Textos

Compose

Text()

```
@Composable
fun Text(text: String,
        modifier: Modifier = Modifier,

        softWrap: Boolean = true,
        minLines: Int = 1,
        maxLines: Int = Int.MAX_VALUE,
        overflow: TextOverflow = TextOverflow.Clip,

        color: Color = Color.Unspecified,
        textDecoration: TextDecoration? = null,
        fontFamily: FontFamily? = null,
        fontSize: TextUnit = TextUnit.Unspecified,
        fontWeight: FontWeight? = null,
        fontStyle: FontStyle? = null,
        textAlign: TextAlign? = null,

        style: TextStyle = LocalTextStyle.current
       )
```

Text()

text:

String → valor a mostrar

softWrap:

Boolean → Indica si el texto debe distribuirse en varias líneas cuando excede el ancho disponible

softWrap = true → el texto se distribuye en varias líneas.
softWrap = false → el texto se mantiene en una sola línea.

minLines:

Int = 1 → Establece el número mínimo de líneas que se mostrarán en el texto. Debe ser mayor que 1

maxLines:

Int = Int.MAX_VALUE → Establece el número máximo de líneas que se mostrarán en el texto. Debe ser >= que minLines

overflow:

TextOverflow → Define cómo se maneja el texto que excede el espacio disponible

TextOverflow.Ellipsis → muestra puntos suspensivos (…) al final del texto
TextOverflow.Clip → corta el texto sin indicación.
TextOverflow.Visible → el texto se muestra en su totalidad

Text()

color:

Color → color del texto

Color.Red
Color.White

textDecoration

TextDecoration → Aplicar decoraciones al texto

TextDecoration.Underline → Subrayado.
TextDecoration.LineThrough → Tachado.

fontSize:

TextUnit → Tamaño del texto

fontSize = 12.sp

fontWeight:

FontWeight → Grosor del texto

FontWeight.Thin (100)
FontWeight.ExtraLight (200)
FontWeight.Light (300)
FontWeight.Normal (400)
FontWeight.Medium (500)
FontWeight.SemiBold (600)
FontWeight.Bold (700)
FontWeight.ExtraBold (800)
FontWeight.Black (900)

Text()

fontFamily:

FontFamily → Tipo de letra

FontFamily.Default
FontFamily.SansSerif
FontFamily.Serif
FontFamily.Monospace
FontFamily.Cursive

```
val fontFamily = FontFamily( Font(R.font.fuente_regular, FontWeight.Normal),
                             Font(R.font.fuente_bold,    FontWeight.Bold)
                           )

Text(..., fontFamily = fontFamily)
```

fontStyle:

FontStyle → Aplicar un estilo al texto

FontStyle.Normal
FontStyle.Italic

```
val fontStyle = TextStyle( fontFamily = FontFamily.Serif,
                           fontWeight = FontWeight.SemiBold,
                           fontStyle = FontStyle.Italic,
                           fontSize = 18.sp,
                          )

Text(..., fontStyle = fontStyle)
```

TextField()

```
@Composable
fun TextField(value: String,
        onValueChange: (String) -> Unit,
        modifier: Modifier = Modifier,

        label: @Composable (() -> Unit)? = null,
        placeholder: @Composable (() -> Unit)? = null,
        leadingIcon: @Composable (() -> Unit)? = null,
        trailingIcon: @Composable (() -> Unit)? = null,

        enabled: Boolean = true,
        readOnly: Boolean = false,

        singleLine: Boolean = false,
        maxLines: Int = if (singleLine) 1 else Int.MAX_VALUE,
        minLines: Int = 1,

        keyboardOptions: KeyboardOptions = KeyboardOptions.Default,
        keyboardActions: KeyboardActions = KeyboardActions.Default,

        supportingText: @Composable (() -> Unit)? = null,
        isError: Boolean = false,
        visualTransformation: VisualTransformation = VisualTransformation.None,

        textStyle: TextStyle = LocalTextStyle.current,
)
```

TextField()

value:

String → valor a mostrar

onChangeValue:

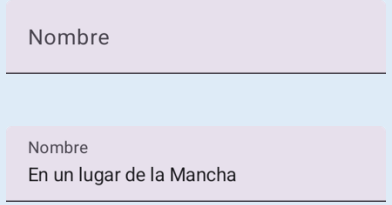
(String) -> Unit → lambda que representa qué hacer si se modifica el valor.

```
var texto by rememberSaveable { mutableStateOf("...") }

TextField(value = texto,
        onChangeValue = { texto = it },
        ...
       );
```

label:

(@Composable () -> Unit)? = null → Etiqueta opcional que se muestra dentro del campo de texto.

```
var texto by rememberSaveable { mutableStateOf("En un lugar de la Mancha") }

TextField(value = text,
        onValueChange = { text = it },
        label = { Text("Nombre") },
       )
```

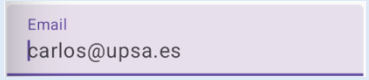


TextField()

placeholder:

(@Composable () -> Unit)? = null → Texto mostrado cuando el valor está vacío.

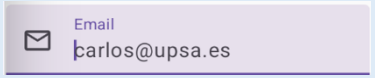
```
var email by rememberSaveable { mutableStateOf("") }

TextField(value = email,
        onValueChange = { email = it },
        label = { Text("Nombre") },
        placeholcer = { Text("carlos@gmail.com" ) },
       )
```



leadingIcon / trailingIcon:

(@Composable () -> Unit)? = null → Imagen opcional que se muestra al inicio/final (puede ser un IconButton)

```
var email by rememberSaveable { mutableStateOf("") }

 TextField(value = email,
         onValueChange = { email = it },
         label = { Text(stringResource(R.string.email_label) ) },
         leadingIcon = { Icon( Icons.Filled.MailOutline, stringResource(R.string.email_description)) }
        )
```



TextField()

readOnly:

Boolean = false → indica si es editable o no

enabled:

Boolean = true → Indica si el campo está habilitado.

singleLine:

Boolean = true → Indica si el texto debe ocupar una única línea.

minLines:

Int = 1 → Si (singleLine == false) indica el número mínimo de líneas.

maxLines:

Int = if (singleLine) 1 else Int.MAX_VALUE → Si (singleLine == false) indica el número máximo de líneas.

lineHeight:

TextUnit = TextUnit.Unspecified → Altura de las líneas (medido en **.sp** ó **.em**).

TextField()

keyboardOptions:

KeyboardOptions = KeyboardOptions.Default → Opciones del teclado según el texto a introducir.

```
KeyboardOptions(keyboardType = KeyboardType.Email,
                capitalization = KeyboardCapitalization.None,
                autoCorrect = true,
                imeAction = ImeAction.Next
               ),
```

KeyBoardType → Tipo de teclado

- **KeyBoardType.Email**
- **KeyBoardType.Text**
- **KeyBoardType.Uri**
- **KeyBoardType.Number**
- **KeyBoardType.Ascii**
- **KeyBoardType.Decimal**
- **KeyBoardType.NumberPassword**
- **KeyBoardType.Password**
- **KeyBoardType.Phone**

ImeAction → Tecla de acción del teclado

- **ImeAction.None**
- **ImeAction.Next**
- **ImeAction.Previous**
- **ImeAction.Go**
- **ImeAction.Done**
- **ImeAction.Send**
- **ImeAction.Default**
- **ImeAction.Search**

KeyboardCapitalization → Mayúsculas

- **KeyboardCapitalization.None**
- **KeyboardCapitalization.Characters**
- **KeyboardCapitalization.Words**
- **KeyboardCapitalization.Sentences**

keyboardActions:

KeyboardActions = KeyboardActions.Default → Qué ejecutar en el caso de pulsar la tecla de acción del teclado.

```
KeyboardActions(onDone = {},    → ImeAction.Done
                onSend = {},    → ImeAction.Send
                onGo = {},      → ImeAction.Go
                onSearch = {},  → ImeAction.Search
                onNext = {},    → ImeAction.Next
                onPrevious = {} → ImeAction.Previous
               )

KeyboardActions(onAny = {})
```

supportingText:

(@Composable () -> Unit)? = null → Muestra un composable en la parte inferior del TextField (para textos de ayuda).

```
var email by rememberSaveable { mutableStateOf("") }
val emailMaxLength = 40

TextField(value = email,
        onValueChange = { if (it.length <= emailMaxLength) email = it },
        label = { Text( stringResource(R.string.email_label) ) },
        placeholder = { Text( stringResource(R.string.email_hint) ) },
        leadingIcon = { Icon( Icons.Filled.MailOutline, stringResource(R.string.email_description)) }
        keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Email,
                                  capitalization = KeyboardCapitalization.None,
                                  autoCorrect = true,
                                  imeAction = ImeAction.Next
                                 ),
        supportingText = {
                     Row()
                     {
                       Spacer(modifier = Modifier.weight(1f))
                       Text(text="${email.length}/$emailMaxLength")
                     }
                   }

      )
```

TextField()

isError:

Boolean = false → Indica si el valor es erróneo o no.

```
var email by rememberSaveable { mutableStateOf("") }
var emailErrorMessage by rememberSaveable { mutableStateOf("") }
val emailMaxLength = 40

val validateEmail: (String) -> String = {  when {  it.isEmpty()                          -> "Email vacío"
                                                   !Patterns.EMAIL_ADDRESS.matcher(it).matches() -> "No es un email válido"
                                                   else                                  -> ""
                                                 }
                                          }
TextField(value = email,
        onValueChange = {  if ( (0..emailMaxLength).contains( it.length ) )
                          {
                            email = it
                            emailErrorMessage = validateEmail(it)
                          }
                        },
        label = { Text(stringResource(R.string.email_label)) },
        placeholder = { Text( stringResource(R.string.email_placeholder) ) },
        singleLine = true,
        leadingIcon = { Icon( Icons.Filled.MailOutline, stringResource(R.string.email_description)) },
        keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Email,
                                          capitalization = KeyboardCapitalization.None,
                                          autoCorrect = true,
                                          imeAction = ImeAction.Next
                                          ),
        isError = !emailErrorMessage.isBlank(),
        supportingText = { SupportingText(emailErrorMessage, email.length, emailMaxLength) }
       )
```


```
@Composable
fun SupportingText(message: String, currentLength: Int, maxLength: Int)
{
  Row()
  {
    Text(text = message)
    Spacer(modifier = Modifier.weight(1f))
    Text(text="${currentLength}/$maxLength")
  }
}
```

visualTransformation:

TextField()

VisualTransformation= VisualTransformation.None → Transforma el value para su visualización (ej. ocultación password).

```
var password by rememberSaveable { mutableStateOf("") }
var passwordErrorMessage by rememberSaveable { mutableStateOf("") }
var passwordVisible by rememberSaveable { mutableStateOf(false) }
val passwordMaxLength = 40

val validatePassword: (String) -> String = {  if ( it.length < 8 ) -> "Debe tener al menos 8 caracteres"
                                              else              -> ""
                                            }

TextField(value = password,
        modifier = Modifier.fillMaxWidth(0.9f),
        onValueChange = {
                    if ( (0..passwordMaxLength).contains( it.length ) )
                    {
                      password = it
                      passwordErrorMessage = validatePassword(it)
                    }
                  },
        label = { Text(stringResource(R.string.password_label)) },
        placeholder = { Text(stringResource(R.string.password_placeholder) ) },
        singleLine = true,
        trailingIcon = {  IconButton(onClick = { passwordVisible = !passwordVisible })
                    {
                      Icon(painter = painterResource( if (passwordVisible) R.drawable.visibility_off else R.drawable.visibility_on),
                          contentDescription = stringResource(R.string.email_description)
                         )
                    }
                  },
        visualTransformation = if (passwordVisible) VisualTransformation.None else PasswordVisualTransformation(),
        keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Email,
                                  capitalization = KeyboardCapitalization.None,
                                  autoCorrect = true,
                                  imeAction = ImeAction.Next
                                  ),
        isError = !passwordErrorMessage.isBlank(),
        supportingText = { SupportingText(passwordErrorMessage, password.length, passwordMaxLength) }
       )
```


```
@Composable
fun SupportingText(message: String, currentLength: Int, maxLength: Int)
{
  Row()
  {
    Text(text = message)
    Spacer(modifier = Modifier.weight(1f))
    Text(text="${currentLength}/$maxLength")
  }
}
```