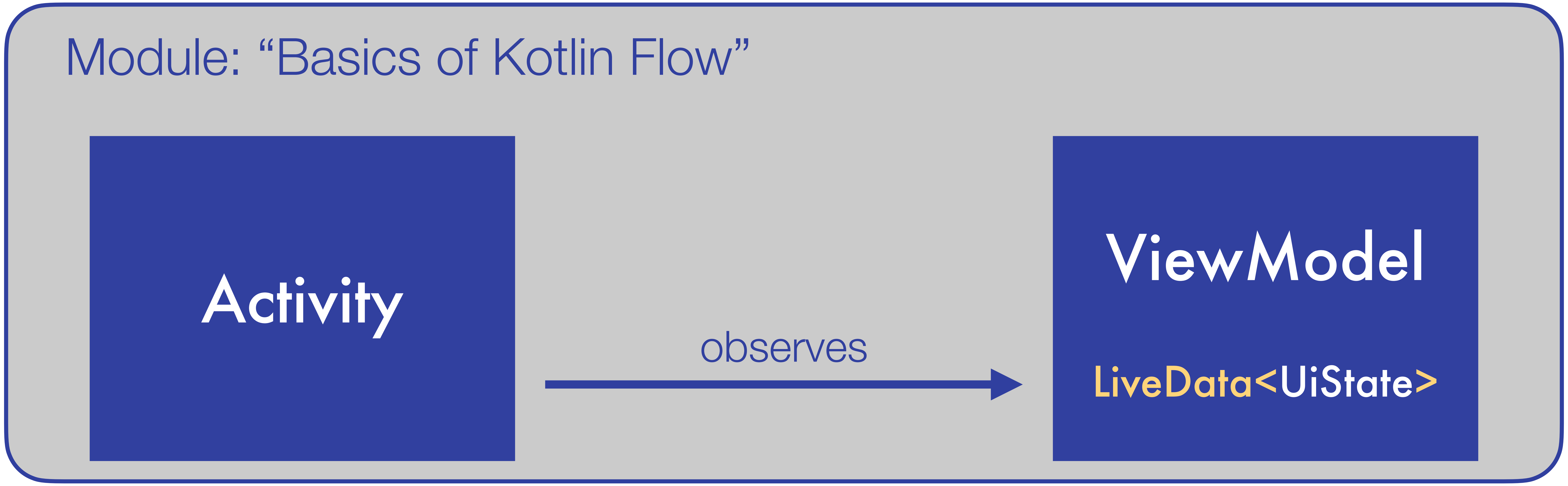
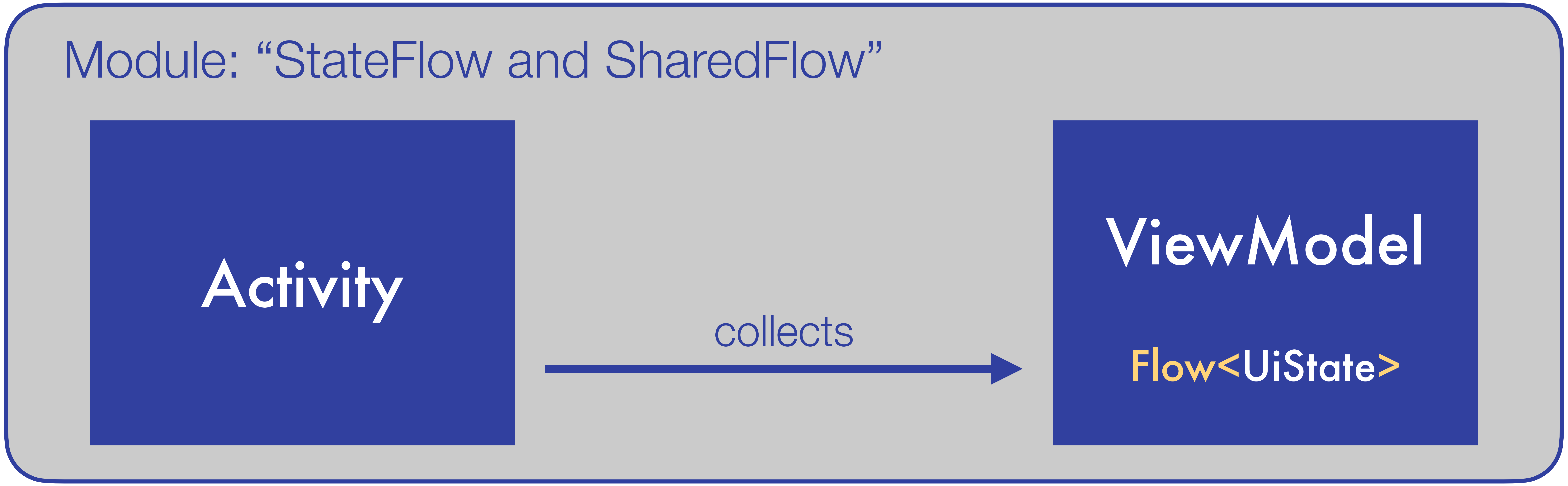
StateFlow

