1. Antes de começar

Jetpack Compose é um kit de ferramentas moderno criado para simplificar o desenvolvimento de IUs. Ele combina um modelo de programação reativa com a concisão e a facilidade de uso da linguagem de programação Kotlin. Ele é totalmente declarativo, ou seja, você descreve a IU chamando uma série de funções que transformam dados em uma hierarquia de IUs. Quando os dados subjacentes mudam, o framework reexecuta automaticamente essas funções, atualizando a hierarquia de IUs.

Um app Compose é formado por funções combináveis, que são funções normais marcadas com `@Composable` e que podem chamar outras funções combináveis. Basta usar uma função para criar um novo componente de IU. A anotação instrui o Compose a adicionar suporte especial à função para atualizar e manter a IU ao longo do tempo. O Compose permite estruturar o código em pequenos blocos. As funções combináveis costumam ser chamadas de "combináveis" para abreviar.

Ao criar pequenas funções combináveis que podem ser reutilizadas, é fácil criar uma biblioteca de elementos de IU usados no app. Cada um é responsável por uma parte da tela e pode ser editado de forma independente.

Observação: neste tutorial, os termos "componentes de interface", "funções combináveis" e "combináveis" são usados de forma intercambiável para se referir ao mesmo conceito.

Para receber mais suporte durante este tutorial, confira as orientações neste vídeo (em inglês):

Observação: o Material 2 é usado no vídeo, mas este tutorial foi atualizado para uso do Material 3. Portanto, algumas etapas serão diferentes.

Pré-requisitos

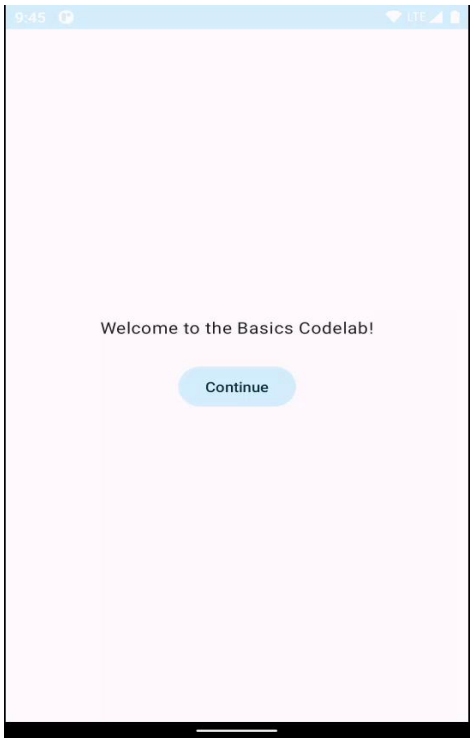
- Experiência com a sintaxe do Kotlin, incluindo lambdas.

O que você vai fazer

Neste tutorial, você vai aprender:

- O que é o Compose
- Como criar IUs com o Compose
- Como gerenciar o estado em funções combináveis
- Como criar uma lista de desempenho
- Como adicionar animações
- Como definir o estilo e o tema de um app

Você vai criar um app com uma tela de integração e uma lista de itens com animações de abertura:

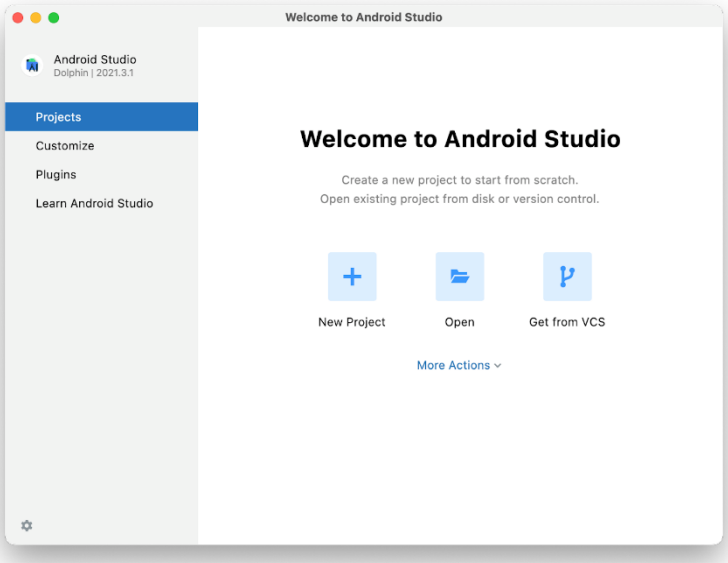


O que é necessário

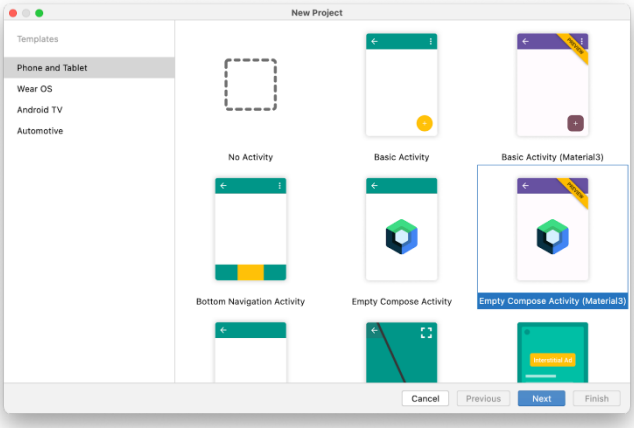
- Versão mais recente do Android Studio

2. Como iniciar um novo projeto do Compose

Abra o Android Studio e selecione **Start a new Android Studio project**, conforme mostrado:



Caso a tela acima não seja mostrada, acesse **File** > **New** > **New Project**.
Ao criar um novo projeto, selecione a opção **Empty Compose Activity (Material3)** nos modelos disponíveis.



Clique em **Next** e configure o projeto normalmente, chamando-o de "**Basics Codelab**". Selecione uma ***minimumSdkVersion*** com API de nível 21 ou mais recente. Esse é o nível mínimo de API com suporte no Compose.
Ao selecionar o modelo **Empty Compose Activity (Material3),** o seguinte código vai ser gerado no seu projeto:

- O projeto já está configurado para usar o Compose.
- O arquivo AndroidManifest.xml foi criado.
- Os arquivos build.gradle e app/build.gradle contêm opções e dependências necessárias para o Compose.

Depois de sincronizar o projeto, abra MainActivity.kt e confira o código.

```
class MainActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
      BasicsCodelabTheme {
        // 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!")
}

@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
   BasicsCodelabTheme {
      Greeting("Android")
   }
}
```

Aviso: o tema do app usado em **setContent** depende da forma como o projeto é nomeado. Este tutorial considera que o projeto é chamado de ***BasicsCodelab*.** Se você copiar e colar o código do tutorial não se esqueça de atualizar o **BasicsCodelabTheme** com o nome do tema, disponível no arquivo **ui/Theme.kt.**
Na próxima seção, você vai aprender sobre o que cada método faz e como é possível melhorá-los para criar layouts flexíveis e reutilizáveis.

3. Começar a usar o Compose

Analise as classes e os métodos diferentes relacionados ao Compose que foram gerados pelo Android Studio.
Funções de composição
Uma **função de composição** é uma função regular anotada com @Composable. Isso permite que a função chame outras funções @Composable dentro dela. É possível ver como a função Greeting é marcada como @Composable. Essa função produzirá uma parte da hierarquia de IUs que exibe a entrada fornecida, String. Text é uma função de composição fornecida pela biblioteca.

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

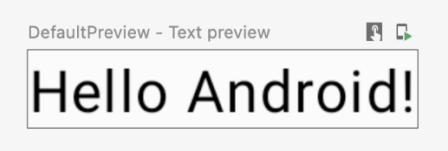
Observação: funções combináveis são funções Kotlin marcadas com a anotação @Composable, como você pode conferir no snippet de código acima.
Compose em um app Android
No Compose, a Activity ainda é o ponto de entrada para um app Android. No nosso projeto, a MainActivity é iniciada quando o usuário abre o app, conforme especificado no arquivo AndroidManifest.xml. Use setContent para definir o layout, mas, em vez de usar um arquivo XML, como você faria no sistema de visualização tradicional, chame funções de composição.

```
class MainActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContent {
         BasicsCodelabTheme {
            // A surface container using the 'background' color from the theme
            Surface(
              modifier = Modifier.fillMaxSize(),
              color = MaterialTheme.colorScheme.background
            ) {
               Greeting("Android")
            }
         }
      }
   }
}
```

BasicsCodelabTheme é uma maneira de definir o estilo de funções combináveis. Saiba mais sobre isso na seção **Como aplicar temas ao app**. Para ver como o texto aparece na tela, execute o app em um emulador ou dispositivo ou use a visualização do Android Studio.

Para usar a visualização do Android Studio, basta marcar qualquer função de composição sem parâmetros ou funções com parâmetros padrão com a anotação `@Preview` e criar seu projeto. Você já pode ver uma função `Preview Composable` no arquivo `MainActivity.kt`. É possível ter várias visualizações no mesmo arquivo e atribuir nomes a elas.

```
@Preview(showBackground = true, name = "Text preview")
@Composable
fun DefaultPreview() {
   BasicsCodelabTheme {
      Greeting(name = "Android")
   }
}
```



Observação: ao importar classes relacionadas ao Jetpack Compose nesse projeto, use o seguinte:

- androidx.compose.* para classes de compilador e ambiente de execução
- androidx.compose.ui.* para kit de ferramentas e bibliotecas de IU

A visualização talvez não apareça se a opção **Code** estiver selecionada. Clique em **Split** para abrir a visualização.

