COROUTINES AND RXJAVA

AN ASYNCHRONICITY COMPARISON

MANUEL VICENTE VIVO

@MANUELVICNT

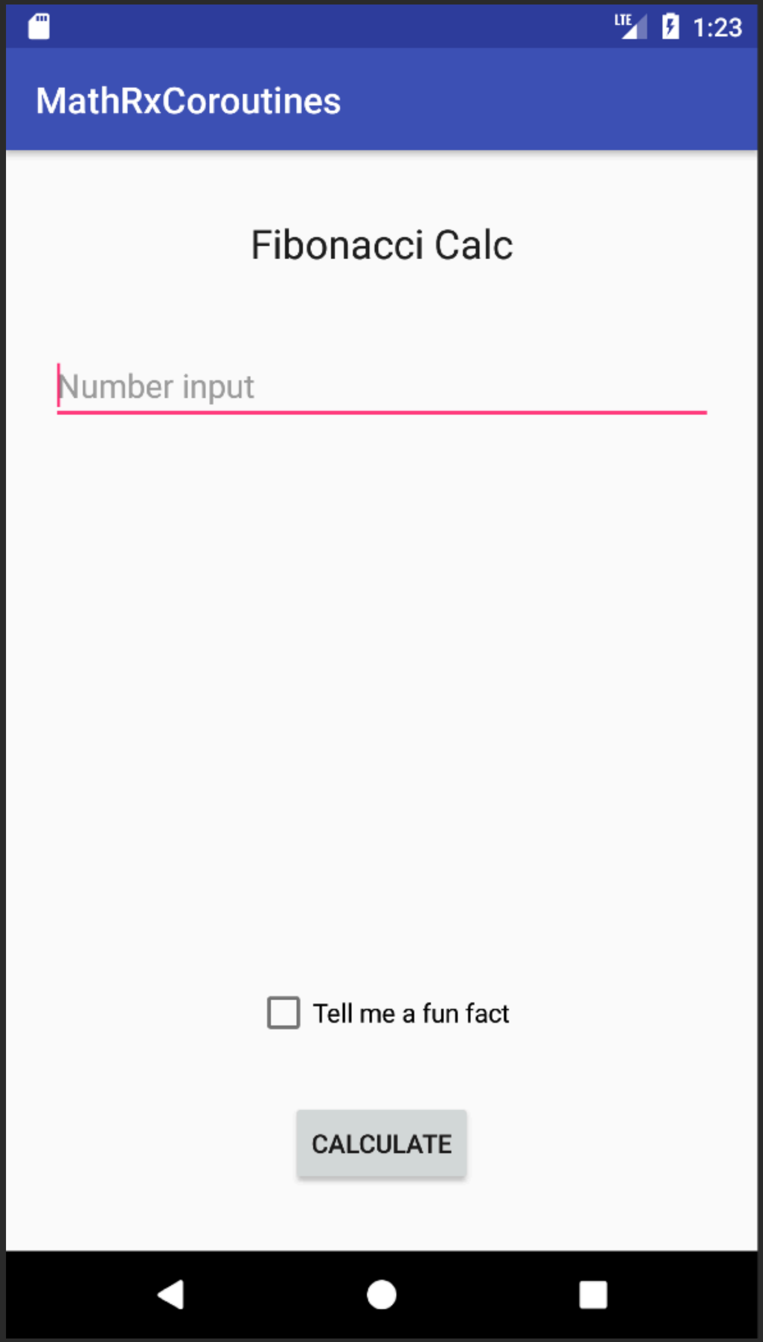
AGENDA

- Coroutines Recap
- RxJava & Coroutines concepts
- Build an App

GOALS

- Learn Coroutines with basic RxJava knowledge
- Compare both libraries on different topics
- Use what we learned to build the App
    - Code available