and

SharedFlow


Module: “Basics of Kotlin Flow”
Activity
observes
ViewModel
LiveData<UiState>


Module: “StateFlow and SharedFlow”
Activity
collects
ViewModel
Flow<UiState>

Antipattern

using LiveData in other Layers than the UI Layer

Advantages: Exposing Flows instead of LiveData in ViewModels

- A Single type of observable data holder throughout your architecture
- No knowledge about LiveData necessary
- More flow operators
- ViewModels are decoupled from Android Dependencies
- Simplified testing

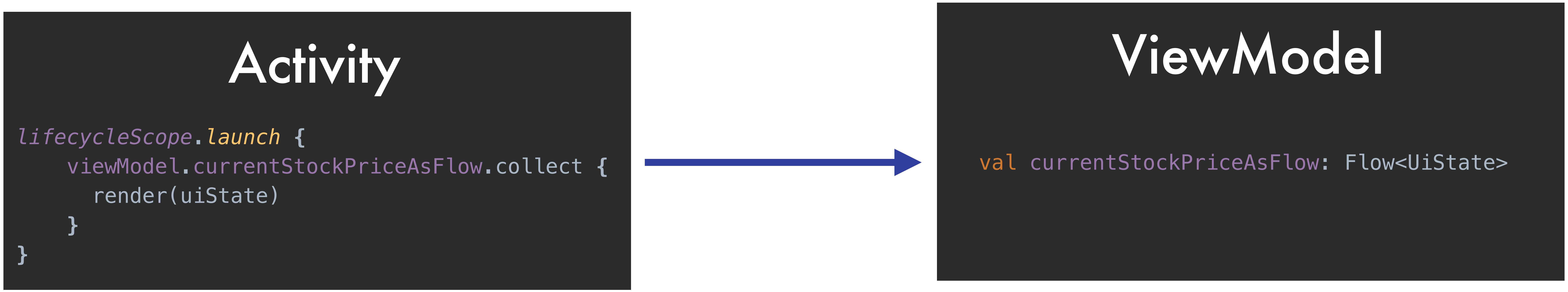
Disadvantages: Exposing Flows instead of LiveData in ViewModels

