Lecture 11

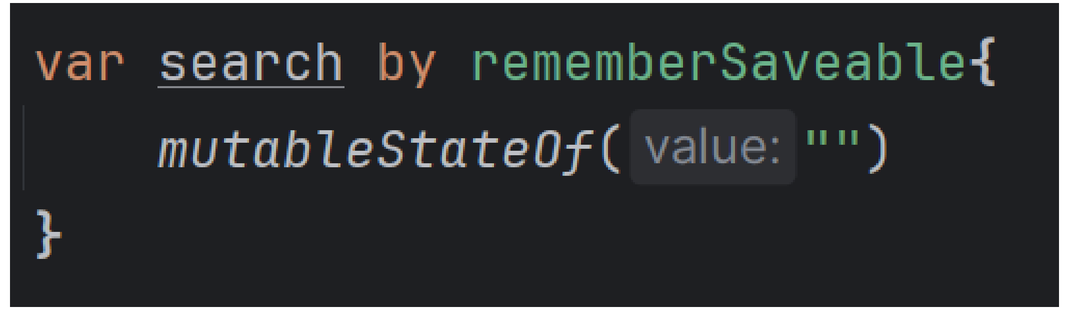
COMP 3717- Mobile Dev with Android Tech

第11讲

COMP 3717 - 使用Android技术进行移动开发

rememberSaveable

- rememberSaveable works like the remember composable

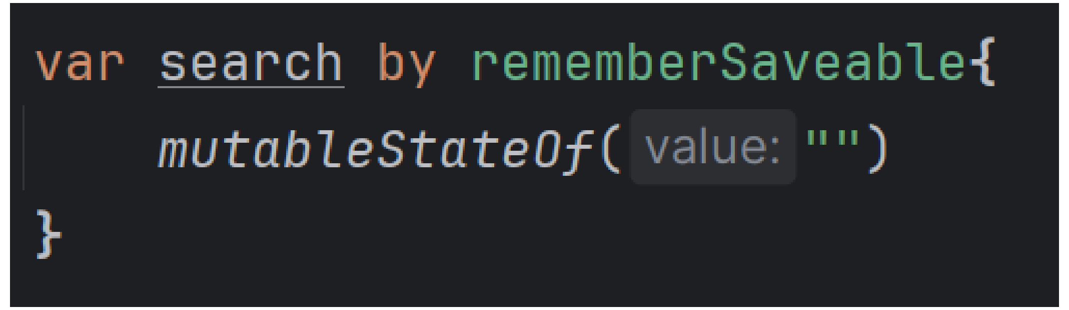


```
var search by rememberSaveable{
    mutableStateOf( value: "")
}
```

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

rememberSaveable

- rememberSaveable 的工作方式类似于 remember 可组合项

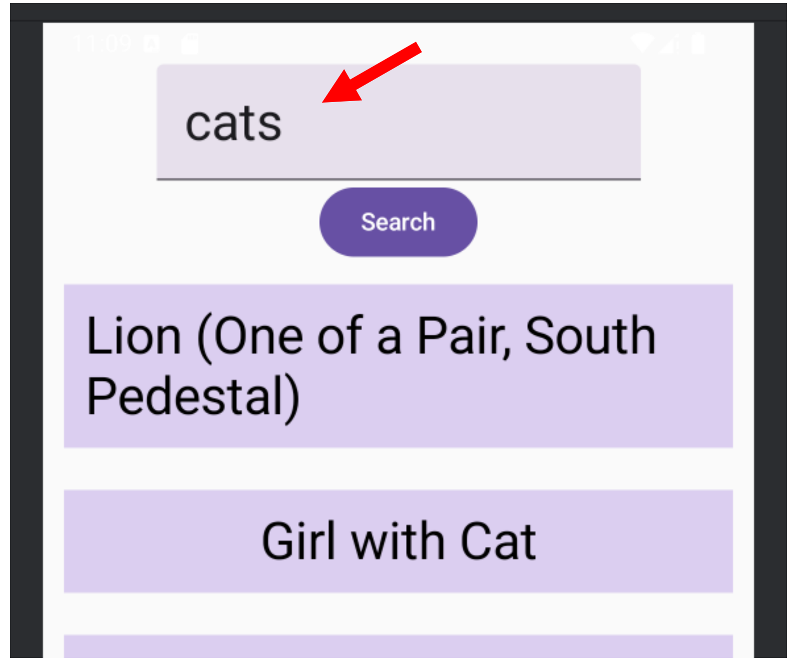


```
var search by rememberSaveable{
    mutableStateOf( value: "")
}
```

- 不同之处在于，状态也将在配置更改期间被保留

rememberSaveable (cont.)

Portrait

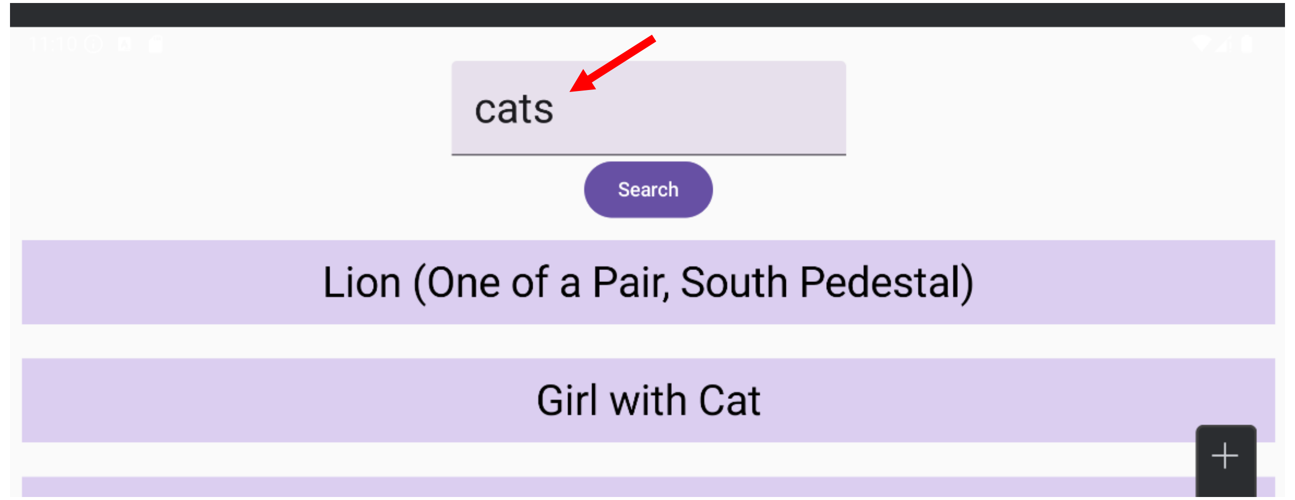


cats

Lion (One of a Pair, South Pedestal)

Girl with Cat

Landscape



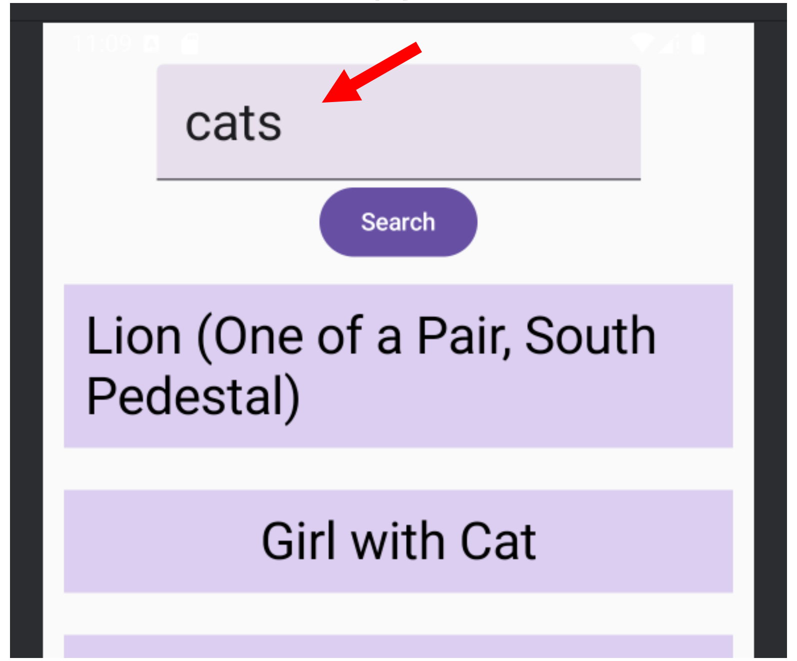
cats

Lion (One of a Pair, South Pedestal)

Girl with Cat

rememberSaveable（续）

纵向

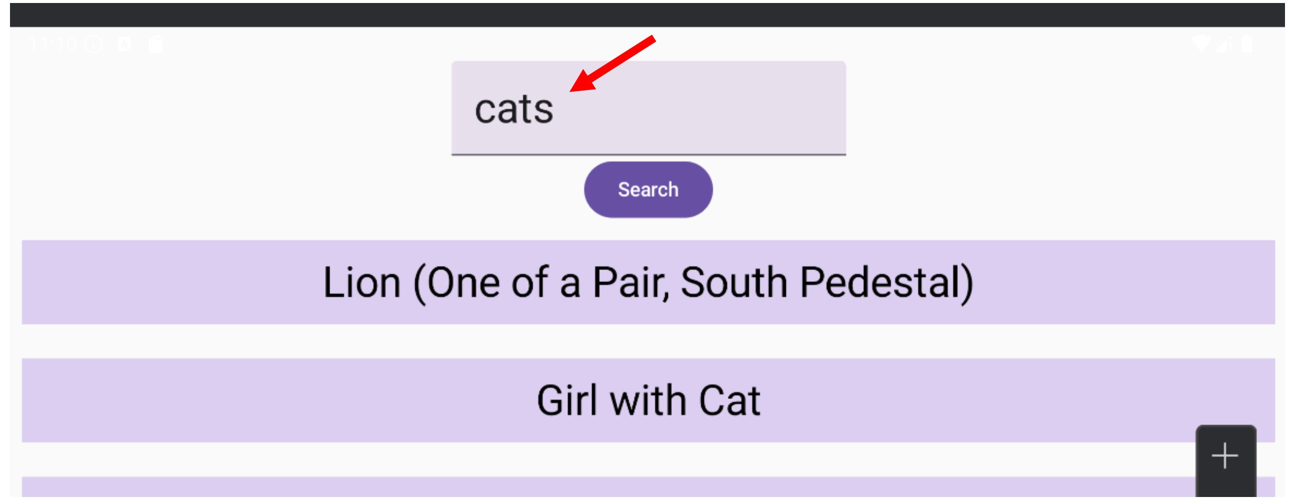


cats

Lion (One of a Pair, South Pedestal)

Girl with Cat

横向



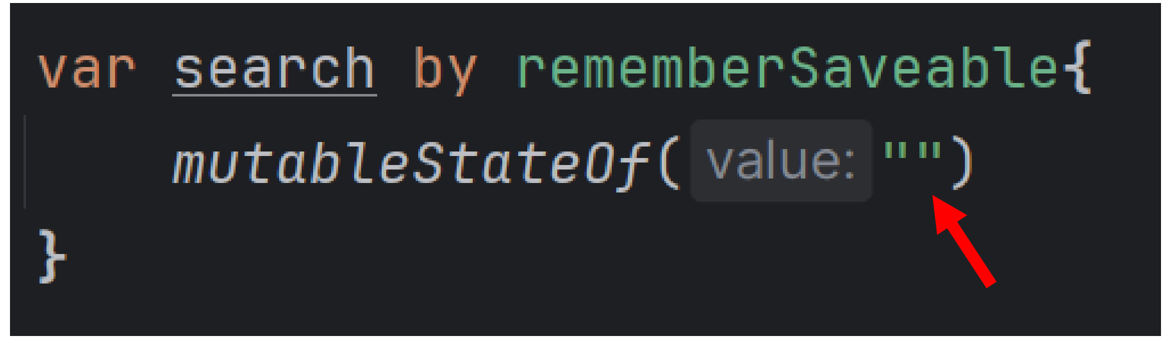
cats

Lion (One of a Pair, South Pedestal)

Girl with Cat

rememberSaveable (cont.)

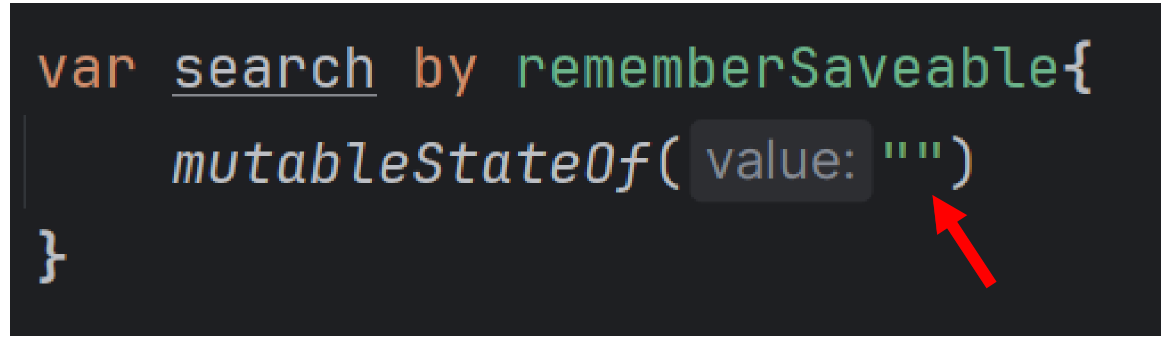
- *rememberSavable* can only save state across configuration changes for types that can be stored in a Bundle
  - Primitives and Strings



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

rememberSaveable（续）

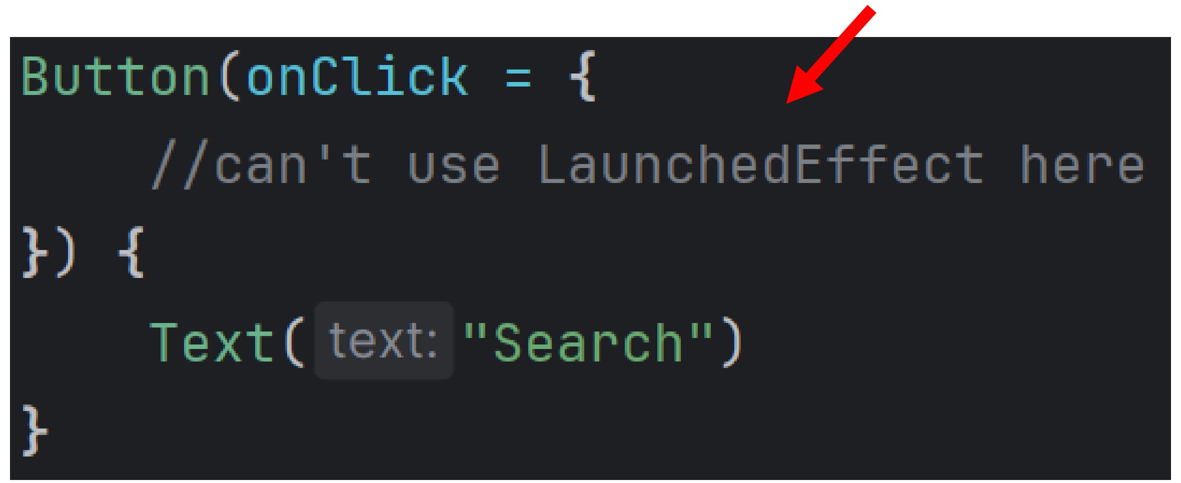
- *rememberSavable* 只能在配置更改期间保存可存储在 Bundle 中的类型的状态types that can be stored in a Bundle
  - 基本类型和字符串



