Flow Operators provide a concise and effective way to handle your asynchronous events.

Understanding !!

1. State is an Immutable property which can’t be Updated through the UI. Purpose of the UI is just to Display the States.
2. Immutable part of State is constructed and managed inside ViewModel where we do Have either MutableStateOf or MutableStateFlowOf states.
3. :: “In kotlin this is callable Reference Operator” it is used to reference functions, properties and constructor as objects
    a. Function reference example

    ```
    fun greet(name : String ) = “hello,$name”
    fun main(){
    val greeter : ((String)->String) = ::greet
    println(greeter(kotlin))
    }
    ```

    b. Property reference example

    ```
        val name = “kotlin”
        val nameRef = ::name
        println(nameRef.get())
    ```

4. Flows : Hot and Cold Flows 😀
5. **Summary of `stateIn` Differences**

Feature	**Without `stateIn` (`Flow`)**	**With `stateIn` (`StateFlow`)**

Cold/Hot Behavior	Cold (starts fresh on each collect)	Hot (always active, holds last value)
Retains Last Value	No	Yes
Immediate Emission to New Collectors	No	Yes
Typical Use Case	One-time data fetch, simple streams	Continuous state management

6. By using `viewModelScope` and `SharingStarted.WhileSubscribed()`, you only keep `_contacts` active while there are active subscribers, minimizing unnecessary work when no one is observing it

7. combine()
    a. Combine function merge Multiple stateflow objects (_states, _sortType, _contacts) into a single StateFlow called state 😀
    b. Due to this if any one these change combine function is called again and the state is constructed with new values.
    c. Since combine function return Flow<R> which is cold flow to convert this flow into hot flow we use stateIn

8. Potential Problem with Collecting Cold Flow directly in UI
    a. Harek collector haru ko lagin new execution hunxa different network call, database operation etc.
    b. Yedi recomposition vayo vane whole process start again hunxa
    c. Life cycle awareness hudaina

d. Does not retain last values

   i. A **cold `Flow`** in Kotlin does *not* retain its last emitted value. Instead, a cold flow replays its emissions from scratch each time it is collected. This means:

   ii. **No Last Value Storage**: When no one is collecting a cold flow, it does not hold onto its last value. As soon as a new collector starts, the flow begins producing emissions from the start, potentially triggering fresh operations (like a database query or network call).

   iii. **New Collection on Each Subscriber**: Because it doesn't retain the last value, each time a new subscriber collects the flow, the flow restarts its emission from the beginning.

   iv. **Cold `Flow`**: Since a cold flow doesn't retain any values when it's not being collected, it does not occupy memory to store emitted values. It simply defines a sequence of operations that are executed each time a collector starts, without holding any of its past emissions. So, when it's not actively in use, it effectively has a minimal memory footprint.

   v. **`StateFlow`**: `StateFlow`, on the other hand, always retains its most recent emission in memory, ensuring it can instantly provide that value to new collectors. This retained value does occupy some memory, but it's usually small and only the latest value is stored.