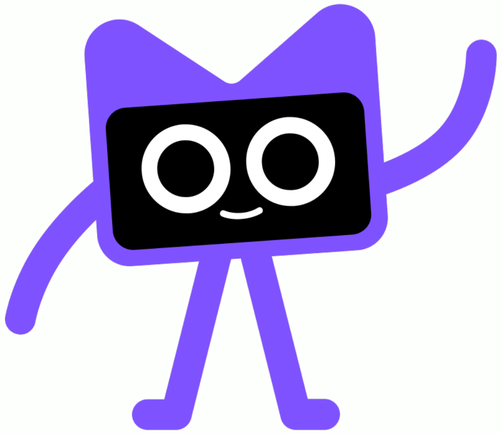
UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Modificadores e Restrições

QXD0276 - Desenvolvimento de Software para Dispositivos Móveis



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

Conteúdo

- Introdução
- Modifier
- Restrições
- Um pouco mais de Kotlin

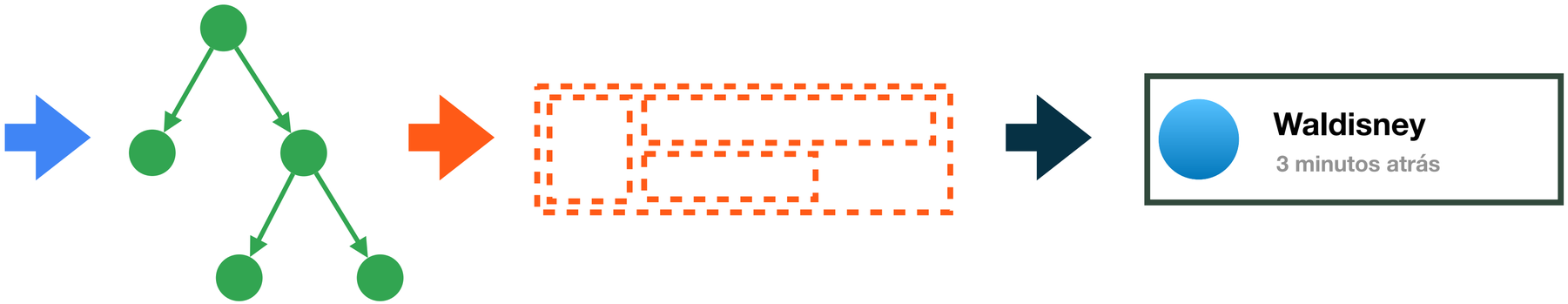
Introdução

Introdução

- O Jetpack Compose transforma estados em elementos da UI da seguinte forma:

O que ? *Onde ?* *Como ?*

```
Row {
   Image()
   Column {
     Text()
     Text()
   }
}
```

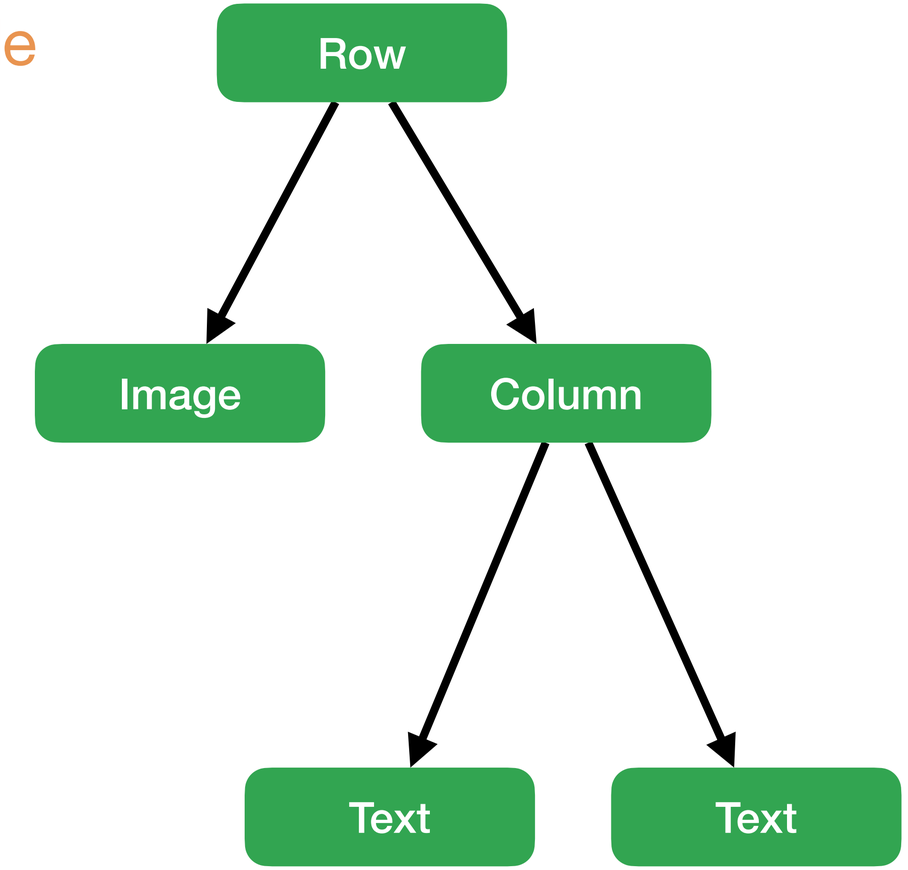


Introdução

Composition

- Na fase de composição, as funções @Composable são executas e gerando uma estrutura em árvore que representa a interface
- Cada função @Composable é mapeada para um único nó de layout na árvore de UI

```
Row {
  Image( )
  Column {
    Text( )
    Text( )
  }
}
```



Introdução

Layout

- A árvore produzida na fase de composition é utilizada como entrada
- Durante esta fase, a árvore de UI é percorrida em um único passo
- Os nós da árvore possuem as informações necessárias para decidir o tamanho e a localização de cada um
- O algoritmo usado para percorrer árvore possui três etapas como veremos a seguir
- Ao final dessa etapa, cada nó sabe sua posição e seu tamanho

