Lecture 11

COMP 3717- Mobile Dev with Android Tech

rememberSaveable

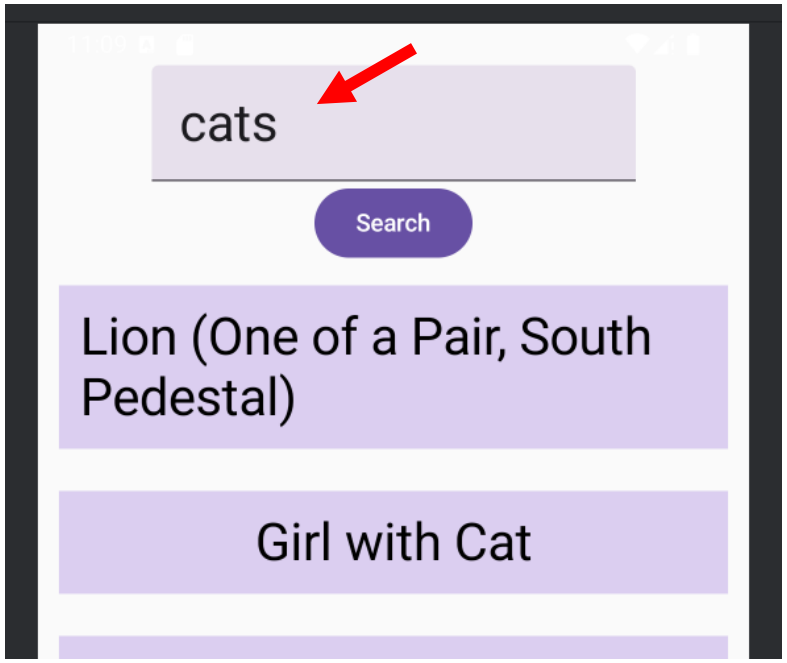
- rememberSaveable works like the remember composable

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

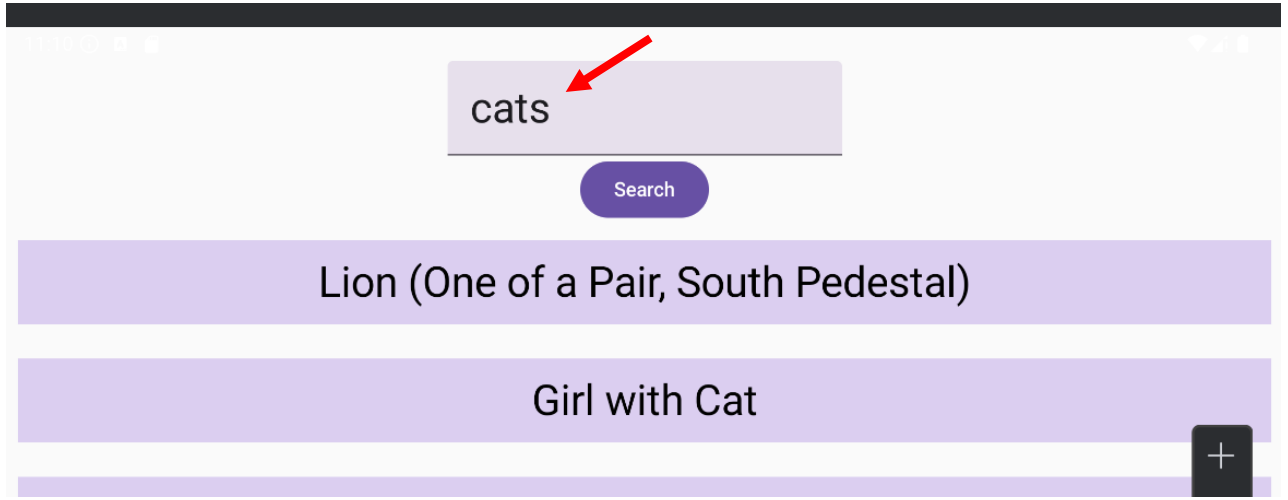
- The difference is that the state will also be remembered across configuration changes

rememberSaveable (cont.)