4. Como ajustar a IU

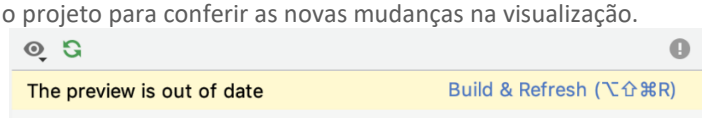
Vamos começar definindo uma cor de segundo plano diferente para `Greeting`. Para isso, envolva o `Text` de composição com uma `Surface`. Como a `Surface` usa uma cor, use `MaterialTheme.colorScheme.primary`.

```
@Composable
private fun Greeting(name: String) {
   Surface(color = MaterialTheme.colorScheme.primary) {
      Text (text = "Hello $name!")
   }
}
```

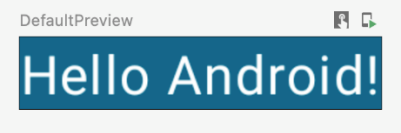
Aviso: Seu código não está sendo compilado? Talvez você tenha usado o modelo de projeto errado no início do tutorial. Use o modelo "Empty Compose Activity (Material3)".

Os componentes aninhados em `Surface` serão renderizados sobre a cor do segundo plano.

Ao adicionar esse código ao projeto, você vai notar um botão **Build & Refresh** no canto direito de cima do Android Studio. Toque nele ou crie o projeto para conferir as novas mudanças na visualização.



Veja as novas mudanças na visualização:



Talvez você não tenha observado um detalhe importante: **o texto agora está em branco**. Quando definimos isso?

Não foi você! Os componentes do Material Design, como `androidx.compose.material3.Surface`, são criados para melhorar sua experiência ao cuidar dos recursos comuns que você provavelmente quer usar no app, como escolher uma cor adequada para o texto. Dizemos que o Material Design *tem opinião* porque fornece bons padrões comuns à maioria dos apps. Os componentes do Material Design no Compose são criados com base em outros componentes básicos (em `androidx.compose.foundation`), que também podem ser acessados nos componentes do app caso você precise de mais flexibilidade.

Nesse caso, a `Surface` entende que, quando a cor do segundo plano é definida como `primary`, qualquer texto nela precisa usar a cor `onPrimary`, que também é definida no tema. Saiba mais sobre isso na seção **Como aplicar temas no app**.

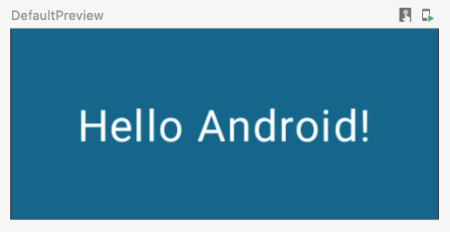
.

Modificadores

A maioria dos elementos de IU do Compose, como Surface e Text, aceita um parâmetro modifier opcional. Os modificadores informam para um elemento da IU como serão dispostos, exibidos ou se comportarão no layout pai.
Por exemplo, o modificador padding aplicará um pouco de espaço ao redor do elemento que ele decora. Você pode criar um modificador de padding com Modifier.padding().
Agora, adicione padding ao Text na tela:

```
import androidx.compose.foundation.layout.padding
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
...

@Composable
private fun Greeting(name: String) {
    Surface(color = MaterialTheme.colorScheme.primary) {
        Text(text = "Hello $name!", modifier = Modifier.padding(24.dp))
    }
}
```

Clique em **Build & Refresh** para ver as novas mudanças.



Existem dezenas de modificadores que podem ser usados para alinhar, animar, dispor, tornar clicáveis ou roláveis, transformar etc. Para ver uma lista abrangente, consulte a **Lista de modificadores do Compose**. Você usará alguns deles nas próximas etapas.

5. Como reutilizar composições

Quanto mais componentes adicionar à IU, mais níveis de aninhamento você vai criar. Isso poderá afetar a legibilidade se uma função se tornar muito grande. Ao criar pequenos componentes reutilizáveis, é fácil criar uma biblioteca de elementos de IU usados no app. Cada um é responsável por uma parte da tela e pode ser editado de forma independente.
Como prática recomendada, a função precisa incluir um parâmetro de modificador atribuído a um modificador vazio por padrão. Encaminhe esse modificador para o primeiro elemento de composição chamado dentro da função. Assim, o local da chamada pode adaptar instruções e comportamentos de layout fora da função de composição.
Crie um elemento combinável chamado MyApp que inclua a saudação.

```
@Composable
private fun MyApp(modifier: Modifier = Modifier) {
    Surface(
        modifier = modifier,
        color = MaterialTheme.colorScheme.background
    ) {
        Greeting("Android")
    }
}
```

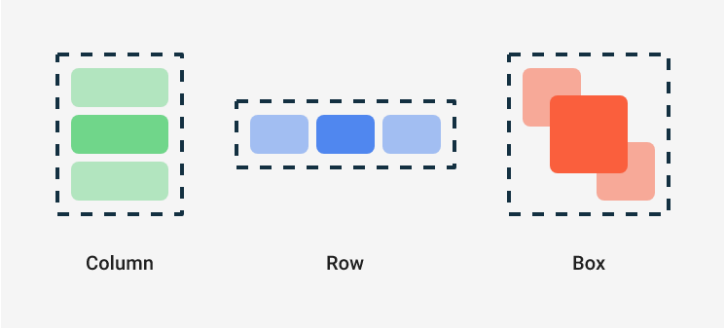
Isso permite limpar o callback onCreate e a visualização, porque é possível reutilizar a composição MyApp, evitando a duplicação de código. O arquivo MainActivity.kt ficará assim:

```
package com.example.basicscodelab

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
```

```
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import com.example.basicscodelab.ui.theme.BasicsCodelabTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            BasicsCodelabTheme {
                MyApp(modifier = Modifier.fillMaxSize())
            }
        }
    }
}

@Composable
private fun MyApp(modifier: Modifier = Modifier) {
    Surface(
        modifier = modifier,
        color = MaterialTheme.colorScheme.background
    ) {
        Greeting("Android")
    }
}

@Composable
private fun Greeting(name: String) {
    Surface(color = MaterialTheme.colorScheme.primary
    ) {
        Text(text = "Hello $name!", modifier = Modifier.padding(24.dp))
    }
}

@Preview(showBackground = true)
@Composable
private fun DefaultPreview() {
    BasicsCodelabTheme {
        MyApp()
    }
}
```

6. Como criar colunas e linhas

Os três elementos básicos de layout padrão do Compose são Column, Row e Box.

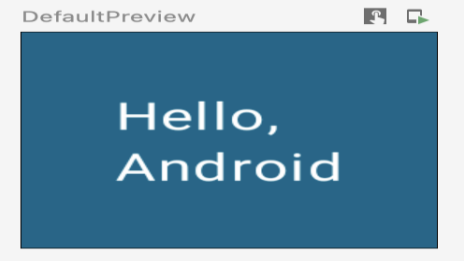


Essas são funções de composição, ou seja, você pode colocar itens nelas. Por exemplo, cada filho dentro de uma Column vai ser colocado na vertical.

```
// Don't copy over
Column {
```

```
    Text("First row")
    Text("Second row")
}
```

Agora, tente mudar Greeting para que ela mostre uma coluna com dois elementos de texto, como no exemplo a seguir:



Talvez seja necessário mover o padding.
Compare seu resultado com esta solução:

```
import androidx.compose.foundation.layout.Column
...

@Composable
private fun Greeting(name: String) {
    Surface(color = MaterialTheme.colorScheme.primary) {
        Column(modifier = Modifier.padding(24.dp)) {
            Text(text = "Hello,")
            Text(text = name)
        }
    }
}
```

Compose e Kotlin

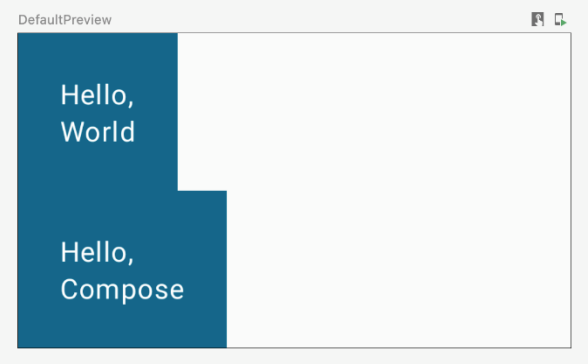
As funções de composição podem ser usadas como qualquer outra função no Kotlin. Isso torna a criação de IUs muito eficiente, já que é possível adicionar instruções para influenciar como a IU vai ser mostrada.
Por exemplo, é possível usar uma repetição for para adicionar elementos à Column:

```
@Composable
fun MyApp(
    modifier: Modifier = Modifier,
    names: List<String> = listOf("World", "Compose")
) {
    Column(modifier) {
        for (name in names) {
            Greeting(name = name)
        }
    }
}
```

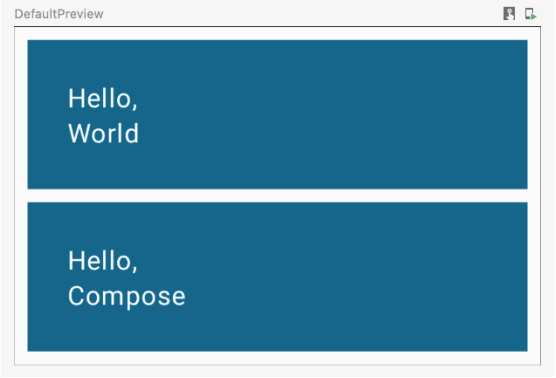
Você ainda não definiu dimensões ou não adicionou restrições ao tamanho das composições. Portanto, cada linha ocupa o mínimo de espaço possível, e a visualização faz o mesmo. Vamos mudar a visualização para emular a largura comum de um smartphone pequeno de 320 dp. Adicione um parâmetro widthDp à anotação @Preview desta maneira:

```
@Preview(showBackground = true, widthDp = 320)
@Composable
fun DefaultPreview() {
    BasicsCodelabTheme {
        MyApp()
```

```
    }
}
```



Como os modificadores são muito usados no Compose, vamos praticar com um exercício mais avançado: tentar replicar o layout abaixo usando os modificadores fillMaxWidth e padding.