Introdução

Layout

- Passos do algoritmo
    1. Medir filhos: um nó mede os filhos, se houver
    2. Decidir o próprio tamanho: com base nisso, um nó decide o próprio tamanho
    3. Posicionar filhos: cada nó filho é posicionado em relação à posição do nó
- Resumidamente, os pais são medidos antes dos filhos, porém são dimensionados e posicionados depois deles
- Ao final dessa etapa cada nó possui:
    - Uma largura e uma altura definidas
    - Uma coordenada x, y que corresponde ao posição onde o nó será renderizados

Layouts básicos

A fase de layout

```
Row {
  Image(
  // ...
  )
  Column {
    Text(
    // ...
    )
    Text(
      // ...
    )
  }
}
```

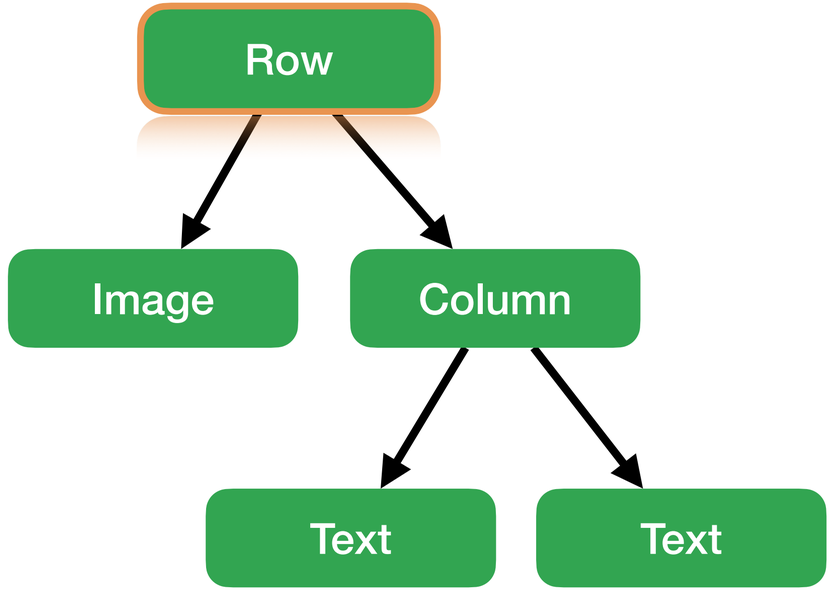
1 Qual o seu tamanho? 10 Tamanho calculado Coordenadas

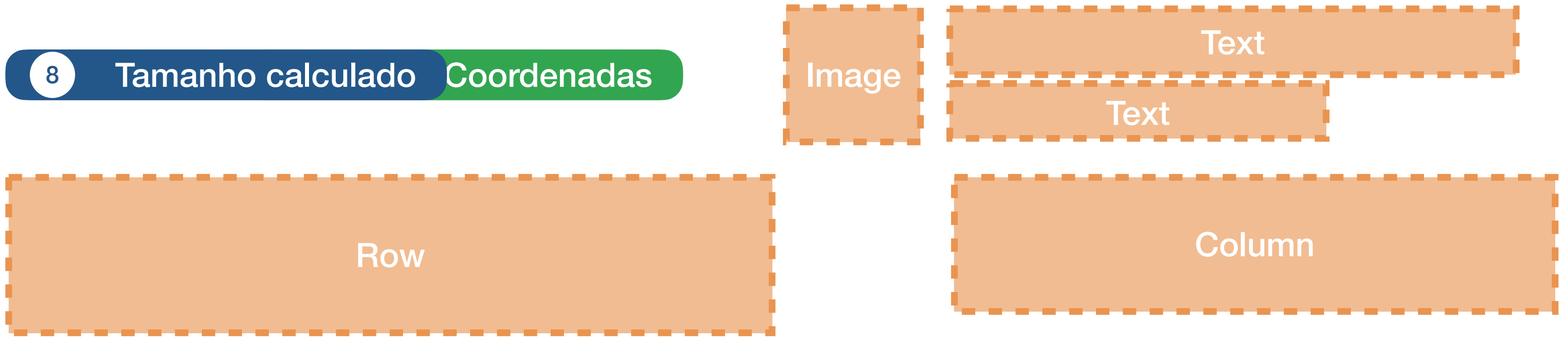
2 Qual o seu tamanho? 3 Tamanho calculado Coordenadas

4 Qual o seu tamanho? 9 Tamanho calculado Coordenadas

5 Qual o seu tamanho? 6 Tamanho calculado Coordenadas

7 Qual o seu tamanho? 8 Tamanho calculado Coordenadas

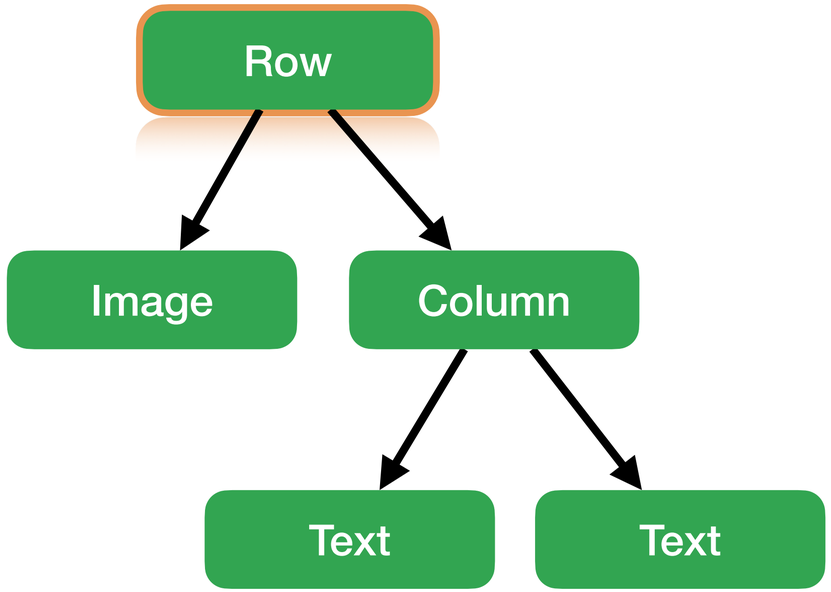




Introdução

Drawing

- Nesta fase, cada nó é visitado mais uma vez
  - Cada nó sabe onde deve ser desenhado