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

rememberCoroutineScope

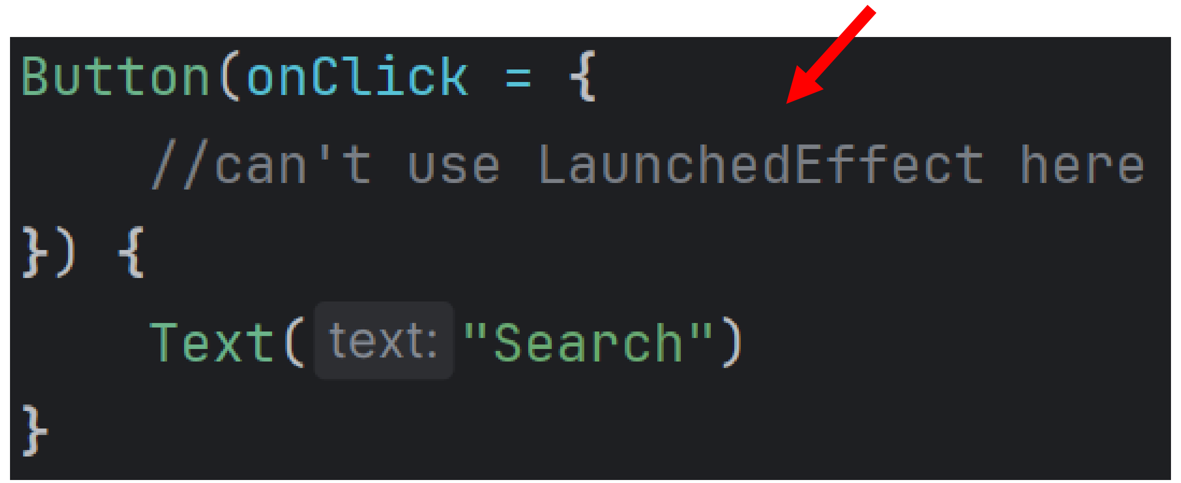
- Often, we need to launch a coroutine that is not within a composable directly
  - i.e. a button’s onClick event



```
Button(onClick = {
    //can't use LaunchedEffect here
}) {
    Text( text: "Search")
}
```

rememberCoroutineScope

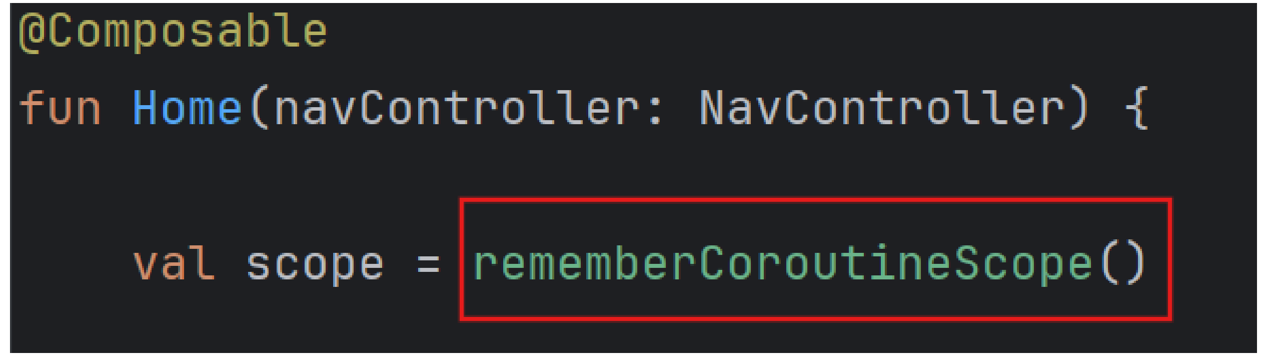
- 通常，我们需要启动一个不在可组合函数内部的协程直接
  - 例如按钮的 onClick 事件



```
Button(onClick = {
    //can't use LaunchedEffect here
}) {
    Text( text: "Search")
}
```

rememberCoroutineScope (cont.)

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

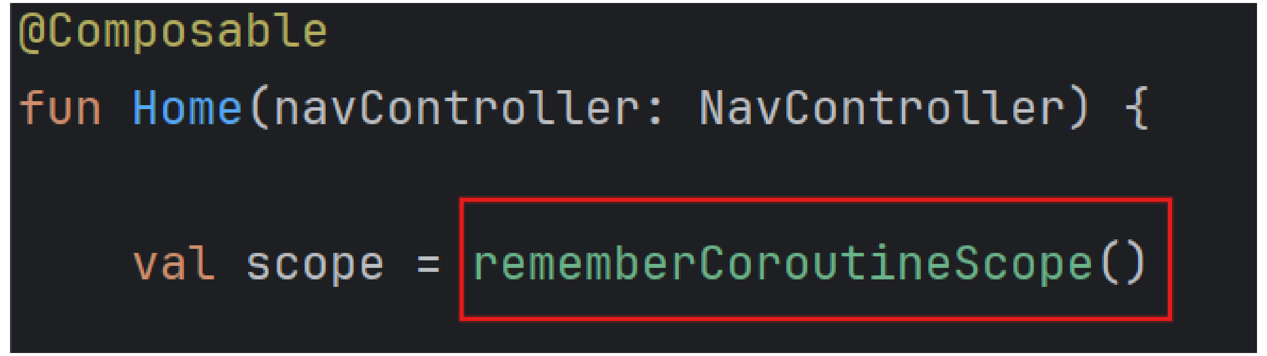


```
@Composable
fun Home(navController: NavController) {

    val scope = rememberCoroutineScope()
```

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

rememberCoroutineScope（续）

- *rememberCoroutineScope* 是一个可组合函数，返回一个与父级生命周期绑定的 *CoroutineScope* ，该作用域 父级 生命周期

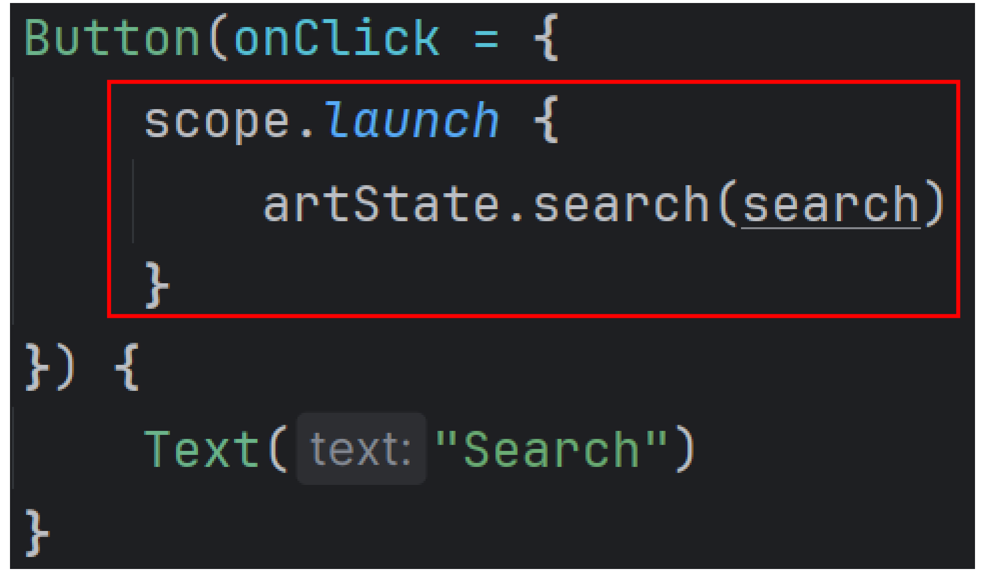


```
@Composable
fun Home(navController: NavController) {

    val scope = rememberCoroutineScope()
```

- 如果父级离开组合，所有使用此作用域的协程都将被取消

rememberCoroutineScope (cont.)

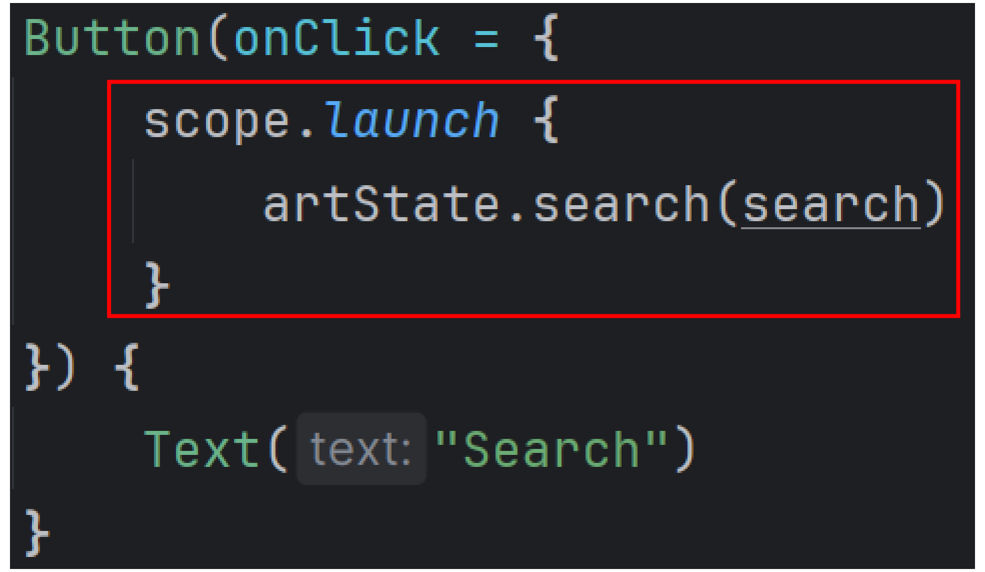
- We can then use the scope to launch coroutines within callback events



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

rememberCoroutineScope (续)

- 然后我们可以使用该作用域在回调事件中启动协程



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

ViewModel

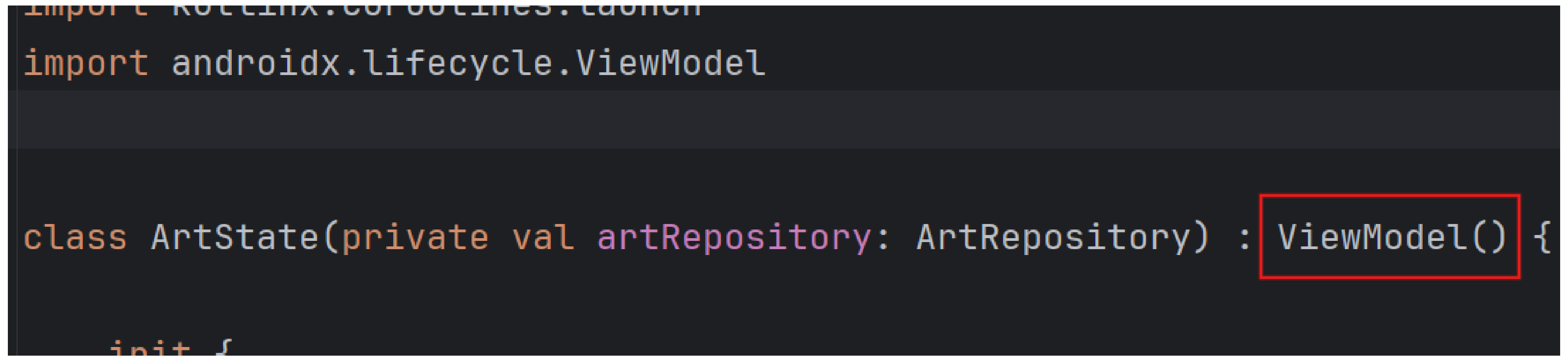
- A *ViewModel* is a type of state holder that is lifecycle aware
  - Survives configuration changes
- It is bound to the activity
  - We can share data easily across entire activity
- Allows us to use launch coroutines within its own scope
- Integrates well with other jetpack libraries

ViewModel

- *ViewModel* 是一种具有生命周期感知能力的状态持有器
  - 能够经受住配置更改
- 它与 Activity 绑定
  - 我们可以轻松地在整个 Activity 中共享数据
- 允许我们在其自身作用域内启动协程
- 与其他 Jetpack 库集成良好

ViewModel (cont.)

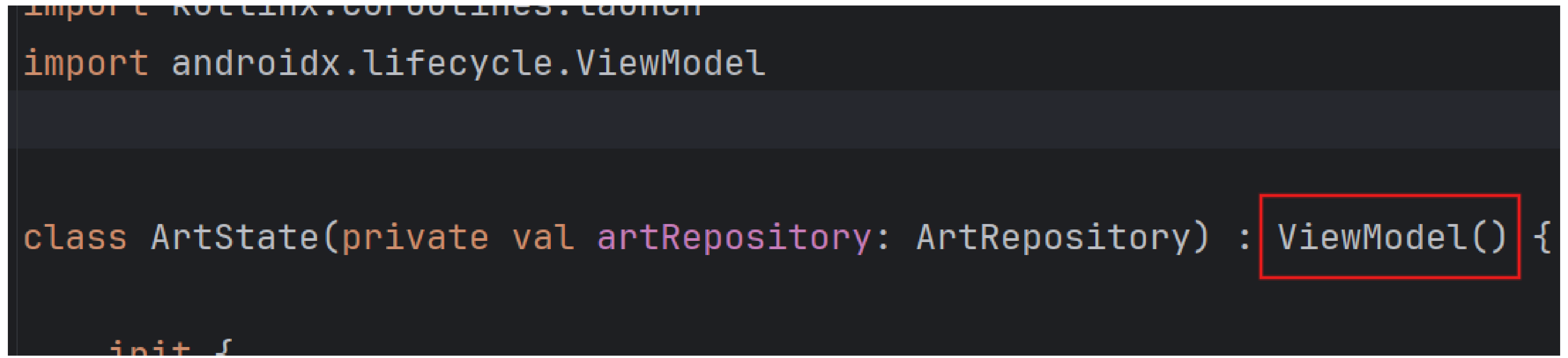
- To make a class a ViewModel, extend the *ViewModel* class



```
import kotlinx.coroutines.launch
import androidx.lifecycle.ViewModel

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

    init {
```

