



UD4.2 – Componentes Jetpack Compose

2º CFGS
Desarrollo de Aplicaciones Multiplataforma
2023-24

1.- Composición y recomposición

La **composición** y la **recomposición** son un comportamiento habitual en las interfaces de usuario declarativas.

- **Composición:** ejecución por primera vez de una función de composición (@Composable) → pintar un componente en la interfaz.
- **Recomposición:** ejecución por segunda o más vez de una función de composición (@Composable) → repintar la interfaz pintando solo los componentes afectados.

Si en tiempo de ejecución se modifica un componente, ese componente se recompone.

De esta manera la interfaz de usuario siempre tiene la última versión del componente.

2.- Estados

En Jetpack Compose un **estado** es una variable conectada al estado de la aplicación.

Si durante la ejecución de la aplicación un estado (variable conectada al estado de la aplicación) cambia su valor, las partes de la interfaz afectadas se recomponen (se vuelven a pintar).

En el ejemplo de la UD3 "Contador de clics" se utilizó un estado para recomponer la interfaz al pulsar un botón.

```
@Composable
fun Content() {
    var times by rememberSaveable { mutableStateOf( value: 0) }
    Column(
```

2.- Estados

```
@Composable
fun Content() {
    var times by rememberSaveable { mutableStateOf( value: 0) }
    Column()
```

mutableStateOf: indica que la variable es un estado (variable conectada al estado de la aplicación). Si esta variable cambia su valor se deben recomponer las funciones @Composable que la utilicen.

El problema es que cuando se vuelve a ejecutar la función @Composable el estado volverá a tener el valor inicial.

Para solucionar esto se usa:

- **remember**: indica a Android que una variable se **recuerde en la recomposición**.
- **rememberSaveable**: lo anterior y también que se **recuerde al destruir y crear la Activity**.

2.- Estados

Los estados se pueden crear con `=` pero esto conlleva que para acceder al valor del estado se debe usar **.value**.

```
val quantity = rememberSaveable{ mutableStateOf( value: 0) }  
quantity.value++
```

Para **facilitar el uso de estados** se creo el delegado **by**, visto en los ejemplos, que permite acceder al valor del estado con el propio nombre de la variable.

```
var quantity by rememberSaveable{ mutableStateOf( value: 0) }  
quantity++
```

2.- Estados

Cuando se explicó el parámetro **modifier** se indicó que si el contenido de un componente no cabe en la pantalla se pueden usar los modificadores **verticalScroll** y **horizontalScroll** para deslizar y poder alcanzar todo el contenido.

Para esto se hizo uso de **rememberScrollState**.

Esta función crea un estado con **rememberSaveable** por lo que el estado del scroll se recordará tanto al cambiar la orientación como al cambiar la configuración:

```
    porque es el auténtico guardián de su hermano y el  
    | descubridor de los niños perdidos. ..." """.trimMargin(),  
    modifier = Modifier  
        .height(100.dp)  
        .width(200.dp)  
        .verticalScroll(rememberScrollState())  
        .horizontalScroll(rememberScrollState())  
    )
```

```
@Composable  
fun rememberScrollState(initial: Int = 0): ScrollState {  
    return rememberSaveable(saver = ScrollState.Saver) {  
        ScrollState(initial = initial)  
    }  
}
```

2.- Estados

Gracias a la recomposición es muy sencillo cambiar la apariencia de la interfaz gráfica:

```
var quantity by rememberSaveable{ mutableStateOf( value: 0) }
var addButtonEnabled by rememberSaveable{ mutableStateOf( value: true) }
var deleteButtonEnabled by rememberSaveable{ mutableStateOf( value: false) }

Text( text: "Añadidos: $quantity")
Row { this: RowScope
    Button(
        enabled = addButtonEnabled,
        onClick = {
            if (++quantity == 5) addButtonEnabled = false
            deleteButtonEnabled = true
        }
    ) { this: RowScope
        Text( text: "Añadir")
    }
    Spacer(Modifier.width(8.dp))
    if (deleteButtonEnabled) {
        Button(
            enabled = deleteButtonEnabled,
            onClick = {
                quantity = 0
                addButtonEnabled = true
                deleteButtonEnabled = false
            }
        ) { this: RowScope
            Icon(
                imageVector = Icons.Default.DeleteOutline,
                contentDescription = "Borrar"
            )
        }
    }
}
Text( text: "Máximo 5 productos")
```



2.- Estados

Otro ejemplo:

```
val colors = arrayListOf<Color>(Color.Red, Color.Green, Color.Blue, Color.DarkGray)
var buttonColor by rememberSaveable{ mutableStateOf((0 ≤ until < colors.size).random()) }
Button(
    onClick = {
        buttonColor = (0 ≤ .. ≤ 2).random()
    },
    colors = ButtonDefaults.buttonColors(containerColor = colors[buttonColor])
) { this: RowScope
    Text( text: "Cambia el color")
}
```



Cambia el color

Cambia el color

Cambia el color

Cambia el color

3.- Componentes para introducir información

A continuación, se van a estudiar una serie de componentes Jetpack Compose que permiten **recoger información del usuario**.

Estos componentes se suelen utilizar en formularios o en pantallas de ajustes.

Como estos componentes cambian conforme el usuario introduce los datos o los selecciona, se deben usar estados para su correcto funcionamiento.

4.- Componentes TextField

Los componentes **TextField** permiten al usuario introducir datos desde el teclado del dispositivo.

Existen tres componentes TextField:

- **BasicTextField**: campo de texto básico.
- **TextField**: utiliza los principios de Material.
- **OutlinedTextField**: como TextField pero con un borde predefinido en el parámetro **shape**.

```
var textFieldValue by rememberSaveable { mutableStateOf( value: "") }  
TextField(  
    value = textFieldValue,  
    onChange = { textFieldValue = it }  
)
```

4.- Componentes TextField

Como se estudió anteriormente es importante conocer los parámetros que admiten los componentes.

Params:

- `value` - the input text to be shown in the text field
- `onValueChange` - the callback that is triggered when the input service updates the text. An updated text comes as a parameter of the callback
- `modifier` - the `Modifier` to be applied to this text field
- `enabled` - controls the enabled state of this text field. When `false`, this component will not respond to user input, and it will appear visually disabled and disabled to accessibility services.
- `readOnly` - controls the editable state of the text field. When `true`, the text field cannot be modified. However, a user can focus it and copy text from it. Read-only text fields are usually used to display pre-filled forms that a user cannot edit.
- `textStyle` - the style to be applied to the input text. Defaults to `LocalTextStyle`.
- `label` - the optional label to be displayed inside the text field container. The default text style for internal `Text` is `Typography.bodySmall` when the text field is in focus and `Typography.bodyLarge` when the text field is not in focus
- `placeholder` - the optional placeholder to be displayed when the text field is in focus and the input text is empty. The default text style for internal `Text` is `Typography.bodyLarge`
- `leadingIcon` - the optional leading icon to be displayed at the beginning of the text field container
- `trailingIcon` - the optional trailing icon to be displayed at the end of the text field container
- `prefix` - the optional prefix to be displayed before the input text in the text field
- `suffix` - the optional suffix to be displayed after the input text in the text field
- `supportingText` - the optional supporting text to be displayed below the text field
- `isError` - indicates if the text field's current value is in error. If set to `true`, the label, bottom indicator and trailing icon by default will be displayed in error color
- `visualTransformation` - transforms the visual representation of the input `value`. For example, you can use `PasswordVisualTransformation` to create a password text field. By default, no visual transformation is applied.
- `keyboardOptions` - software keyboard options that contains configuration such as `KeyboardType` and `ImeAction`.
- `keyboardActions` - when the input service emits an IME action, the corresponding callback is called. Note that this IME action may be different from what you specified in `KeyboardOptions.imeAction`.
- `singleLine` - when `true`, this text field becomes a single horizontally scrolling text field instead of wrapping onto multiple lines. The keyboard will be informed to not show the return key as the `ImeAction`. Note that `maxLines` parameter will be ignored as the `maxLines` attribute will be automatically set to 1.
- `maxLines` - the maximum height in terms of maximum number of visible lines. It is required that `1 <= minLines <= maxLines`. This parameter is ignored when `singleLine` is `true`.
- `minLines` - the minimum height in terms of minimum number of visible lines. It is required that `1 <= minLines <= maxLines`. This parameter is ignored when `singleLine` is `true`.
- `interactionSource` - the `MutableInteractionSource` representing the stream of `Interactions` for this text field. You can create and pass in your own remembered instance to observe `Interactions` and customize the appearance / behavior of this text field in different states.
- `shape` - defines the shape of this text field's container
- `colors` - `TextFieldColors` that will be used to resolve the colors used for this text field in different states. See `TextFieldDefaults.colors`.

```
fun TextField(  
    value: String,  
    onValueChange: (String) -> Unit,  
    modifier: Modifier = Modifier,  
    enabled: Boolean = true,  
    readOnly: Boolean = false,  
    textStyle: TextStyle = LocalTextStyle.current,  
    label: @Composable (() -> Unit)? = null,  
    placeholder: @Composable (() -> Unit)? = null,  
    leadingIcon: @Composable (() -> Unit)? = null,  
    trailingIcon: @Composable (() -> Unit)? = null,  
    prefix: @Composable (() -> Unit)? = null,  
    suffix: @Composable (() -> Unit)? = null,  
    supportingText: @Composable (() -> Unit)? = null,  
    isError: Boolean = false,  
    visualTransformation: VisualTransformation = VisualTransformation.None,  
    keyboardOptions: KeyboardOptions = KeyboardOptions.Default,  
    keyboardActions: KeyboardActions = KeyboardActions.Default,  
    singleLine: Boolean = false,  
    maxLines: Int = if (singleLine) 1 else Int.MAX_VALUE,  
    minLines: Int = 1,  
    interactionSource: MutableInteractionSource = remember { MutableInteractionSource() },  
    shape: Shape = TextFieldDefaults.shape,  
    colors: TextFieldColors = TextFieldDefaults.colors()  
) {
```

4.- Componentes TextField

Todos los parámetros que aceptan una función lambda permiten que se puedan realizar acciones.