Modifier

Modifier

- Permitem decorar/customizar funções @Composable
  - Mudar o tamanho, o layout, o comportamento e a aparência do elemento
  - Adicionar informações de acessibilidade
  - Processar a entrada do usuário
  - Tornar um elemento clicável, rolável, arrastável ou redimensionável

Modifier

- Modifiers são objetos “normais”
  - São criado por meios de extension functions da Interface Modifier
  - Podem ser encadeados

```
@Composable
private fun Greeting(name: String) {
    Column(
        modifier = Modifier
            .padding(24.dp)
            .fillMaxWidth()
    ) { }
}
```

Todos as funções @Composable devem aceitar um parâmetro modifier e passá-lo ao primeiro filho. Torna seu código mais reutilizável e o comportamento mais previsível e intuitivo

Boa prática

Modifier

Boa prática

```
@Composable
fun FancyButton(
    text: String,
    onClick: () -> Unit,
    modifier: Modifier = Modifier
) = Text(
    text = text,
    modifier = modifier.surface(elevation = 4.dp)
        .clickable(onClick)
        .padding(horizontal = 32.dp, vertical = 16.dp)
)
```

Modifier

A ordem importa

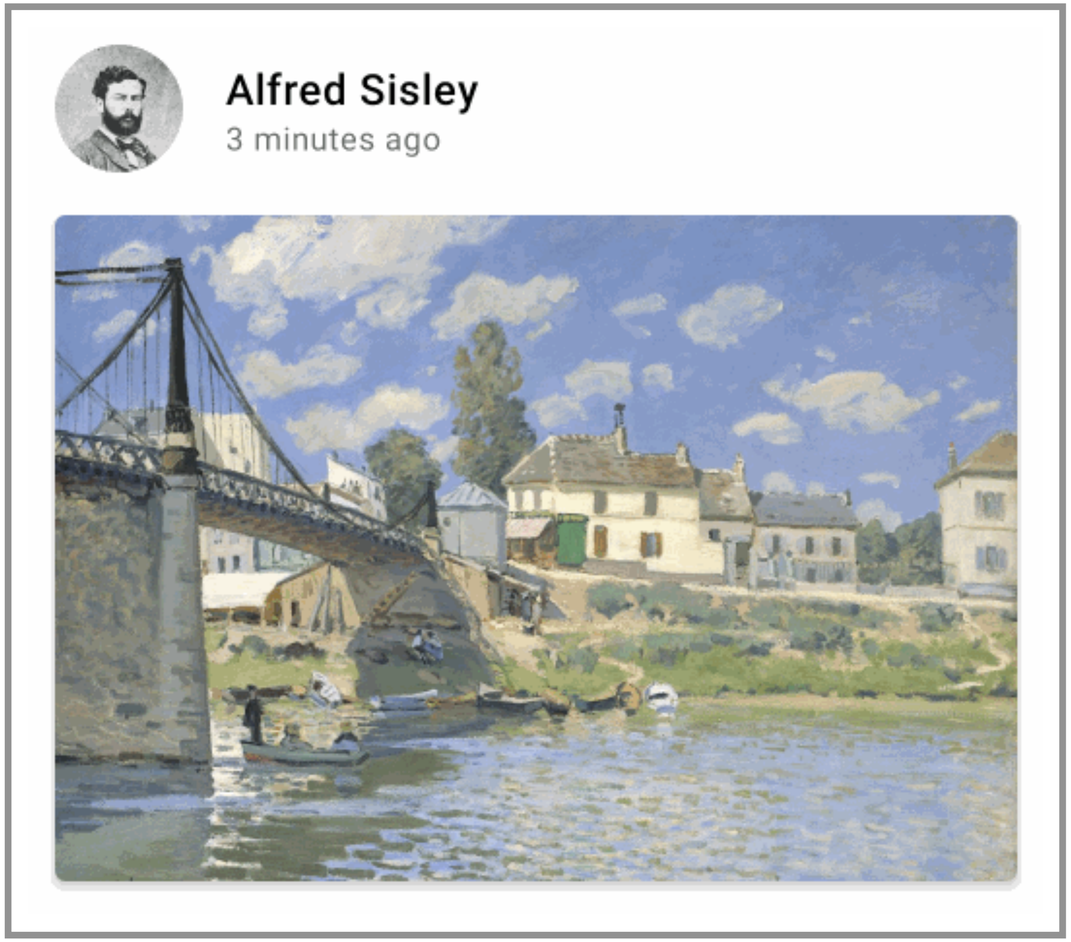
- A ordem dos modifiers é importante
- Cada função realiza mudanças no Modifier retornado pela função anterior