视图模型（续）

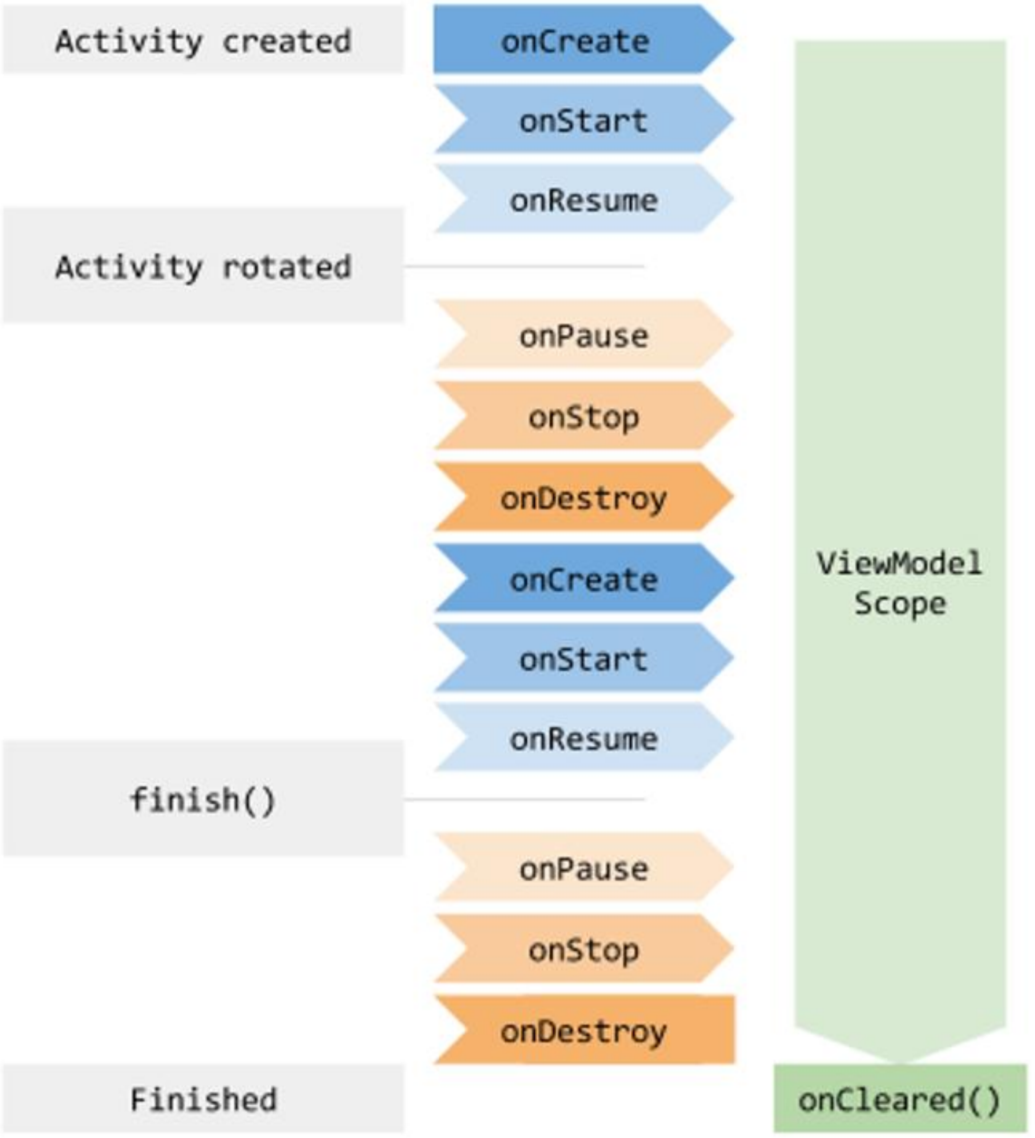
- 要将一个类作为 ViewModel，需继承 *ViewModel* 类



```
import kotlinx.coroutines.launch
import androidx.lifecycle.ViewModel

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

    init {
```

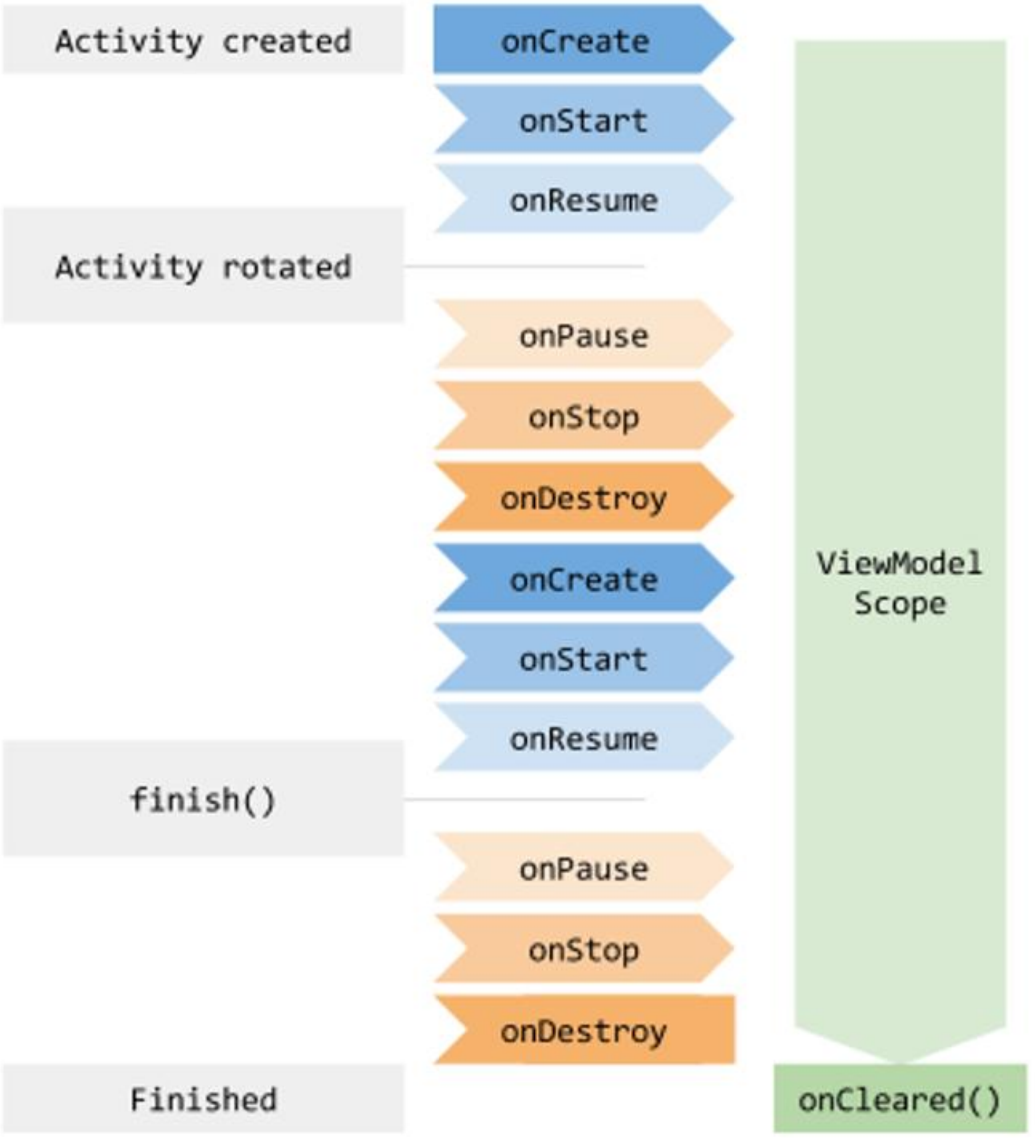
ViewModel (cont.)

- 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



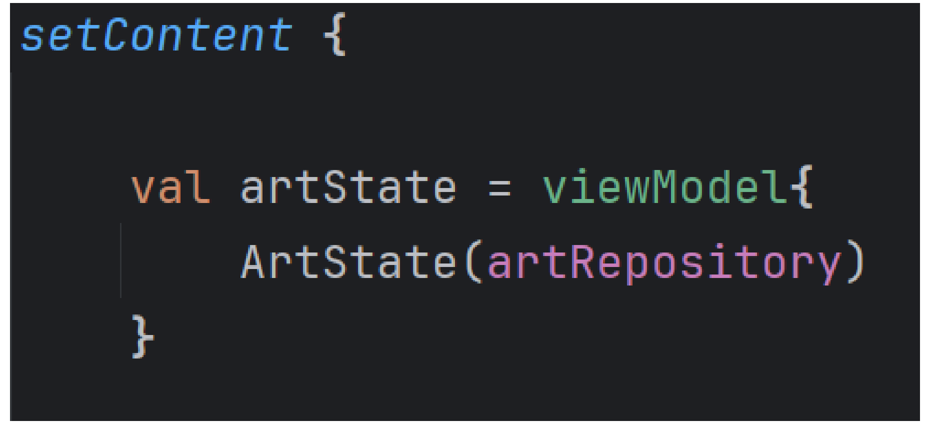
视图模型（续）

- *ViewModelStore* 对象在配置更改期间会被保留
- 创建 ViewModel 时，我们会将其限定于它所属的 ViewModelStoreOwner
  - Activity（默认）
  - NavBackStackEntry
    - 使用导航时很有用



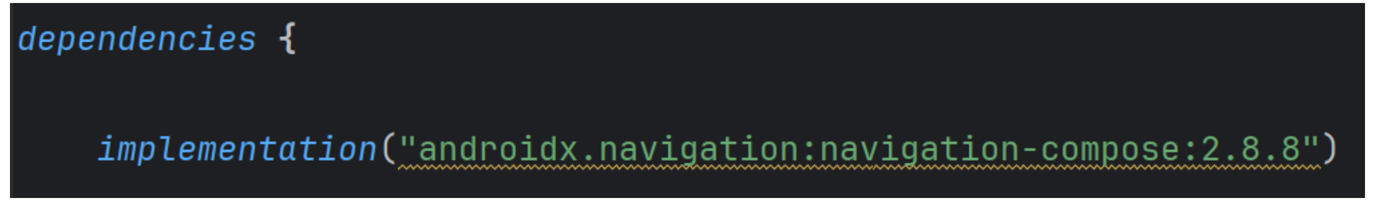
ViewModel (cont.)

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



```
setContent {

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

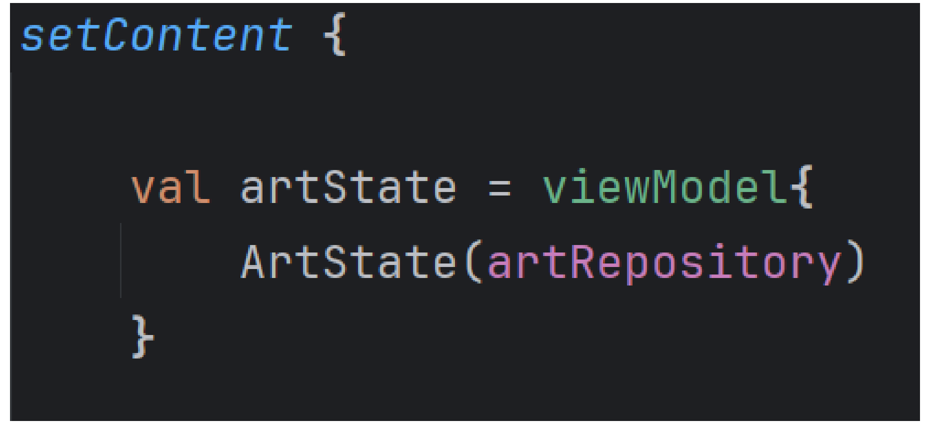
- You will need the navigation dependency to use this



```
dependencies {

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

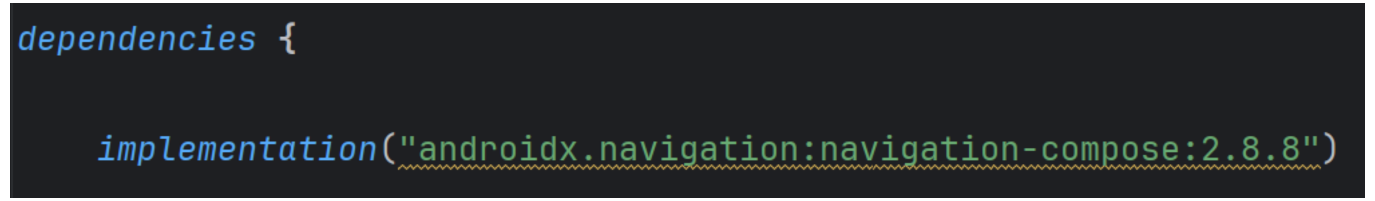
ViewModel（续）

- 要创建一个，我们使用可组合项 *viewModel*



```
setContent {

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

- 您需要导航依赖项才能使用此功能

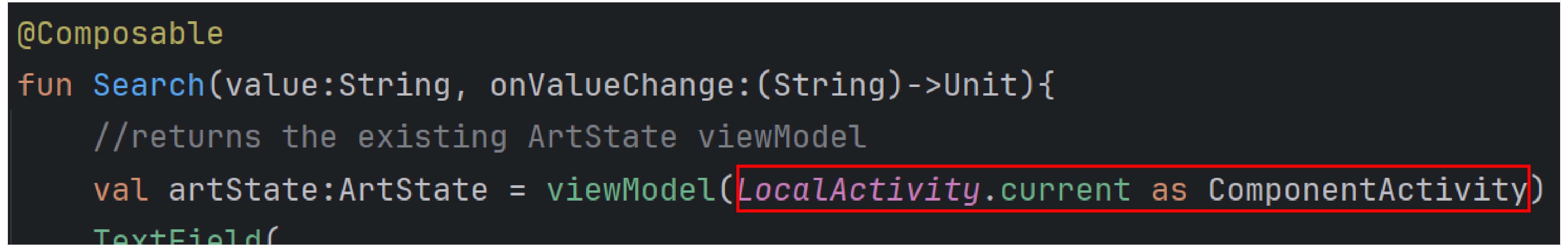


```
dependencies {

   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*

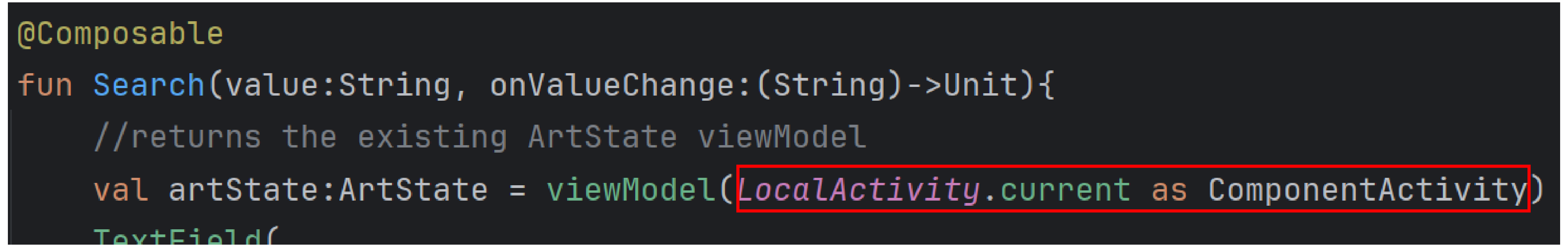


```
@Composable
fun Search(value:String, onValueChange:(String)->Unit){
    //returns the existing ArtState viewModel
    val artState:ArtState = viewModel(LocalActivity.current as ComponentActivity)
    TextField(
```