Si además la función lambda es `@Composable`, dentro se podrán incorporar otros componentes.

```
value: String,  
onValueChange: (String) -> Unit,  
modifier: Modifier = Modifier,  
enabled: Boolean = true,  
readOnly: Boolean = false,  
textStyle: TextStyle = LocalTextStyle.current,  
label: @Composable (() -> Unit)? = null,  
placeholder: @Composable (() -> Unit)? = null,  
leadingIcon: @Composable (() -> Unit)? = null,  
trailingIcon: @Composable (() -> Unit)? = null,  
prefix: @Composable (() -> Unit)? = null,  
suffix: @Composable (() -> Unit)? = null,  
supportingText: @Composable (() -> Unit)? = null,  
isError: Boolean = false
```

4.- Componentes TextField

Ejemplo con dos campos, uno para el correo y otro para la contraseña:

```
var textFieldValue by rememberSaveable { mutableStateOf( value: "" ) }
TextField(
    value = textFieldValue,
    onChange = { textFieldValue = it },
    label = { Text( text: "Introduce tu email" ) },
    placeholder = { Text( text: "tudireccion@server.com" ) },
    leadingIcon = { Icon(imageVector = Icons.Default.Mail, contentDescription = null) },
    trailingIcon = { Icon(imageVector = Icons.Default.PriorityHigh, contentDescription = null) },
)

Spacer(modifier = Modifier.size(8.dp))

var outlinedTextFieldValue by rememberSaveable { mutableStateOf( value: "" ) }
OutlinedTextField(
    value = outlinedTextFieldValue,
    onChange = { outlinedTextFieldValue = it },
    label = { Text( text: "Introduce tu contraseña" ) },
    visualTransformation = PasswordVisualTransformation()
)
```

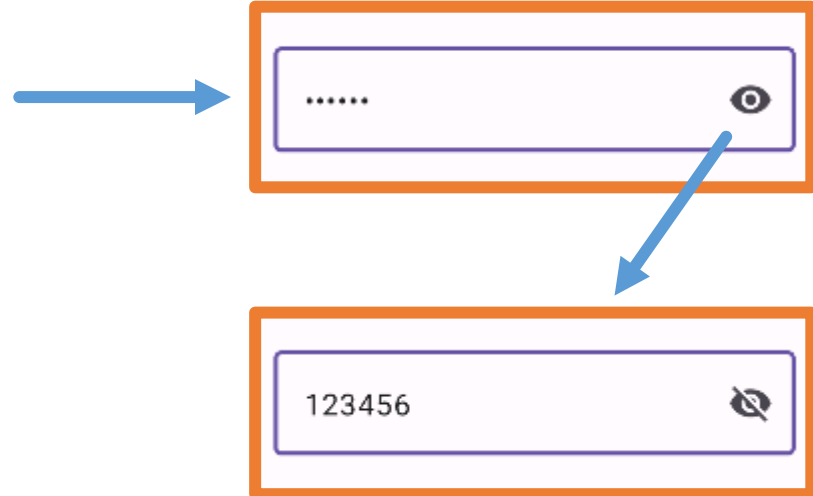


The image shows a visual representation of the login form generated by the code. It consists of two main input fields stacked vertically. The top field is for the email, with a light purple header containing the text 'Introduce tu email' and a dark purple placeholder text 'tudireccion@server.com'. It features a mail icon on the left and a warning icon on the right. The bottom field is for the password, with a light purple header containing the text 'Introduce tu contraseña' and a placeholder of six dots. It has a rounded rectangular border.

4.- Componentes TextField

Ejemplo de campo contraseña que permite ver el texto escrito:

```
var passFieldState by rememberSaveable { mutableStateOf( value: "" ) }
var passVisibleState by rememberSaveable { mutableStateOf( value: false ) }
OutlinedTextField(
    value = passFieldState,
    onValueChange = { passFieldState = it },
    trailingIcon = {
        val icon = if (passVisibleState) {
            Icons.Default.VisibilityOff
        } else {
            Icons.Default.Visibility
        }
        IconButton(onClick = { passVisibleState = !passVisibleState }) {
            Icon(
                imageVector = icon,
                contentDescription = "Ver contraseña"
            )
        }
    },
    visualTransformation = if (passVisibleState) {
        VisualTransformation.None
    } else {
        PasswordVisualTransformation()
    }
)
```

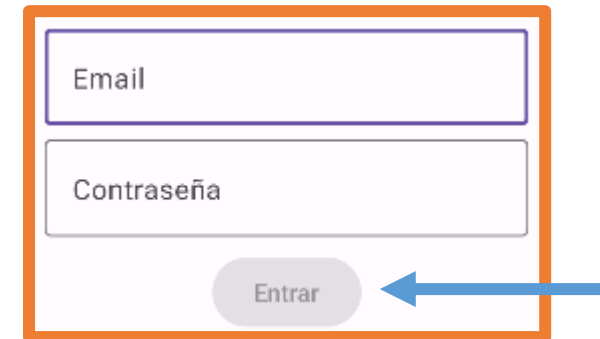


4.- Componentes TextField

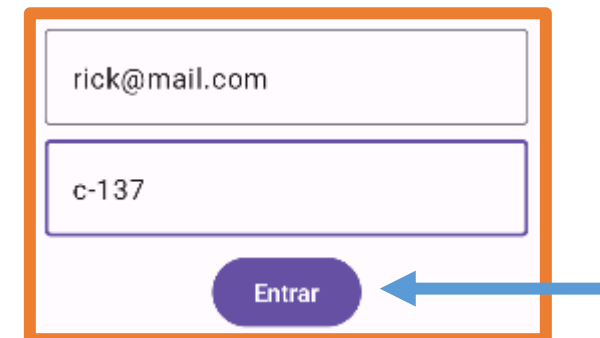
Ejemplo en el que el botón solo se activa si los campos tienen datos:

Gracias a los estados y la recomposición no es necesario un if para activar el botón:

```
var nameField by rememberSaveable { mutableStateOf( value: "" ) }
var passwordField by rememberSaveable { mutableStateOf( value: "" ) }
OutlinedTextField(
    value = nameField,
    onValueChange = { nameField = it },
    placeholder = { Text( text: "Email" ) }
)
Spacer(Modifier.size(8.dp))
OutlinedTextField(
    value = passwordField,
    onValueChange = { passwordField = it },
    placeholder = { Text( text: "Contraseña" ) }
)
Spacer(Modifier.size(8.dp))
Button(
    onClick = { /*TODO*/ },
    enabled = nameField.isNotEmpty() && passwordField.isNotEmpty()
) { this: RowScope
    Text( text: "Entrar" )
}
```



A diagram of a login form with two text input fields. The first field is labeled "Email" and the second is labeled "Contraseña". Below the fields is a button labeled "Entrar". A blue arrow points from the "Entrar" button to the code block on the left, indicating that the button's state is controlled by the code.



A diagram of a login form with two text input fields. The first field contains the text "rick@mail.com" and the second field contains the text "c-137". Below the fields is a button labeled "Entrar". A blue arrow points from the "Entrar" button to the code block on the left, indicating that the button's state is controlled by the code.

4.- Componentes TextField

El parámetro **keyboardOptions** permite indicar cómo será el teclado que se muestra.