Agora, compare o código com a solução:

```
@Composable
fun MyApp(
    modifier: Modifier = Modifier,
    names: List<String> = listOf("World", "Compose")
) {
    Column(modifier = modifier.padding(vertical = 4.dp)) {
        for (name in names) {
            Greeting(name = name)
        }
    }
}

@Composable
private fun Greeting(name: String) {
    Surface(
        color = MaterialTheme.colorScheme.primary,
        modifier = Modifier.padding(vertical = 4.dp, horizontal = 8.dp)
    ) {
        Column(modifier = Modifier.fillMaxWidth().padding(24.dp)) {
            Text(text = "Hello, ")
            Text(text = name)
        }
    }
}
```

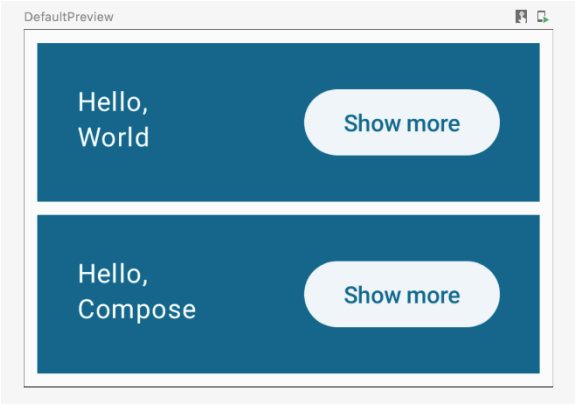
Algumas considerações:

- Os modificadores podem ter sobrecargas. Por exemplo, é possível especificar formas diferentes de criar um padding.
- Para adicionar vários modificadores a um elemento, basta encadeá-los.

Há várias maneiras de alcançar esse resultado. Sendo assim, caso seu código não corresponda a esse snippet, isso não significa que ele está errado. No entanto, copie e cole esse código para continuar com o tutorial.

Como adicionar um botão

Na próxima etapa, você vai adicionar um elemento clicável que abre a Greeting. Precisamos adicionar esse botão primeiro. O objetivo é criar o layout abaixo:



Button é uma composição fornecida pelo pacote do Material 3, que usa uma composição como o último argumento. Como os lambdas finais podem ser movidos para fora dos parênteses, você pode adicionar qualquer conteúdo ao botão como filho. Por exemplo, Text:

```
// Don't copy yet
Button(
   onClick = { } // You'll learn about this callback later
) {
   Text("Show less")
}
```

Observação: o Compose oferece diferentes tipos de Button de acordo com a especificação dos botões do Material Design (em inglês): Button, ElevatedButton, FilledTonalButton, OutlinedButton e TextButton. No seu caso, você vai usar um ElevatedButton que envolve um Text como o conteúdo do ElevatedButton.

Para fazer isso, você precisa aprender a posicionar uma composição no final de uma linha. Como não há um modificador alignEnd, você define um weight para a composição no início. O modificador weight faz com que o elemento preencha todo o espaço disponível, tornando-o *flexível*, eliminando efetivamente os outros elementos que não têm peso, que são chamados de *inflexíveis*. Isso também torna o modificador fillMaxWidth redundante.

Agora tente adicionar o botão e colocá-lo como mostrado na imagem anterior.

Confira a solução aqui:

```
import androidx.compose.foundation.layout.Row
import androidx.compose.material3.ElevatedButton
...

@Composable
private fun Greeting(name: String) {

   Surface(
      color = MaterialTheme.colorScheme.primary,
      modifier = Modifier.padding(vertical = 4.dp, horizontal = 8.dp)
   ) {
      Row(modifier = Modifier.padding(24.dp)) {
         Column(modifier = Modifier.weight(1f)) {
            Text(text = "Hello, ")
            Text(text = name)
         }
         ElevatedButton(
            onClick = { /* TODO */ }
         ) {
            Text("Show more")
         }
      }
   }
}
```

7. Estado no Compose

Nesta seção, você vai adicionar interação à tela. Até aqui, você criou layouts estáticos, mas agora vai fazer com que eles reajam às mudanças do usuário:

Antes de descobrir como tornar um botão clicável e como redimensionar um item, é necessário armazenar um valor que indique se cada item está aberto ou não, como o ***estado*** do item. Como precisamos ter um desses valores por saudação, o local lógico para isso é na composição `Greeting`. Confira esse booleano `expanded` e como ele é usado no código:

```
// Don't copy over
@Composable
private fun Greeting(name: String) {
   var expanded = false // Don't do this!

   Surface(
      color = MaterialTheme.colorScheme.primary,
      modifier = Modifier.padding(vertical = 4.dp, horizontal = 8.dp)
   ) {
      Row(modifier = Modifier.padding(24.dp)) {
         Column(modifier = Modifier.weight(1f)) {
            Text(text = "Hello, ")
            Text(text = name)
         }
         ElevatedButton(
            onClick = { expanded = !expanded }
         ) {
            Text(if (expanded) "Show less" else "Show more")
         }
      }
   }
}
```

Adicionamos também uma ação `onClick` e um texto de botão dinâmico. Vamos explicar melhor esses componentes mais adiante.

No entanto, **isso não funcionará como esperado**. Definir um valor diferente para a variável `expanded` não vai fazer com que o Compose a detecte como uma *mudança de estado*. Portanto, nada acontecerá.

Os apps do Compose transformam dados em interface chamando funções combináveis. Se os dados mudam, o Compose executa novamente essas funções com os novos dados, criando uma IU atualizada. Isso é chamado de **recomposição**. O Compose também analisa quais dados são necessários para uma composição individual. Assim, ele só precisa recompor os componentes cujos dados mudaram e pular a recomposição daqueles que não são afetados.

Como mencionado em **Como trabalhar com o Compose:**

As funções de composição podem ser executadas com frequência e em qualquer ordem. Você não precisa depender da ordem em que o código é executado ou de quantas vezes essa função vai ser recomposta.

A modificação dessa variável não aciona recomposições porque **ela não está sendo acompanhada pelo Compose**. Além disso, sempre que `Greeting` for chamada, a variável vai ser redefinida como falsa.

Para adicionar um estado interno a uma composição, use a função `mutableStateOf`, que faz com que o Compose recomponha funções que leiam esse `State`.

`State` e `MutableState` são interfaces que contêm algum valor e acionam atualizações de IU (recomposições) sempre que esse valor muda.

```
import androidx.compose.runtime.mutableStateOf
...
```

```
// Don't copy over
@Composable
fun Greeting() {
   val expanded = mutableStateOf(false) // Don't do this!
}
```

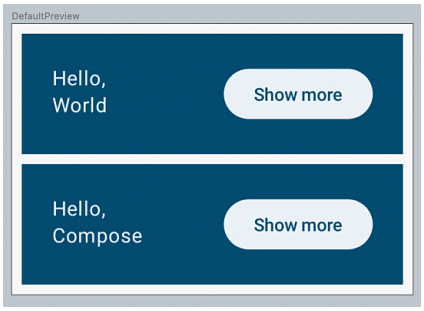
No entanto, **não é possível atribuir** `mutableStateOf` **a uma variável dentro de uma composição**. Como explicado anteriormente, a recomposição pode ocorrer a qualquer momento em que a composição é chamada novamente, redefinindo o estado para um novo estado mutável com um valor de `false`.
Para preservar o estado nas recomposições, *lembre-se* do estado mutável usando `remember`.

```
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
...

@Composable
fun Greeting() {
   val expanded = remember { mutableStateOf(false) }
   ...
}
```

`remember` é usado para ***proteger*** contra a recomposição, para que o estado não seja redefinido.
Se você chamar a mesma composição em diferentes partes da tela, vai criar elementos de IU diferentes, cada um com sua própria versão do estado. **Pense no estado interno como uma variável particular em uma classe**.
A função de composição será automaticamente "assinada" para o estado. Se o estado mudar, as composições que lerem esses campos serão recompostas para exibir as atualizações.
Como mudar o estado e reagir às mudanças de estado
Para mudar o estado, você pode ter notado que `Button` tem um parâmetro chamado `onClick`, mas não aceita um valor. Em vez disso, **ele usa uma função**.
No Compose, esse uso de funções é um recurso muito útil e bastante usado do Kotlin. As funções são `cidadãs de primeira classe no Kotlin`. Por isso, elas podem ser atribuídas a uma variável, transmitidas para outras funções e até mesmo retornadas delas.
Você pode definir a ação a ser tomada *ao clicar* atribuindo uma **expressão lambda** a ela. Por exemplo, vamos alternar o valor do estado expandido e mostrar um texto diferente, dependendo do valor.

```
ElevatedButton(
   onClick = { expanded.value = !expanded.value },
) {
  Text(if (expanded.value) "Show less" else "Show more")
}
```

Se você executar o app em um emulador, verá que, quando o botão é clicado, o estado `expanded` é alternado, acionando uma recomposição do texto dentro do botão. Cada `Greeting` mantém o próprio estado expandido porque pertence a diferentes elementos da IU.



