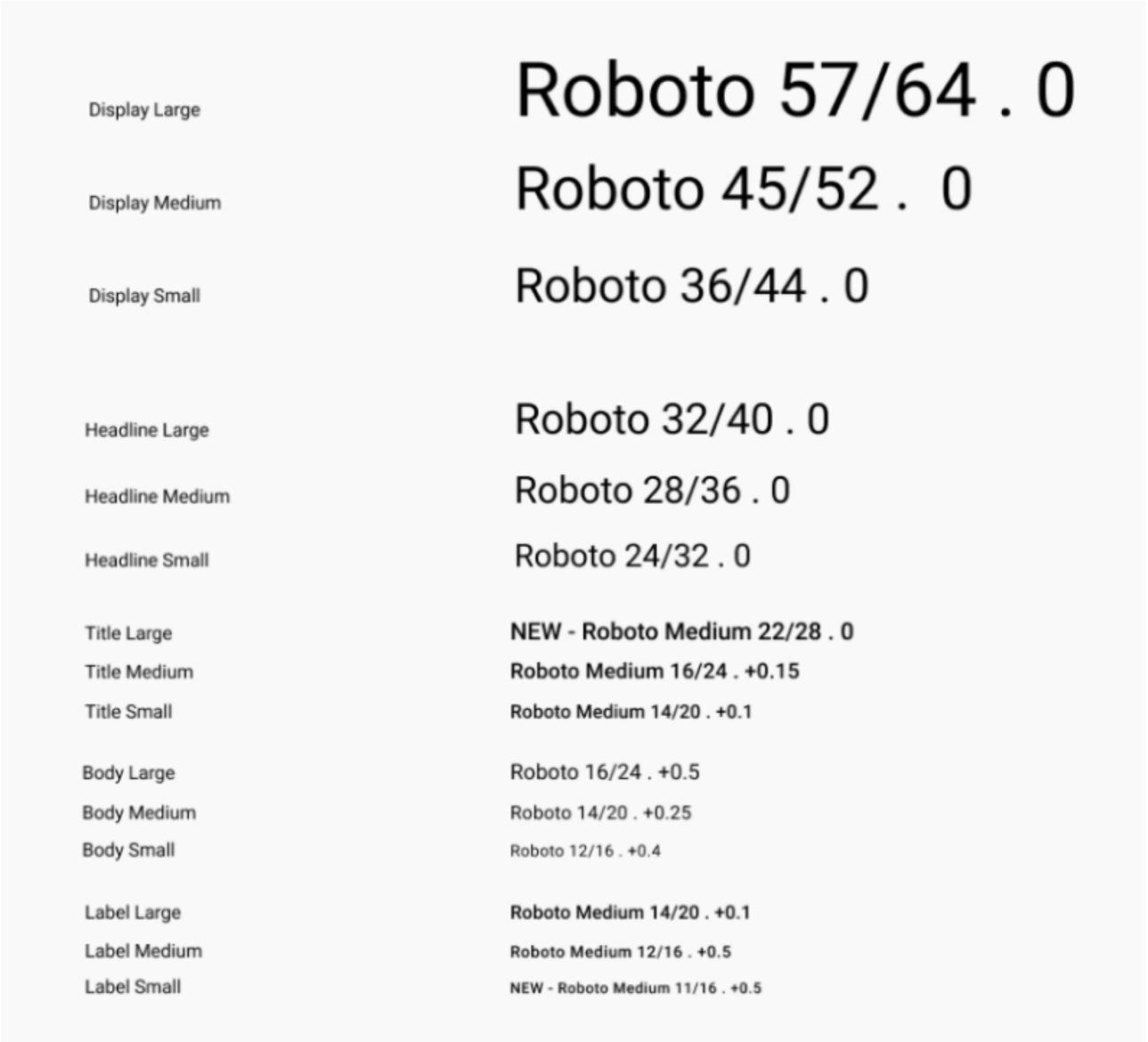
- O Material You usa um algoritmo extrai cores do papel de parede do usuário para serem aplicadas aos seus aplicativos e à interface do sistema
- Essa paleta de cores é usada como ponto de partida para gerar esquemas de cores claras e escuras



Material Design

Tipografia

- 🅰️ Hierarquia tipográfica
  - Display → títulos grandes
  - Headline → seções
  - Title → subtítulos
  - Body → texto corrido
  - Label → elementos de interface
- Fonte padrão: Roboto / Google Sans



Material Design

Tipografia

M3	Default Font Size/Line Height
displayLarge	Roboto 57/64
displayMedium	Roboto 45/52
displaySmall	Roboto 36/44
headlineLarge	Roboto 32/40
headlineMedium	Roboto 28/36
headlineSmall	Roboto 24/32
titleLarge	New Roboto Medium 22/28
titleMedium	Roboto Medium 16/24
titleSmall	Roboto Medium 14/20
bodyLarge	Roboto 16/24
bodyMedium	Roboto 14/20
bodySmall	Roboto 12/16
labelLarge	Roboto Medium 14/20
labelMedium	Roboto Medium 12/16
labelSmall	New Roboto Medium, 11/16