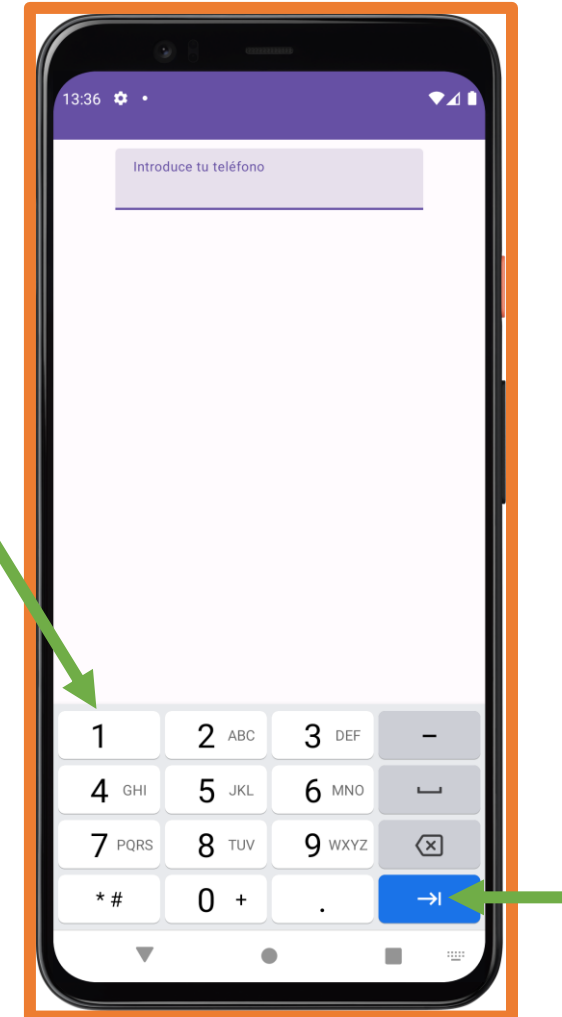
```
var textFieldValue by rememberSaveable { mutableStateOf( value: "" ) }  
TextField(  
    value = textFieldValue,  
    onValueChange = { textFieldValue = it },  
    label = { Text( text: "Introduce tu teléfono" ) },  
    keyboardOptions = KeyboardOptions(  
        keyboardType = KeyboardType.Phone,  
        imeAction = ImeAction.Next  
    ),  
    keyboardActions = KeyboardActions(  
        onNext = { this: KeyboardActionScope  
    }  
    )  
)
```

```
keyboardType = KeyboardType.|  
imeAction = ImeAction.Next  
boardActions = KeyboardAct  
onNext = { this: KeyboardAct  
}  
modifier = Modifier.size(8
```

- ☒ Email
- ☒ Decimal
- ☒ Password
- ☒ Text
- ☒ Ascii
- ☒ Number
- ☒ NumberPassword
- ☒ Phone
- ☒ Uri

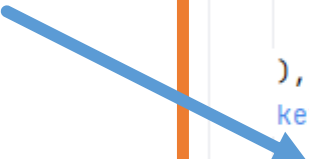
```
keyboardOptions = KeyboardOptions(  
    keyboardType = KeyboardType.Phone,  
    imeAction = ImeAction.  
),  
keyboardActions = Keybo  
onNext = { this: Keybo  
}  
cer(modifier = Modifier,
```

- ☒ Next
- ☒ Done
- ☒ Go
- ☒ Default
- ☒ Send
- ☒ None
- ☒ Previous
- ☒ Search



4.- Componentes TextField

El parámetro **keyboardActions** permite indicar la acción que se ejecutará al pulsar el botón (imeAction).



```
TextField(  
    value = textFieldValue,  
    onValueChange = { textFieldValue = it },  
    label = { Text( text: "Introduce tu teléfono") },  
    keyboardOptions = KeyboardOptions(  
        keyboardType = KeyboardType.Phone,  
        imeAction = ImeAction.Next  
    ),  
    keyboardActions = KeyboardActions(  
        onDone = { /* TODO */ },  
        onNext = { /* TODO */ },  
        on|  
        onGo = (KeyboardActionScope() -> Unit)?  
        onPrevious = (KeyboardActionScope() -> Unit)?  
        onSearch = (KeyboardActionScope() -> Unit)?  
        onSend = (KeyboardActionScope() -> Unit)?  
    )  
)
```

5.- State Hoisting

Como se estudió a principio de la unidad, es muy habitual crear componentes propios que envuelven uno o más componentes Jetpack Compose.

En este punto se pueden dar dos opciones:

- Componente **stateless**: componente propio en el que no se declara un estado.
- Componente **stateful**: componente propio en el que se declara un estado.

Stateless:

```
@Composable
fun MyHeaderText(
    text: String,
    modifier: Modifier = Modifier
) {
    Text(
        text = text,
        fontSize = 30.sp,
        fontWeight = FontWeight.Bold,
        color = Color.White,
        modifier = modifier
        .background(Color.DarkGray)
        .padding(16.dp)
        .then(modifier)
    )
}
```

Stateful:

```
@Composable
fun MyTextField() {
    var textFieldState by rememberSaveable { mutableStateOf( value: "" ) }
    TextField(
        value = textFieldState,
        onChange = { textFieldState=it }
    )
}
```



5.- State Hoisting

Desde el exterior de un componente **stateful** no se tendrá acceso a su estado.

Imaginemos un **formulario** en el que **todos los componentes son propios y stateful**, ¿Cómo se podría acceder a los datos que ha introducido el usuario?

En el ejemplo anterior, tanto los componentes como las variables de estado se encuentran en el mismo elemento y no existe ese problema, pero ¿y si se hubieran extraído los componentes a componentes propios como se recomienda?

```
var nameField by rememberSaveable { mutableStateOf( value: "" ) }
var passwordField by rememberSaveable { mutableStateOf( value: "" ) }
OutlinedTextField(
    value = nameField,
    onValueChange = { nameField = it },
    placeholder = { Text( text: "Email" ) }
)
Spacer(Modifier.size(8.dp))
OutlinedTextField(
    value = passwordField,
    onValueChange = { passwordField = it },
    placeholder = { Text( text: "Contraseña" ) }
)
Spacer(Modifier.size(8.dp))
Button(
    onClick = { /*TODO*/ },
    enabled = nameField.isNotEmpty() && passwordField.isNotEmpty()
) { this: RowScope
    Text( text: "Entrar" )
}
```

```
NameTextField()
PasswordTextField()
Spacer(Modifier.size(8.dp))
AccessButton()
```

5.- State Hoisting

Para solucionar esto se utiliza el **State Hoisting** (elevación de estado) para poder acceder al estado desde el exterior.

Mediante el **State Hoisting** el estado no se declara dentro del componente si no en el **componente más externo** en el que se vaya a utilizar dicho estado.

Al declarar el estado fuera del componente, este componente se convertirá en **stateless**.

Se recomienda que todos los componentes sean stateless en la medida de lo posible.

Si un componente utiliza un estado y este estado no se necesita en el exterior, en ese caso el componente puede ser **stateful**.

5.- State Hoisting

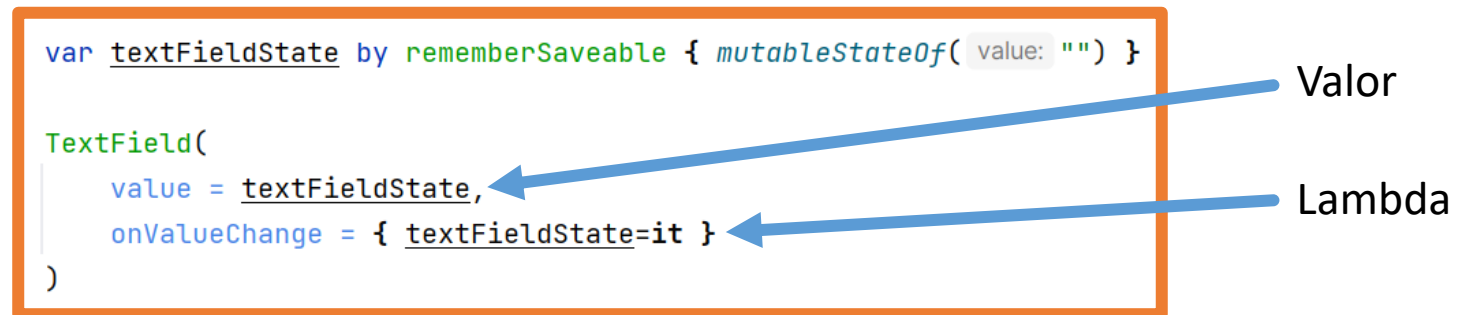
La técnica del **State Hoisting** consiste en eliminar la declaración del estado de un componente **stateful** (y así convertirlo en stateless) y sustituirlo por uno o dos parámetros que se le deberán proporcionar al utilizarlo:

- Uno para proporcionar el valor al componente
- Otro que es una lambda para modificar ese valor.