Codifique até este ponto:

```
@Composable
private fun Greeting(name: String) {
   val expanded = remember { mutableStateOf(false) }

   Surface(
      color = MaterialTheme.colorScheme.primary,
      modifier = Modifier.padding(vertical = 4.dp, horizontal = 8.dp)
```

```
    ) {
        Row(modifier = Modifier.padding(24.dp)) {
            Column(modifier = Modifier.weight(1f)) {
                Text(text = "Hello, ")
                Text(text = name)
            }
            ElevatedButton(
                onClick = { expanded.value = !expanded.value }
            ) {
                Text(if (expanded.value) "Show less" else "Show more")
            }
        }
    }
}
```

Como abrir o item

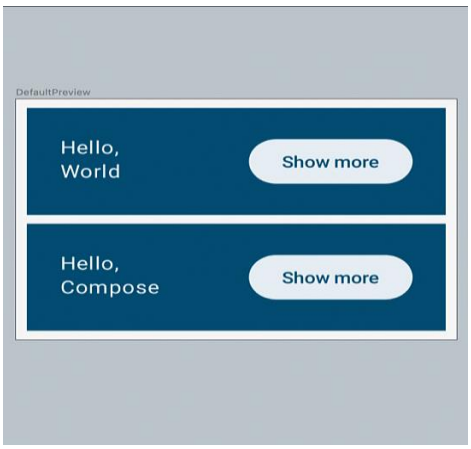
Agora vamos expandir um item quando solicitado. Adicione outra variável que dependa do nosso estado:

```
@Composable
private fun Greeting(name: String) {

    val expanded = remember { mutableStateOf(false) }

    val extraPadding = if (expanded.value) 48.dp else 0.dp
...
```

Você não precisa se lembrar de `extraPadding` em relação à recomposição porque ela está fazendo um cálculo simples. Agora podemos aplicar um novo modificador de padding à coluna:

```
@Composable
private fun Greeting(name: String) {

    val expanded = remember { mutableStateOf(false) }

    val extraPadding = if (expanded.value) 48.dp else 0.dp

    Surface(
        color = MaterialTheme.colorScheme.primary,
        modifier = Modifier.padding(vertical = 4.dp, horizontal = 8.dp)
    ) {
        Row(modifier = Modifier.padding(24.dp)) {
            Column(modifier = Modifier
                .weight(1f)
                .padding(bottom = extraPadding)
            ) {
                Text(text = "Hello, ")
                Text(text = name)
            }
            ElevatedButton(
                onClick = { expanded.value = !expanded.value }
            ) {
                Text(if (expanded.value) "Show less" else "Show more")
            }
        }
    }
}
```

Se você executar em um emulador, vai notar que cada item pode ser aberto de maneira independente:

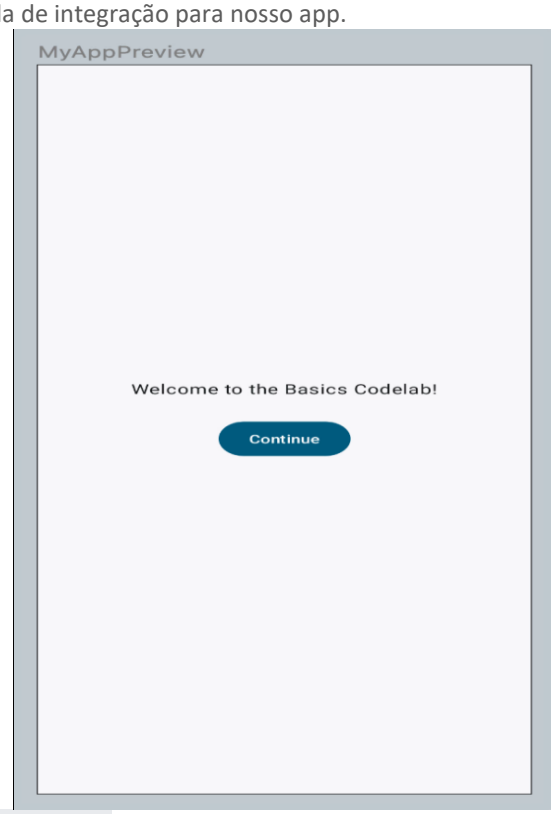


8. Elevação de estado

Nas funções combináveis, o estado lido ou modificado por várias funções precisa estar em um ancestral comum. Esse processo é chamado de **elevação de estado**. *Elevar* significa *levantar* ou *aumentar*.

A elevação de estado evita a duplicação do estado e a introdução de bugs, ajuda a reutilizar as funções combináveis e facilita muito o teste delas. Por outro lado, o estado que não precisa ser controlado pelo pai de uma função combinável não deve ser elevado. A **fonte da verdade** pertence a quem cria e controla esse estado.

Por exemplo, vamos criar uma tela de integração para nosso app.



Adicione o código a seguir a MainActivity.kt:

```
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.material3.Button
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
...

@Composable
fun OnboardingScreen(modifier: Modifier = Modifier) {
    // TODO: This state should be hoisted
    var shouldShowOnboarding by remember { mutableStateOf(true) }

    Column(
```

```
        modifier = modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text("Welcome to the Basics Codelab!")
        Button(
            modifier = Modifier.padding(vertical = 24.dp),
            onClick = { shouldShowOnboarding = false }
        ) {
            Text("Continue")
        }
    }
}

@Preview(showBackground = true, widthDp = 320, heightDp = 320)
@Composable
fun OnboardingPreview() {
    BasicsCodelabTheme {
        OnboardingScreen()
    }
}
```

Esse código contém muitos recursos novos:

- Você adicionou uma nova composição chamada `OnboardingScreen` e uma nova **visualização**. Se você criar o projeto, vai perceber que pode ter várias visualizações ao mesmo tempo. Também adicionamos uma altura fixa para verificar se o conteúdo está alinhado corretamente.
- A `Column` pode ser configurada para exibir o conteúdo no centro da tela.
- `shouldShowOnboarding` está usando uma palavra-chave `by` em vez de `=`. É um delegado de propriedade que evita que você digite `.value` todas as vezes.
- Quando o botão é clicado, `shouldShowOnboarding` é definido como `false`, mas você ainda não está lendo o estado de nenhum lugar.

Agora podemos adicionar essa nova tela de integração ao nosso app. Queremos que ela apareça na inicialização e depois seja ocultada quando o usuário pressionar "Continue".

No Compose, **você não oculta elementos de IU**. Em vez disso, você simplesmente não os adiciona à composição. Eles não são adicionados à árvore da IU gerada pelo Compose. Para isso, use uma lógica condicional de Kotlin simples. Por exemplo, para mostrar a tela de integração ou a lista de saudações, você pode fazer algo como:

```
// Don't copy yet
@Composable
fun MyApp(modifier: Modifier = Modifier) {
    Surface(modifier) {
        if (shouldShowOnboarding) { // Where does this come from?
            OnboardingScreen()
        } else {
            Greetings()
        }
    }
}
```

No entanto, não temos acesso a `shouldShowOnboarding`. É claro que precisamos compartilhar o estado que criamos em `OnboardingScreen` com a composição `MyApp`.

Em vez de compartilhar o valor do estado com o pai, ***elevamos*** o estado. Basta o mover para o ancestral comum que precisa acessá-lo.

Primeiro, mova o conteúdo de `MyApp` para uma nova composição, chamada `Greetings`: Adapte também a visualização para chamar o método `Greetings`:

```
@Composable
fun MyApp(modifier: Modifier = Modifier) {
    Greetings()
}

@Composable
```

```
private fun Greetings(
    modifier: Modifier = Modifier,
    names: List<String> = listOf("World", "Compose")
) {
    Column(modifier = modifier.padding(vertical = 4.dp)) {
        for (name in names) {
            Greeting(name = name)
        }
    }
}

@Preview(showBackground = true, widthDp = 320)
@Composable
private fun GreetingsPreview() {
    BasicsCodelabTheme {
        Greetings()
    }
}
```

Adicione uma visualização do novo elemento combinável MyApp de nível superior para que possamos testar o comportamento dele:

```
@Preview
@Composable
fun MyAppPreview() {
    BasicsCodelabTheme {
        MyApp(Modifier.fillMaxSize())
    }
}
```

Agora adicione a lógica para mostrar as diferentes telas no MyApp e **elevar** o estado.

```
@Composable
fun MyApp(modifier: Modifier = Modifier) {

    var shouldShowOnboarding by remember { mutableStateOf(true) }

    Surface(modifier) {
        if (shouldShowOnboarding) {
            OnboardingScreen(/* TODO */)
        } else {
            Greetings()
        }
    }
}
```

Também precisamos compartilhar shouldShowOnboarding com a tela de integração, mas a transmissão não vai ser direta. Em vez de permitir que OnboardingScreen mude nosso estado, é melhor nos avisar quando o usuário clicar no botão *Continue*.

