Tipos de variables

val

`val`: Variable inmutable

Una `val` es una variable de solo lectura, es decir, una vez que se le asigna un valor, no puede cambiar. Esto es útil para definir datos constantes.

Uso correcto:

```
val nombre = "Juan"

// nombre = "Pedro" // Error: no puedes reasignar una `val`
```

Razón de ser: Garantiza que el valor no cambie una vez asignado, promoviendo la inmutabilidad y la seguridad en el manejo de datos.

var: Variable mutable

Por otro lado, una `var` es una variable mutable, lo que significa que su valor puede cambiar a lo largo del tiempo.

Uso correcto:

```
var edad = 25

edad = 26 // Esto es válido
```

Uso incorrecto: Usar `var` para datos que no deben cambiar.

```
var nombre = "Juan"

nombre = "Pedro" // Si no quieres que cambie, usa `val`
```

Razón de ser: Permitir cambios de estado cuando se requiere modificar datos durante la ejecución.

MutableState: Estado en Compose

`MutableState` es una clase en Jetpack Compose que permite que el estado sea observado y la UI se recomponga automáticamente al cambiar dicho estado.

Ejemplo de uso:

```
var contador by remember { mutableStateOf(0) }

Button(onClick = { contador++ }) {

    Text(text = "Contador: $contador")

}
```

Razón de ser: Facilitar la reactividad en la UI, permitiendo que Compose observe y recomposite automáticamente los elementos cuando el estado cambia.

LiveData y MutableLiveData: Observando datos para el ciclo de vida

`LiveData` es un contenedor de datos observables diseñado para funcionar con el ciclo de vida de Android (actividades y fragmentos).

`LiveData`: Dato inmutable observable

Ejemplo:

```
val usuario: LiveData<Usuario> = MutableLiveData()
```

LiveData y MutableLiveData: Observando datos para el ciclo de vida

`MutableLiveData`: Permite modificar el valor
Uso correcto:

```
private val _usuario = MutableLiveData<Usuario>()

val usuario: LiveData<Usuario> get() = _usuario


// Actualizar el valor

_usuario.value = nuevoUsuario
```

Razón de ser: `LiveData` permite que la UI observe los datos de forma segura, mientras que `MutableLiveData` permite modificarlos solo desde el `ViewModel`.

Uso incorrecto: Hacer que la UI modifique el `MutableLiveData` directamente.

LiveData y MutableLiveData: Observando datos para el ciclo de vida

```
class ContadorViewModel : ViewModel() {

    // MutableLiveData: puedo modificarlo desde aquí
    private val _contador = MutableLiveData<Int>(0)

    // LiveData: solo lectura para la UI
    val contador: LiveData<Int> = _contador

    fun aumentar() {
```

LiveData y MutableLiveData: Observando datos para el ciclo de vida

```
val viewModel: ContadorViewModel = viewModel()

// Observar los cambios del contador
viewModel.contador.observe(this) { valor ->
    textView.text = "Contador: $valor"
}

// Cuando pulsas un botón
button.setOnClickListener {
    viewModel.aumentar()
}
```

La **UI no toca el dato directamente**, solo lo observa.

El **ViewModel es el único que puede cambiarlo**.

Cada vez que el valor cambia (`_contador.value = ...`), **la UI se actualiza sola**.

StateFlow y MutableStateFlow: Flujo de datos reactivo con Kotlin Flow

`StateFlow` es una forma moderna y segura para trabajar con datos observables, reemplazando gradualmente a `LiveData`. Está basado en corrutinas y es más flexible.

`MutableStateFlow`: Un `StateFlow` que puede cambiar de valor

Ejemplo de uso:

```
private val _estado = MutableStateFlow(0)

val estado: StateFlow<Int> = _estado


fun incrementarContador() {

    _estado.value += 1

}
```

collectAsState: Convertir StateFlow a un estado de Compose

`collectAsState` permite observar un `StateFlow` dentro de un `Composable`, asegurando que la UI se actualice automáticamente.

Ejemplo de uso:

```
val valor = viewModel.estado.collectAsState().value

Text("Contador: $valor")
```

Convierte un StateFlow (flujo reactivo de datos) → en una variable que Compose puede observar y redibujar en pantalla cuando cambia.

SharedFlow: Manejo de eventos de un solo uso

```
private val  eventoNavegacion =
MutableSharedFlow<Unit>()

val eventoNavegacion = _eventoNavegacion.asSharedFlow()



fun navegar() {

    viewModelScope.launch {

        _eventoNavegacion.emit(Unit)

    }

}

LaunchedEffect(Unit) {

    viewModel.eventoNavegacion.collect {

        navController.navigate("detalle")

    }
```

`SharedFlow` es como `StateFlow`, pero:

- No guarda un “último estado”.
- Emite los valores solo a los que estén escuchando en ese momento.
- Es ideal para **eventos puntuales** (toasts, navegación, mensajes, etc.).