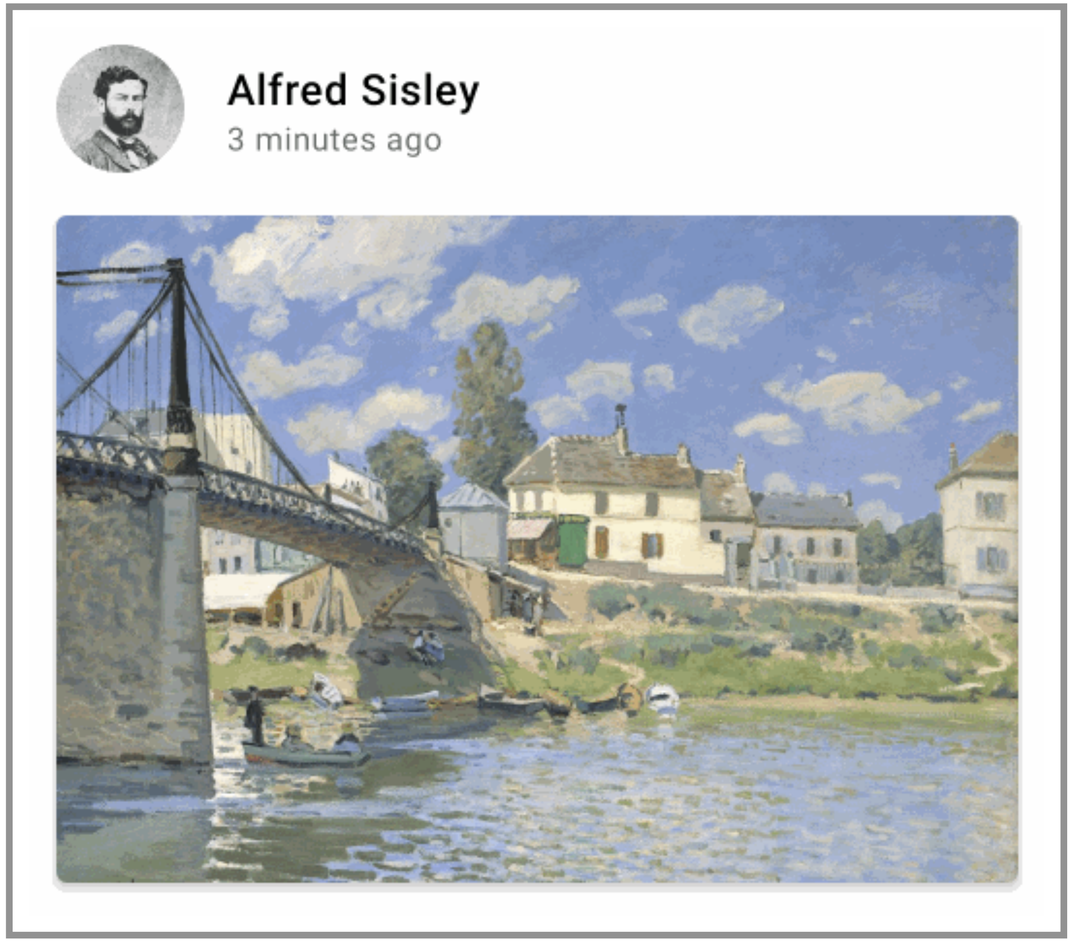
```
@Composable
fun ArtistCard(/*...*/) {
    val padding = 16.dp
    Column(
        Modifier
            .clickable(onClick = onClick)
            .padding(padding)
            .fillMaxWidth()
    ) {
        // rest of the implementation
    }
}
```

```
@Composable
fun ArtistCard(/*...*/) {
    val padding = 16.dp
    Column(
        Modifier
            .padding(padding)
            .clickable(onClick = onClick)
            .fillMaxWidth()
    ) {
        // rest of the implementation
    }
}
```

Modifier

A ordem importa





Modifier

Modificadores prontos

- O Jetpack Compose oferece vários modificadores para serem utilizados
  - Confira a lista completa
- A seguir veremos alguns dos mais utilizados para ajustar layouts

Modifier

size

- Por padrão os layouts englobam os seus elementos filhos
- Consequentemente, o tamanho do layout depende dos seus filhos
- Contudo, podemos definir o tamanho de um elemento

```
@Composable
fun ArtistCard(/*...*/) {
    Row(
        modifier = Modifier.size(width = 400.dp, height = 100.dp)
    ) {
        Image(/*...*/)
        Column { /*...*/ }
    }
}
```

Modifier

padding

```
@Composable
fun ArtistCard(/*...*/) {
    Row(
        modifier = Modifier.size(width = 400.dp, height = 100.dp)
    ) {
        Image(/*...*/)
         Column {
            Text(
                text = artist.name,
                modifier = Modifier.padding(top = 10.dp)
            )
            Text(artist.lastSeenOnline)
        }
    }
}
```

Modifier

offset

- Usado para posicionar um layout em relação à posição original
  - Permite o deslocamento no eixo x e y
  - O deslocamento pode ser positivo ou negativo
- A diferença entre padding e offset é que ao adicionar um offset a um elemento sua dimensão não é alterada
- Importante: O offset aplicado horizontalmente é sensível ao contexto
  - Alternativa: absoluteOffset

Modifier

padding

```
@Composable
fun ArtistCard(artist: Artist) {
    Row(/*...*/) {
        Column {
            Text(artist.name)
            Text(
                text = artist.lastSeenOnline,
                modifier = Modifier.offset(x = 4.dp)
            )
        }
    }
}
```

Modifier

Escopo

- Há modifiers que só podem ser usados quando aplicados a filhos de determinadas funções composable
- Ex:
  - matchParentSize disponível apenas no BoxScope
  - weight disponível apenas no ColumnScope e RowScope

Modifier

matchParentSize

- Usado para que um elemento filho tenha o mesmo tamanho de uma Box mãe sem afetar o tamanho da Box
- Disponível no escopo de Box, ele se aplica apenas a filhos das funções Box

```
@Composable
fun MatchParentSizeComposable() {
    Box {
        Spacer(
            Modifier
                .fillMaxSize()
                .background(Color.LightGray)
        )
        ArtistCard()
    }
}
```

Modifier

weight

- Em geral, o tamanho de um elemento é definido pelo seu conteúdo
- Podemos flexibilizar o cálculo do tamanho com o modifier weight

```
@Composable
fun ArtistCard(/*...*/) {
    Row(
        modifier = Modifier.fillMaxWidth()
    ) {
        Image(
            /*...*/
            modifier = Modifier.weight(2f)
        )
        Column(
            modifier = Modifier.weight(1f)
        ) {
            /*...*/
        }
    }
}
```