Portrait

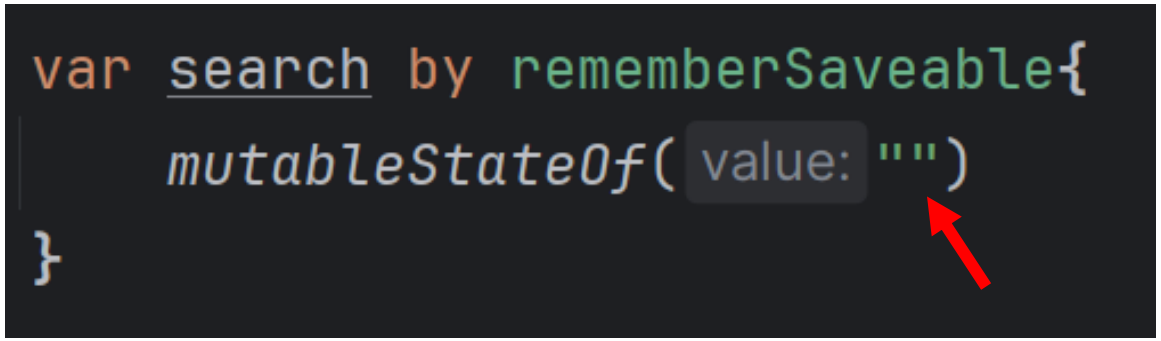


Landscape



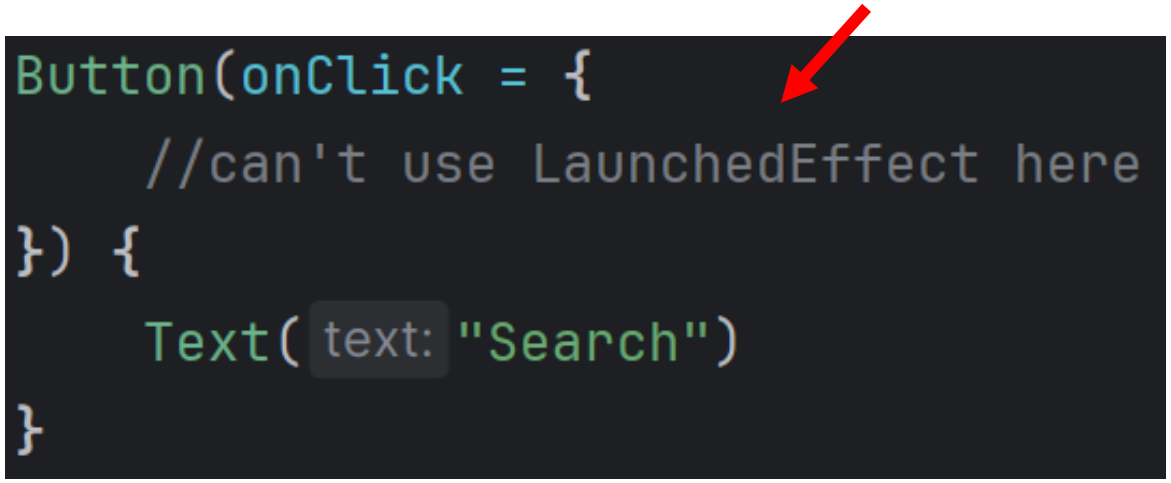
rememberSaveable (cont.)

- *rememberSavable* can only save state across configuration changes for types that can be stored in a Bundle
  - Primitives and Strings