Como transmitimos eventos? **Transmitindo callbacks**. Callbacks são funções transmitidas como argumentos para outras funções e executadas quando o evento ocorre.

Tente adicionar um parâmetro de função à tela de integração definida como onContinueClicked: () -> Unit para que você possa modificar o estado da MyApp.

Solução:

```
@Composable
fun MyApp(modifier: Modifier = Modifier) {

    var shouldShowOnboarding by remember { mutableStateOf(true) }

    Surface(modifier) {
        if (shouldShowOnboarding) {
            OnboardingScreen(onContinueClicked = { shouldShowOnboarding = false })
```

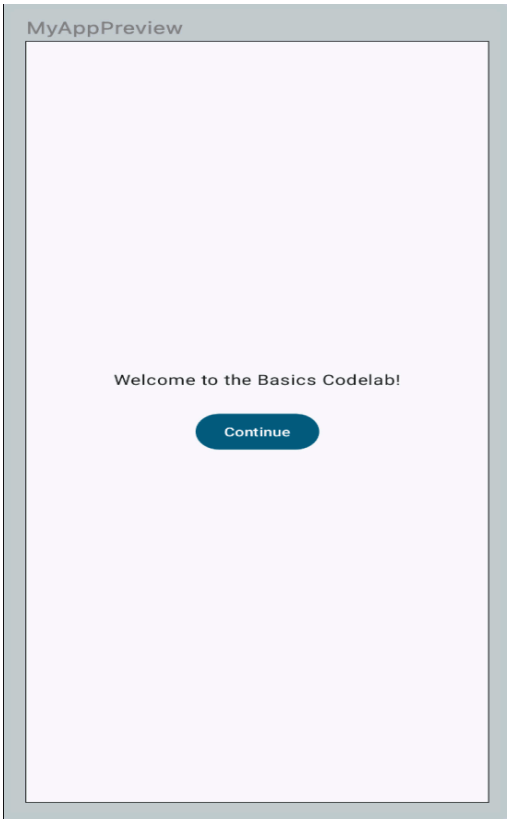
```
        } else {
            Greetings()
        }
    }
}

@Composable
fun OnboardingScreen(
    onContinueClicked: () -> Unit,
    modifier: Modifier = Modifier
) {

    Column(
        modifier = modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text("Welcome to the Basics Codelab!")
        Button(
            modifier = Modifier
                .padding(vertical = 24.dp),
            onClick = onContinueClicked
        ) {
            Text("Continue")
        }
    }

}
```

Ao transmitir uma função e não um estado para `OnboardingScreen`, estamos tornando essa composição mais reutilizável e protegendo o estado contra mutação por outras composições. Em geral, é simples. Um bom exemplo é como a visualização de integração precisa ser modificada para chamar `OnboardingScreen` agora:

```
@Preview(showBackground = true, widthDp = 320, heightDp = 320)
@Composable
fun OnboardingPreview() {
    BasicsCodelabTheme {
        OnboardingScreen(onContinueClicked = {}) // Do nothing on click.
    }
}
```

Atribuir `onContinueClicked` a uma expressão lambda vazia significa "não fazer nada", o que é perfeito para uma visualização.

O app está bem parecido com um app real. Bom trabalho!



Código completo até agora:

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Button
import androidx.compose.material3.ElevatedButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import com.codelab.basics.ui.theme.BasicsCodelabTheme

class MainActivity : ComponentActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContent {
         BasicsCodelabTheme {
            MyApp(modifier = Modifier.fillMaxSize())
         }
      }
   }
}
```

```
@Composable
fun MyApp(modifier: Modifier = Modifier) {

    var shouldShowOnboarding by remember { mutableStateOf(true) }

    Surface(modifier) {
        if (shouldShowOnboarding) {
            OnboardingScreen(onContinueClicked = { shouldShowOnboarding = false })
        } else {
            Greetings()
        }
    }
}

@Composable
fun OnboardingScreen(
    onContinueClicked: () -> Unit,
    modifier: Modifier = Modifier
) {

    Column(
        modifier = modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text("Welcome to the Basics Codelab!")
        Button(
            modifier = Modifier.padding(vertical = 24.dp),
            onClick = onContinueClicked
        ) {
            Text("Continue")
        }
    }
}

@Composable
private fun Greetings(
    modifier: Modifier = Modifier,
    names: List<String> = listOf("World", "Compose")
) {
    Column(modifier = modifier.padding(vertical = 4.dp)) {
        for (name in names) {
            Greeting(name = name)
        }
    }
}

@Preview(showBackground = true, widthDp = 320, heightDp = 320)
@Composable
fun OnboardingPreview() {
    BasicsCodelabTheme {
        OnboardingScreen(onContinueClicked = {})
    }
}

@Composable
private fun Greeting(name: String) {

    val expanded = remember { mutableStateOf(false) }
```

```
    val extraPadding = if (expanded.value) 48.dp else 0.dp

    Surface(
        color = MaterialTheme.colorScheme.primary,
        modifier = Modifier.padding(vertical = 4.dp, horizontal = 8.dp)
    ) {
        Row(modifier = Modifier.padding(24.dp)) {
            Column(modifier = Modifier
                .weight(1f)
                .padding(bottom = extraPadding)
            ) {
                Text(text = "Hello, ")
                Text(text = name)
            }
            ElevatedButton(
                onClick = { expanded.value = !expanded.value }
            ) {
                Text(if (expanded.value) "Show less" else "Show more")
            }
        }
    }
}

@Preview(showBackground = true, widthDp = 320)
@Composable
fun DefaultPreview() {
    BasicsCodelabTheme {
        Greetings()
    }
}

@Preview
@Composable
fun MyAppPreview() {
    BasicsCodelabTheme {
        MyApp(Modifier.fillMaxSize())
    }
}
```

9. Como criar uma lista de desempenho lento

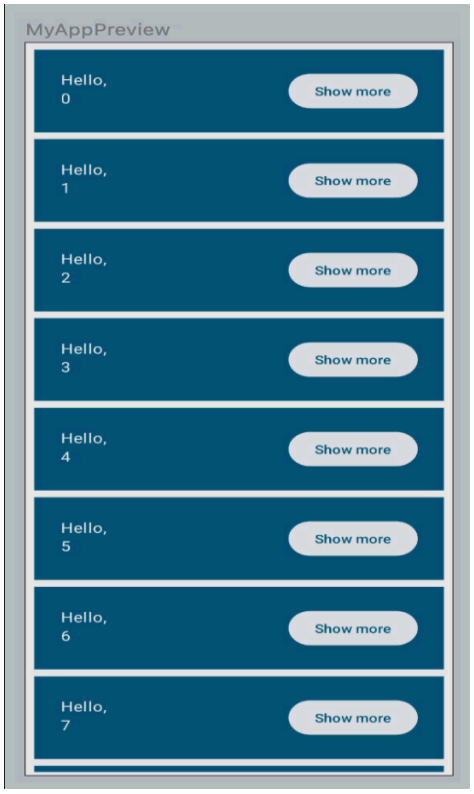
Agora vamos deixar a lista de nomes mais realista. Até agora, você exibiu duas saudações em uma `Column`. No entanto, ela consegue lidar com milhares de saudações?
Mude o valor da lista padrão nos parâmetros de `Greetings` para usar outro construtor de lista que permita definir o tamanho e preenchê-la com o valor contido na lambda (aqui, `$it` representa o índice da lista):
`names: List<String> = List(1000) { "$it" }`

Isso cria 1.000 saudações, mesmo as que não cabem na tela. Obviamente, isso não é um bom desempenho. Você pode tentar executá-lo em um emulador. Aviso: esse código pode congelar o emulador.
Para exibir uma coluna rolável, usamos uma `LazyColumn`. `LazyColumn` renderiza somente os itens visíveis na tela, permitindo ganhos de desempenho ao renderizar uma lista grande.
Observação: `LazyColumn` e `LazyRow` são equivalentes a `RecyclerView` nas visualizações do Android.
No uso básico, a API `LazyColumn` fornece um elemento `items` no escopo, em que a lógica de renderização de itens individuais é escrita:

```
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
...