Modifier

Reuso

- Às vezes, pode ser vantajoso reutilizar as mesmas instâncias modifiers
- Pode melhorar a legibilidade do código
- Pode ajudar a melhorar a performance do app por alguns motivos:
  - A nova alocação dos modificadores não acontece durante a *recomposition*
  - Modifiers encadeados podem ser complexos, portanto, ao reutilizar a mesma instância podemos diminuir a carga de trabalho necessária para compará-los
- Promove a legibilidade, a consistência e a manutenção do código

Modifier

Extraindo e reusando modifers sem escopo

- Modifiers sem escopo podem ser extraídos para fora de funções composable

```
val myButtonModifier = Modifier
    .padding(8.dp)
    .width(200.dp)
    .background(Color.Gray, RoundedCornerShape(20.dp))

@Composable
fun MyColumn() {
    Column {
        MyButton("meu primeiro botão”, myButtonModifier)
        MyButton("meu segundo botão", myButtonModifier)
        Box(modifier = myButtonModifier.height(200.dp))
    }
}
```

Modifier

Extraindo e reusando modifers sem escopo

```
val myButtonModifier = Modifier
    .padding(8.dp)
    .fillMaxWidth()
    .background(Color.Gray, RoundedCornerShape(20.dp))

@Composable
fun MyLazyColumn() {
    LazyColumn {
        items(1000) {
            MyButton(
                "meu segundo botão",
                myButtonModifier
            )
        }
    }
}
```

Modifier

Extraindo e reusando modifers com escopo

- Modifiers sem escopo podem ser extraídos para fora de funções composable

```
@Composable
fun MyColumn() {
    Column {
        val myButtonModifier = Modifier
            .padding(8.dp)
            .width(200.dp)
            .background(Color.Gray, RoundedCornerShape(20.dp))
            .align(CenterHorizontally)
        MyButton("meu primeiro botão”, myButtonModifier)
        MyButton("meu segundo botão", myButtonModifier)
        Box(modifier = myButtonColumnModifier.height(200.dp))
    }
}
```

Modifier

reuso

```
val reusableModifier = Modifier
    .fillMaxWidth()
    .background(Color.Red)
    .padding(12.dp)

// Append to your reusableModifier
reusableModifier.clickable { /*...*/ }

// Append your reusableModifier
otherModifier.then(reusableModifier)
```

Restrições

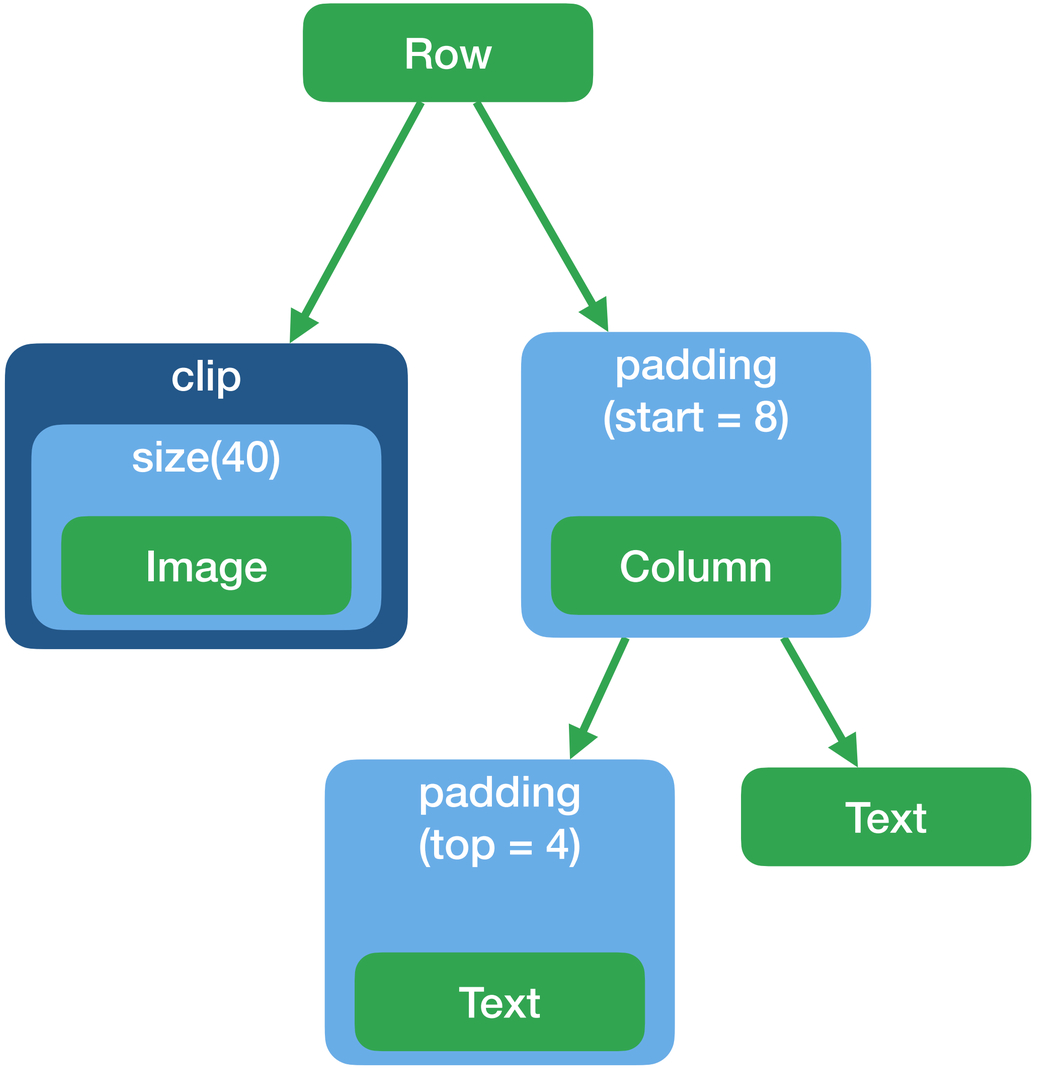
Restrições

- Modifiers podem impor restrições a funções @Composables
- Podemos pensar em modifiers como nós que encapsulam outros os nós
- Na fase de layout, o algoritmo que percorre a árvore permanece o mesmo, mas cada nó modificador também é visitado
- Assim, um modifier pode mudar os requisitos de tamanho e posicionamento de um modifier ou do nó ele envolve