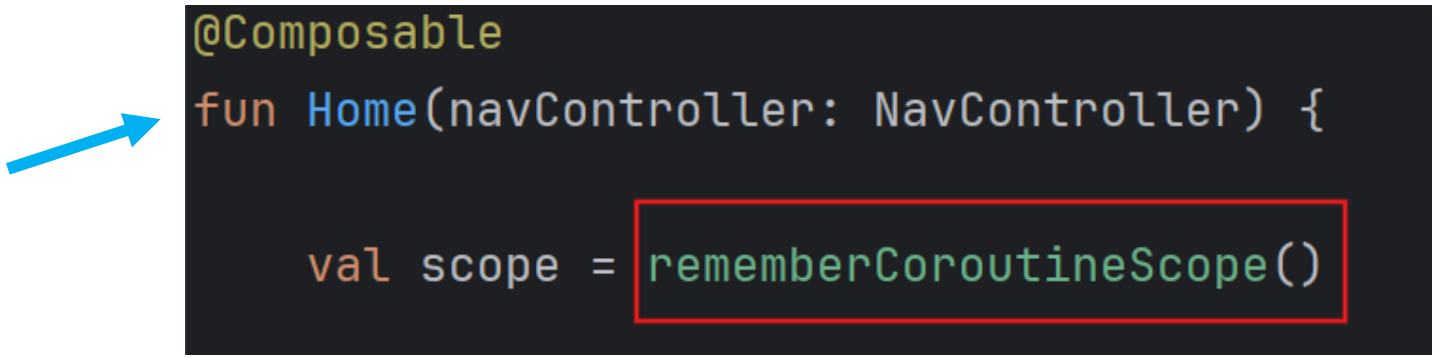
rememberCoroutineScope

- Often, we need to launch a coroutine that is not within a composable directly
  - i.e. a button's onClick event



rememberCoroutineScope (cont.)

- *rememberCoroutineScope* is a composable that returns a *CoroutineScope* that is bound to its parent's lifecycle



- If the parent leaves composition, all coroutines using this scope will be cancelled

rememberCoroutineScope (cont.)

- We can then use the scope to launch coroutines within callback events

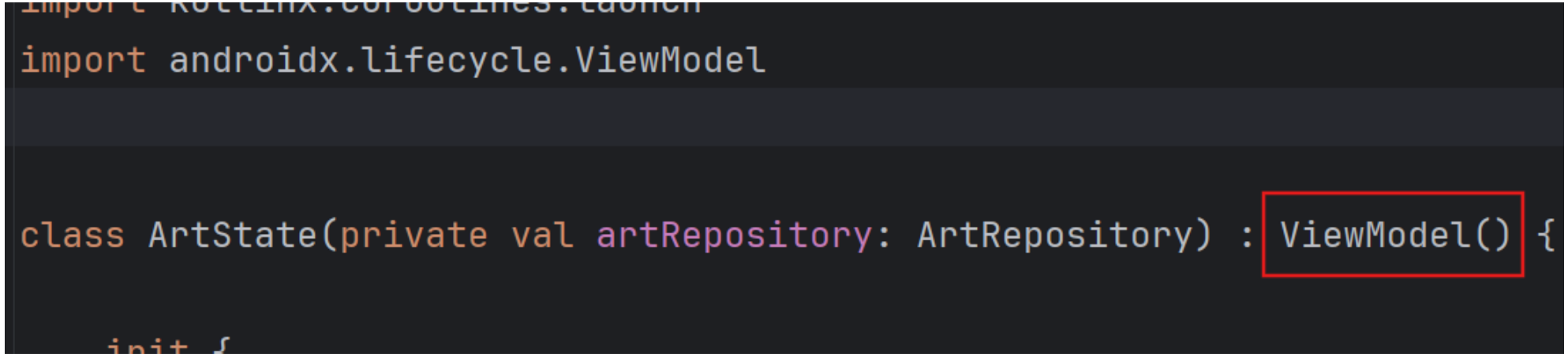
```
Button(onClick = {
    scope.launch {
        artState.search(search)
    }
}) {
    Text( text: "Search")
}
```

ViewModel

- A *ViewModel* is a type of state holder that is lifecycle aware
  - Survives configuration changes
- It is bound to the activity
  - We can share data easily across entire activity
- Allows us to use launch coroutines within its own scope
- Integrates well with other jetpack libraries