- more “boilerplate” code in the view

Summary: Exposing Flows instead of LiveData in ViewModels

- existing Code: 🤔
- new Code: 🙂

Approach 1: Exposing regular flow



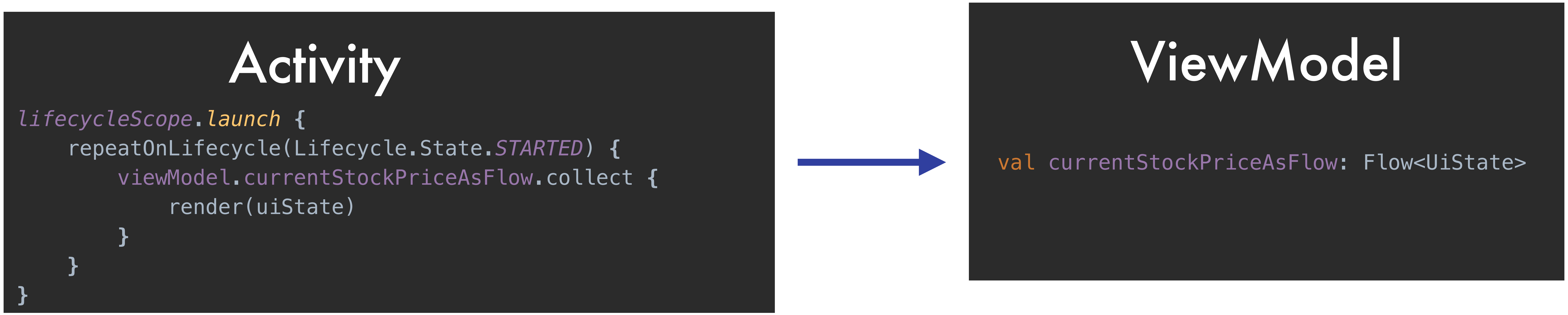
```kotlin
lifecycleScope.launch {
    viewModel.currentStockPriceAsFlow.collect {
      render(uiState)
    }
}
```

```kotlin
val currentStockPriceAsFlow: Flow<UiState>
```

Problems ❌

- Flow Producer continues to run when the app is in background
- Activity receives emissions and renders UI when it is in the background
- Multiple collectors create multiple flows
- Configuration Change re-starts the flow

Approach 2: “lifecycle-aware” collecting coroutine



Activity

```
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        viewModel.currentStockPriceAsFlow.collect {
            render(uiState)
        }
    }
}
```

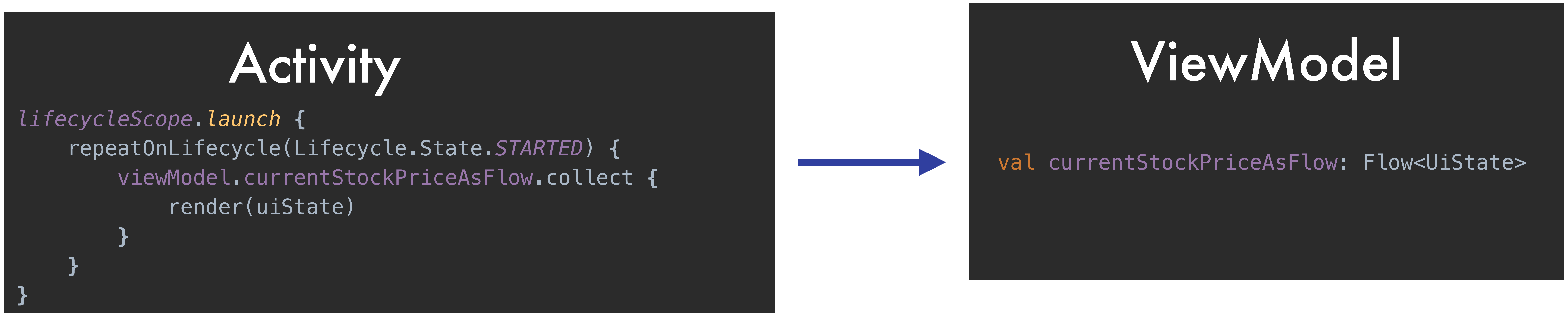
ViewModel

```
val currentStockPriceAsFlow: Flow<UiState>
```

Problems ❌

- Flow Producer continues to run when the app is in background
- Activity receives emissions and renders UI when it is in the background
- Multiple collectors create multiple flows
- Configuration Change re-starts the flow

Approach 2: “lifecycle-aware” collecting coroutine



```kotlin
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        viewModel.currentStockPriceAsFlow.collect {
            render(uiState)
        }
    }
}
```

```kotlin
val currentStockPriceAsFlow: Flow<UiState>
```

Problems ❌

- Flow Producer continues to run when the app is in background
- Activity receives emissions and renders UI when it is in the background
- Multiple collectors create multiple flows
- Configuration Change re-starts the flow

Approach 2: “lifecycle-aware” collecting coroutine

Activity

```
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        viewModel.currentStockPriceAsFlow.collect {
            render(uiState)
        }
    }
}
```