Restrições

A fase de layout

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



Restrições

Restrições na fase de layout

- Constraints (restrições) ajudam a encontrar os tamanhos certos para os nós durante as duas primeiras etapas do algoritmo da fase de layout
  - Definem os limites mínimos e máximos para largura e altura
  - Quando um nó decide o seu tamanho, eles devem obedecer esses limites

Restrições

Tipos de restrição

- Uma restrição pode ser de um dos seguintes tipos
- Delimitado:
  - O nó tem largura e uma altura máximas e mínimas
- Ilimitado:
  - O nó não possui nenhuma restrição de tamanho (infinito)
- Exata:
  - O nó precisa ter um tamanho exato
- Combinado:
  - O nó combina os tipos anteriores

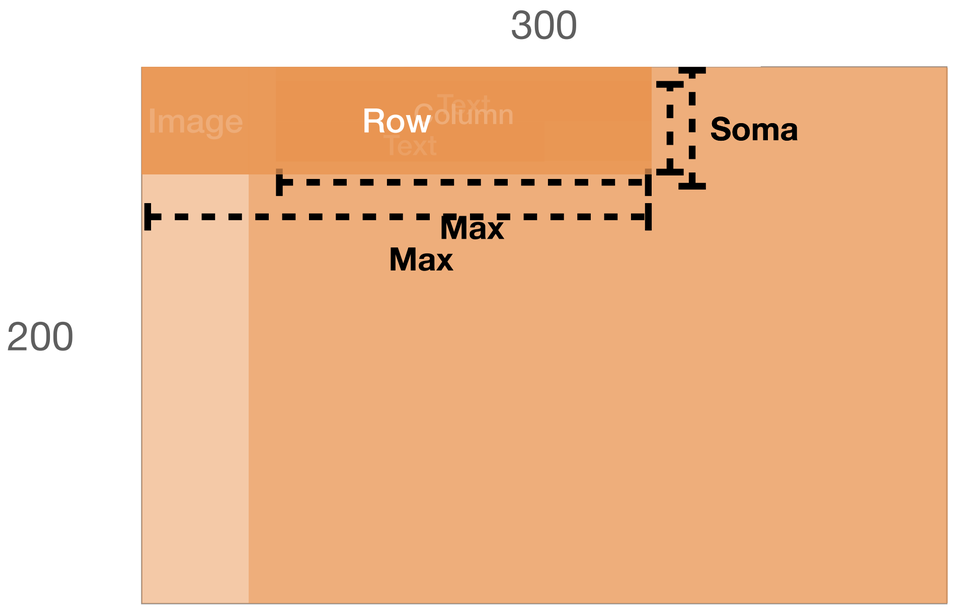
Restrições

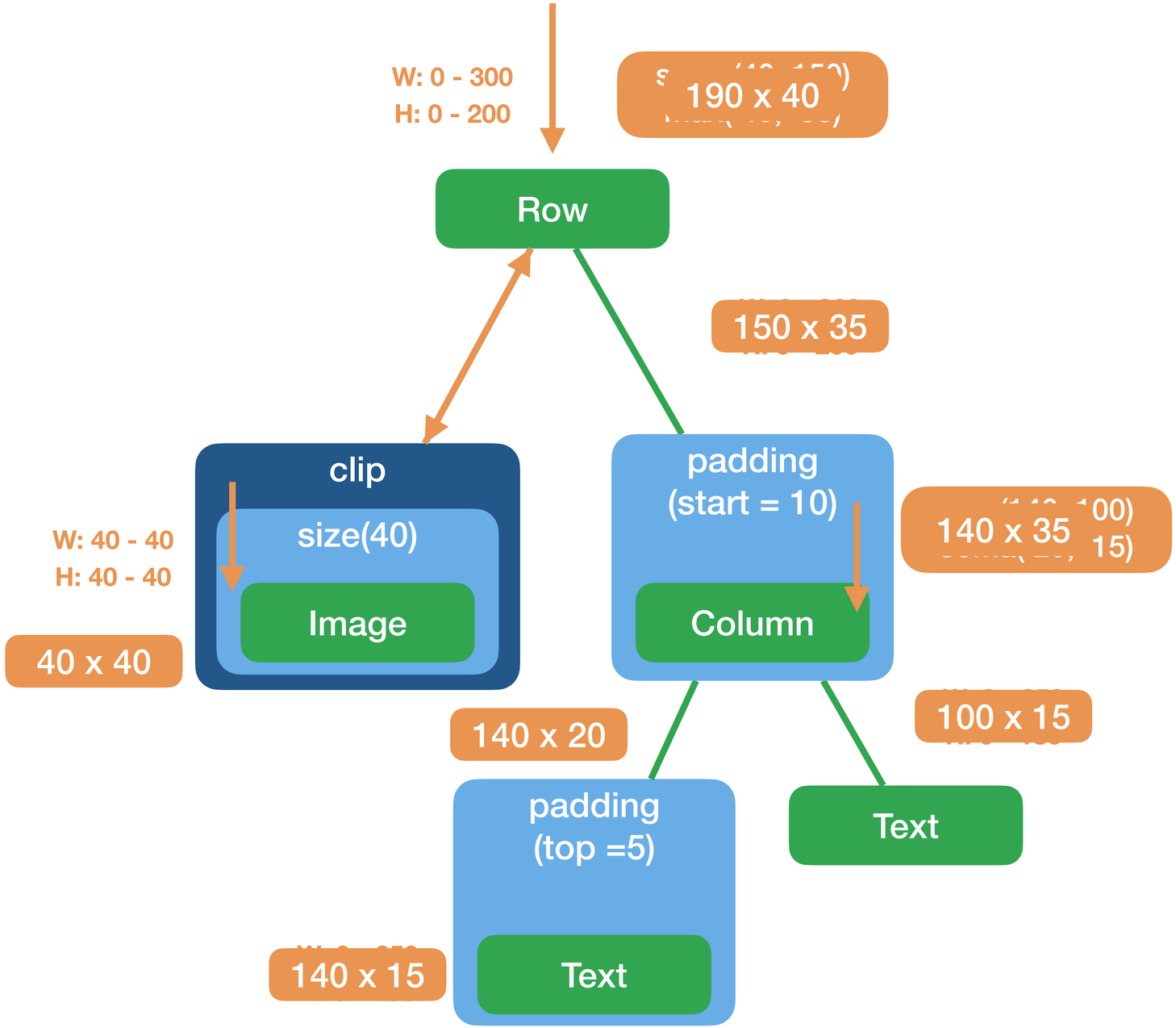
Transmitindo restrições

1. Para decidir o tamanho que quer ocupar, a raiz da árvore mede os filhos e encaminha as restrições ao primeiro filho