ViewModel（续）

- 创建后，我们通过提供相同的viewModelStoreOwner 来返回已存在的实例

- 在此情况下，即为 *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

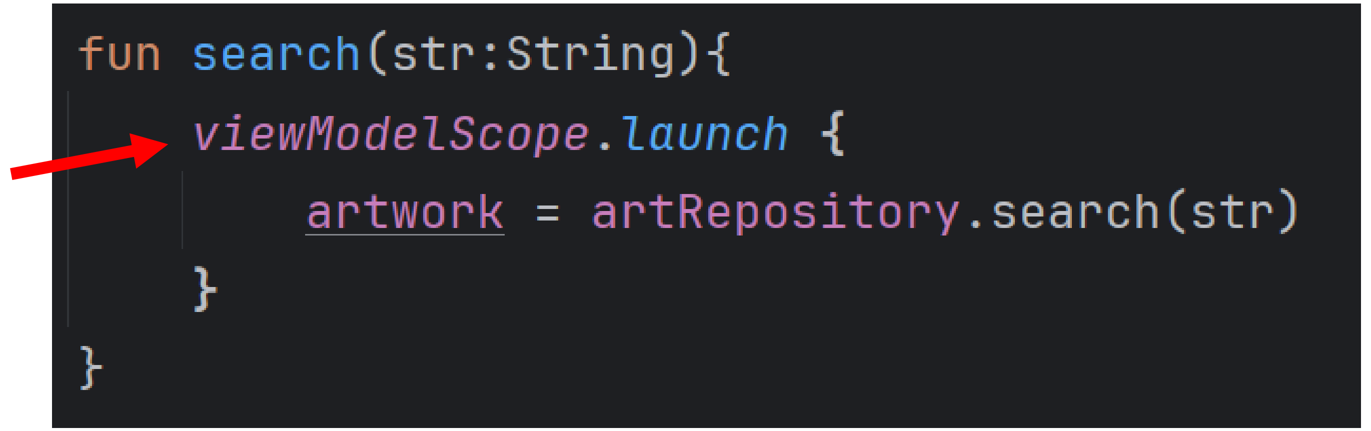
ViewModel（续）

- 这很有用，因为我们现在可以在多个可组合项和目的地之间共享状态
  - 无需将状态逐级向下传递

- 如果您正在使用 *ViewModel*，请不要将其传递给其他可组合项
  - 这样做就失去了使用 ViewModel 的意义

ViewModel (cont.)

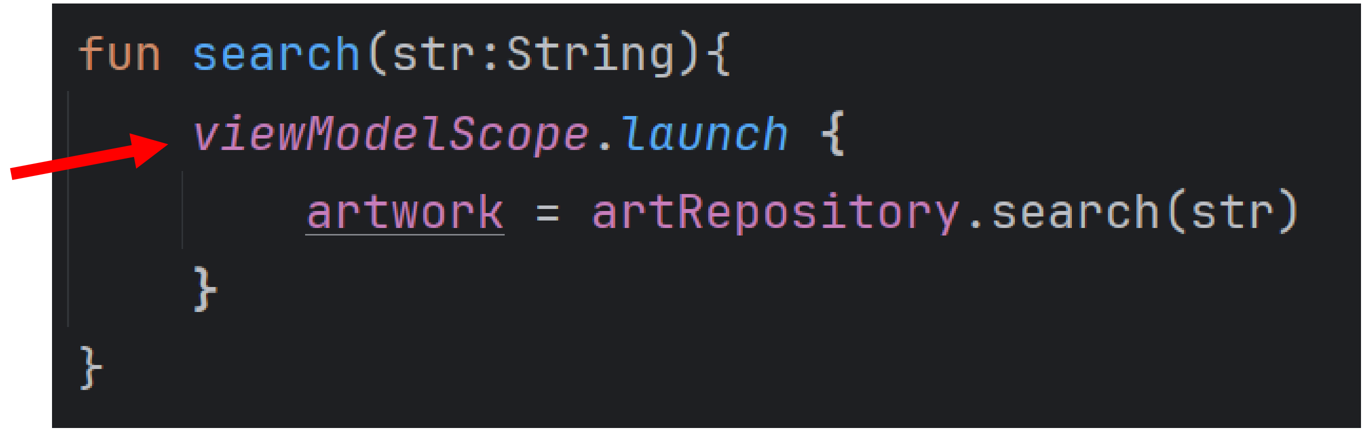
- To launch a coroutine within the scope of the ViewModel use *viewModelScope*



```
fun search(str:String){
    viewModelScope.launch {
        artwork = artRepository.search(str)
    }
}
```

ViewModel（续）

- 要在 ViewModel 的作用域内启动协程，请使用 viewModelScope



```
fun search(str:String){
    viewModelScope.launch {
        artwork = artRepository.search(str)
    }
}
```

ViewModel (cont.)

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

ViewModel（续）

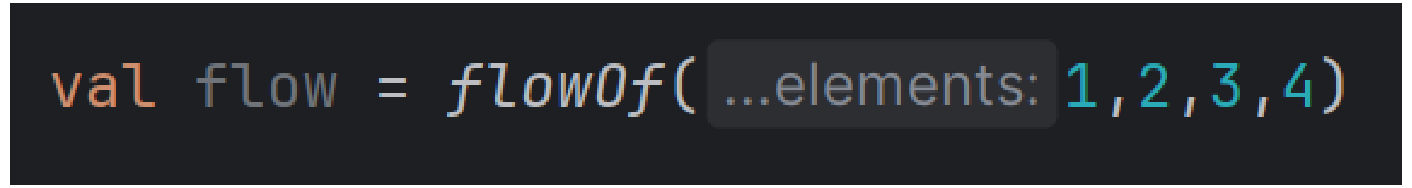
- 使用 *viewModelScope* 的协程仅在以下情况下被取消
  - Activity 被销毁时
    - 不包括配置更改
  - 它们被手动取消时

GUESS WHAT ?
BREAK TIME !
memegenerator.net

GUESS WHAT ?
BREAK TIME !
memegenerator.net

Flows

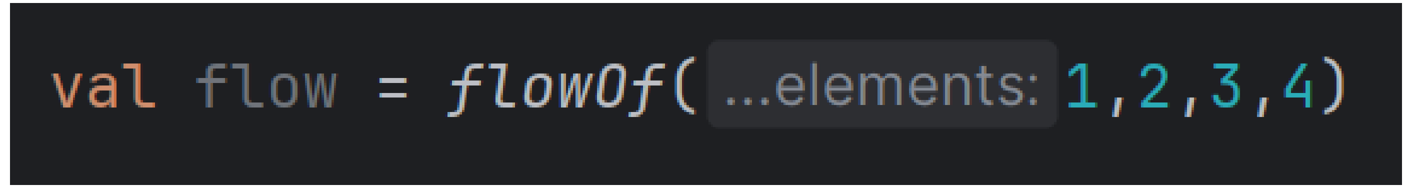
- A flow is like a collection, but its elements are processed lazily, and we consume the elements asynchronously



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

流

- 流类似于集合，但其元素是惰性处理的，且我们以异步方式消费这些元素

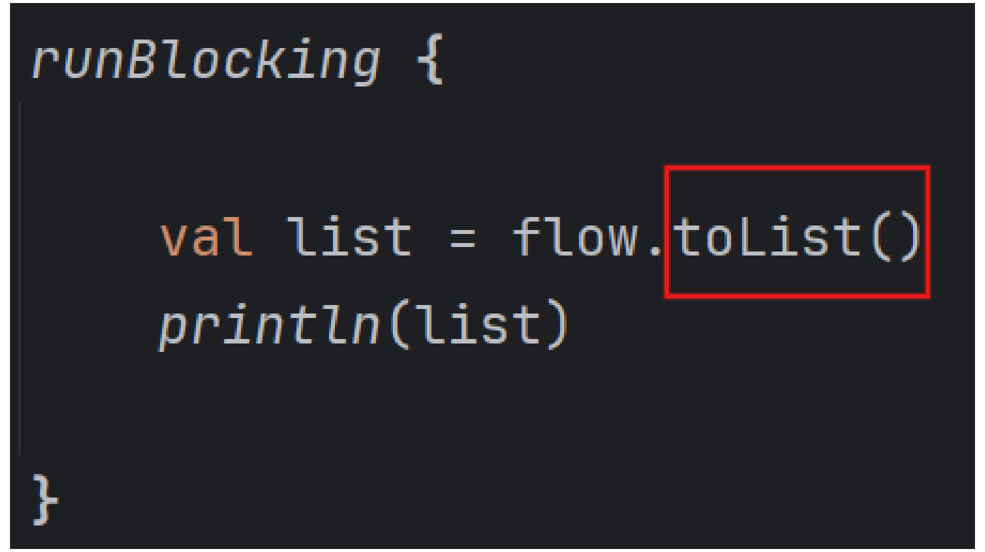


```
val flow = flowOf( ...elements: 1,2,3,4)
```

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

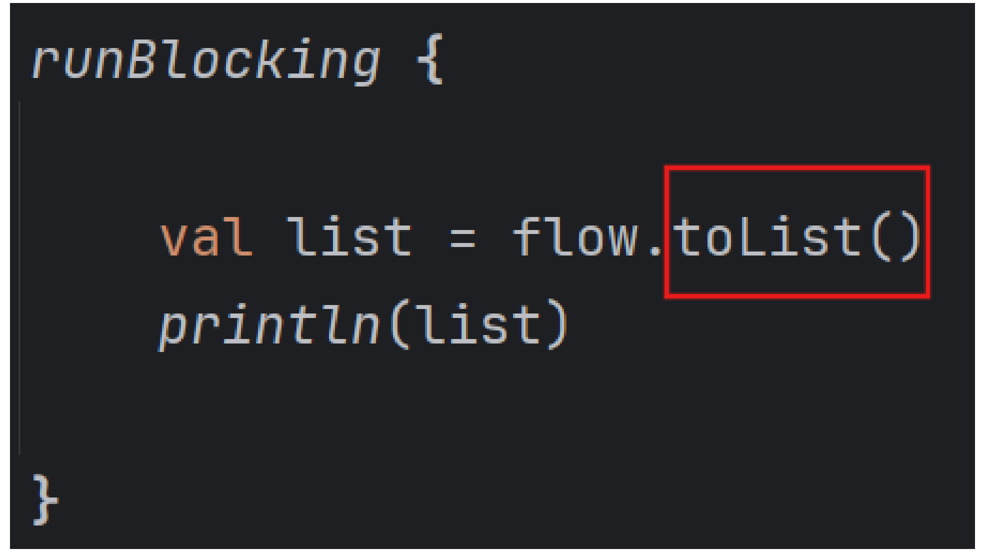


```
runBlocking {

    val list = flow.toList()
    println(list)

}
```

流（续）

- 要消费流中的元素，我们使用终端操作

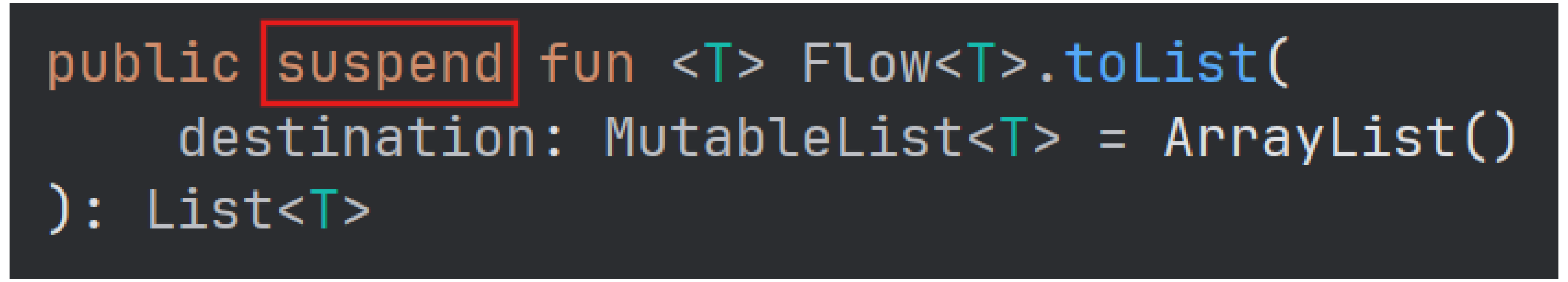
- 终端操作是指对值进行迭代的任何操作
  - toList ，*sum,* count ，etc



```
runBlocking {

    val list = flow.toList()
    println(list)

}
```

Flows (cont.)

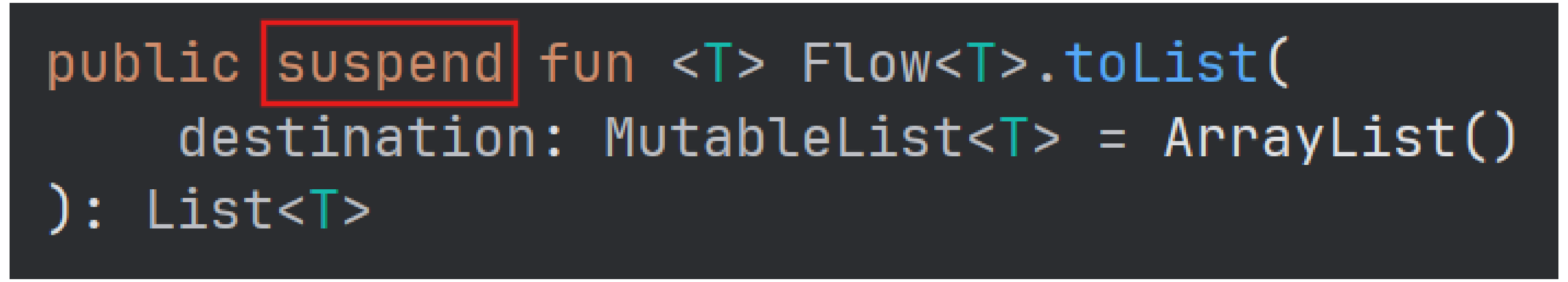
- All terminal operations are suspend functions



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

流（续）

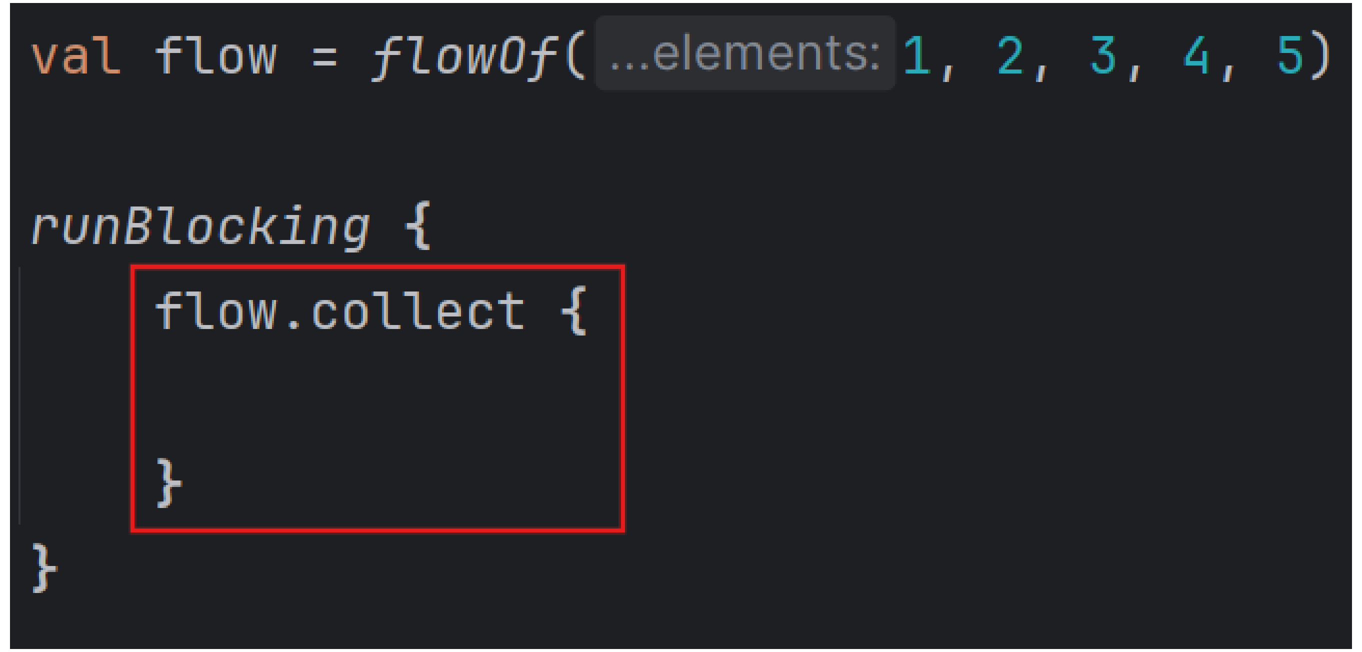
- 所有终端操作 都是 挂起函数



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

Flows (cont.)