Esta técnica ya se utiliza en el componente TextField:

```
var textFieldState by rememberSaveable { mutableStateOf( value: "" ) }

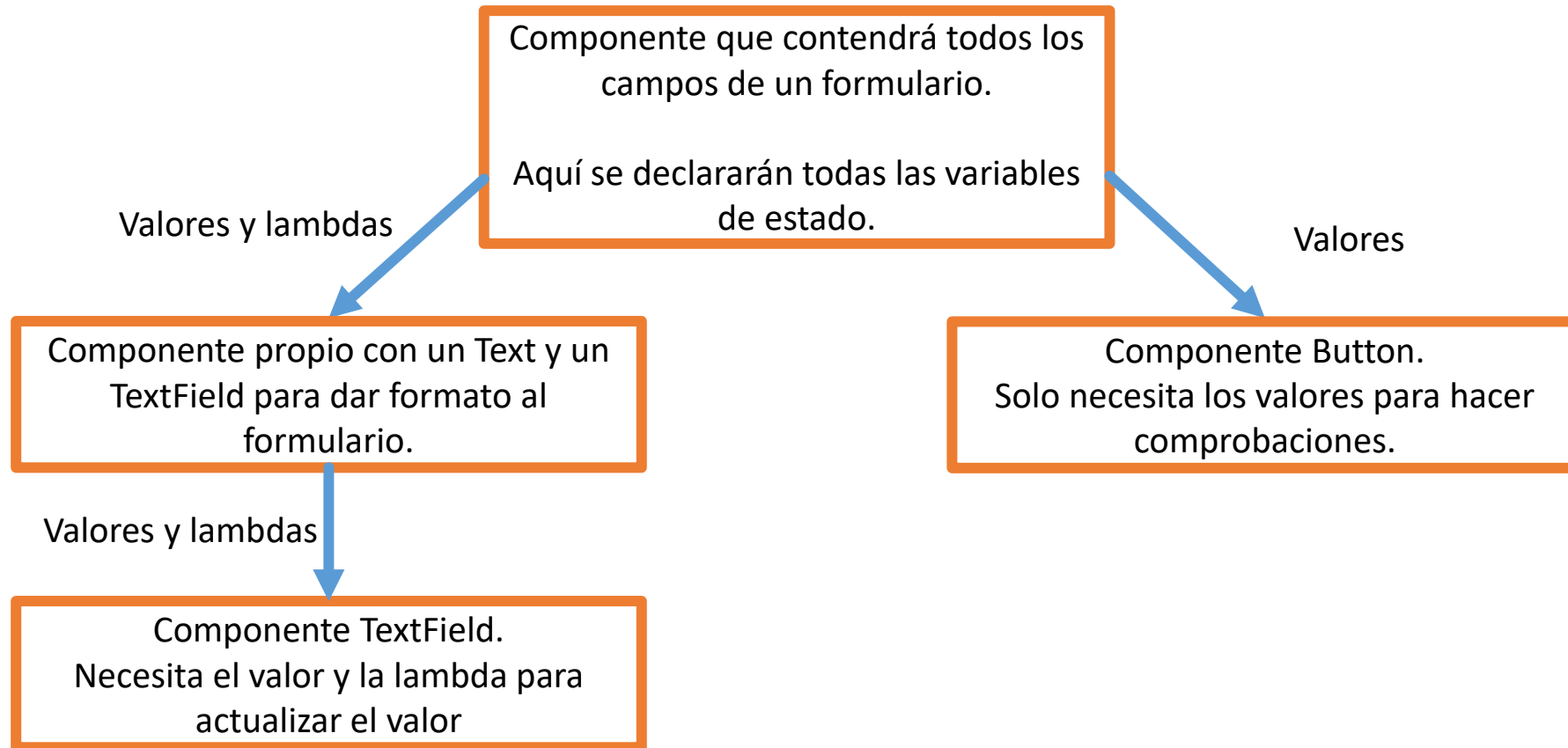
TextField(
    value = textFieldState,
    onChange = { textFieldState=it }
)
```



En esta caso concreto el componente TextField necesita tanto el parámetro para el valor como la lambda para cambiarlo.

5.- State Hoisting

Cuando se utiliza la técnica del **State Hoisting** se debe decidir si el componente propio **recibirá el valor y la lambda para cambiarlo** o solo uno de esos parámetros según la funcionalidad del componente propio.



5.- State Hoisting

Ejemplo anterior aplicando **State Hoisting**:

Componentes propios **stateless**

```
@Composable
fun NameTextField(value: String, onValueChange: (String) -> Unit) {
    TextField(
        value = value,
        onValueChange = { onValueChange(it) }
    )
}

@Composable
fun PasswordTextField(value: String, onValueChange: (String) -> Unit) {
    TextField(
        value = value,
        onValueChange = { onValueChange(it) },
        visualTransformation = PasswordVisualTransformation()
    )
}

@Composable
fun AccessButton(value: Boolean, onClick: () -> Unit) {
    Button(
        onClick = onClick,
        enabled = value
    ) { this: RowScope
        Text( text: "Entrar")
    }
}
```

Uso de esos componentes

```
var nameState by rememberSaveable { mutableStateOf( value: "" ) }
var passState by rememberSaveable { mutableStateOf( value: "" ) }

NameTextField(
    value = nameState,
    onValueChange = { nameState = it }
)

PasswordTextField(
    value = passState,
    onValueChange = { passState = it }
)

Spacer( Modifier.size(8.dp) )
AccessButton(
    value = nameState.isNotEmpty() && passState.isNotEmpty(),
    onClick = { /* TODO */ }
)
```

5.- State Hoisting

Si las variables de estado **solo se van a usar dentro de un componente** se puede realizar de una manera más limpia usando `=` en lugar de `by`.

Un solo estado


```
val (value, onValueChange) = rememberSaveable { mutableStateOf( value: "" ) }  
NameTextField(  
    value = value,  
    onValueChange = onValueChange  
)
```

Varios estados

```
val (nameValue, onNameValueChange) = rememberSaveable { mutableStateOf( value: "" ) }  
val (passValue, onPassValueChange) = rememberSaveable { mutableStateOf( value: "" ) }  
NameTextField(  
    value = nameValue,  
    onValueChange = onNameValueChange  
)  
PasswordTextField(  
    value = passValue,  
    onValueChange = onPassValueChange  
)  
Spacer(Modifier.size(8.dp))  
AccessButton(  
    value = nameValue.isNotEmpty() && passValue.isNotEmpty(),  
    onClick = { /* TODO */ }  
)
```

Esta manera de realizar el State **Hoisting** **no es adecuada si se quiere modificar las variables de estado desde otros componentes** ya que en ese caso se debe usar la variable `onChangeValue` lo cual no es muy intuitivo.

```
Button(onClick = {  
    onChangeSurname("")  
}) { this: RowScope  
    Text(text = "Borrar formulario")  
}
```

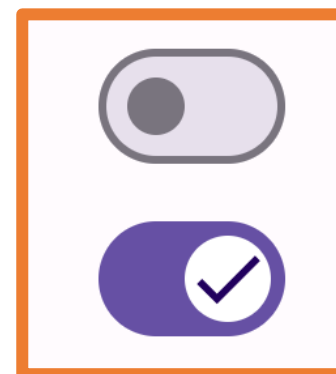


6.- Componente Switch

Los **Switch** son interruptores que se utilizan para activar y desactivar funciones. Son habituales en las preferencias.

El parámetro **thumbContent** permite añadir un icono al interruptor.

```
var switch by rememberSaveable { mutableStateOf( value: false) }  
Switch(  
    checked = switch,  
    onCheckedChange = { it: Boolean  
        |  
        switch = it  
    }  
)  
  
var checked by rememberSaveable { mutableStateOf( value: false) }  
Switch(  
    checked = checked,  
    onCheckedChange = { it: Boolean  
        |  
        checked = it  
    },  
    thumbContent = {  
        Icon(  
            imageVector = Icons.Default.Check,  
            contentDescription = null  
        )  
    }  
)
```



Práctica

Actividad 2

Conversor v2.0.

6.- Componente Switch

Se pueden personalizar los colores:

```
var switch by rememberSaveable { mutableStateOf( value: false) }  
Switch(  
    checked = switch,  
    onCheckedChange = { it: Boolean  
        switch = it  
    },  
    colors = SwitchDefaults.colors(  
        uncheckedBorderColor = Color.Red,  
        checkedBorderColor = Color.Blue  
    )  
)
```

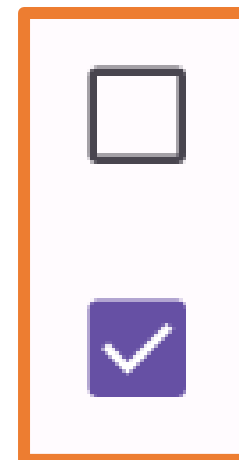
checkedThumbColor - the color used for the thumb when enabled and checked
checkedTrackColor - the color used for the track when enabled and checked
checkedBorderColor - the color used for the border when enabled and checked
checkedIconColor - the color used for the icon when enabled and checked
uncheckedThumbColor - the color used for the thumb when enabled and unchecked
uncheckedTrackColor - the color used for the track when enabled and unchecked
uncheckedBorderColor - the color used for the border when enabled and unchecked
uncheckedIconColor - the color used for the icon when enabled and unchecked
disabledCheckedThumbColor - the color used for the thumb when disabled and checked
disabledCheckedTrackColor - the color used for the track when disabled and checked
disabledCheckedBorderColor - the color used for the border when disabled and checked
disabledCheckedIconColor - the color used for the icon when disabled and checked
disabledUncheckedThumbColor - the color used for the thumb when disabled and unchecked
disabledUncheckedTrackColor - the color used for the track when disabled and unchecked
disabledUncheckedBorderColor - the color used for the border when disabled and unchecked
disabledUncheckedIconColor - the color used for the icon when disabled and unchecked

7.- Componente Checkbox

Los **Checkbox** permiten al usuario marcar una o varias opciones.