→

ViewModel

```
val currentStockPriceAsFlow: Flow<UiState>
```

Problems ❌

- ~~Flow Producer continues to run when the app is in background~~ ✅
- Activity receives emissions and renders UI when it is in the background
- Multiple collectors create multiple flows
- Configuration Change re-starts the flow

Approach 2: “lifecycle-aware” collecting coroutine

Activity

```
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        viewModel.currentStockPriceAsFlow.collect {
            render(uiState)
        }
    }
}
```

→

ViewModel

```
val currentStockPriceAsFlow: Flow<UiState>
```

Problems ❌

- ~~Flow Producer continues to run when the app is in background~~ ✅
- ~~Activity receives emissions and renders UI when it is in the background~~ ✅
- Multiple collectors create multiple flows
- Configuration Change re-starts the flow

Approach 3: Exposing a “hot” flow in the ViewModel

Activity

```kotlin
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        viewModel.currentStockPriceAsFlow.collect {
            render(uiState)
        }
    }
}
```

→

ViewModel

```kotlin
val currentStockPriceAsFlow: SharedFlow<UiState>
// = coldflow
.shareIn(
    scope = viewModelScope,
    started = SharingStarted.WhileSubscribed()
)
```

Problems ❌

- ~~Flow Producer continues to run when the app is in background~~ ✅
- ~~Activity receives emissions and renders UI when it is in the background~~ ✅
- Multiple collectors create multiple flows
- Configuration Change re-starts the flow

Approach 3: Exposing a “hot” flow in the ViewModel

Activity

```kotlin
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        viewModel.currentStockPriceAsFlow.collect {
            render(uiState)
        }
    }
}
```

→

ViewModel

```kotlin
val currentStockPriceAsFlow: SharedFlow<UiState>
// = coldflow
.shareIn(
    scope = viewModelScope,
    started = SharingStarted.WhileSubscribed()
)
```

Problems ❌

- ~~Flow Producer continues to run when the app is in background~~ ✅
- ~~Activity receives emissions and renders UI when it is in the background~~ ✅
- ~~Multiple collectors create multiple flows~~ ✅
- Configuration Change re-starts the flow