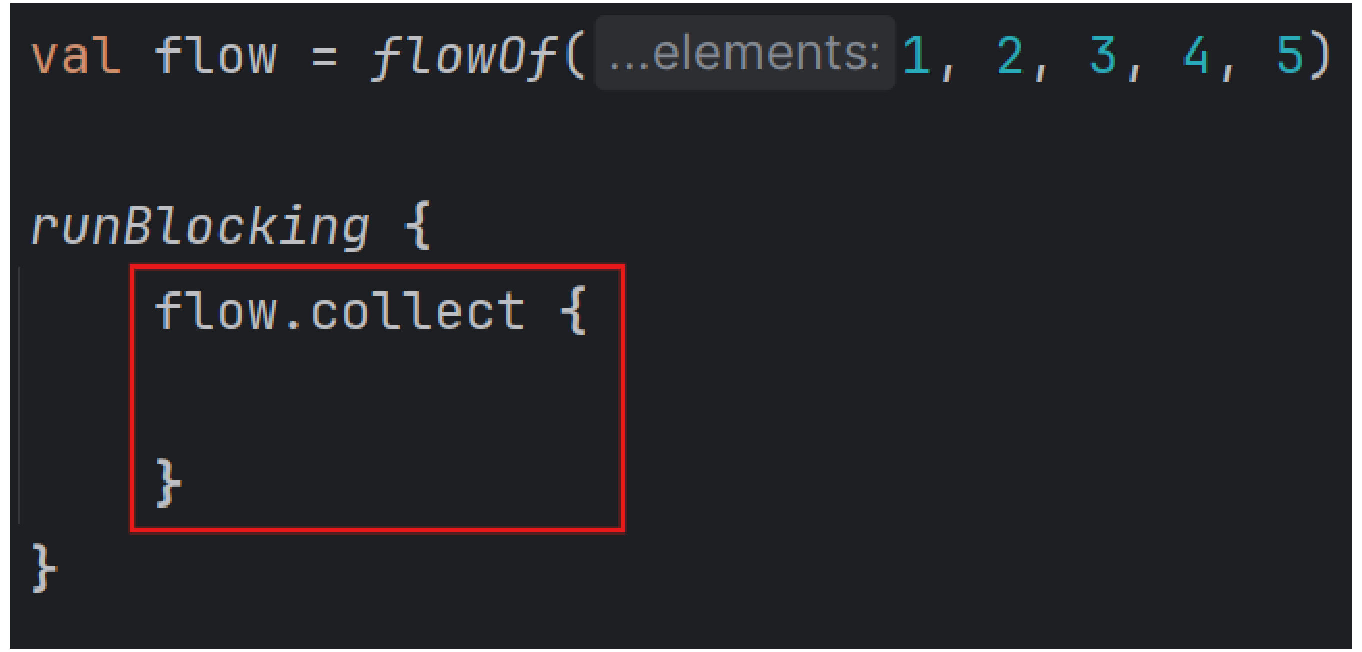
- The most common terminal operation is collect



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

runBlocking {
    flow.collect {

    }
}
```

流（续）

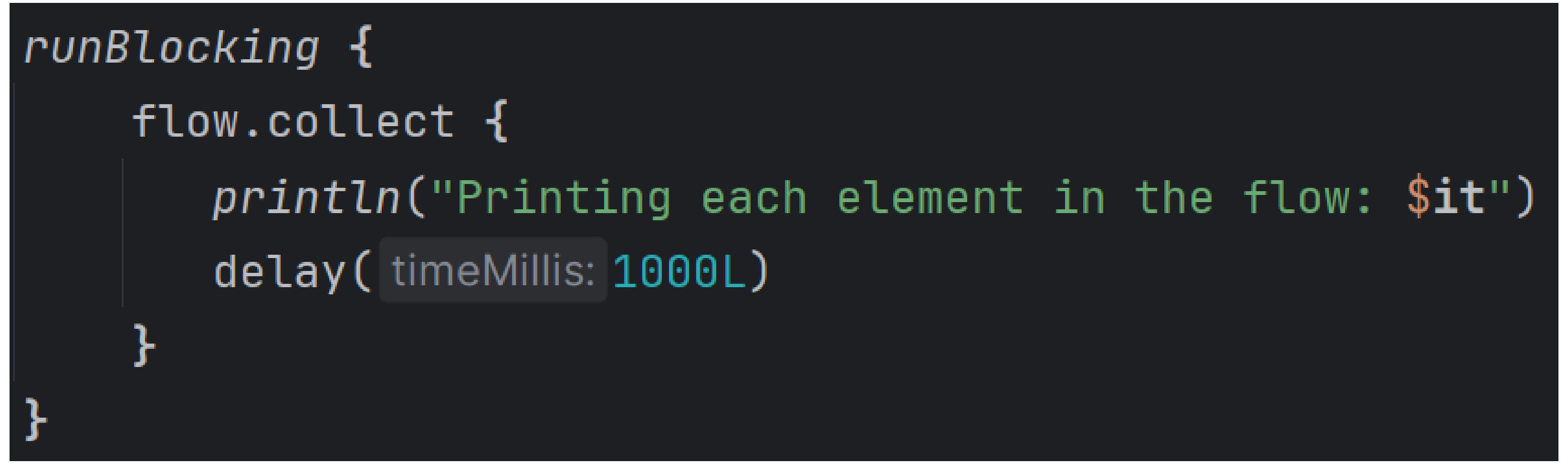
- 最常见的 终端操作 是 collect



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

runBlocking {
    flow.collect {

    }
}
```

Flows (cont.)

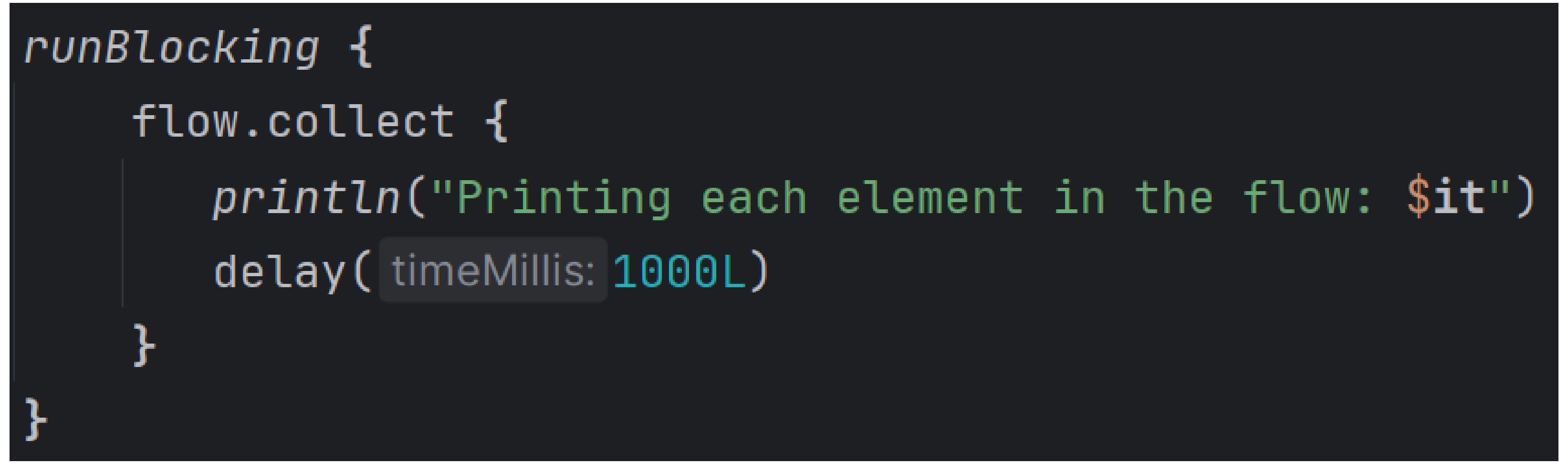
- *collect* allows us to *subscribe* to a flow and perform logic on each consumed element



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

流（续）

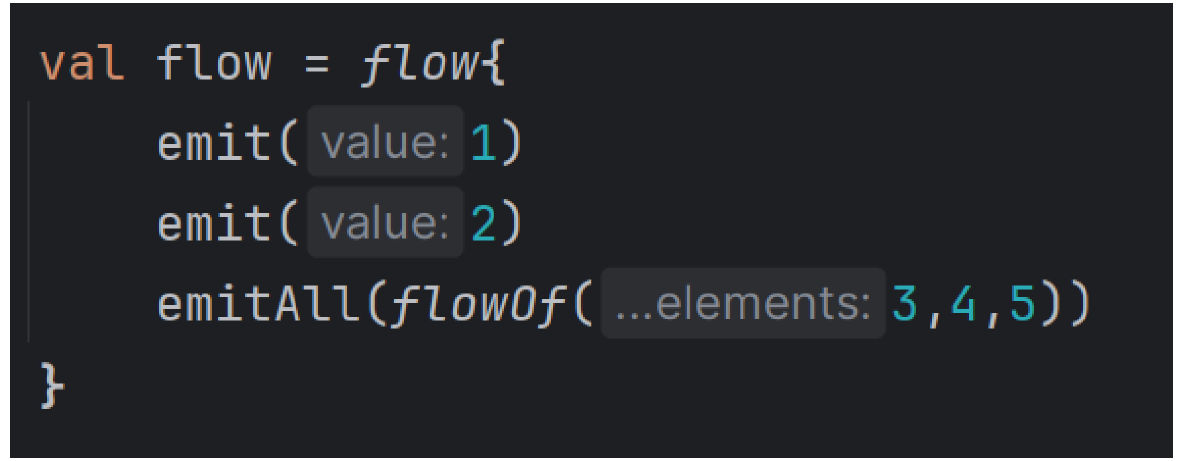
- *collect* 允许我们 *subscribe* 到一个流并对每个消费的元素执行逻辑



```
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*

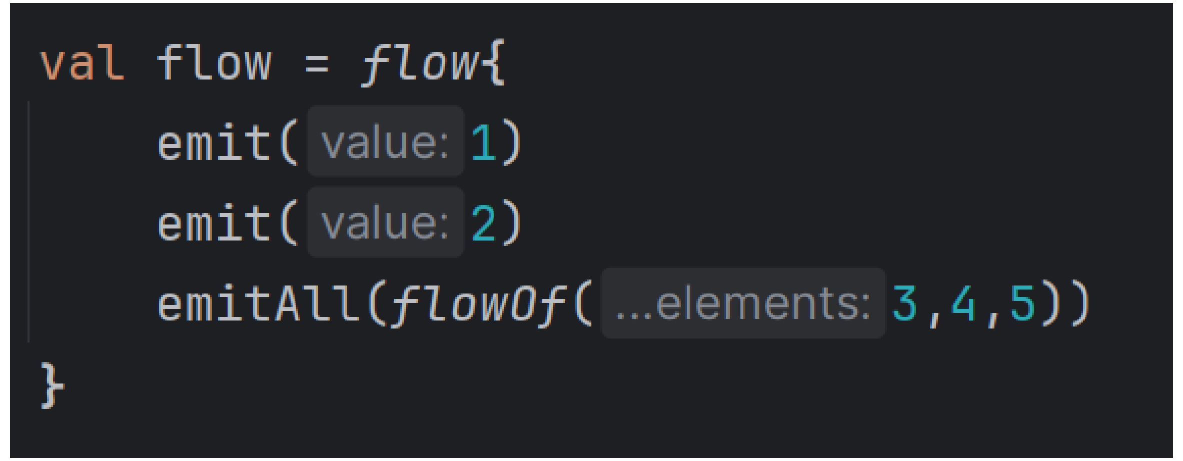


```
val flow = flow{
    emit( value: 1)
    emit( value: 2)
    emitAll(flowOf( ...elements: 3,4,5))
}
```

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

流（续）

- 我们还可以使用 *流函数* 创建相同的流

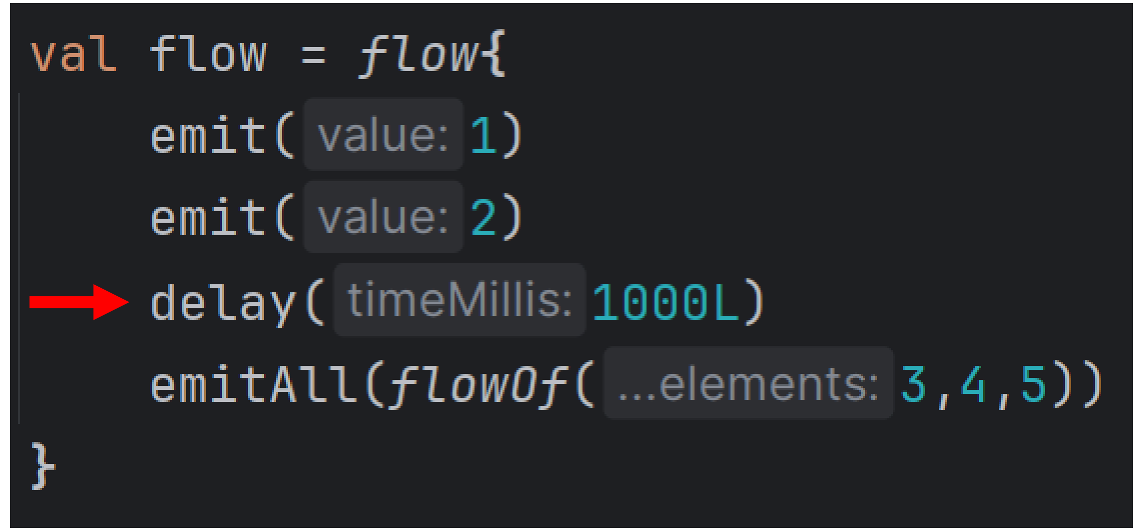