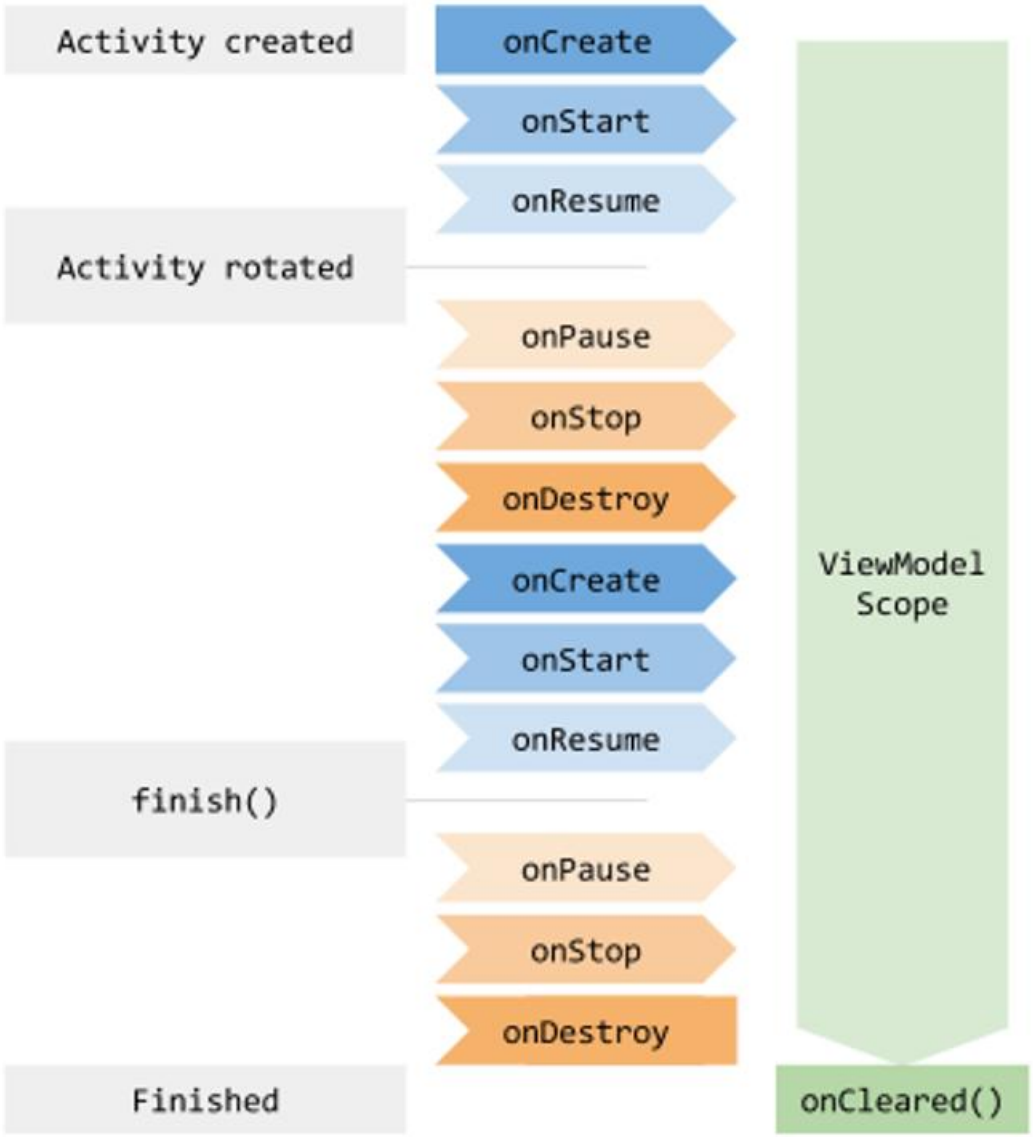
ViewModel (cont.)

- To make a class a ViewModel, extend the *ViewModel* class



ViewModel (cont.)

- A *ViewModelStore* object is retained through configuration changes
- When creating a ViewModel, we scope it to a *ViewModelStoreOwner*
  - Activity (default)
  - NavBackStackEntry
    - Useful when using Navigation



ViewModel (cont.)

- To create one, we use the composable *viewModel*

```
setContent {

    val artState = viewModel{
        ArtState(artRepository)
    }
```

- You will need the navigation dependency to use this

```
dependencies {

    implementation("androidx.navigation:navigation-compose:2.8.8")
```

ViewModel (cont.)

- Once it’s created, we return the existing one by providing the same *viewModelStoreOwner*
- Which in this case, is the *Activity*

```
@Composable
fun Search(value:String, onValueChange:(String)->Unit){
    //returns the existing ArtState viewModel
    val artState:ArtState = viewModel(LocalActivity.current as ComponentActivity)
    TextField(
```

ViewModel (cont.)