Los colores se pueden personalizar de manera similar a los Switch.

```
var checked by rememberSaveable { mutableStateOf( value: false) }  
Checkbox(  
    checked = checked,  
    onCheckedChange = { it: Boolean  
        checked = it  
    }  
)
```



Como los Checkbox suelen ir acompañados de texto se puede crear un componente propio que agrupe en una Row un Text, un Spacer y un Checkbox para poder utilizarlo en todos los lugares.

7.- Componente Checkbox

Ejemplo con varios CheckBox y StateHoisting.

```
// Data class para los objetos MyCheckbox
data class Check(
    val title: String,
    var selected: Boolean = false,
    var onCheckedChange: (Boolean) -> Unit
)

// Función que recibe un conjunto de títulos y los convierte en objetos Check
@Composable
fun getChecks(vararg titles: String): List<Check> {
    return titles.map { it: String
        var state by rememberSaveable { mutableStateOf( value: false) }
        Check(
            title = it,
            selected = state,
            onCheckedChange = { state = it }
        ) ^map
    }
}

// Componente Checkbox stateless propio
@Composable
fun MyCheckbox(check: Check) {
    Row(verticalAlignment = Alignment.CenterVertically) { this: RowScope
        Checkbox(checked = check.selected,
            onCheckedChange = { check.onCheckedChange(it) })
        Text(text = check.title)
    }
}
```

```
val myOptions = getChecks( ...titles: "Opción 1", "Opción 2", "Opción 3")
myOptions.forEach { it: Check
    MyCheckbox(check = it)
}

Button(onClick = {
    // Se recorren las opciones y se puede ver el dato almacenado en cada opción
    myOptions.map { option ->
        Log.i( tag: "--->", msg: "${option.title}: ${option.selected}")
    }
}) { this: RowScope
    Text(text = "Guardar")
}
```

8.- Componente TriStateCheckbox

El componente **TriStateCheckbox** es como un Checkbox pero con tres posiciones.



Se utiliza en conjunto a otros Checkbox de manera que el icono cambiará dependiendo de la cantidad de Checkbox seleccionados:

- Nada seleccionado.
- Algo seleccionado.
- Todo seleccionado.

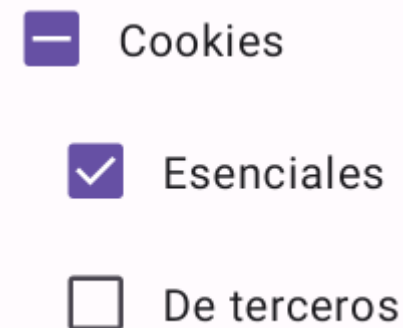
8.- Componente TriStateCheckbox

Ejemplo con TriStateCheckbox:

```
var essentialsState by rememberSaveable { mutableStateOf( value: true) }
var thirdsState by rememberSaveable { mutableStateOf( value: true) }
val cookiesState = rememberSaveable(
    essentialsState,
    thirdsState
) {
    if (essentialsState && thirdsState) ToggleableState.On ^rememberSaveable
    else if (!essentialsState && !thirdsState) ToggleableState.Off ^rememberSaveable
    else ToggleableState.Indeterminate ^rememberSaveable
}

Row(verticalAlignment = Alignment.CenterVertically) { this: RowScope
    TriStateCheckbox(
        state = cookiesState,
        onClick = {
            val s = cookiesState != ToggleableState.On
            essentialsState = s
            thirdsState = s
        },
    )
    Text( text: "Cookies")
}

Column(modifier = Modifier.padding(16.dp, 0.dp, 0.dp, 0.dp)) { this: ColumnScope
    Row(verticalAlignment = Alignment.CenterVertically) { this: RowScope
        Checkbox(
            checked = essentialsState,
            onCheckedChange = { essentialsState = it }
        )
        Text(text = "Esenciales")
    }
    Row(verticalAlignment = Alignment.CenterVertically) { this: RowScope
        Checkbox(
            checked = thirdsState,
            onCheckedChange = { thirdsState = it }
        )
        Text(text = "De terceros")
    }
}
```



9.- Componente RadioButton

Los **RadioButton** permiten seleccionar una única opción entre varias.

Los RadioButton de un conjunto **se deben de agrupar** en un layout al que se indique el **parámetro Modifier.selectableGroup()**.


```
var selected by rememberSaveable { mutableStateOf( value: "" ) }
Column(modifier = Modifier.selectableGroup().fillMaxWidth()) { this: ColumnScope
    Row(verticalAlignment = Alignment.CenterVertically) { this: RowScope
        RadioButton(
            selected = selected == "DAM",
            onClick = {
                selected = "DAM"
            }
        )
        Text( text: "DAM")
    }
    Row(verticalAlignment = Alignment.CenterVertically) { this: RowScope
        RadioButton(
            selected = selected == "DAW",
            onClick = {
                selected = "DAW"
            }
        )
        Text( text: "DAW")
    }
}
```



9.- Componente RadioButton

Es interesante que se pueda hacer clic sobre toda la fila no solo sobre el RadioButton:

```
var selected by rememberSaveable { mutableStateOf( value: "" ) }
Column(modifier = Modifier.selectableGroup().fillMaxWidth()) { this: ColumnScope
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier.selectable(
            selected = selected == "DAM",
            onClick = { selected = "DAM" }
        )
    ) { this: RowScope
        RadioButton(
            selected = selected == "DAM",
            onClick = { selected = "DAM" }
        )
        Text( text: "DAM")
    }
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier.selectable(
            selected = selected == "DAW",
            onClick = { selected = "DAW" }
        )
    ) { this: RowScope
        RadioButton(
            selected = selected == "DAW",
            onClick = { selected = "DAW" }
        )
        Text( text: "DAW")
    }
}
```



```
val radioOptions = listOf("SMR", "ASIR", "DAM", "DAW")
var selected by rememberSaveable { mutableStateOf( value: "" ) }
Column(modifier = Modifier.selectableGroup().fillMaxWidth()) { this: ColumnScope
    radioOptions.forEach { option ->
        Row(
            verticalAlignment = Alignment.CenterVertically,
            modifier = Modifier.selectable(
                selected = selected == option,
                onClick = { selected = option }
            )
        ) { this: RowScope
            RadioButton(
                selected = selected == option,
                onClick = { selected = option }
            )
            Text(option)
        }
    }
}
```

9.- Componente RadioButton

Ejemplo con varios **RadioButton** y **StateHoisting**.

```
@Composable
fun MyRadioButton(vararg options: String, selected: String, onClick: (String) -> Unit) {
    options.forEach { option ->
        Row(verticalAlignment = Alignment.CenterVertically) { this: RowScope
            RadioButton(
                selected = selected == option,
                onClick = { onClick(option) }
            )
            Text(text = option)
        }
    }
}
```

```
var option by rememberSaveable { mutableStateOf( value: "" ) }
MyRadioButton( ...options:
    "Opción 1", "Opción 2", "Opción 3",
    selected = option,
    onClick = { option = it }
)
```

10.- Componentes Slider

Los **Slider** son barras con un indicador que se puede deslizar para seleccionar un valor.

Por defecto se puede seleccionar entre los valores 0.0 y 1.0 pero se puede cambiar.