```
val flow = flow{
    emit( value: 1)
    emit( value: 2)
    emitAll(flowOf( ...elements: 3,4,5))
}
```

- 我们使用 *emit* 和 *emitAll* 将元素添加到我们的流中

Flows (cont.)

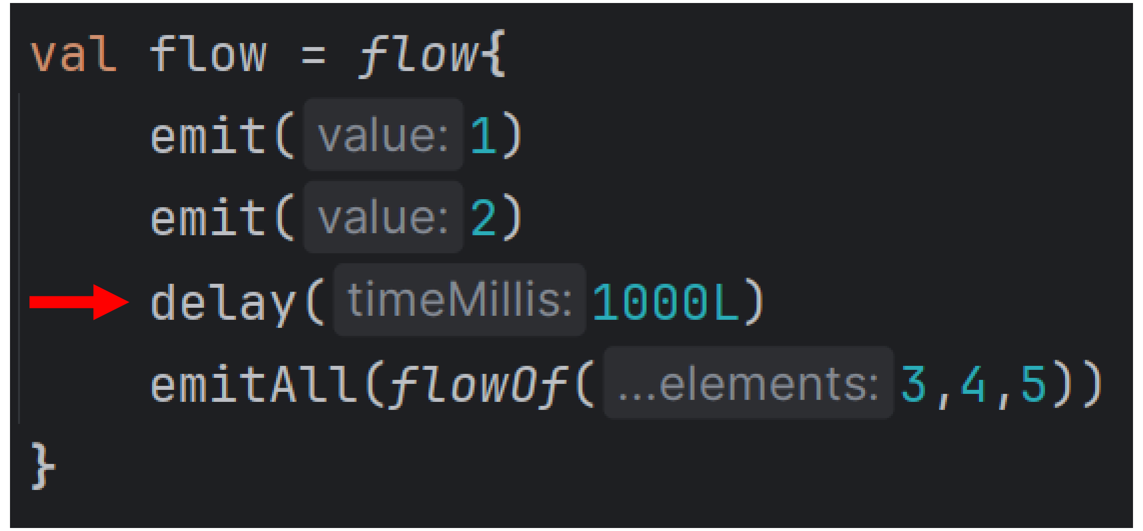
- The flow function allows us to emit elements along side other suspending operations



```
val flow = flow{
    emit( value: 1)
    emit( value: 2)
    delay( timeMillis: 1000L)
    emitAll(flowOf( ...elements: 3,4,5))
}
```

流（续）

- 流函数允许我们在执行其他挂起操作的同时发射元素



```
val flow = flow{
    emit( value: 1)
    emit( value: 2)
    delay( timeMillis: 1000L)
    emitAll(flowOf( ...elements: 3,4,5))
}
```

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

流（续）

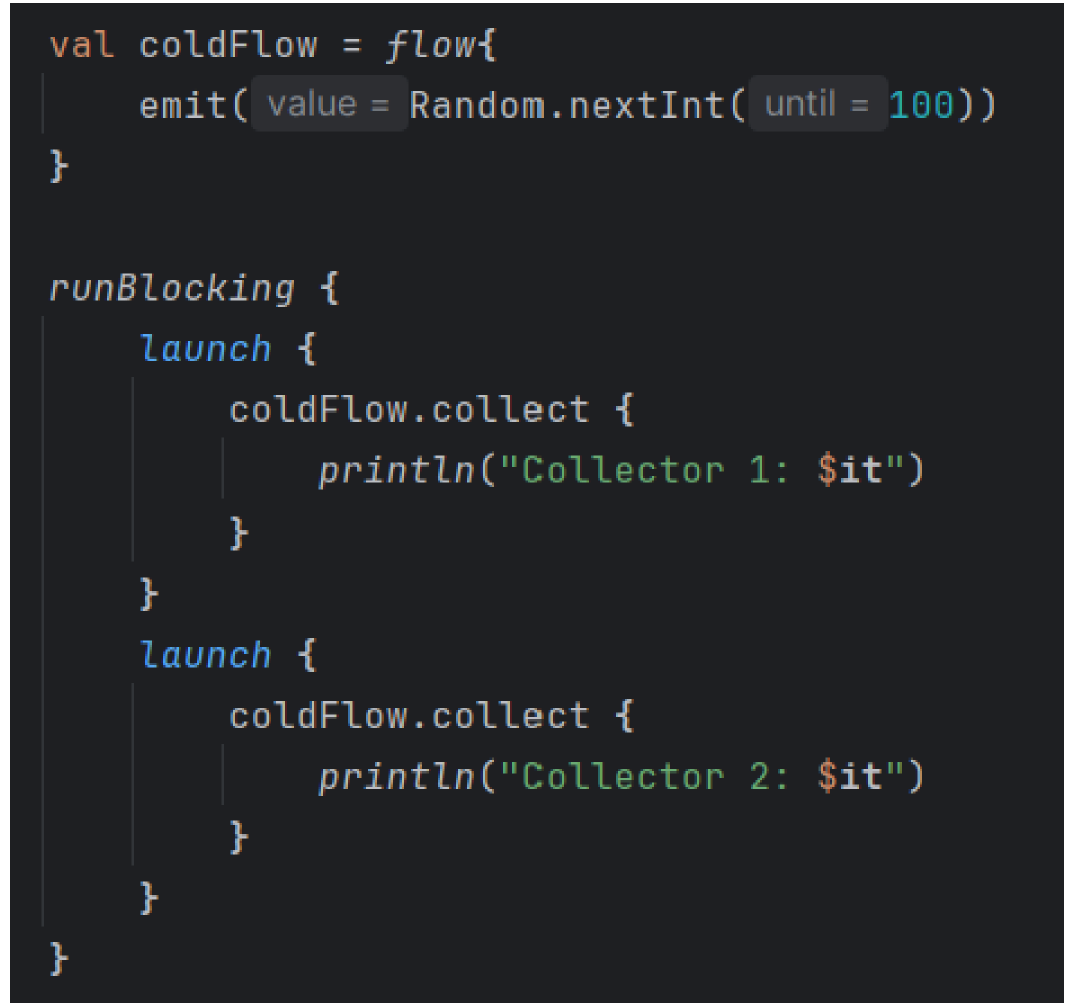
- 流函数是冷流的一个示例

- 冷流
  - 仅当有收集器在收集时，元素才会发出
  - 每个收集器都会收集属于自己的一份元素实例

- 热流
  - 元素的发射与收集器无关（即始终处于激活状态）
  - 发射数据在所有订阅者之间共享

Cold Flow

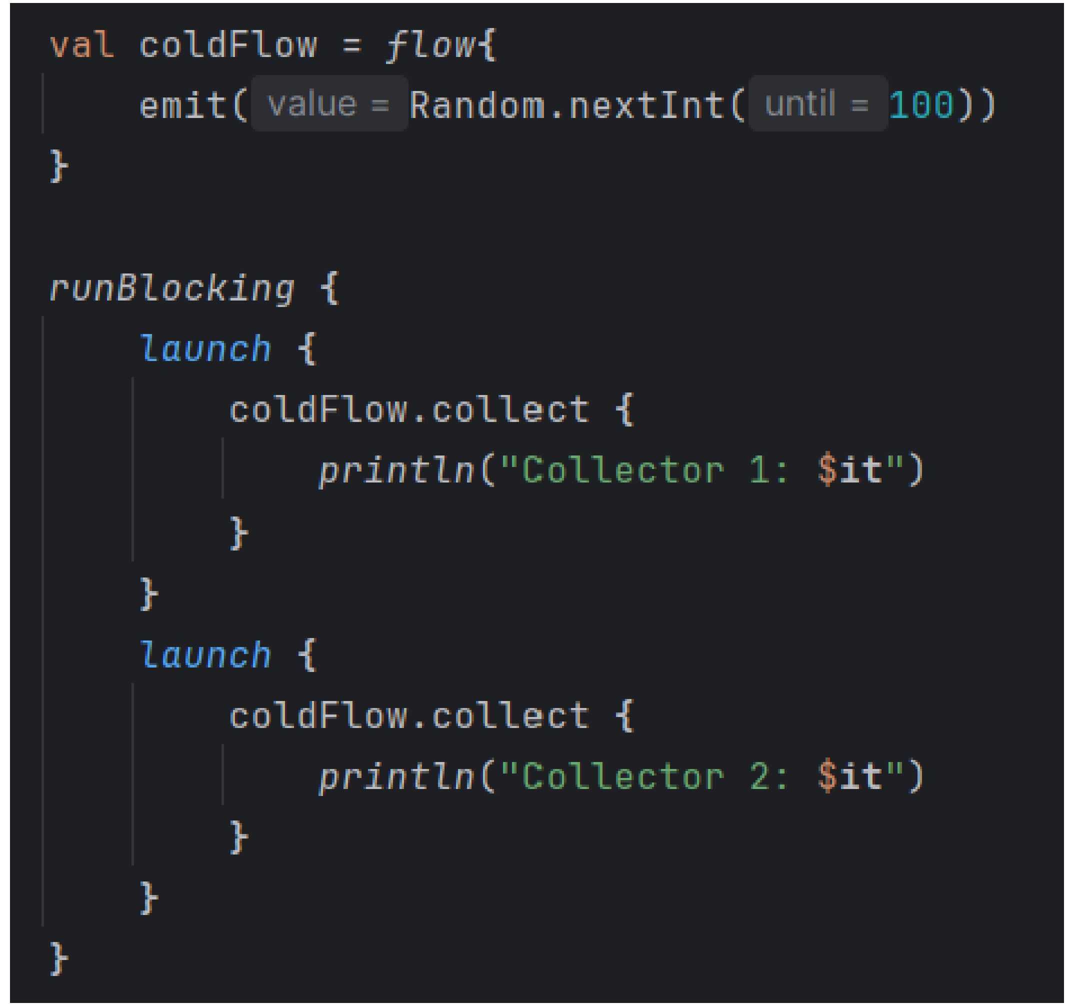
- Notice that each Collector prints a different random value
  - Each collector collects its own instance of elements



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

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

冷流

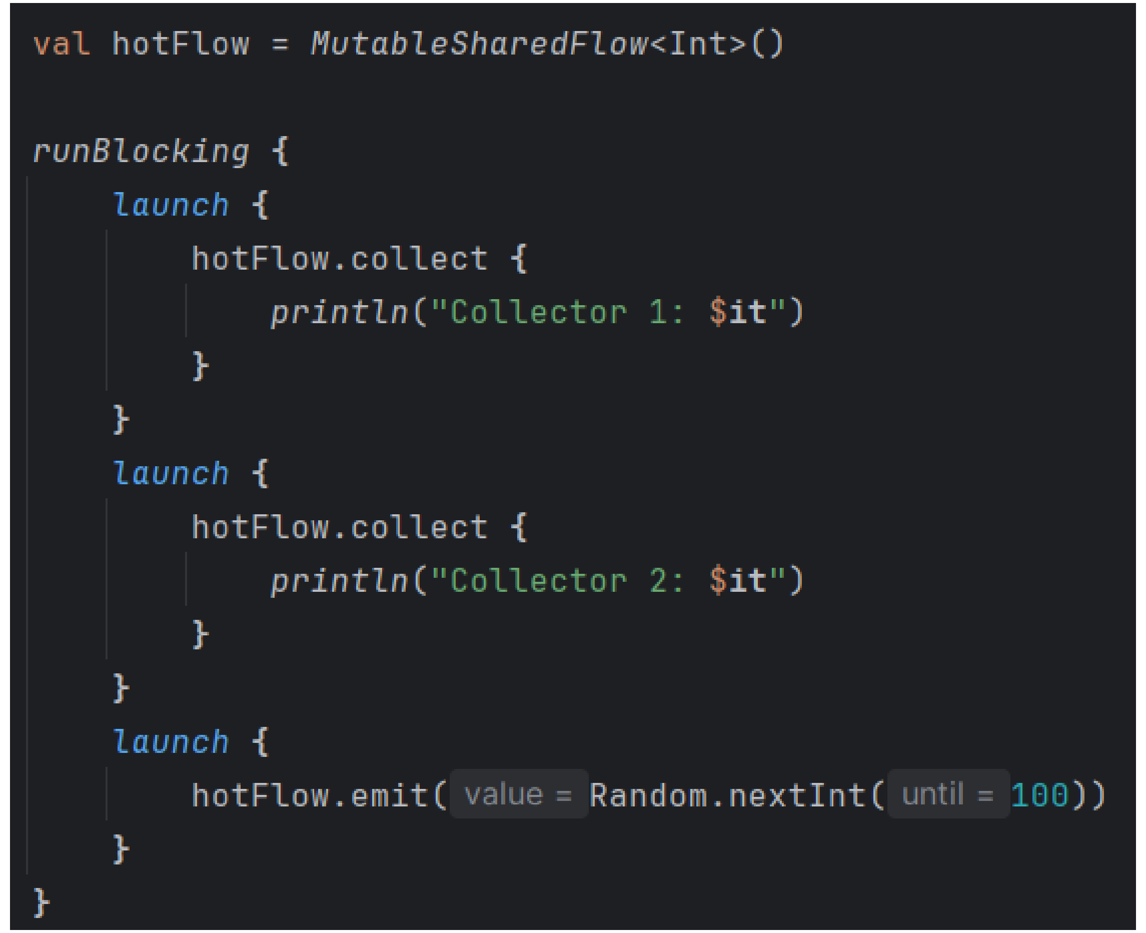
- 请注意，每个收集器打印的随机值都不同
  - 每个收集器收集其自己的元素实例