2. Se o filho for um modifier que não afeta as restrições, ela as encaminhará para o próximo modificador
    - As restrições são transmitidas pela cadeia de modifiers sem alterações. A menos que um modifier que afete a medição seja alcançado
3. Quando uma folha é alcançada, ela decide seu tamanho com base nas restrições e o informa
4. O nó pai ajusta as restrições com base nos seus do filhos e chama o próximo filho com as novas restrições
5. Após todos filhos de um pai serem medidos, o nó pai decide o seu tamanho e comunica ao pai dele
6. Toda a árvore é percorrida em profundidade. Eventualmente, todos os nós decidem os tamanhos

Restrições

Transmitindo restrições

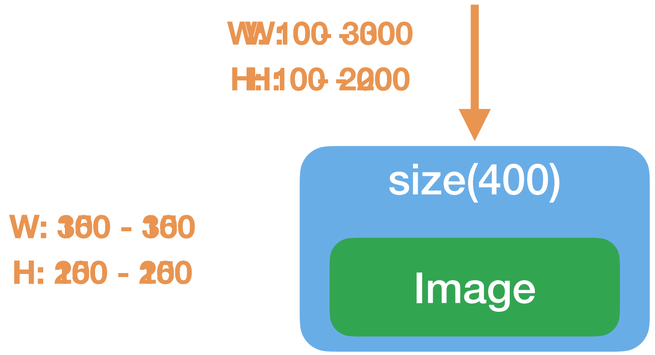




Restrições

Modificadores que afetam as restrições

- size: Adapta as restrições ao valor informado nele

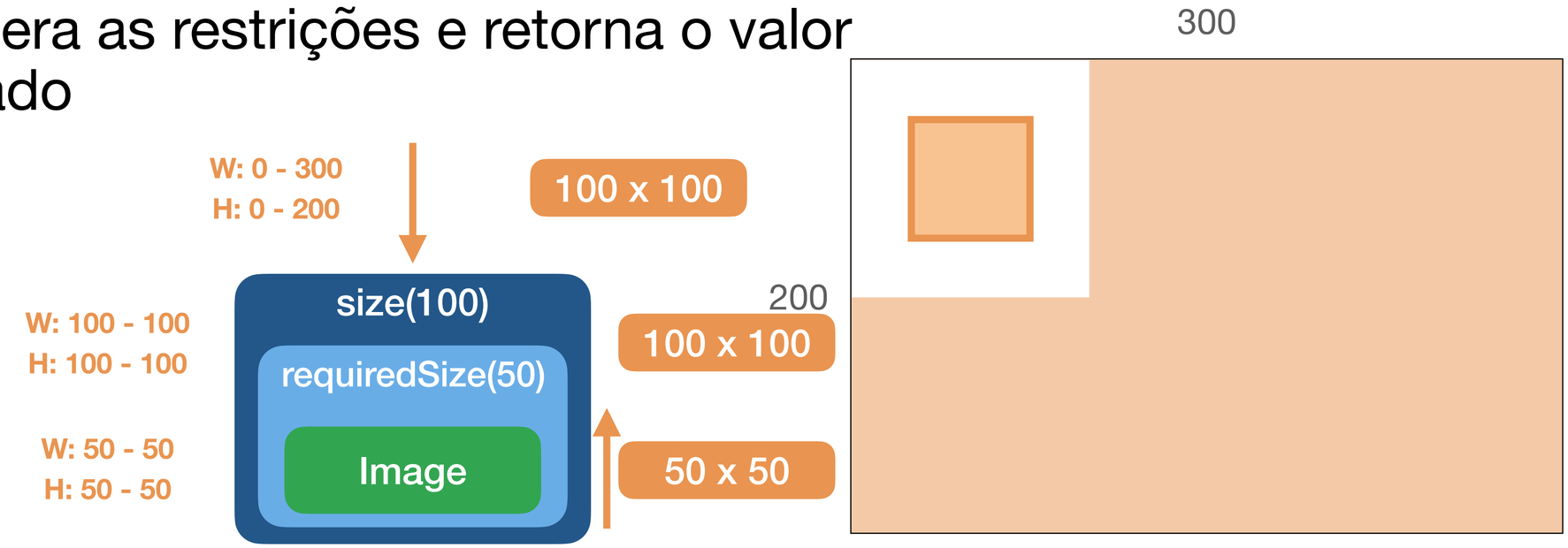


Ao usar o size, os valores mínimos e máximos das restrições sempre serão respeitados