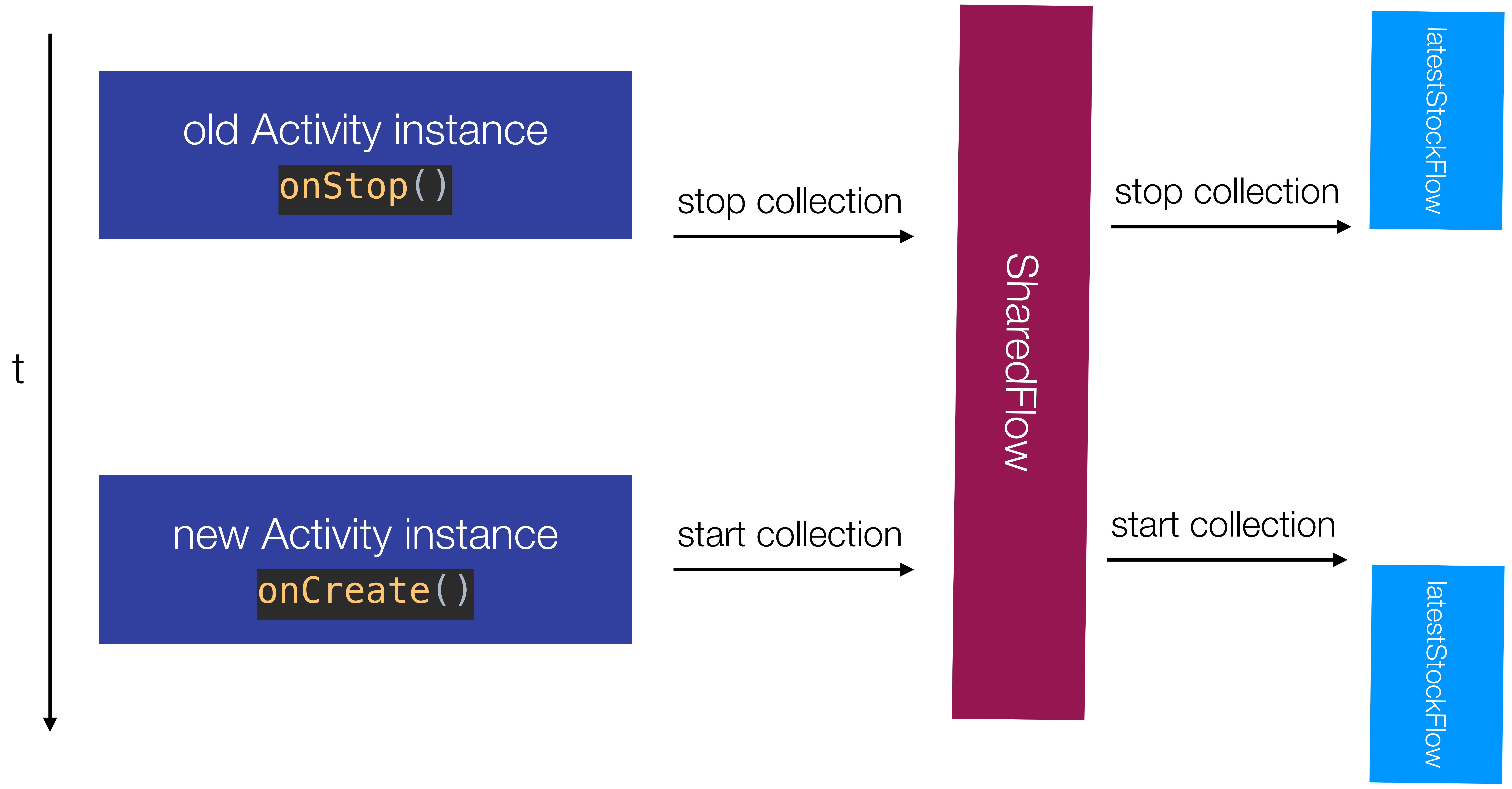
Cold Flows ❄️

- become active on collection
- become inactive on cancellation of the collecting coroutine
- emit individual emissions to every collector