```
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

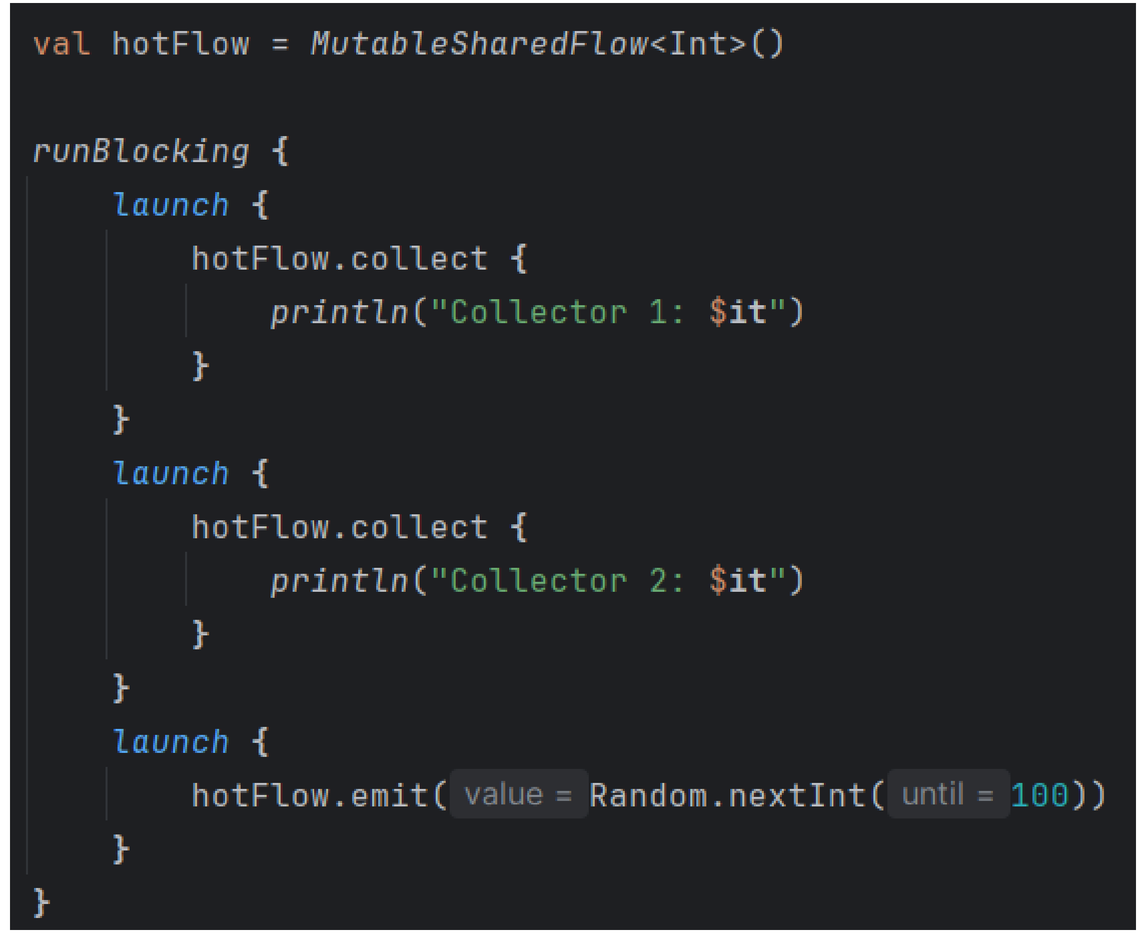
- Notice that each Collector prints the same random value
  - Emissions are shared across all subscribers



```
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))
    }
}
```

热门 流程

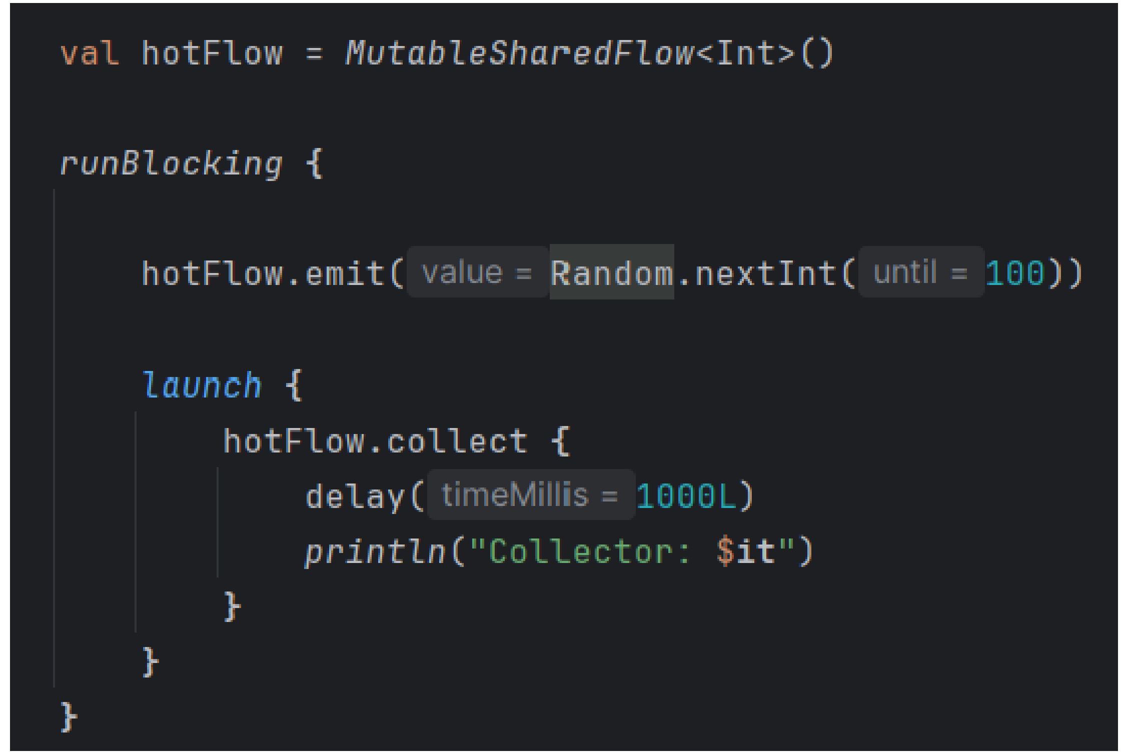
- 请注意，每个收集器都会打印出相同的随机值
  - 发射的数据对所有订阅者是共享的



```
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.)

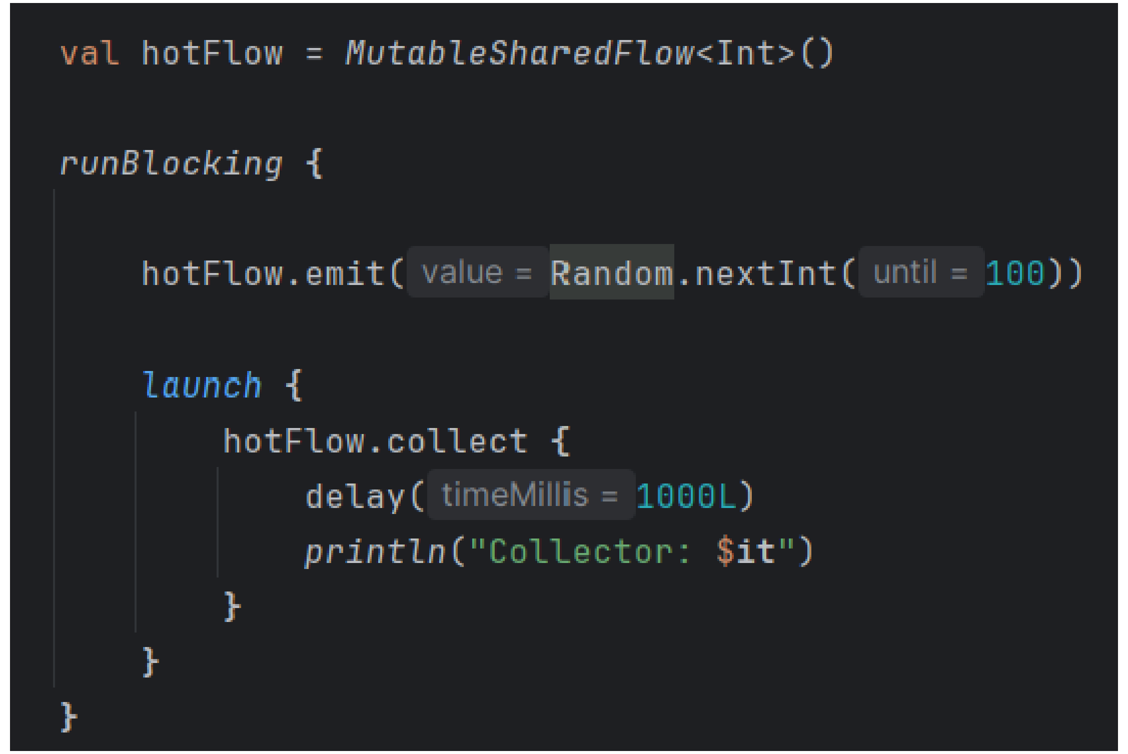
- Notice the Collector misses this emission
    - Elements emit independently of collectors (aka. always active)



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

runBlocking {

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

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

热门 流程（续）

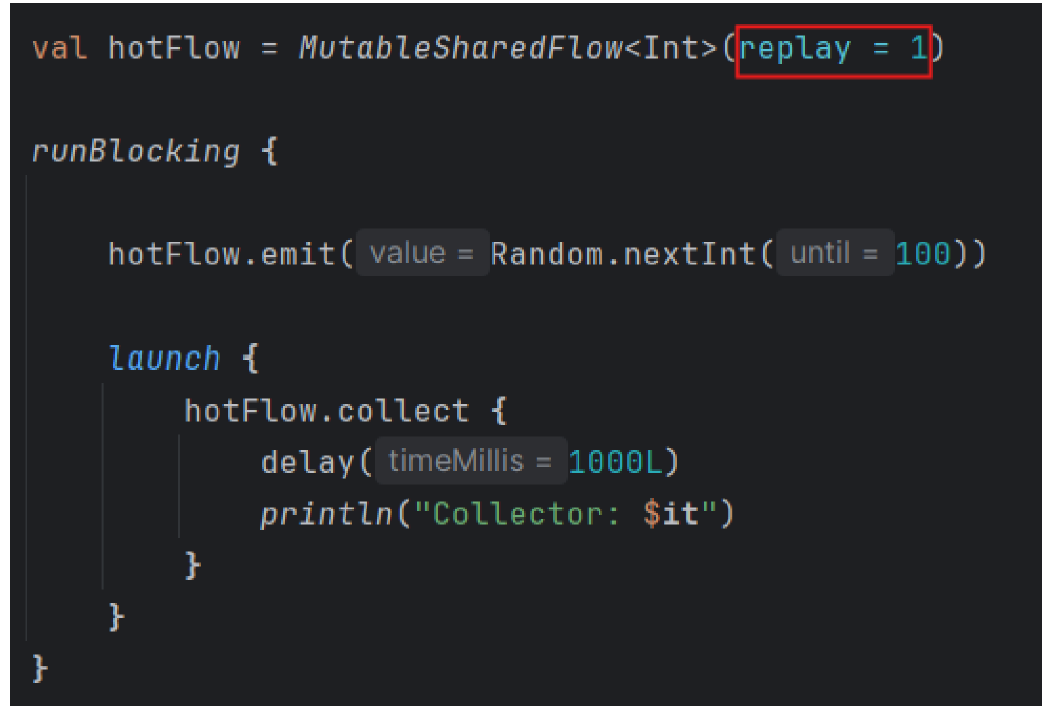
- 注意收集器会遗漏此项发射
    - 元素的发射与收集器无关（即始终处于激活状态）



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

runBlocking {

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

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

Hot Flow (cont.)

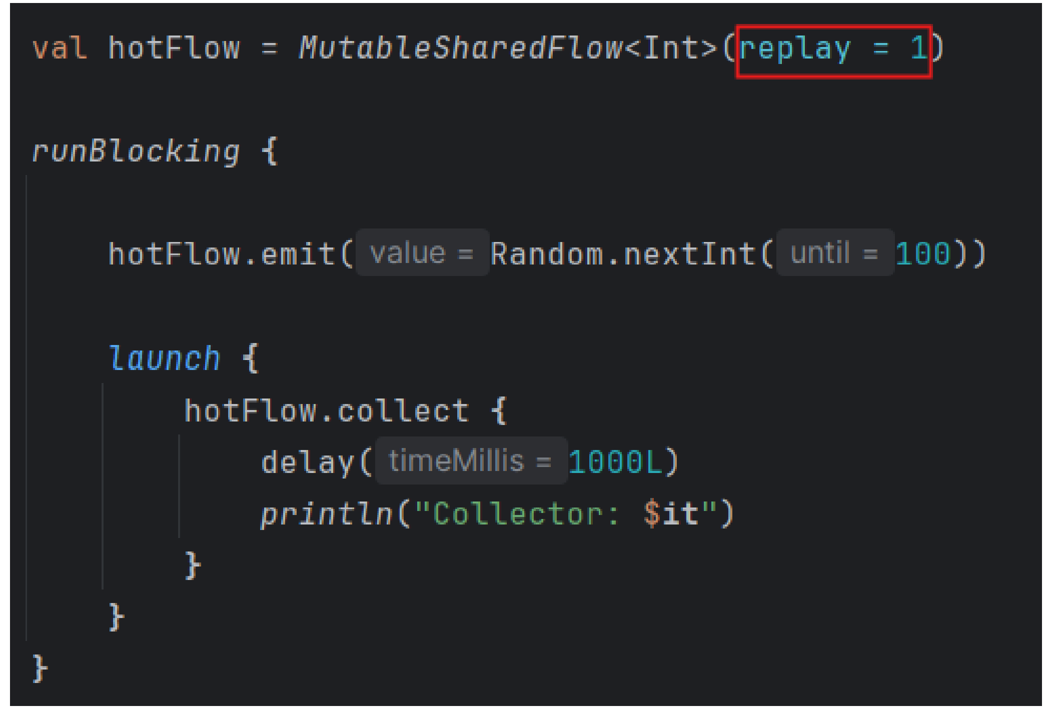
- We can avoid misses by increasing the replay value
  - This will keep a certain number of emissions in a cache
  - Benefits late collectors



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

runBlocking {

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

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

热门 流程（续）

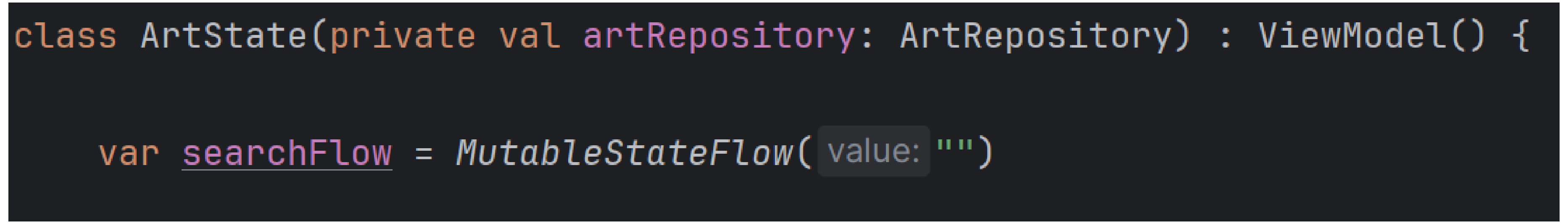
- 我们可以通过增加 重播 值
  - 这将保留一定数量的排放到缓存中
  - 有利于延迟收集者



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

runBlocking {

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

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

StateFlow

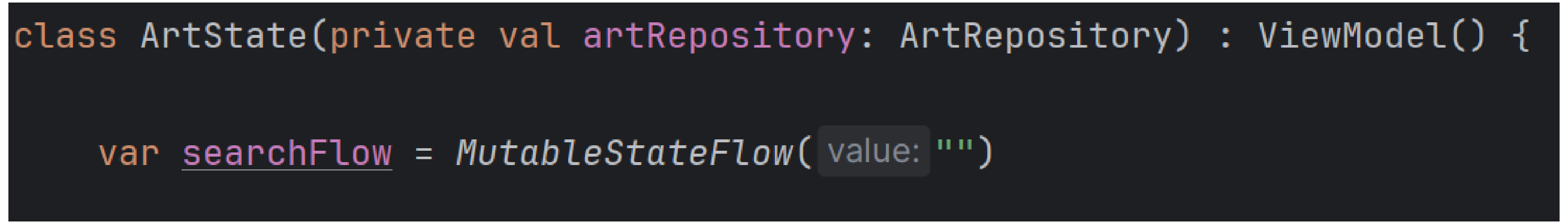
- A *StateFlow* is another example of a hot flow



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

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

状态流

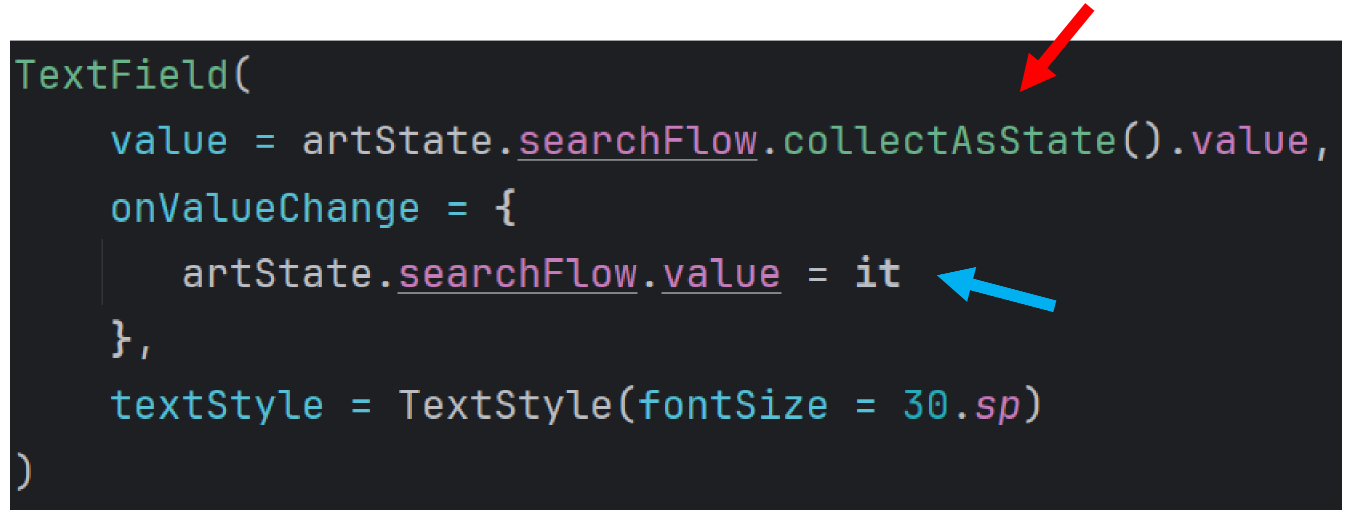
- 一个 *StateFlow* 是另一种热流的例子



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

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

StateFlow (cont.)

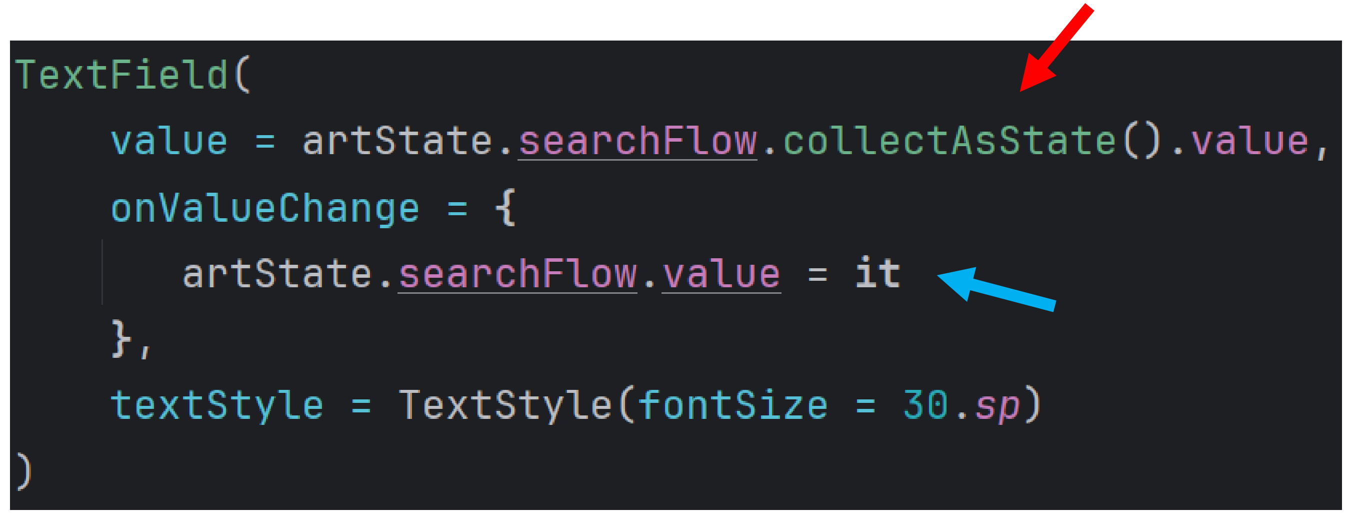
- *collectAsState* allows us to subscribe to the flow as *MutableState*
- Setting the value, emits new data into the flow



