APUNTES DATA CLASS

1. Una **`data class`** es una clase especial en Kotlin utilizada para **almacenar datos.**
2. No se necesita escribir **constructores, setters ni getters** ni sobrescribir **toString(), equals() ni hashCode().**
3. Con una clase normal, `==` compararía la referencia de memoria, pero con una `data class`, se comparan los valores de sus propiedades.
4. HashCode() calcula un valor de hash único para la clase en función de sus propiedades. Se usa en Set, HashMap
5. toString() Muestra la información de la clase de forma legible, sin necesidad de sobrescribir el método
6. **copy() Crea una copia del objeto y te permite modificar solo las propiedades que necesites. La nueva instancia tiene una nueva referencia en memoria**. **Se usa:**
    1. **Cuando necesitas inmutabilidad y no quieres modificar el objeto original.**
    2. **Para hacer pequeñas modificaciones sin tener que crear una instancia manualmente.**
    3. **Ejemplo de uso: personaActual?.let { _persona.value = it.copy(edad = it.edad + 1) }**
7. Desestructuración de una data class. En Kotlin, puedes "extraer" los valores de las propiedades de una data class con una sola línea, gracias a la desestructuración. Ejemplo:

    ```
    val persona = Persona(id = 1, nombre = "Juan", edad = 30)

    // Desestructuramos en variables

    val (id, nombre, edad) = persona

    println(id) // 1

    println(nombre) // Juan

    println(edad) // 30
    ```

    También se puede usar en listas:

    ```
    for ((id, nombre, edad) in lista) {
        println("$nombre tiene $edad años")
    }
    ```

8. persona1 == persona2 Compara el contenido de las propiedades.
   persona1 === persona2 Compara la referencia de memoria.

FE DE ERRATAS. ACLARACIONES IMPORTANTES SOBRE DATA CLASS

1. Data class NO es inmutble. Ya que admite propiedades VAL y VAR. Solo las propiedades VAL son inmutables. Por convención, la mayoría de las data classes usan VAR en todas sus propiedades pero no es inmutable por el hecho de ser dataclass
2. Copy hace justo lo que dice. Una copia del objeto con otra referencia de la memoria. Por lo que no tiene la misma referencia que el objeto original.
3. La copia del objeto puede incluir modificaciones en varios atributos, el resto se mantienen.
4. Para que sirve? Evitar errores y asegurarnos de que nuestros objetos se mantienen inalterados.

Aclaración de dudas:

1. P: Si ya he cambiado el objeto con copy(), el original cambió?

   R: NO por eso se hace la copia, para mantener el original intacto

2. P: Si quiero hacer un cambio en el original y es inmutable ¿Cómo lo hago?

   R: Haciendo por ejemplo que el objeto original apunte al nuevo objeto.

   Situación típica. En una lista de objetos quiero modificar uno, pero son inmutables.

   Hago copy(), pero entonces ya es otro objeto, no el que está en la lista. Tengo que hacer que el objeto que está en la lista apunte al nuevo objeto recién creado.

3. P: Y en el ejemplo del ViewModel. Cambió el objeto original?

   R: Sí. Por esta razón:

```
// Reactivo con LiveData
   private val _persona = MutableLiveData(Persona())
   val persona: LiveData<Persona> get() = _persona
fun incrementarEdad(valor:Int) {
      _persona.value = _persona.value?.copy(edad = valor)
   }
```

   // Aquí ya se está haciendo coincidir a numero3 con _numero3. Por tanto lo que sucede es:

   1. Se crea la copia de persona
   2. Se asocia a _persona (persona es otro objeto diferente)
   3. Debido a esto: val persona: *LiveData<Persona> get() = _persona,* Cuando se quiera acceder a persona se estará devolviendo una referencia al objeto _persona
   4. Como _persona ahora tiene una referencia a memoria distinta, es cuando LiveData lo detecta y notifica a sus observadores. **LiveData notifica los cambios solo cuando cambia la referencia, no el contenido. Una de las razones (no la única) para crear objetos data class: Poder ser usados con LiveData**