

Desenvolvimento para Dispositivos Móveis

Fundamentos dos Layouts

Prof. Bruno Azevedo

UNIP – Universidade Paulista



Layout



Column



Row



Box

Column

- O composable `Column` organiza seus filhos em uma coluna vertical.
- Cada elemento é exibido um abaixo do outro, na ordem em que aparece no código.
- Pode-se controlar alinhamentos e espaçamentos com parâmetros.

```
Column {  
    Text("Texto 1")  
    Text("Texto 2")  
    Text("Texto 3")  
}
```

- Os textos aparecerão empilhados verticalmente.
- A ordem no código define a ordem visual na tela.

Row

- O composable Row organiza seus filhos horizontalmente.
- Cada elemento é exibido lado a lado, da esquerda para a direita.

```
Row {  
    Text("Esquerda")  
    Text("Centro")  
    Text("Direita")  
}
```

- Os textos serão colocados em uma linha, na ordem do código.
- É possível adicionar espaçamento e alinhamento.

Column e Row – Personalização com Modifier

```
Column(  
    modifier = Modifier.fillMaxSize(),  
    verticalArrangement = Arrangement.SpaceEvenly,  
    horizontalAlignment = Alignment.CenterHorizontally  
) {  
    Text("Topo")  
    Text("Centro")  
    Text("Base")  
}
```

- `fillMaxSize()` faz a `Column` ocupar todo o espaço disponível.
- `Arrangement.SpaceEvenly` distribui os elementos com espaçamento igual.
- `Alignment.CenterHorizontally` centraliza horizontalmente os elementos.

Box

- O composable Box permite empilhar elementos visuais, um sobre o outro.
- Útil para criar layouts com sobreposição, como texto sobre imagem ou ícones sobre botões.
- A ordem dos elementos no código determina a ordem de desenho (o último fica por cima).
- Podemos controlar o posicionamento de cada elemento com `Modifier.align()`.
- O modificador `Modifier.size()` define o tamanho da área da Box.

```
Box(modifier = Modifier.size(200.dp)) {  
    Text("Fundo", modifier = Modifier.align(Alignment.Center))  
    Text("Topo", modifier = Modifier.align(Alignment.TopEnd))  
}
```

- A Box terá 200x200 dp de tamanho.
- O texto “Fundo” será centralizado.
- O texto “Topo” será desenhado no canto superior direito, sobre o anterior.

LazyColumn e LazyRow

- LazyColumn e LazyRow são usadas para listas com um grande número de itens, otimizando a performance.
- **A renderização dos itens só ocorre conforme o usuário rola a lista.**
- Exemplo de uso do LazyColumn:

```
LazyColumn {  
    // Um único item  
    item {  
        Text(text = "First item")  
    }  
    // Cinco items. Cria uma lista de 5 elementos numerados de 0 a 4.  
    // Dentro do bloco, mostra o texto correspondente a cada índice.  
    items(5) { index ->  
        Text(text = "Item: $index")  
    }  
    // Outro item único  
    item {  
        Text(text = "Last item")  
    }  
}
```

- A **LazyRow** funciona da mesma forma, mas com uma rolagem horizontal.

Tipos anuláveis e Não-anuláveis

- Em Kotlin, qualquer tipo pode ser tornado **anulável** ao adicionar ? no final do tipo.
- Isso vale para tipos primitivos e tipos de objetos.
- Exemplos:

```
val nome: String = "Maria"    // Não pode ser null
val nome2: String? = null    // Pode ser null ou string

val idade: Int = 20           // Não pode ser null
val idade2: Int? = null       // Pode ser null ou inteiro

val nota: Double = 8.5        // Não pode ser null
val nota: Double? = 8.5       // Pode ser null ou real
```

- Sempre que usamos um tipo com ?, devemos tratar o caso de null antes de acessar o valor.
- Isso ajuda a evitar falhas como NullPointerException.

Tipos anuláveis e Não-anuláveis

- Considere o tipo `Double`, que representa um número de ponto flutuante. Este **nunca pode ser nulo**.
- Já o tipo `Double?` representa um número que **pode ser nulo**.
- Isso é útil para representar valores que podem ou não ter sido fornecidos pelo usuário.
- Exemplos:

```
// Tipo não-nulo:  
val x: Double = 5.0      // OK  
val y: Double = null     // ERRO: Double não aceita null  
  
// Tipo anulável:  
val a: Double? = 5.0     // OK  
val b: Double? = null    // OK
```

- O uso de tipos anuláveis exige tratamento com `if`, `?.`, `?:`, etc., para evitar falhas em tempo de execução.

Tratamento de Entradas Nulas em Kotlin

- Em aplicações Android, ao obter entradas do usuário, é comum precisar converter texto para número.
- Se o texto for inválido, o método `toDouble()` lançará uma exceção.
- Para evitar isso, usamos `toDoubleOrNull()`, que retorna `null` em caso de erro.
- Exemplo:

```
val numero = texto.toDoubleOrNull()
```

- A variável `numero` será do tipo `Double?`.
- Precisamos então tratar o caso em que o valor seja `null`.

Operador Elvis (?:)

- O operador `?:` permite definir um valor padrão caso a expressão à esquerda seja `null`.
- Exemplo:

```
val numero = texto.toDoubleOrNull() ?: 0.0
```

- Se `toDoubleOrNull()` retornar `null`, será usado `0.0`.
- Esse recurso é útil para evitar falhas de execução causadas por valores nulos.

Conversão segura com `toDoubleOrNull()` e operador Elvis

- `toDoubleOrNull()` tenta converter uma `String` para `Double`, retornando `null` se falhar.
- O operador Elvis (`?:`) define um valor padrão caso o resultado seja `null`:

```
fun converterParaDouble(campoTexto: String): Double {  
    val numero = campoTexto.toDoubleOrNull() ?: 0.0  
    return numero  
}
```

- Exemplos:

```
println(converterParaDouble("3.14"))  
println(converterParaDouble("abc"))
```

- Saída:

```
3.14  
0.0
```

Conversão segura com `toIntOrNull()` e operador Elvis

- `toIntOrNull()` tenta converter uma `String` para `Int`, retornando `null` se falhar.
- O operador Elvis (`?:`) define um valor padrão caso o resultado seja `null`:

```
fun converterTexto(campoTexto: String): Int {  
    val num1 = campoTexto.toIntOrNull() ?: 0  
    return num1  
}
```

- Exemplos:

```
println(converterTexto("123"))  
println(converterTexto("abc"))
```

- Saída:

```
123  
0
```

Tratando Divisão por Zero

- Mesmo com conversão correta, a operação de divisão pode gerar erro se o denominador for zero.
- Para evitar isso, fazemos uma verificação antes da divisão.
- Exemplo:

```
if (denominador != 0.0) {  
    val resultado = numerador / denominador  
} else {  
    //Divisão por zero, tratar erro ou mostrar mensagem  
}
```

- Nunca devemos assumir que o valor digitado é sempre válido.

Atividade 7

- Façam a atividade 7.