```
TextField(
    value = artState.searchFlow.collectAsState().value,
    onValueChange = {
       artState.searchFlow.value = it
    },
    textStyle = TextStyle(fontSize = 30.sp)
)
```

StateFlow（续）

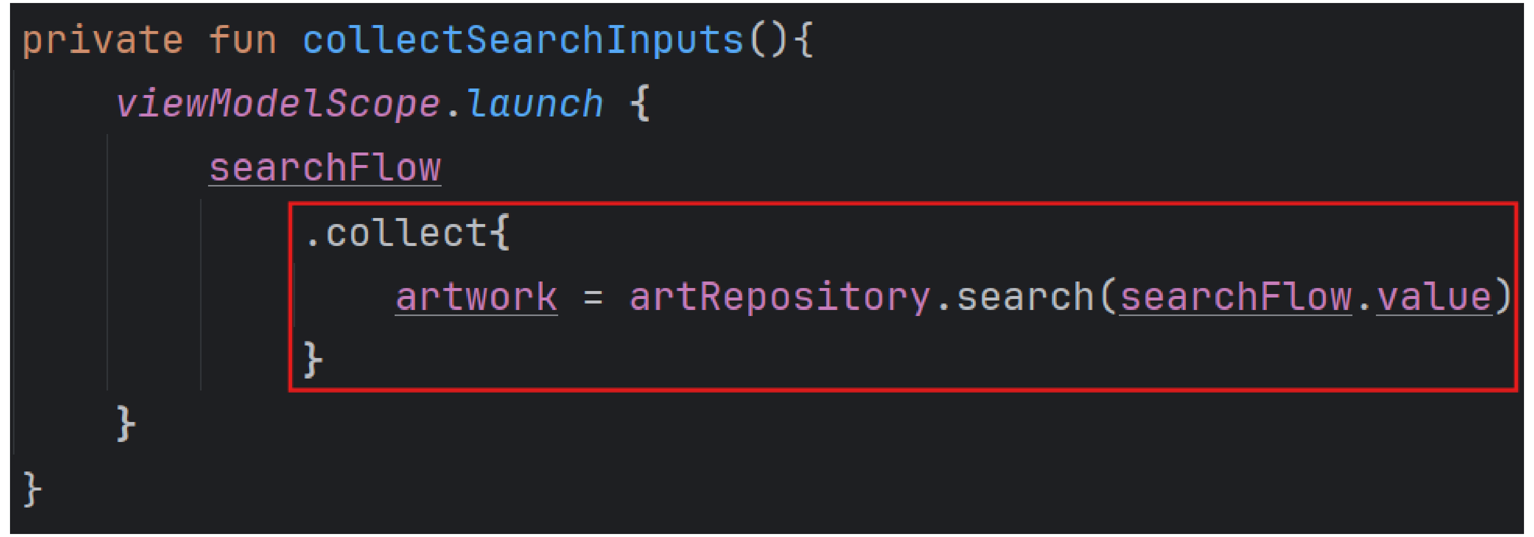
- *collectAsState* 允许我们以 *MutableState*的形式订阅该流
- 设置值，将新数据发射到流中



```
TextField(
    value = artState.searchFlow.collectAsState().value,
    onValueChange = {
       artState.searchFlow.value = it
    },
    textStyle = TextStyle(fontSize = 30.sp)
)
```

StateFlow (cont.)

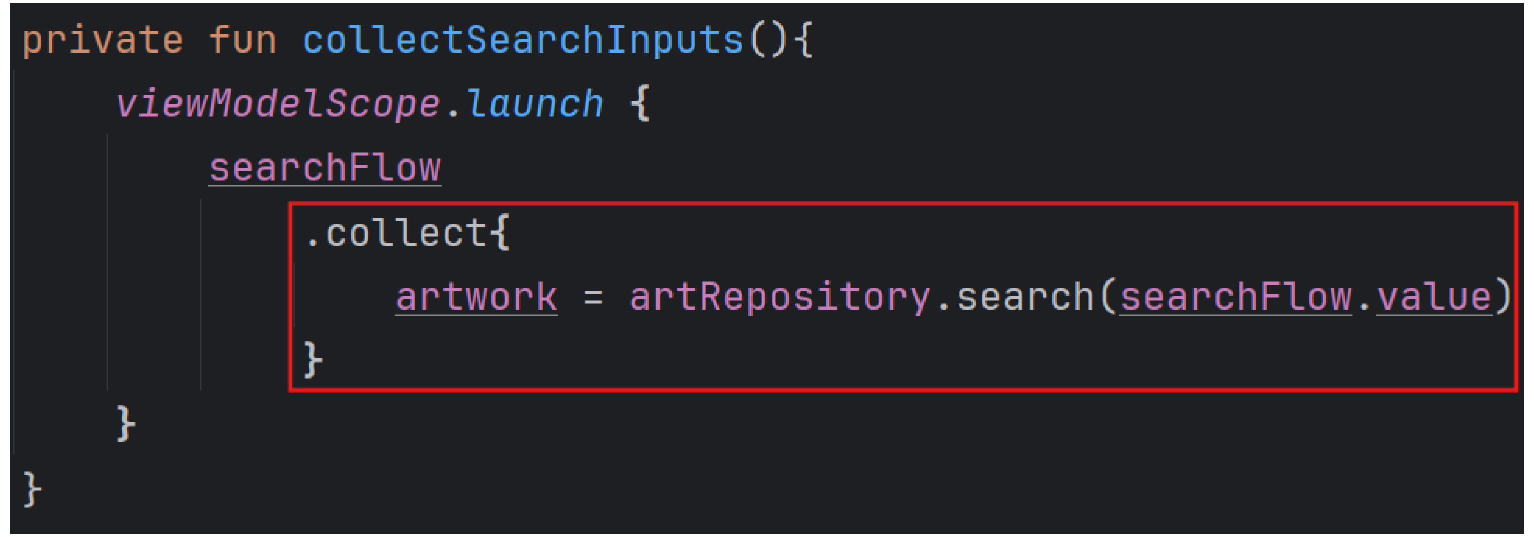
- We can also subscribe and perform a search request for each emission



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

StateFlow（续）

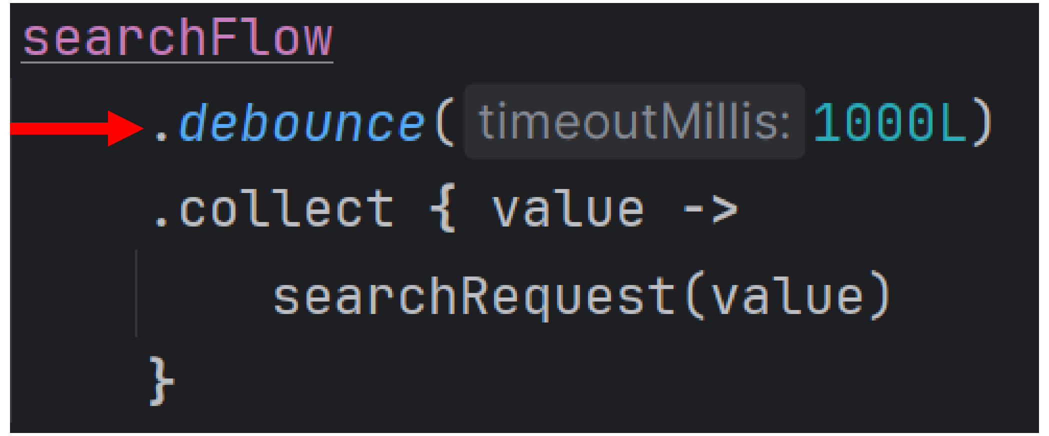
- 我们还可以订阅，并在每次发射时执行搜索请求



```
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

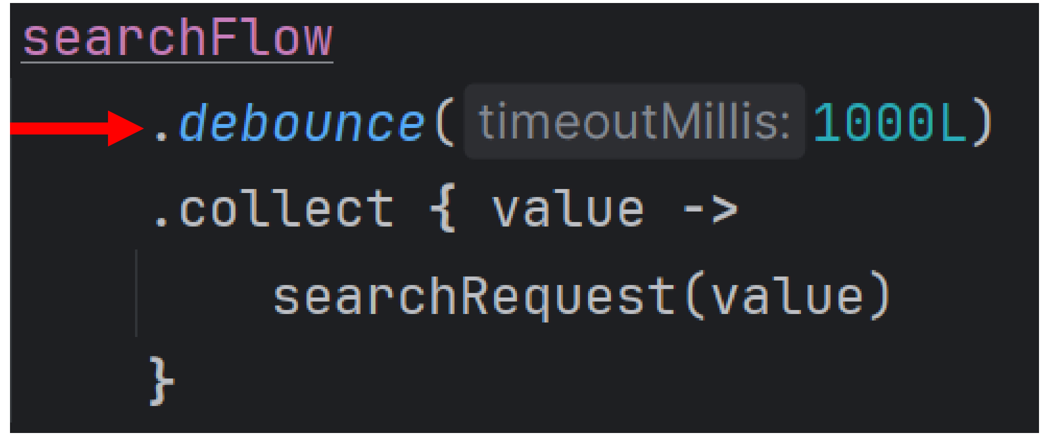


```
searchFlow
    .debounce( timeoutMillis: 1000L)
    .collect { value ->
        searchRequest(value)
    }
```

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

StateFlow（续）

- 为了避免一次性发送过多请求，debounce 属性非常有用



```
searchFlow
    .debounce( timeoutMillis: 1000L)
    .collect { value ->
        searchRequest(value)
    }
```

- 在被收集之前，发射需要停止一段特定的时间

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

流（续）

- 何时使用流可能有用
  - 你的 API 支持数据流式传输
  - 你希望从 API 接收定期更新
  - 从数据库接收实时更新（firebase, mongodb）
  - 防抖；等待输入停止后再发送服务器请求