También se puede modificar el indicador que se desliza.

```
var sliderValue by rememberSaveable { mutableStateOf( value: 0) }
Row(verticalAlignment = Alignment.CenterVertically) { this: RowScope
    Text(sliderValue.toString())
    Spacer(Modifier.width(8.dp))
    Slider(
        value = sliderValue.toFloat(),
        onValueChange = { it: Float
            sliderValue = it.toInt()
        },
        valueRange = 0f ≤ .. ≤ 10f,
        steps = 9,
    )
}
```



```
var sliderValue by rememberSaveable { mutableStateOf( value: 0) }
Row(verticalAlignment = Alignment.CenterVertically) { this: RowScope
    Text(sliderValue.toString())
    Spacer(Modifier.width(8.dp))
    Slider(
        value = sliderValue.toFloat(),
        onValueChange = { it: Float
            sliderValue = it.toInt()
        },
        valueRange = 0f ≤ .. ≤ 10f,
        steps = 9,
        thumb = { it: SliderPositions
            Icon(
                imageVector = Icons.Default.Adb,
                contentDescription = null
            )
        }
    )
}
```



10.- Componentes Slider

El componente **RangeSlider** funciona de manera similar a un **Slider** pero permite seleccionar un rango de valores.

```
var rangeMinState by rememberSaveable { mutableStateOf( value: 0) }
var rangeMaxState by rememberSaveable { mutableStateOf( value: 10) }
Column { this: ColumnScope
    Text( text: "$rangeMinState -> $rangeMaxState")
    RangeSlider(
        value = rangeMinState.toFloat() ≤ .. ≤ rangeMaxState.toFloat(),
        onValueChange = { it: ClosedFloatingPointRange<Float>
            rangeMinState = it.start.toInt()
            rangeMaxState = it.endInclusive.toInt()
        },
        valueRange = 0f ≤ .. ≤ 10f,
        steps = 9
    )
}
```



Se pueden modificar los indicadores de inicio (startThumb) y final (endThumb).

En el ejemplo se utilizan dos estados para los valores mínimo y máximo, se podría crear un estado propio que almacenara el rango (ahora mismo su uso es experimental).

11.- Componente ExposedDropdownMenu

El componente **ExposedDropdownMenu** permite mostrar un TextField con un menú desplegable.

Según como se programe el contenido un ExposedDropdownMenu se puede usar de las siguientes maneras:

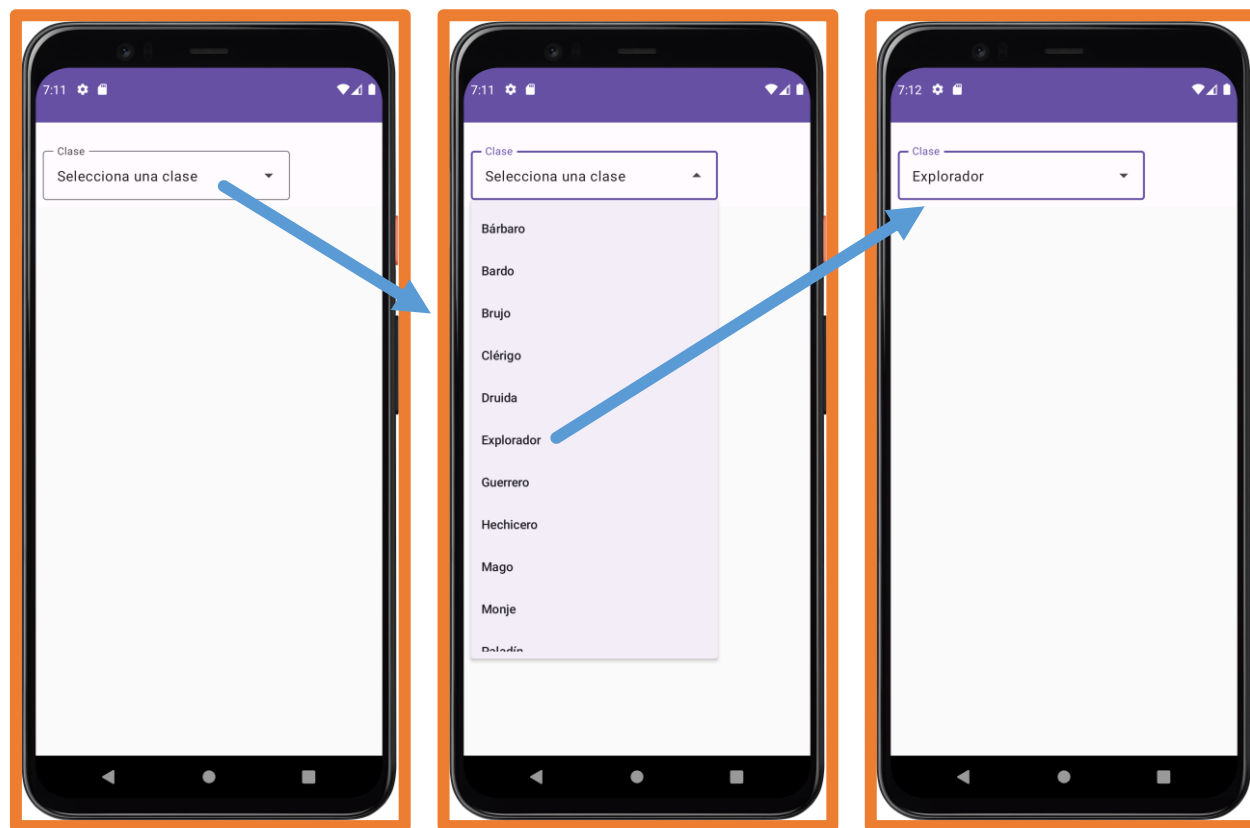
- **No editable**: solo se puede seleccionar una opción.
- **Editable**: se puede elegir una opción y también se puede escribir.
- **Editable y autocompletable**: como editable y que filtra los resultados.

Un ExposedDropdownMenu se compone de un TextField con el modificador **menuAnchor()** y de varios componentes **DropdownMenuItem** que son las opciones disponibles.

11.- Componente ExposedDropDownMenu

Ejemplo de ExposedDropDownMenu no editable:

```
val classes = listOf("Bárbaro", "Bardo", "Brujo", "Clérigo", "Druida", "Explorador",  
                    "Guerrero", "Hechicero", "Mago", "Monje", "Paladín", "Pícaro")  
var showMenu by rememberSaveable { mutableStateOf( value: false) }  
var selectedOptionText by rememberSaveable { mutableStateOf( value: "Selecciona una clase") }  
  
ExposedDropDownMenuBox(  
    expanded = showMenu,  
    onExpandedChange = { showMenu = !showMenu },  
) { this: ExposedDropDownMenuBoxScope  
    OutlinedTextField(  
        modifier = Modifier.menuAnchor(),  
        readOnly = true,  
        value = selectedOptionText, }  
        onValueChange = {},  
        label = { Text( text: "Clase") },  
        trailingIcon = { ExposedDropDownMenuDefaults.TrailingIcon(expanded = showMenu) },  
    )  
    ExposedDropDownMenu(  
        expanded = showMenu,  
        onDismissRequest = { showMenu = false },  
    ) { this: ColumnScope  
        classes.forEach { option ->  
            DropdownMenuItem(  
                text = { Text(option) },  
                onClick = {  
                    selectedOptionText = option  
                    showMenu = false  
                }  
            )  
        }  
    }  
}
```



12.- Componentes DatePicker

El componente **DatePicker** permite seleccionar o introducir manualmente una fecha.

Se puede indicar la fecha seleccionada inicialmente y cambiar la cabecera y el título. También se puede mostrar el icono para cambiar el modo de introducción de fecha.



También existen [DatePickerDialog](#) y [DateRangePicker](#).

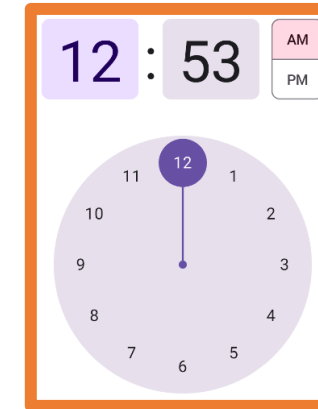
Se necesita usar una versión Alpha de Material 3 en **build.gradle.kts (Module: app)**:

```
implementation("androidx.compose.material3:material3-android:1.2.0-alpha10")
```

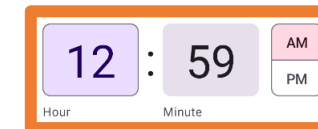
13.- Componentes TimePicker y TimeInput

Los componentes **TimePicker** (con dial) y **TimeInput** (con teclado) permiten seleccionar una hora.

```
val timePickerState = rememberTimePickerState(  
    initialHour = Calendar.getInstance().get(Calendar.HOUR_OF_DAY),  
    initialMinute = Calendar.getInstance().get(Calendar.MINUTE)  
)  
  
TimePicker(  
    state = timePickerState,  
    layoutType = TimePickerLayoutType.Vertical  
)
```



```
val timePickerState = rememberTimePickerState(  
    initialHour = Calendar.getInstance().get(Calendar.HOUR_OF_DAY),  
    initialMinute = Calendar.getInstance().get(Calendar.MINUTE)  
)  
  
TimeInput(  
    state = timePickerState  
)
```



Se necesita usar una versión Alpha de Material 3 en **build.gradle.kts (Module: app)**:

```
implementation("androidx.compose.material3:material3-android:1.2.0-alpha10")
```


14.- Componentes Chips

Los **Chips** son pequeños componentes con un texto que inician el ingreso de información, el inicio de acciones, el filtrado de contenido o realización de selecciones.