- This is useful because we can now share state across multiple composables and destinations
  - No need to pass the state down the tree

- If you are using a *ViewModel*, do not pass it into other composables
  - This defeats the purpose of using one

ViewModel (cont.)

- To launch a coroutine within the scope of the ViewModel use *viewModelScope*

ViewModel (cont.)

- Coroutines using *viewModelScope* will only be cancelled if
  - The Activity is destroyed
    - Not including a configuration change
  - They are cancelled manually


GUESS WHAT ?
BREAK TIME !
memegenerator.net

Flows

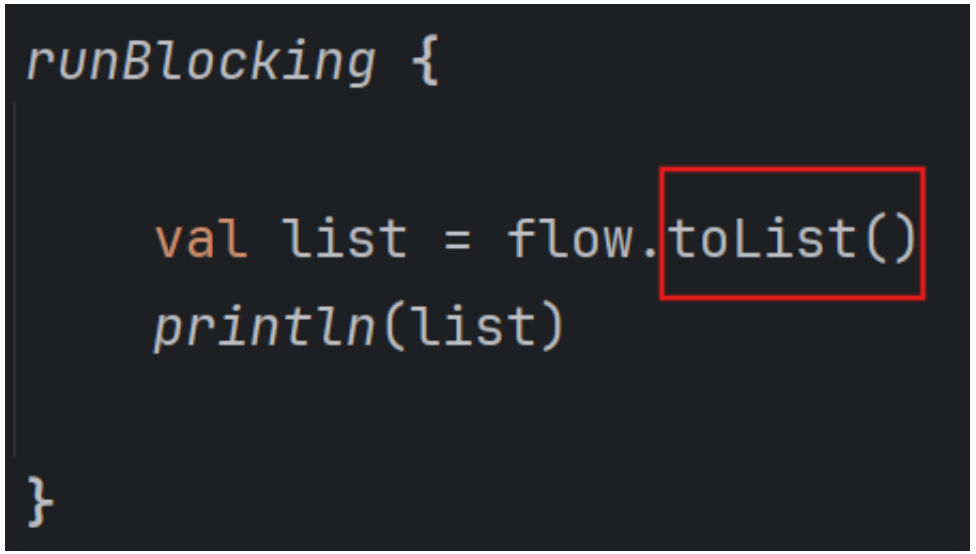
- A flow is like a collection, but its elements are processed lazily, and we consume the elements asynchronously

```
val flow = flowOf( ...elements: 1,2,3,4)
```

Flows (cont.)

- To consume the elements in a flow we use a terminal operation

- A terminal operation is anything that iterates the values
  - *toList, sum, count,* etc



Flows (cont.)

- All terminal operations are suspend functions

```
public suspend fun <T> Flow<T>.toList(
    destination: MutableList<T> = ArrayList()
): List<T>
```

Flows (cont.)

- The most common terminal operation is collect

```
val flow = flowOf( ...elements: 1, 2, 3, 4, 5)

runBlocking {
    flow.collect {

    }
}
```

Flows (cont.)

- *collect* allows us to *subscribe* to a flow and perform logic on each consumed element

```
runBlocking {
    flow.collect {
        println("Printing each element in the flow: $it")
        delay( timeMillis: 1000L)
    }
}
```

Flows (cont.)