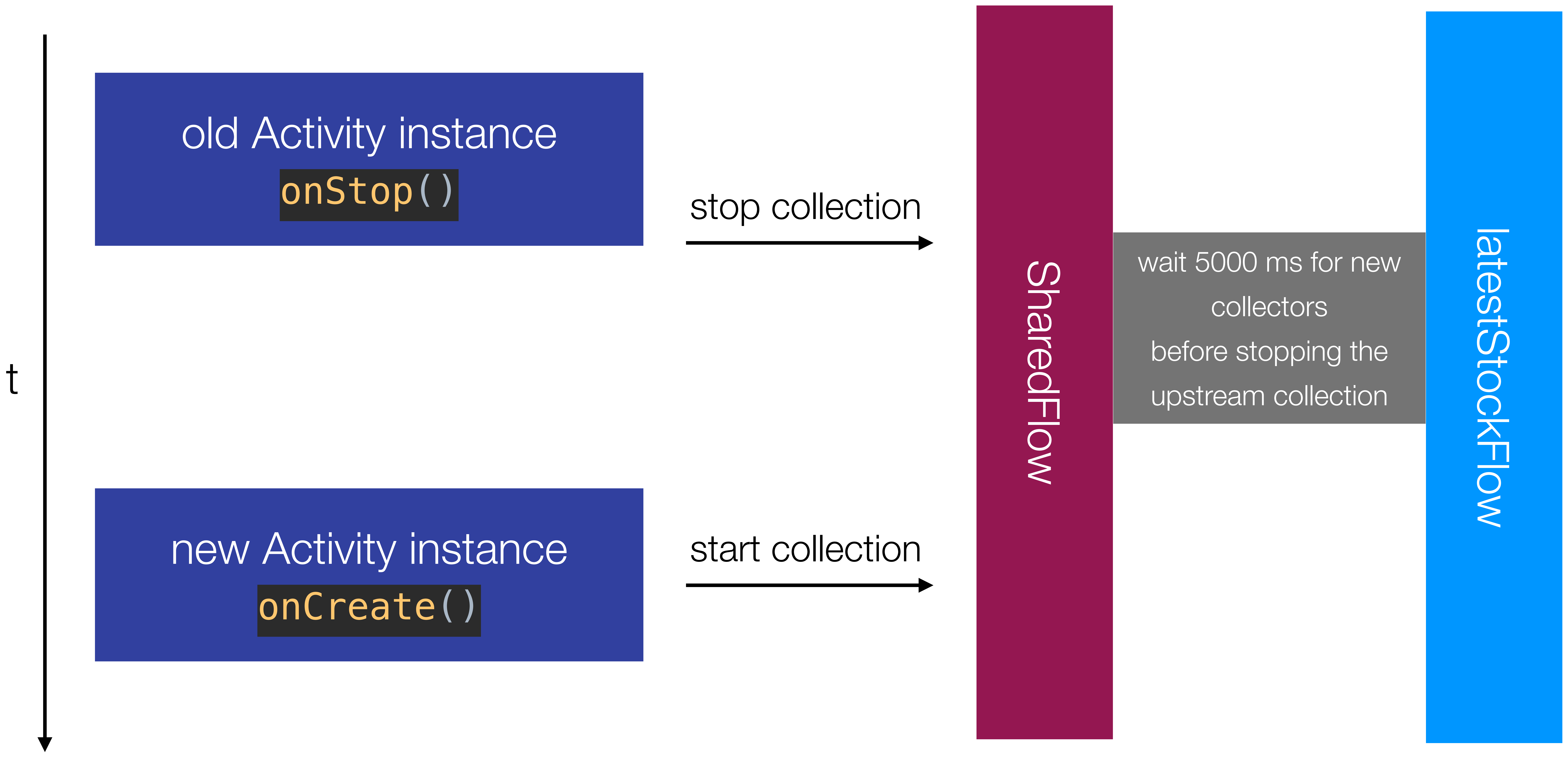
Hot Flows 🔥

- are active regardless of whether there are collectors
- stay active even when there is no more collector
- emissions are shared between all collectors