Material Design

Componentes do Material

- Principais componentes de UI:
- Botões: Filled, Outlined, Text, FAB
- Cards e Containers
- AppBar e Navigation Bar
- TextField, Slider, Switch
- Para conhecer todos os componentes visite: https://m3.material.io/

Material Design

Layout e Espaçamento

📏 Grid de 8dp

- Margens, paddings e espaçamentos múltiplos de 8dp

📱 Responsividade

- Tamanhos adaptáveis:
  - Compact (celulares)
  - Medium (tablets)
  - Expanded (desktops ou telas grandes)

Material Design

Acessibilidade

♿ Boas práticas

- Contraste de cores adequado
- Tamanho mínimo de toque: 48dp
- Suporte a leitores de tela (TalkBack)
- Navegação por teclado

Material Design

No Jetpack Compose

- Baseia-se em quatro pilares fundamentais, todos gerenciados pelo MaterialTheme no Compose:
- Cor (Color): Paletas tonais acessíveis e cores dinâmicas
- Tipografia (Typography): Escala de texto consistente
- Formas (Shape): Cantos arredondados e geometria dos componentes
- Layout (Layout): Diretrizes sobre espaçamento e posicionamento

Material Design

O Composable Raiz: MaterialTheme

- É o Composable raiz que define o sistema de design para todos os componentes dentro dele

```
@Composable
fun MyAppTheme(content: @Composable () -> Unit) {
  MaterialTheme(
   colorScheme = MyColorScheme,  // Define cores (primária, secundária, etc.)
   typography = MyTypography,    // Define fontes (Headline, Body, etc.)
   shapes = MyShapes,            // Define formas (cantos arredondados)
   content = content
  )
}
```

Material Design

No Jetpack Compose

```
// Importar o pacote Material 3:
import androidx.compose.material3.*

//Estrutura típica
@Composable
fun MyApp() {
    MaterialTheme {
        Surface {
            Greeting("Android")
        }
    }
}
```

Material Design

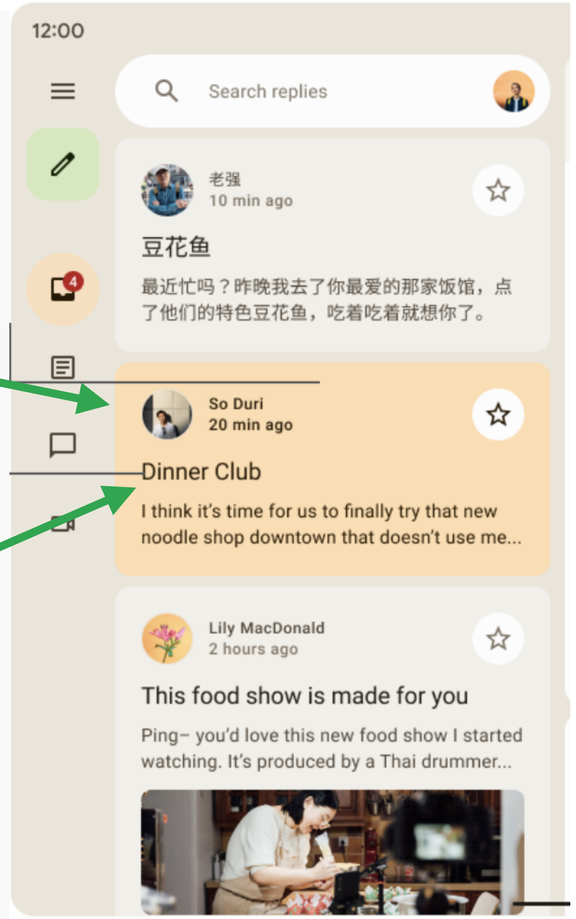
Usando cores

```
Text(
    text = "Hello theming",
    color = MaterialTheme.colorScheme.primary
)
```

Material Design

Usando cores

```
Card(
  colors = CardDefaults.cardColors(
    containerColor =
    if (isSelected) colorScheme.primaryContainer
    else
        colorScheme.surfaceVariant
 )
) {
 Text(
    text = "Dinner club",
    style = MaterialTheme.typography.bodyLarge,
    color =
    if (isSelected)colorScheme.onPrimaryContainer
    else colorScheme.onSurface,
    )
}
```



Material Design

Usando estilos de texto

```
Text(
    text = "Hello M3 theming",
    style = MaterialTheme.typography.titleLarge
)
Text(
    text = "you are learning typography",
    style = MaterialTheme.typography.bodyMedium
)
```

Material Design

Usando shapes

```
Card(shape = MaterialTheme.shapes.medium) { /* card content */ }
FloatingActionButton(
    shape = MaterialTheme.shapes.large,
    onClick = {
    }
) {
    /* fab content */
}
```

Qualquer Composable do Material (ex: Button, Card, FloatingActionButton) herda automaticamente as cores e estilos definidos neste tema

Bibliografia

- Documentação oficial do Kotlin
- Kotlin: Herança
- Lists and Grids - Documentação oficial do Android
- Jetpack Compose — When should I use derivedStateOf?
- Material Design
- Material Design 3 in Compose
- Material Theme Builder

Por hoje é só