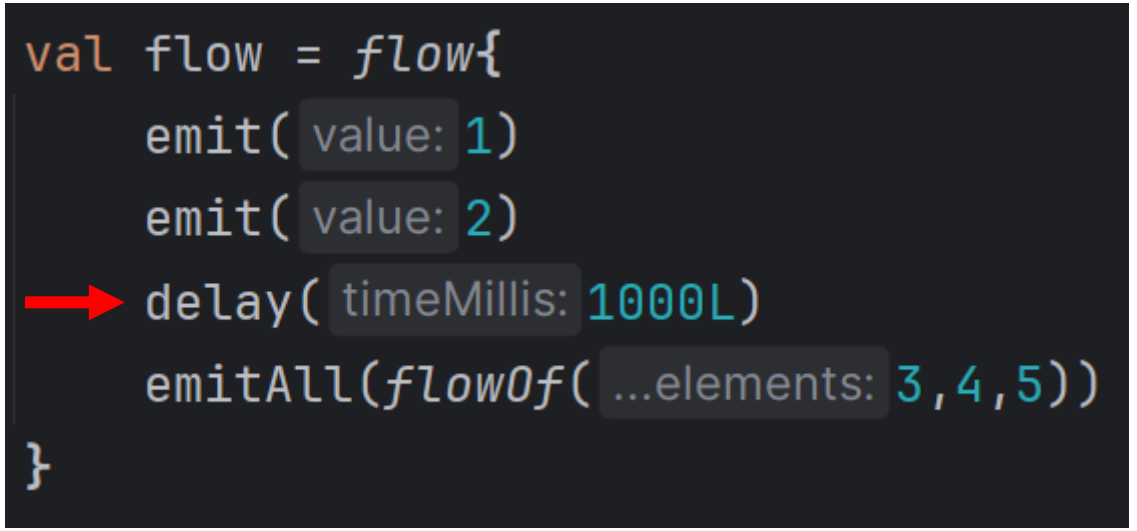
- We can also create the same flow with a *flow function*

```
val flow = flow{
    emit( value: 1)
    emit( value: 2)
    emitAll(flowOf( ...elements: 3,4,5))
}
```

- We use *emit* & *emitAll* to add elements into our flow

Flows (cont.)

- The flow function allows us to emit elements along side other suspending operations



Flows (cont.)

- The flow function is an example of a *cold flow*

- Cold flow
  - Elements only emit if a collector is collecting
  - Each collector collects its own instance of elements

- Hot flow
  - Elements emit independently of collectors (aka. always active)
  - Emissions are shared across all subscribers

Cold Flow

- Notice that each Collector prints a different random value
  - Each collector collects its own instance of elements

```
val coldFlow = flow{
    emit( value = Random.nextInt( until = 100))
}

runBlocking {
    launch {
        coldFlow.collect {
            println("Collector 1: $it")
        }
    }
    launch {
        coldFlow.collect {
            println("Collector 2: $it")
        }
    }
}
```

Hot Flow

- Notice that each Collector prints the same random value
  - Emissions are shared across all subscribers

```
val hotFlow = MutableSharedFlow<Int>()

runBlocking {
    launch {
        hotFlow.collect {
            println("Collector 1: $it")
        }
    }
    launch {
        hotFlow.collect {
            println("Collector 2: $it")
        }
    }
    launch {
        hotFlow.emit( value = Random.nextInt( until = 100))
    }
}
```

Hot Flow (cont.)

- Notice the Collector misses this emission
  - Elements emit independently of collectors (aka. always active)

```
val hotFlow = MutableSharedFlow<Int>()

runBlocking {

    hotFlow.emit( value = Random.nextInt( until = 100))

    launch {
        hotFlow.collect {
            delay( timeMillis = 1000L)
            println("Collector: $it")
        }
    }
}
```

Hot Flow (cont.)

- We can avoid misses by increasing the replay value
  - This will keep a certain number of emissions in a cache
  - Benefits late collectors

```
val hotFlow = MutableSharedFlow<Int>(replay = 1)

runBlocking {

    hotFlow.emit( value = Random.nextInt( until = 100))

    launch {
        hotFlow.collect {
            delay( timeMillis = 1000L)
            println("Collector: $it")
        }
    }
}
```

StateFlow

- A *StateFlow* is another example of a hot flow

```
class ArtState(private val artRepository: ArtRepository) : ViewModel() {

    var searchFlow = MutableStateFlow( value: "")
```

StateFlow (cont.)

- *collectAsState* allows us to subscribe to the flow as *MutableState*
- Setting the value, emits new data into the flow

StateFlow (cont.)

- We can also subscribe and perform a search request for each emission

```
private fun collectSearchInputs(){
    viewModelScope.launch {
        searchFlow
            .collect{
                artwork = artRepository.search(searchFlow.value)
            }
    }
}
```

StateFlow (cont.)

- To avoid sending too many requests at once, the debounce property comes in handy

- Emissions will need to stop for a certain amount time before being collection

Flows (cont.)

- When a flow might be useful
  - Your API supports data streaming
  - You want to receive periodic updates from an API
  - Receiving real time updates from a database (firebase, mongodb)
  - Debouncing; wait until input stops before sending a server request