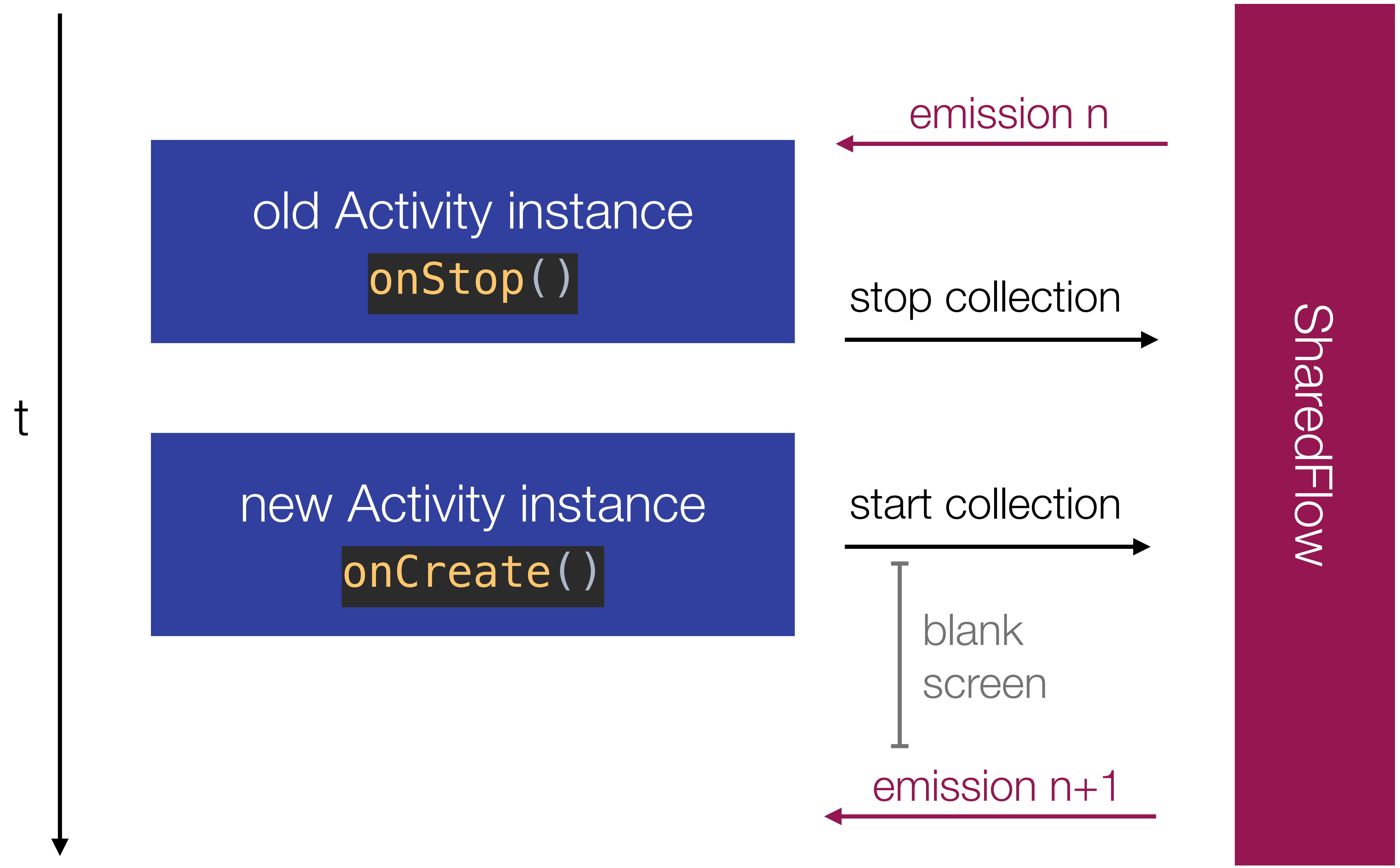
configuration change without timeout



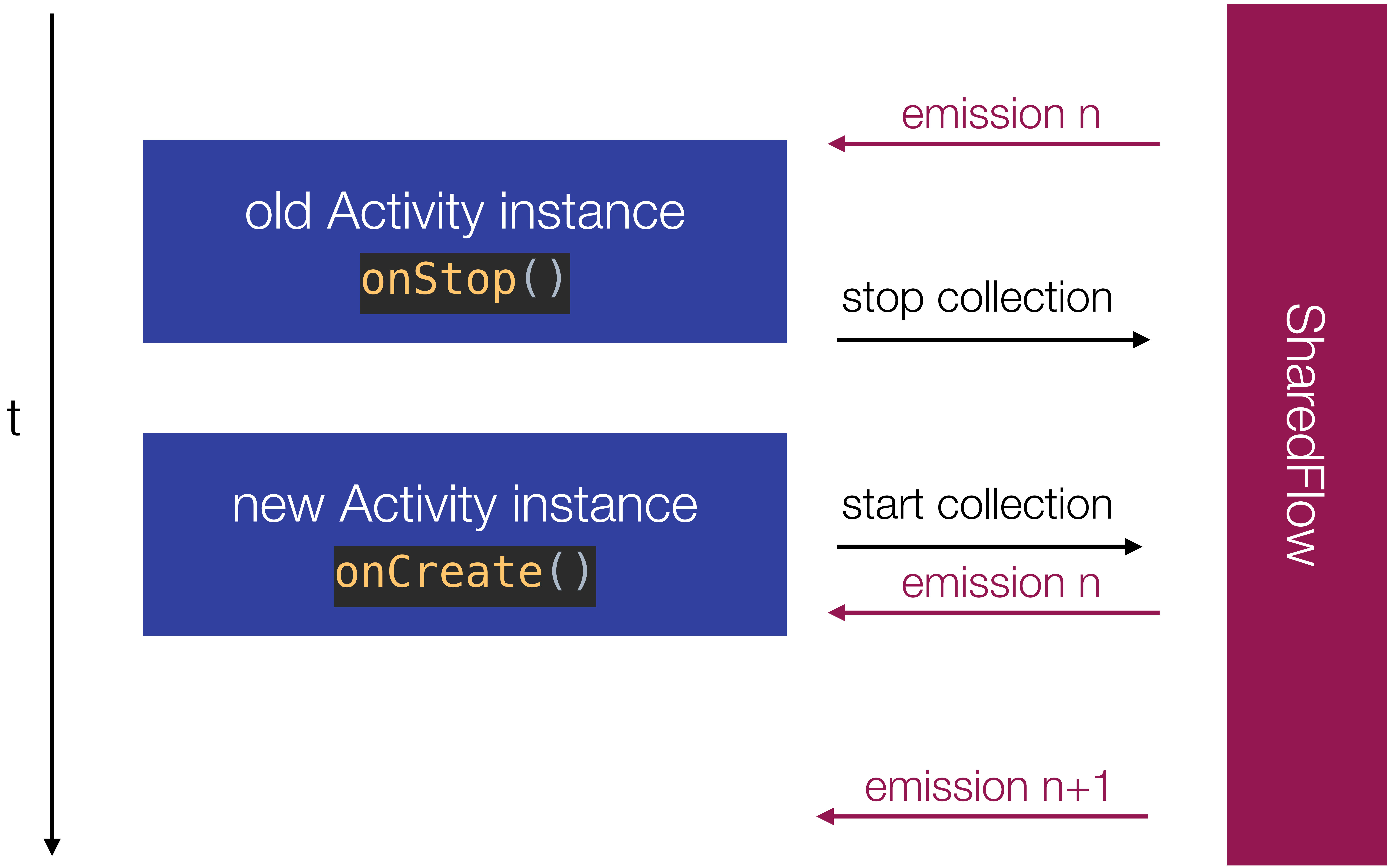
configuration change with timeout of 5000ms