@Composable
```

```
private fun Greetings(
    modifier: Modifier = Modifier,
    names: List<String> = List(1000) { "$it" }
) {
    LazyColumn(modifier = modifier.padding(vertical = 4.dp)) {
        items(items = names) { name ->
            Greeting(name = name)
        }
    }
}
```

Observação: importe androidx.compose.foundation.lazy.items, porque o Android Studio escolherá uma função de itens diferente por padrão.
Observação: a LazyColumn não recicla os filhos como RecyclerView. Ela emite novas composições à medida que você rola e ainda apresenta uma boa performance, já que a emissão de composições é relativamente mais barata em comparação com a instanciação de Views do Android.



10. Estado persistente

Nosso app tem um problema: a tela de integração vai ser mostrada de novo se ele for executado em um dispositivo e você clicar nos botões e girá-lo. A função remember funciona **somente enquanto a composição for mantida**. Quando você faz a rotação, toda a atividade é reiniciada e o estado é perdido. Isso também acontece com qualquer mudança de configuração e após a interrupção do processo.
Em vez de usar remember, use rememberSaveable. Isso salvará cada estado que sobreviveu a mudanças de configuração (como rotações) e à interrupção do processo.
Agora substitua o uso de remember em shouldShowOnboarding por rememberSaveable:

```
var shouldShowOnboarding by rememberSaveable { mutableStateOf(true) }
```

Execute, gire, mude para o modo escuro ou elimine o processo. A tela de integração não é mostrada, a menos que você tenha fechado o app antes.
Com cerca de 120 linhas de código até agora, você conseguiu mostrar uma lista de rolagem dos itens longa e de alta performance, em que cada item mantém o próprio estado. Além disso, como você pode notar, o app tem um modo escuro perfeitamente correto sem linhas extras de código. Você vai aprender sobre a aplicação de temas nas próximas unidades.

11. Como animar a lista

No Compose, existem várias maneiras de animar a IU: de APIs de alto nível para animações simples a métodos de baixo nível para controle total e transições complexas.

Nesta seção, você vai usar uma das APIs de baixo nível, mas não se preocupe, porque elas também podem ser muito simples. Vamos animar a mudança no tamanho que já implementamos:

Para isso, você usará o combinável animateDpAsState. Ela retorna um objeto State cujo value será atualizado continuamente pela animação até que ela termine. É necessário um "valor de destino" com o tipo Dp.

Crie um extraPadding animado que dependa do estado expandido. Além disso, vamos usar o delegado da propriedade (a palavra-chave by):

```
@Composable
private fun Greeting(name: String) {

    var expanded by remember { mutableStateOf(false) }

    val extraPadding by animateDpAsState(
        if (expanded) 48.dp else 0.dp
    )
    Surface(
        color = MaterialTheme.colorScheme.primary,
        modifier = Modifier.padding(vertical = 4.dp, horizontal = 8.dp)
    ) {
        Row(modifier = Modifier.padding(24.dp)) {
            Column(modifier = Modifier
                .weight(1f)
                .padding(bottom = extraPadding)
            ) {
                Text(text = "Hello, ")
                Text(text = name)
            }
            ElevatedButton(
                onClick = { expanded = !expanded }
            ) {
                Text(if (expanded) "Show less" else "Show more")
            }

        }
    }
```

```
}
```

Execute o app e teste a animação.

Observação: se você abrir o item número 1, rolar para o número 20 e voltar para o número 1, vai perceber que 1 voltou ao tamanho original. Você poderia salvar esses dados com `rememberSaveable` se isso fosse um requisito, mas o exemplo devia ser simples.

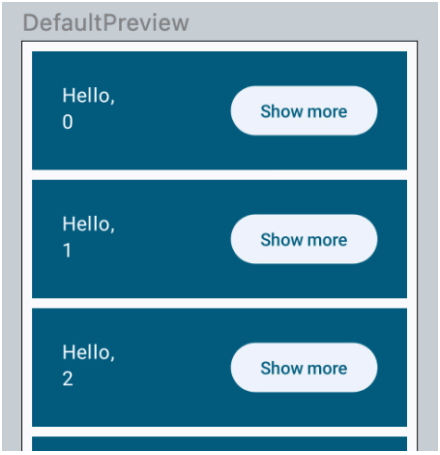
`animateDpAsState` usa um parâmetro `animationSpec` opcional que permite personalizar a animação. Vamos fazer algo mais divertido, como adicionar uma animação com molas:

```
@Composable
private fun Greeting(name: String) {

   var expanded by remember { mutableStateOf(false) }

   val extraPadding by animateDpAsState(
      if (expanded) 48.dp else 0.dp,
      animationSpec = spring(
         dampingRatio = Spring.DampingRatioMediumBouncy,
         stiffness = Spring.StiffnessLow
      )
   )

   Surface(
   ...
         Column(modifier = Modifier
            .weight(1f)
            .padding(bottom = extraPadding.coerceAtLeast(0.dp))

   ...

   )
}
```

Também vamos garantir que o padding nunca seja negativo. Caso contrário, ele pode causar falhas no app. Isso introduz um bug de animação sutil que corrigiremos mais tarde na seção **Toques finais**.

A especificação `spring` não aceita parâmetros relacionados ao tempo. Em vez disso, as propriedades físicas (amortecimento e rigidez) dependem deles para tornar as animações mais naturais. Execute o app agora para testar a nova animação:

Nenhuma animação criada com animate*AsState pode ser interrompida. Isso significa que, se o valor de destino mudar no meio da animação, o animate*AsState vai reiniciar a animação e apontar para o novo valor. As interrupções parecem especialmente naturais com animações com molas:



Se você quiser analisar os diferentes tipos de animação, experimente usar outros parâmetros para spring, especificações diferentes (tween, repeatable) e mais funções: animateColorAsState ou um tipo diferente de animação da API Animation.

Código completo desta seção

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.animation.core.Spring
import androidx.compose.animation.core.animateDpAsState
import androidx.compose.animation.core.spring
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material3.Button
import androidx.compose.material3.ElevatedButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.saveable.rememberSaveable
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import com.codelab.basics.ui.theme.BasicsCodelabTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            BasicsCodelabTheme {
                MyApp(modifier = Modifier.fillMaxSize())
            }
        }
    }
}
```

```
@Composable
fun MyApp(modifier: Modifier = Modifier) {

    var shouldShowOnboarding by rememberSaveable { mutableStateOf(true) }

    Surface(modifier) {
        if (shouldShowOnboarding) {
            OnboardingScreen(onContinueClicked = { shouldShowOnboarding = false })
        } else {
            Greetings()
        }
    }
}

@Composable
fun OnboardingScreen(
    onContinueClicked: () -> Unit,
    modifier: Modifier = Modifier
) {

    Column(
        modifier = modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text("Welcome to the Basics Codelab!")
        Button(
            modifier = Modifier.padding(vertical = 24.dp),
            onClick = onContinueClicked
        ) {
            Text("Continue")
        }
    }

}

@Composable
private fun Greetings(
    modifier: Modifier = Modifier,
    names: List<String> = List(1000) { "$it" }
) {
    LazyColumn(modifier = modifier.padding(vertical = 4.dp)) {
        items(items = names) { name ->
            Greeting(name = name)
        }
    }
}

@Preview(showBackground = true, widthDp = 320, heightDp = 320)
@Composable
fun OnboardingPreview() {
    BasicsCodelabTheme {
        OnboardingScreen(onContinueClicked = {})
    }
}

@Composable
private fun Greeting(name: String) {
```

```
    var expanded by remember { mutableStateOf(false) }

    val extraPadding by animateDpAsState(
        if (expanded) 48.dp else 0.dp,
        animationSpec = spring(
            dampingRatio = Spring.DampingRatioMediumBouncy,
            stiffness = Spring.StiffnessLow
        )
    )
    Surface(
        color = MaterialTheme.colorScheme.primary,
        modifier = Modifier.padding(vertical = 4.dp, horizontal = 8.dp)
    ) {
        Row(modifier = Modifier.padding(24.dp)) {
            Column(modifier = Modifier
                .weight(1f)
                .padding(bottom = extraPadding.coerceAtLeast(0.dp))
            ) {
                Text(text = "Hello, ")
                Text(text = name)
            }
            ElevatedButton(
                onClick = { expanded = !expanded }
            ) {
                Text(if (expanded) "Show less" else "Show more")
            }
        }
    }
}

@Preview(showBackground = true, widthDp = 320)
@Composable
fun DefaultPreview() {
    BasicsCodelabTheme {
        Greetings()
    }
}

@Preview
@Composable
fun MyAppPreview() {
    BasicsCodelabTheme {
        MyApp(Modifier.fillMaxSize())
    }
}
```

12. Como definir o estilo e aplicar temas no app

Você não definiu o estilo de nenhuma composição até o momento, mas já conseguiu um padrão decente, incluindo o suporte ao modo escuro. Vamos ver o que são `BasicsCodelabTheme` e `MaterialTheme`.