Existen cuatro tipos:

- **Assist**: representan acciones predefinidas en la aplicación, por ejemplo, añadir al calendario o cómo llegar a un lugar.
- **Filter**: representan filtros a aplicar a los elementos de una colección.
- **Input**: representan pequeños trozos de información introducidos por el usuario, por ejemplo, tras introducir una dirección de email.
- **Suggestion**: limitan las acciones del usuario al mostrar sugerencias generadas dinámicamente.

Existen las variantes **ElevatedAssistChip**, **ElevatedFilterChip** y **ElevatedSuggestionChip**.

14.- Componentes Chips

Los Chips:

- No son botones.
- No se deben usar como acciones finales, para eso están los botones.
- Son reactivas a las acciones del usuario.
- Son contextuales al contenido de la aplicación.
- Deberían aparecer siempre en grupo.
- Se deben organizar para que se muestren con scroll horizontal.

14.- Componentes Chips

Todos los componentes Chip se basan en el componente **privado Chip** por lo que son similares visualmente aunque cada uno admite unos parámetros según su función.

```
SuggestionChip(  
    onClick = { /* TODO */ },  
    label = { Text( text: "Suggestion Chip") }  
)  
  
AssistChip(  
    onClick = { /* TODO */ },  
    label = { Text( text: "Assist Chip") },  
    leadingIcon = {  
        Icon(  
            Icons.Filled.Settings,  
            contentDescription = "Localized description",  
            Modifier.size(AssistChipDefaults.IconSize)  
        )  
    }  
)
```

Suggestion Chip



Assist Chip

```
var selectedFilterChip by rememberSaveable { mutableStateOf( value: false) }  
FilterChip(  
    selected = selectedFilterChip,  
    onClick = { selectedFilterChip = !selectedFilterChip },  
    label = { Text( text: "Filter chip") },  
    leadingIcon = {  
        if (selectedFilterChip) {  
            Icon(  
                imageVector = Icons.Filled.Done,  
                contentDescription = "Localized Description",  
                modifier = Modifier.size(FilterChipDefaults.IconSize)  
            )  
        }  
    }  
)  
  
var selectedInputChip by rememberSaveable { mutableStateOf( value: false) }  
InputChip(  
    selected = selectedInputChip,  
    onClick = { selectedInputChip = !selectedInputChip },  
    label = { Text( text: "Input Chip") },  
)
```

✓ Filter chip

Input Chip

15.- Componentes IconButton

Los **IconButton** se utilizan cuando se necesita tener botones compactos, se usan habitualmente **en barras de herramientas**.

Hay varios tipos:

- IconButton
- IconToggleButton
- FilledIconButton
- FilledToggleIconButton
- FilledTonalIconButton
- FilledTonalIconButton
- OutlinedIconButton
- OutlinedIconToggleButton

```
FilledIconButton(onClick = { /*TODO*/ }) {  
    Icon(  
        imageVector = Icons.Default.Settings,  
        contentDescription = "Configuración"  
    )  
}
```



```
var selected by rememberSaveable { mutableStateOf( value: false) }  
FilledIconToggleButton(  
    checked = selected,  
    onCheckedChange = { selected = !selected }  
) {  
    Icon(  
        imageVector = Icons.Default.Settings,  
        contentDescription = "Configuración"  
    )  
}
```



Las variantes **Toggle** tienen dos estados marcado o no.

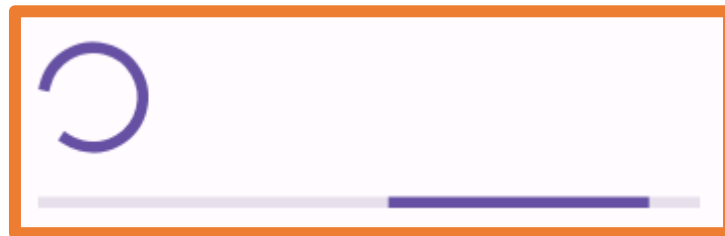
16.- Componentes ProgressIndicator

Los componentes **LinearProgressIndicator** y **CircularProgressIndicator** permiten informar al usuario de que se está ejecutando un.

Las líneas de progreso pueden ser:

- **Indeterminadas:** no tienen fin.
- **Determinadas:** empiezan vacías y se rellenan del todo.

Para convertir una línea de progreso en determinada se debe usar el parámetro **progress** con valores de 0.0f a 1.0f.



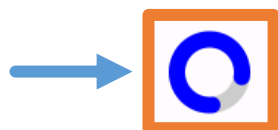
16.- Componentes ProgressIndicator

Los componentes **LinearProgressIndicator** y **CircularProgressIndicator** tienen varios constructores según el aspecto que se quiera dar.

Los parámetros disponibles son:

- **progress**: porcentaje de progreso relleno.
- **color**: color del progreso.
- **trackColor**: color de fondo del progreso.
- **strokeCap**: forma de la línea de progreso.
- **strokeWidth** (solo el circular): grosor de la línea de progreso.

```
CircularProgressIndicator(  
    strokeCap = StrokeCap.Round,  
    trackColor = Color.LightGray,  
    color = Color.Blue,  
    strokeWidth = 8.dp  
)
```



```
LinearProgressIndicator(  
    progress = 0.3f,  
    strokeCap = StrokeCap.Butt,  
    modifier = Modifier.height(16.dp)  
)
```



17.- Componentes SearchBar

Existen dos componentes para crear barras de búsqueda:

- **SearchBar**: el cuerpo ocupa toda la pantalla.
- **DockedSearchBar**: el cuerpo ocupa una parte de la pantalla (configurable).

Para poder usar estos componentes se deben actualizar algunas versiones:

build.gradle.kts (Project):

Kotlin → 1.8.21

build.gradle.kts (Module):

kotlinCompilerExtensionVersion → 1.4.7

compose-bom → 2023.05.01

core-ktx → 1.10.1

Recuerda sincronizar tras realizar cambios en los archivos **gradle**.

17.- Componentes SearchBar

Los **SearchBar** tienen diferentes parámetros para configurarlos, algunos dependen de un estado, así la configuración mínima será la siguiente:

```
var querySearchState by rememberSaveable { mutableStateOf( value: "")}  
var activeSearchState by rememberSaveable { mutableStateOf( value: false) }  
SearchBar(  
    query = querySearchState,  
    onQueryChange = { querySearchState = it },  
    onSearch = { },  
    active = activeSearchState,  
    onActiveChange = { activeSearchState = it }  
) { this: ColumnScope  
    // Cuerpo de la búsqueda  
}
```

Se pueden usar diferentes técnicas:

- Rellenar todo el cuerpo con todo y posteriormente filtrar según lo introducido.
- Dejar el cuerpo vacío y cambiarlo con los resultados según lo introducido.
- Crear un estado para el cuerpo y cambiarlo con los resultados según lo introducido.
- ...