Restrições

Modificadores que afetam as restrições

- requiredSize: altera as restrições e retorna o valor exato especificado



Restrições

Modificadores que afetam as restrições

- Com o modifier width, é possível definir uma largura fixa, e deixar a altura indefinida
- O modificar height funciona de forma análoga
- O modificador sizeIn permite definir restrições mínimas e máximas exatas para largura e altura

Um pouco mais de Kotlin

Um pouco mais de Kotlin

If expressions

- Em Kotlin, if é uma expressão, não apenas uma estrutura condicional
  - Isso significa que ele retorna um valor
- Pode ser usado diretamente em atribuições ou retornos de função

```
val a = 10
val b = 20

val maior = if (a > b) a else b
println(maior) // 20
```

Um pouco mais de Kotlin

If expressions

- Pode ser usado com blocos

```
val resultado = if (a > b) {
    println("A é maior")
    a
} else {
    println("B é maior")
    b
}
```

Um pouco mais de Kotlin

If expressions

- Sempre retorna o último valor avaliado
- Evita o operador ternário de outras linguagens

```
function calc(num1, num2, operator) {
  return operator === '+'
    ? num1 + num2
    : operator === '-'
      ? num1 - num2
      : operator === '*'
        ? num1 * num2
        : operator === '/'
          ? num1 / num2
          : 'Invalid operator';
}
```

```
fun calc(num1: Double, num2: Double, operator:
    return if (operator == "+") {
        num1 + num2
    } else if (operator == "-") {
        num1 - num2
    } else if (operator == "*") {
        num1 * num2
    } else if (operator == "/") {
        num1 / num2
    } else {
        "Invalid operator"
    }
}
```

Um pouco mais de Kotlin

When expressions

- when é o equivalente aprimorado do switch do Java
- Também é uma expressão, ou seja, retorna um valor
- Mais flexível: aceita qualquer tipo e condições complexas

```
val dia = 3
val nome = when (dia) {
    1 -> "Domingo"
    2 -> "Segunda"
    3 -> "Terça"
    else -> "Dia inválido"
}
println(nome) // Terça
```

Um pouco mais de Kotlin

When expressions

- Pode ser usado para verificar tipos

```
when (valor) {
    is String -> println("É uma string")
    is Int -> println("É um inteiro")
}
```

Um pouco mais de Kotlin

When expressions

- Múltiplos casos por linha

```
val tipo = when (dia) {
    1, 7 -> "Fim de semana"
    in 2..6 -> "Dia útil"
    else -> "Inválido"
}
```

Um pouco mais de Kotlin

When expressions

- Pode ser usado com expressões

```
val storedPin = "1234"
val enteredPin = 1234
when (enteredPin) {
    // Expression
    storedPin.toInt() -> print("PIN is correct")
    else -> print("Incorrect PIN")
}
```

O resultado da expressão é comparado com a variável passada como parâmetro

Um pouco mais de Kotlin

When expressions

- Pode ser usado sem argumento
  - Equivalente a usar if/else

```
val x = 10
val resultado = when {
    x % 2 == 0 -> "Par"
    x % 2 != 0 -> "Ímpar"
    else -> "Indefinido"
}
```

Um pouco mais de Kotlin

When expressions

- Pode ser usado como comando (sem retorno) ou expressão (retorna algo)
- Quando usado como expressão é necessário cobrir todos os casos

```
val text = when (x) {
    1 -> "x == 1"
    2 -> "x == 2"
    else -> "x is neither 1 nor 2"
}
```

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> print("x is neither 1 nor 2")
}
```

Opcional

Um pouco mais de Kotlin

When expressions

```
fun calc(num1: Double, num2: Double,
operator: String): Double{
    return if (operator == "+") {
        num1 + num2
    } else if (operator == "-") {
        num1 - num2
    } else if (operator == "*") {
        num1 * num2
    } else if (operator == "/") {
        num1 / num2
    } else {
        "Invalid operator"
    }
}
```

```
fun calculate(num1: Double, num2: Double,
operator: String): Double {
    return when (operator) {
        "+" -> num1 + num2
        "-" -> num1 - num2
        "*" -> num1 * num2
        "/" -> num1 / num2
        else -> "Invalid operator"
    }
}
```

Um pouco mais de Kotlin

Null Safety

- Em Java, qualquer variável pode ser null
  - Isso leva ao erro mais temido: NullPointerException (NPE)
- Kotlin foi criado para eliminar o NPE sempre que possível
- Oferece suporte explícito à nulidade como parte de seu sistema de tipos
  - Por padrão, nenhuma variável pode ser nula
  - É possível declarar explicitamente aquelas podem ser nulas
    - Para isso, é preciso usar ? no tipo

Um pouco mais de Kotlin

Null Safety

```
var nome: String = "Kotlin"
nome = null // ❌ Erro de compilação

var apelido: String? = "K"
apelido = null // ✅ Ok
```

Um pouco mais de Kotlin

O operador seguro e operador Elvis

- O operador ?. só acessa o membro se o valor não for nulo
- Caso contrário, o resultado da expressão é null
- O operador Elvis ?: define o valor padrão em caso de nulo

```
val nome: String? = null
var tamanho = nome?.length
println(tamanho) // null

tamanho = nome?.length ?: 0
println(tamanho) // 0
```

```
val nome: String? = null
val tamanho = if (nome != null) {
  nome.length
} else {
  0
}

println(tamanho) // 0
```

Um pouco mais de Kotlin

Smart cast

- Kotlin detecta quando um valor não pode mais ser null dentro de um bloco

```
val nome: String? = "Kotlin"

if (nome != null) {
    println(nome.length) // Smart cast: String, não String?
}
```

Bibliografia

- Boas práticas com Modifiers no Jetpack Compose
- Conditions and loops
- Null Safety

Por hoje é só