Problem: blank screen after orientation change



Problem: blank screen after orientation change



Approach 3: Exposing a “hot” flow in the ViewModel

Activity

```
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        viewModel.currentStockPriceAsFlow.collect {
            render(uiState)
        }
    }
}
```

→

ViewModel

```
val currentStockPriceAsFlow: SharedFlow<UiState>
// = coldflow
.shareIn(
    scope = viewModelScope,
    started = SharingStarted.WhileSubscribed(5000),
    replay = 1
)
```

Problems ❌

- ~~Flow Producer continues to run when the app is in background~~ ✅
- ~~Activity receives emissions and renders UI when it is in the background~~ ✅
- ~~Multiple collectors create multiple flows~~ ✅
- Configuration Change re-starts the flow

Approach 3: Exposing a “hot” flow in the ViewModel

Activity

```
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        viewModel.currentStockPriceAsFlow.collect {
            render(uiState)
        }
    }
}
```

→

ViewModel

```
val currentStockPriceAsFlow: SharedFlow<UiState>
// = coldflow
.shareIn(
    scope = viewModelScope,
    started = SharingStarted.WhileSubscribed(5000),
    replay = 1
)
```

Problems ❌

- ~~Flow Producer continues to run when the app is in background~~ ✅
- ~~Activity receives emissions and renders UI when it is in the background~~ ✅
- ~~Multiple collectors create multiple flows~~ ✅
- ~~Configuration Change re-starts the flow~~ ✅

SharedFlow

vs 🤺

StateFlow

	SharedFlow	StateFlow
Initial Value	No	Yes
Replay Cache	customizable	fixed size of 1
Emission of subsequent equal values	yes	no

	SharedFlow	StateFlow
Initial Value	No	Yes
Replay Cache	customisable	fixed size of 1
Emission of subsequent equal values	yes	no

Rule of Thumb of Usage:

- Whenever you want to use a hot flow, use a StateFlow by default.
- StateFlows are more efficient when used for state
- StateFlows provide convenient option to read and write its value in a non-suspending fashion by synchronously accessing the `.value` property
- Only if you have special requirements, switch to a SharedFlow.

Attention!

- In Module 17 about “Concurrent Flows”, you will learn about an additional difference between SharedFlows and StateFlows.
- With StateFlows, you can potentially “lose” emissions if you have a slow collector.
- You can find more information about this behaviour in the lecture “Buffers in SharedFlow and StateFlow”