17.- Componentes SearchBar

Ejemplo de uso:

```
// Llista de todos los elementos a buscar
val dragonBallCharacters = listOf(
    "Son Goku",
    "Vegeta",
    "Piccolo",
    "Son Gohan",
    "Trunks",
    "Bulma",
    "Krillin",
    "Master Roshi",
    "Freezer",
    "Cell"
)

// En algunas acciones se necesita el contexto (suele ser la Activity)
val myContext = LocalContext.current

var querySearchState by rememberSaveable { mutableStateOf( value: "") }
var activeSearchState by rememberSaveable { mutableStateOf( value: false) }

SearchBar(
    query = querySearchState,
    onQueryChange = { querySearchState = it },
    onSearch = { it: String
        // Para mostrar mensajes temporales en la parte inferior de la pantalla
        Toast.makeText(myContext, text: "Se busca: $it", Toast.LENGTH_SHORT).show()
    },
    active = activeSearchState,
    onActiveChange = { activeSearchState = it },
    placeholder = { Text( text: "Introduce el personaje a buscar") },
    trailingIcon = {
        val icon = if (activeSearchState) Icons.Default.Close else Icons.Default.Search
        IconButton(onClick = {
            querySearchState = ""

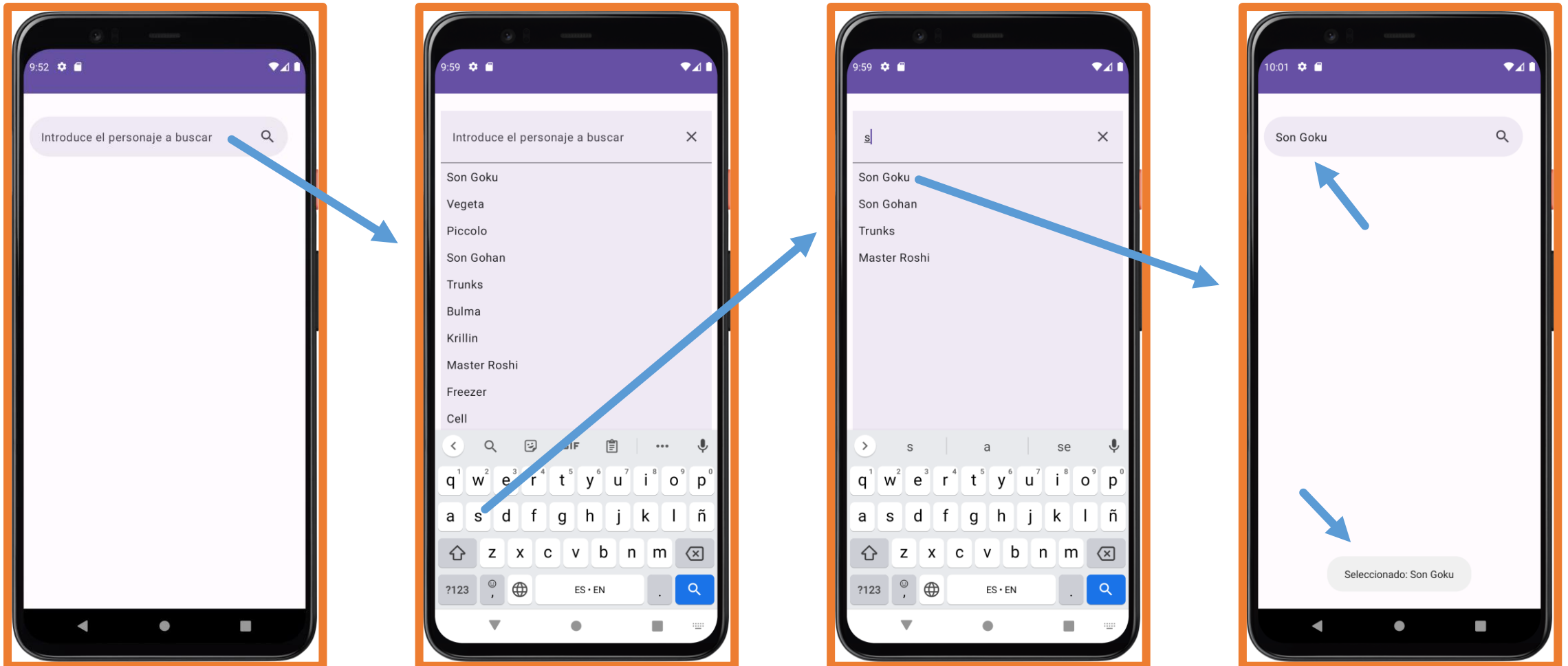
```

```
            activeSearchState = !activeSearchState
        }) {
            Icon(
                imageVector = icon,
                contentDescription = "Cerrar búsqueda"
            )
        }
    }
) { this: ColumnScope
    val charactersToShow = if (querySearchState.isEmpty()) {
        dragonBallCharacters
    } else {
        dragonBallCharacters.filter { it: String
            it.contains(other = querySearchState, ignoreCase = true)
        }
    }

    Column { this: ColumnScope
        charactersToShow.forEach { it: String
            Text(
                text = it,
                modifier = Modifier
                    .padding(8.dp)
                    .clickable {
                        Toast.makeText(myContext, text: "Seleccionado: $it", Toast.LENGTH_SHORT).show()
                        querySearchState = it
                        activeSearchState = false
                    }
            )
        }
    }
}
}
```

17.- Componentes SearchBar

Ejemplo de uso:



18.- Componente AlertDialog

El componente **AlertDialog** permite mostrar un mensaje en una ventana modal (emergente).

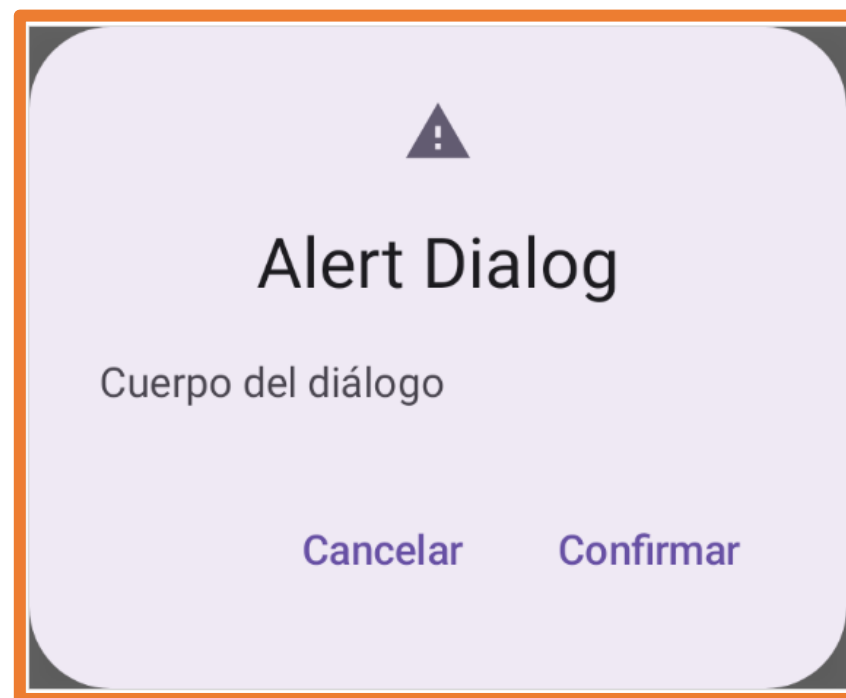
```
fun AlertDialog(  
    onDismissRequest: () -> Unit,  
    confirmButton: @Composable () -> Unit,  
    modifier: Modifier = Modifier,  
    dismissButton: @Composable (() -> Unit)? = null,  
    icon: @Composable (() -> Unit)? = null,  
    title: @Composable (() -> Unit)? = null,  
    text: @Composable (() -> Unit)? = null,  
    shape: Shape = AlertDialogDefaults.shape,  
    containerColor: Color = AlertDialogDefaults.containerColor,  
    iconContentColor: Color = AlertDialogDefaults.iconContentColor,  
    titleContentColor: Color = AlertDialogDefaults.titleContentColor,  
    textContentColor: Color = AlertDialogDefaults.textContentColor,  
    tonalElevation: Dp = AlertDialogDefaults.TonalElevation,  
    properties: DialogProperties = DialogProperties()  
) = AlertDialog(onDismissRequest = onDismissRequest, modifier = modifier
```

Los parámetros **confirmButton**, **dismissButton**, **icon**, **title** y **text** son funciones lambda **@Composable** por lo que dentro admiten cualquier componente que se quiera.

También existe el componente **Dialog** pero no utiliza los principios de Material y se debe configurar totalmente a mano.

18.- Componente AlertDialog

```
AlertDialog(  
  onDismissRequest = {  
    // Se ejecuta pulsar atras o fuera del AlertDialog  
    Log.i( tag: "Dialog", msg: "Se ha cancelado")  
  },  
  confirmButton = {  
    TextButton(onClick = {  
      Log.i( tag: "Dialog", msg: "Se ha aceptado")  
    }) { this: RowScope  
      Text(text = "Confirmar")  
    }  
  },  
  dismissButton = {  
    TextButton(onClick = {  
      Log.i( tag: "Dialog", msg: "Se ha cancelado")  
    }) { this: RowScope  
      Text(text = "Cancelar")  
    }  
  },  
  icon = {  
    Icon(  
      imageVector = Icons.Default.Warning,  
      contentDescription = "Advertencia"  
    )  
  },  
  title = { Text( text: "Alert Dialog") },  
  text = { Text( text: "Cuerpo del diálogo") },  
)
```



18.- Componente AlertDialog

Con el código anterior se **mostrará siempre el AlertDialog** lo cual no es lo más interesante.

El AlertDialog debería estar oculto y mostrarse al producirse una acción del usuario como pulsar un botón.

Para **controlar que esté oculto o visible** se debe utilizar **una variable que debería almacenarse en el estado**, para que al cambiar, en la recomposición de la pantalla aparezca o no.

18.- Componente AlertDialog

```
var dialogVisible by rememberSaveable { mutableStateOf( value: false) }

if (dialogVisible) {
    AlertDialog(
        onDismissRequest = { dialogVisible = false },
        confirmButton = {
            TextButton(onClick = { dialogVisible = false }) { this: RowScope
                Text(text = "Confirmar")
            }
        },
        dismissButton = {
            TextButton(onClick = { dialogVisible = false }) { this: RowScope
                Text(text = "Cancelar")
            }
        },
        icon = {
            Icon(
                imageVector = Icons.Default.Warning,
                contentDescription = "Advertencia"
            )
        },
        title = { Text( text: "Alert Dialog") },
        text = { Text( text: "Cuerpo del diálogo") },
    )
}

Button(onClick = {
    dialogVisible = true
}) { this: RowScope
    Text( text: "Mostrar AlertDialog")
}
```



Práctica

Actividad 3 Registro.