Se você abrir o arquivo `ui/theme/Theme.kt`, verá que `BasicsCodelabTheme` usa `MaterialTheme` na implementação:

```
// Do not copy
@Composable
fun BasicsCodelabTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    // Dynamic color is available on Android 12+
    dynamicColor: Boolean = true,
    content: @Composable () -> Unit
) {
    // ...
```

```
    MaterialTheme(
        colorScheme = colorScheme,
        typography = Typography,
        content = content
    )
}
```

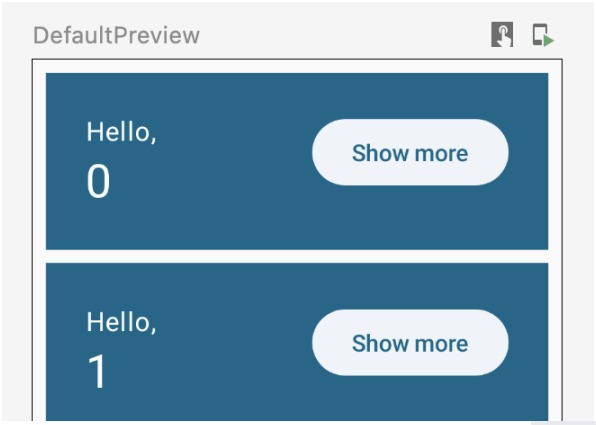
MaterialTheme é uma função combinável que reflete os princípios de estilo da especificação do Material Design (em inglês). Essas informações de estilo são aplicadas em cascata aos componentes que estão dentro do content, que pode ler as informações para definir o estilo. Na IU, você já está usando BasicsCodelabTheme da seguinte maneira:

```
BasicsCodelabTheme {
    MyApp(modifier = Modifier.fillMaxSize())
}
```

Como BasicsCodelabTheme envolve MaterialTheme internamente, MyApp é estilizado com as propriedades definidas no tema. Em qualquer composição descendente, é possível recuperar três propriedades de MaterialTheme: colorScheme, typography e shapes. Use-as para definir o estilo de cabeçalho como um dos Texts:

```
Column(modifier = Modifier
    .weight(1f)
    .padding(bottom = extraPadding.coerceAtLeast(0.dp))
) {
    Text(text = "Hello, ")
    Text(text = name, style = MaterialTheme.typography.headlineMedium)
}
```

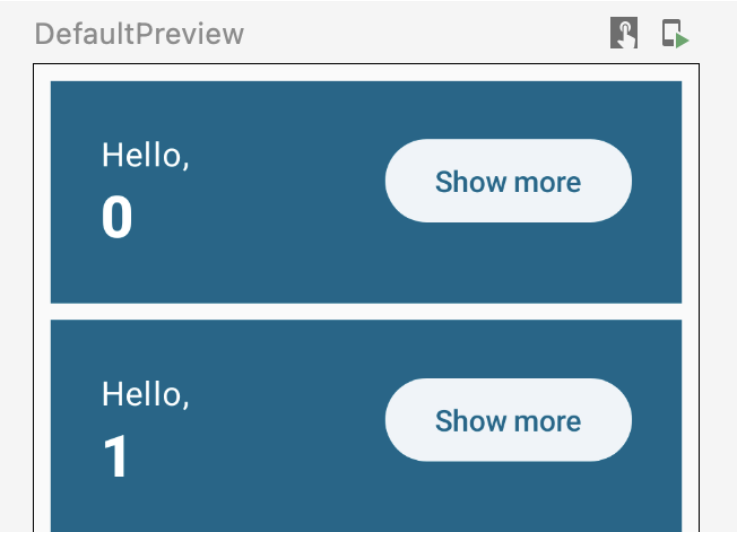
A composição Text no exemplo acima define um novo TextStyle. Você pode criar seu próprio TextStyle ou extrair um estilo definido pelo tema usando MaterialTheme.typography, que é preferencial. Essa construção oferece acesso aos estilos de texto definidos pelo Material Design, como displayLarge, headlineMedium, titleSmall, bodyLarge, labelMedium. No exemplo, usamos o estilo headlineMedium, definido no tema.
Crie o app agora para conferir nosso texto com estilo recém-definido:



Em geral, é muito melhor manter as cores, as formas e os estilos de fonte em um MaterialTheme. Por exemplo, o modo escuro seria difícil de implementar se você codificasse cores e exigiria muito trabalho propenso a erros para corrigir.
No entanto, às vezes, é necessário desviar um pouco da seleção de cores e estilos de fonte. Nessas situações, é melhor basear a cor ou o estilo em uma cor ou um estilo existente.
Para isso, é possível modificar um estilo predefinido usando a função copy. Coloque o número em negrito:

```
Text(
    text = name,
    style = MaterialTheme.typography.headlineMedium.copy(
        fontWeight = FontWeight.ExtraBold
    )
)
```

Dessa forma, se você precisar mudar a família de fontes ou qualquer outro atributo de headlineMedium, não vai ter que se preocupar com os pequenos desvios.
Este vai ser o resultado na janela de visualização:

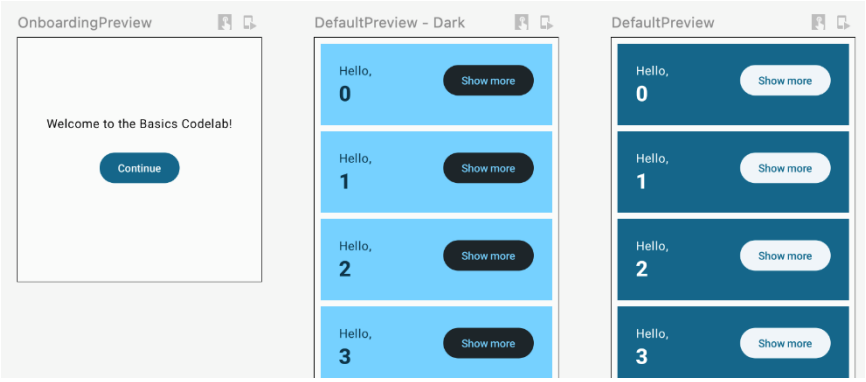


Configurar uma visualização do modo escuro

Por enquanto, a prévia mostra apenas como o app vai ficar no modo claro. Adicione uma anotação `@Preview` extra a `DefaultPreview` com `UI_MODE_NIGHT_YES`:

```
@Preview(
    showBackground = true,
    widthDp = 320,
    uiMode = UI_MODE_NIGHT_YES,
    name = "Dark"
)
@Preview(showBackground = true, widthDp = 320)
@Composable
fun DefaultPreview() {
    BasicsCodelabTheme {
        Greetings()
    }
}
```

Isso adiciona uma visualização no modo escuro.



Ajustar o tema do app

Tudo o que está relacionado ao tema atual pode ser encontrado nos arquivos dentro da pasta `ui/theme`. Por exemplo, as cores padrão que estamos usando até agora são definidas em `Color.kt`.

Vamos começar definindo novas cores. Adicione-as a `Color.kt`:

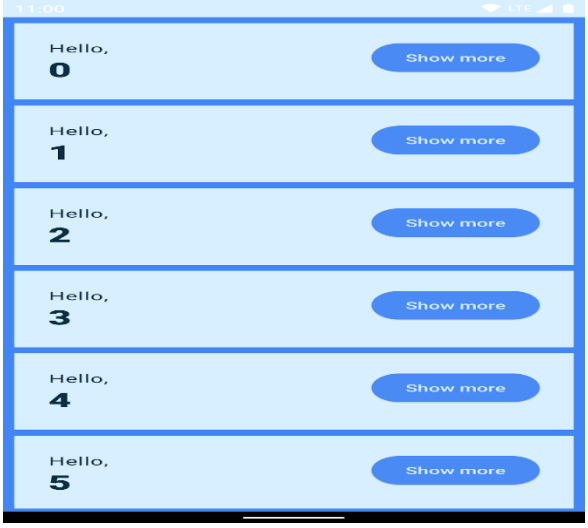
```
val Navy = Color(0xFF073042)
val Blue = Color(0xFF4285F4)
val LightBlue = Color(0xFFD7EFFE)
val Chartreuse = Color(0xFFEFF7CF)
```

Agora atribua-as à paleta do `MaterialTheme` em `Theme.kt`:

```
private val LightColorScheme = lightColorScheme(
    surface = Blue,
    onSurface = Color.White,
    primary = LightBlue,
```

```
    onPrimary = Navy
)
```

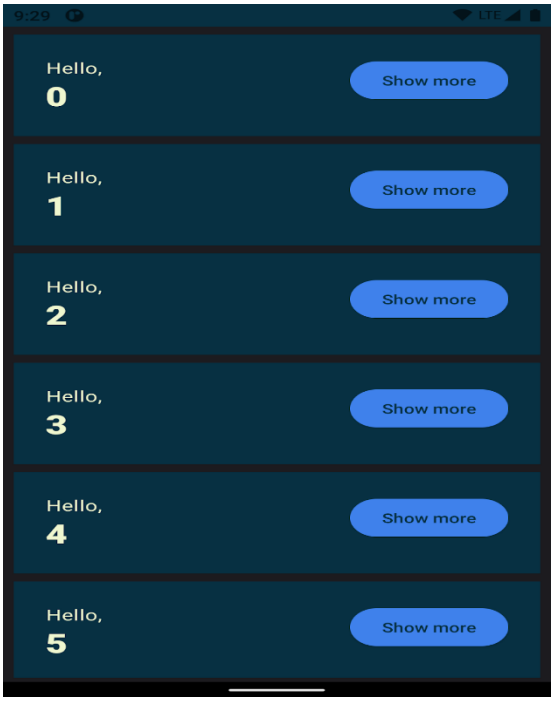
Se você voltar para MainActivity.kt e atualizar a visualização, as cores não vão mudar. Isso ocorre porque, por padrão, a visualização usa **cores dinâmicas**. Você pode ver a lógica para adicionar cores dinâmicas em Theme.kt, usando o parâmetro booleano dynamicColor.
Para ver a versão não adaptável do esquema de cores, execute o app em um dispositivo com API de nível anterior a 31 (que corresponde ao Android S, em que as cores adaptáveis foram introduzidas). Agora você vai ver as novas cores:



Em Theme.kt, defina a paleta para cores escuras:

```
private val DarkColorScheme = darkColorScheme(
    surface = Blue,
    onSurface = Navy,
    primary = Navy,
    onPrimary = Chartreuse
)
```

Agora, ao executar o app, podemos ver como as cores ficam no modo escuro:



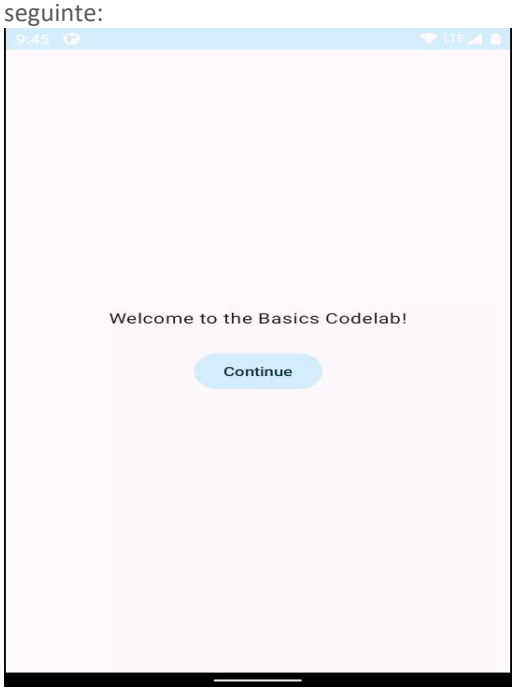
Código final para Theme.kt

```
import android.app.Activity
import android.os.Build
import androidx.compose.foundation.isSystemInDarkTheme
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.darkColorScheme
```

```
import androidx.compose.material3.dynamicDarkColorScheme
import androidx.compose.material3.dynamicLightColorScheme
import androidx.compose.material3.lightColorScheme
import androidx.compose.runtime.Composable
import androidx.compose.runtime.SideEffect
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.toArgb
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.platform.LocalView
import androidx.core.view.ViewCompat

private val DarkColorScheme = darkColorScheme(
    surface = Blue,
    onSurface = Navy,
    primary = Navy,
    onPrimary = Chartreuse
)

private val LightColorScheme = lightColorScheme(
    surface = Blue,
    onSurface = Color.White,
    primary = LightBlue,
    onPrimary = Navy
)

@Composable
fun BasicsCodelabTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    // Dynamic color is available on Android 12+
    dynamicColor: Boolean = true,
    content: @Composable () -> Unit
) {
    val colorScheme = when {
        dynamicColor && Build.VERSION.SDK_INT >= Build.VERSION_CODES.S -> {
            val context = LocalContext.current
            if (darkTheme) dynamicDarkColorScheme(context) else dynamicLightColorScheme(context)
        }
        darkTheme -> DarkColorScheme
        else -> LightColorScheme
    }
    val view = LocalView.current
    if (!view.isInEditMode) {
        SideEffect {
            (view.context as Activity).window.statusBarColor = colorScheme.primary.toArgb()
            ViewCompat.getWindowInsetsController(view)?.isAppearanceLightStatusBars = darkTheme
        }
    }

    MaterialTheme(
        colorScheme = colorScheme,
        typography = Typography,
        content = content
    )
}
```

13. Toques finais!

Nesta etapa, você vai aplicar o que já sabe e aprender novos conceitos com apenas algumas dicas. Você vai criar o seguinte:



Substituir o botão por um ícone

- Use a composição IconButton com um filho Icon.
- Use Icons.Filled.ExpandLess e Icons.Filled.ExpandMore, que estão disponíveis no artefato material-icons-extended. Adicione a seguinte linha às dependências no arquivo app/build.gradle:

```
implementation "androidx.compose.material:material-icons-extended:$compose_version"
```

- Modifique os paddings para corrigir o alinhamento.
- Adicione uma descrição do conteúdo para acessibilidade (consulte "Usar recursos de string" abaixo).

Usar recursos de string

A descrição do conteúdo de "Mostrar mais" e "Mostrar menos" precisa estar presente, e é possível adicioná-la com uma simples instrução if:

```
contentDescription = if (expanded) "Show less" else "Show more"
```

No entanto, strings codificadas não são uma prática recomendada. Você pode usar as strings no arquivo strings.xml. Você pode usar a opção "Extract string resource" em cada string, disponível em "Context Actions" no Android Studio, para fazer isso automaticamente.

Como alternativa, abra app/src/res/values/strings.xml e adicione os seguintes recursos:

```
<string name="show_less">Show less</string>
<string name="show_more">Show more</string>
```

Mostrar mais

O texto "Composem ipsum" aparece e desaparece, acionando uma mudança no tamanho de cada card.

- Adicione um novo Text à coluna dentro de Greeting que vai ser exibida quando o item for expandido.
- Remova o extraPadding e aplique o modificador animateContentSize à Row. Isso automatizará o processo de criação da animação, o que seria difícil de fazer manualmente. Além disso, não é mais necessário usar coerceAtLeast.

Adicionar elevação e formas

- Você pode usar o modificador shadow com o modificador clip para ter a aparência do card. No entanto, há uma composição do Material Design que faz exatamente isso: Card. Para mudar as cores do Card, chame CardDefaults.cardColors e substitua a cor que você quer mudar.

Código final

```
import android.content.res.Configuration.UI_MODE_NIGHT_YES
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.animation.animateContentSize
```

```
import androidx.compose.animation.core.Spring
import androidx.compose.animation.core.spring
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material.icons.Icons.Filled
import androidx.compose.material.icons.filled.ExpandLess
import androidx.compose.material.icons.filled.ExpandMore
import androidx.compose.material3.Button
import androidx.compose.material3.Card
import androidx.compose.material3.CardDefaults
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.saveable.rememberSaveable
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import com.codelab.basics.ui.theme.BasicsCodelabTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            BasicsCodelabTheme {
                MyApp(modifier = Modifier.fillMaxSize())
            }
        }
    }
}

@Composable
fun MyApp(modifier: Modifier = Modifier) {
    var shouldShowOnboarding by rememberSaveable { mutableStateOf(true) }

    Surface(modifier, color = MaterialTheme.colorScheme.background) {
        if (shouldShowOnboarding) {
            OnboardingScreen(onContinueClicked = { shouldShowOnboarding = false })
        } else {
            Greetings()
        }
    }
}

@Composable
fun OnboardingScreen(
```

```
    onContinueClicked: () -> Unit,
    modifier: Modifier = Modifier
) {
    Column(
        modifier = modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text("Welcome to the Basics Codelab!")
        Button(
            modifier = Modifier.padding(vertical = 24.dp),
            onClick = onContinueClicked
        ) {
            Text("Continue")
        }
    }
}

@Composable
private fun Greetings(
    modifier: Modifier = Modifier,
    names: List<String> = List(1000) { "$it" }
) {
    LazyColumn(modifier = modifier.padding(vertical = 4.dp)) {
        items(items = names) { name ->
            Greeting(name = name)
        }
    }
}

@Composable
private fun Greeting(name: String) {
    Card(
        colors = CardDefaults.cardColors(
            containerColor = MaterialTheme.colorScheme.primary
        ),
        modifier = Modifier.padding(vertical = 4.dp, horizontal = 8.dp)
    ) {
        CardContent(name)
    }
}

@Composable
private fun CardContent(name: String) {
    var expanded by remember { mutableStateOf(false) }

    Row(
        modifier = Modifier
            .padding(12.dp)
            .animateContentSize(
                animationSpec = spring(
                    dampingRatio = Spring.DampingRatioMediumBouncy,
                    stiffness = Spring.StiffnessLow
                )
            )
    ) {
        Column(
            modifier = Modifier
                .weight(1f)
                .padding(12.dp)
```

```
        ) {
            Text(text = "Hello, ")
            Text(
                text = name, style = MaterialTheme.typography.headlineMedium.copy(
                    fontWeight = FontWeight.ExtraBold
                )
            )
            if (expanded) {
                Text(
                    text = ("Composem ipsum color sit lazy, " +
                        "padding theme elit, sed do bouncy. ").repeat(4),
                )
            }
        }
        IconButton(onClick = { expanded = !expanded }) {
            Icon(
                imageVector = if (expanded) Filled.ExpandLess else Filled.ExpandMore,
                contentDescription = if (expanded) {
                    stringResource(R.string.show_less)
                } else {
                    stringResource(R.string.show_more)
                }
            )
        }
    }
}

@Preview(
    showBackground = true,
    widthDp = 320,
    uiMode = UI_MODE_NIGHT_YES,
    name = "DefaultPreviewDark"
)
@Preview(showBackground = true, widthDp = 320)
@Composable
fun DefaultPreview() {
    BasicsCodelabTheme {
        Greetings()
    }
}

@Preview(showBackground = true, widthDp = 320, heightDp = 320)
@Composable
fun OnboardingPreview() {
    BasicsCodelabTheme {
        OnboardingScreen(onContinueClicked = {})
    }
}

@Preview
@Composable
fun MyAppPreview() {
    BasicsCodelabTheme {
        MyApp(Modifier.fillMaxSize())
    }
}
```