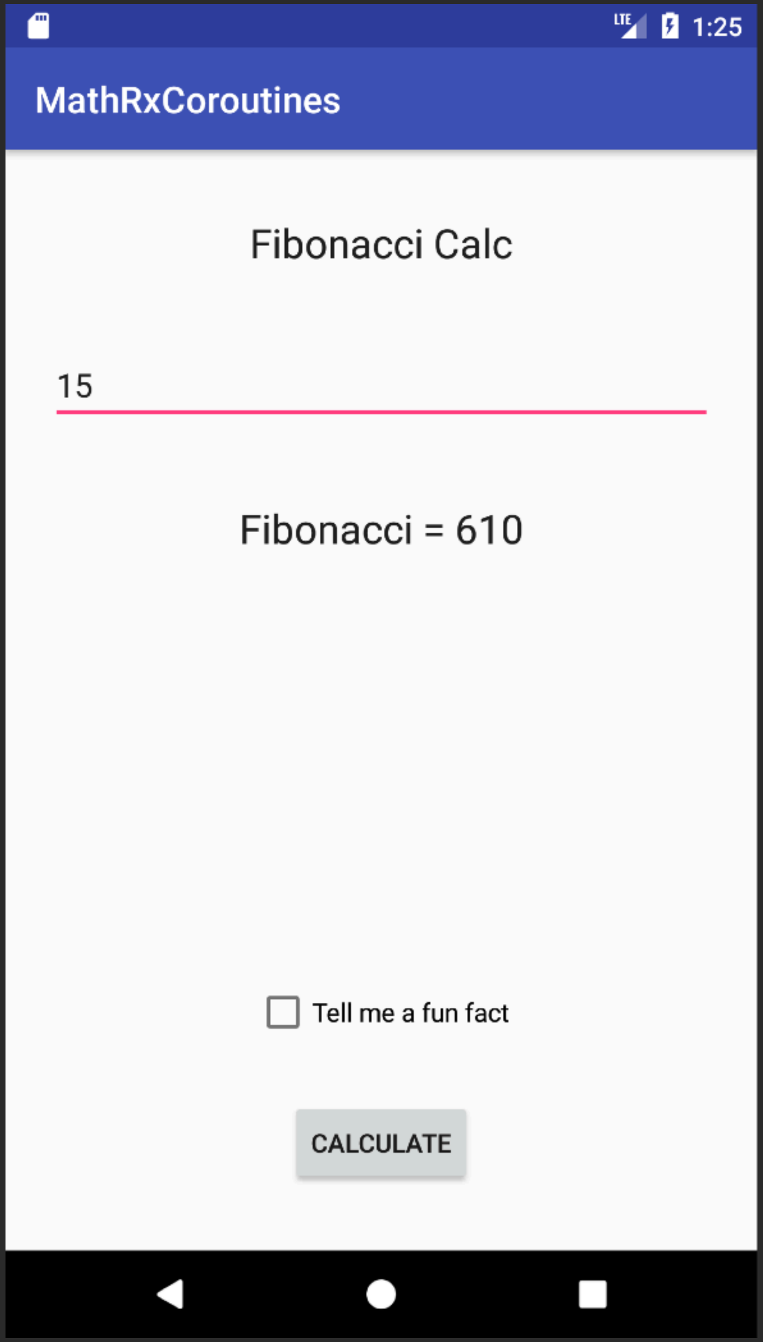
THE APP


MathRxCoroutines

Fibonacci Calc

Number input

☐ Tell me a fun fact

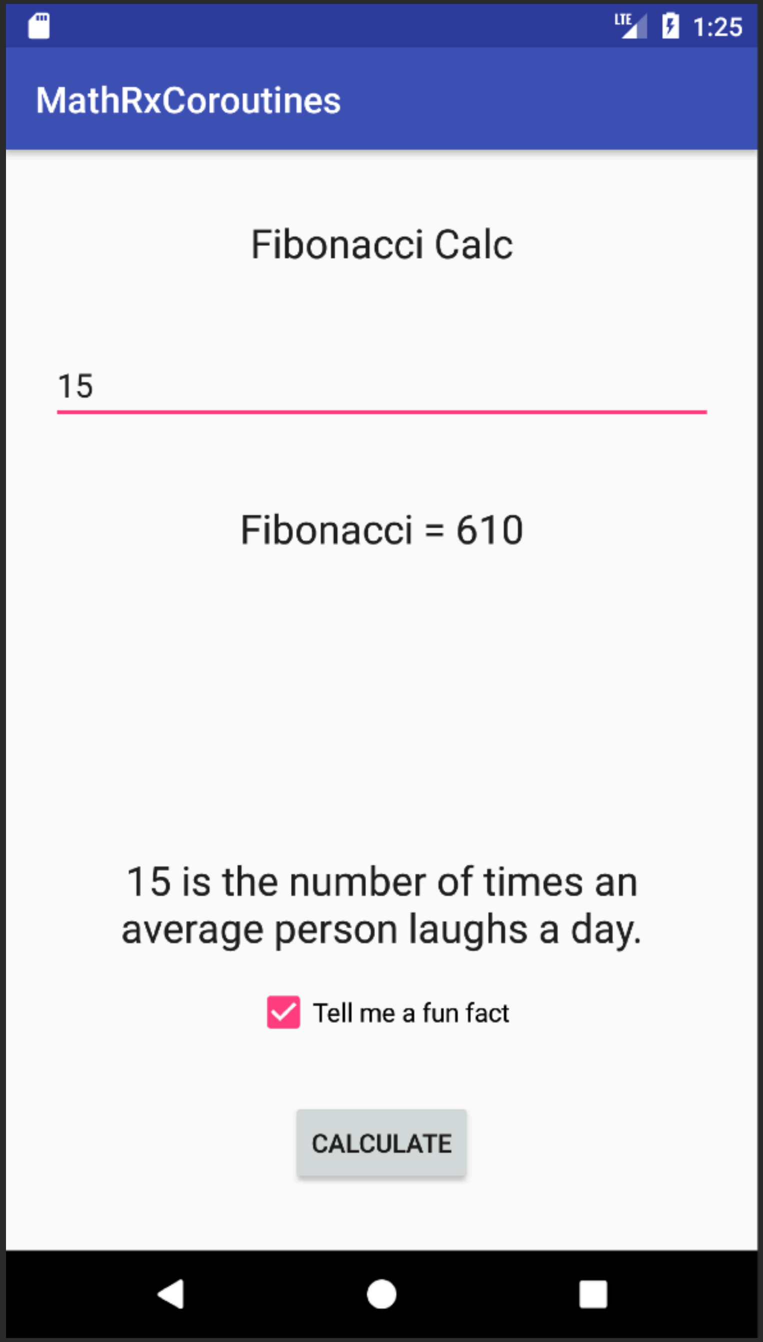

MathRxCoroutines

Fibonacci Calc

15

Fibonacci = 610

☐ Tell me a fun fact


MathRxCoroutines

Fibonacci Calc

15

Fibonacci = 610

15 is the number of times an average person laughs a day.

☑ Tell me a fun fact

INTENDED AUDIENCE

- Able to read Kotlin code
- Basic RxJava experience
- Coroutines 101
- You want to learn differences and similarities between RxJava and Coroutines

I HEARD COROUTINES?

- From the Kotlin documentation…
  - Coroutines simplify asynchronous programming
  - Code can be expressed sequentially and the library handles the asynchronous code for us
  - Computations can be suspended without blocking a Thread

I HEARD COROUTINES?

- From the Kotlin documentation…
  - Coroutines simplify asynchronous programming
  - Code can be expressed sequentially and the library handles the asynchronous code for us
  - Computations can be suspended without blocking a Thread

I HEARD COROUTINES?

- From the Kotlin documentation…
  - Coroutines simplify asynchronous programming
  - Code can be expressed sequentially and the library handles the asynchronous code for us
  - Computations can be suspended without blocking a Thread

I WANT TO
PLAY A GAME

What is this?

```
launch(CommonPool) {
    heavyComputation()
}
```

What is this?

Coroutine

```
launch(CommonPool) {
    heavyComputation()
}
```

What is this?

```
launch(CommonPool) {
    heavyComputation()
}
```

What is this?

Coroutine Builder

```
launch(CommonPool) {

    heavyComputation()

}
```

What is this?

```
launch(CommonPool) {
    heavyComputation()
}
```

What is this?

Coroutine Context

```
launch(CommonPool) {
    heavyComputation()
}
```

What is this?

```
launch(CommonPool) {
    heavyComputation()
}
```

What is this?

Suspending Lambda

```
launch(CommonPool) {

    heavyComputation()

}
```

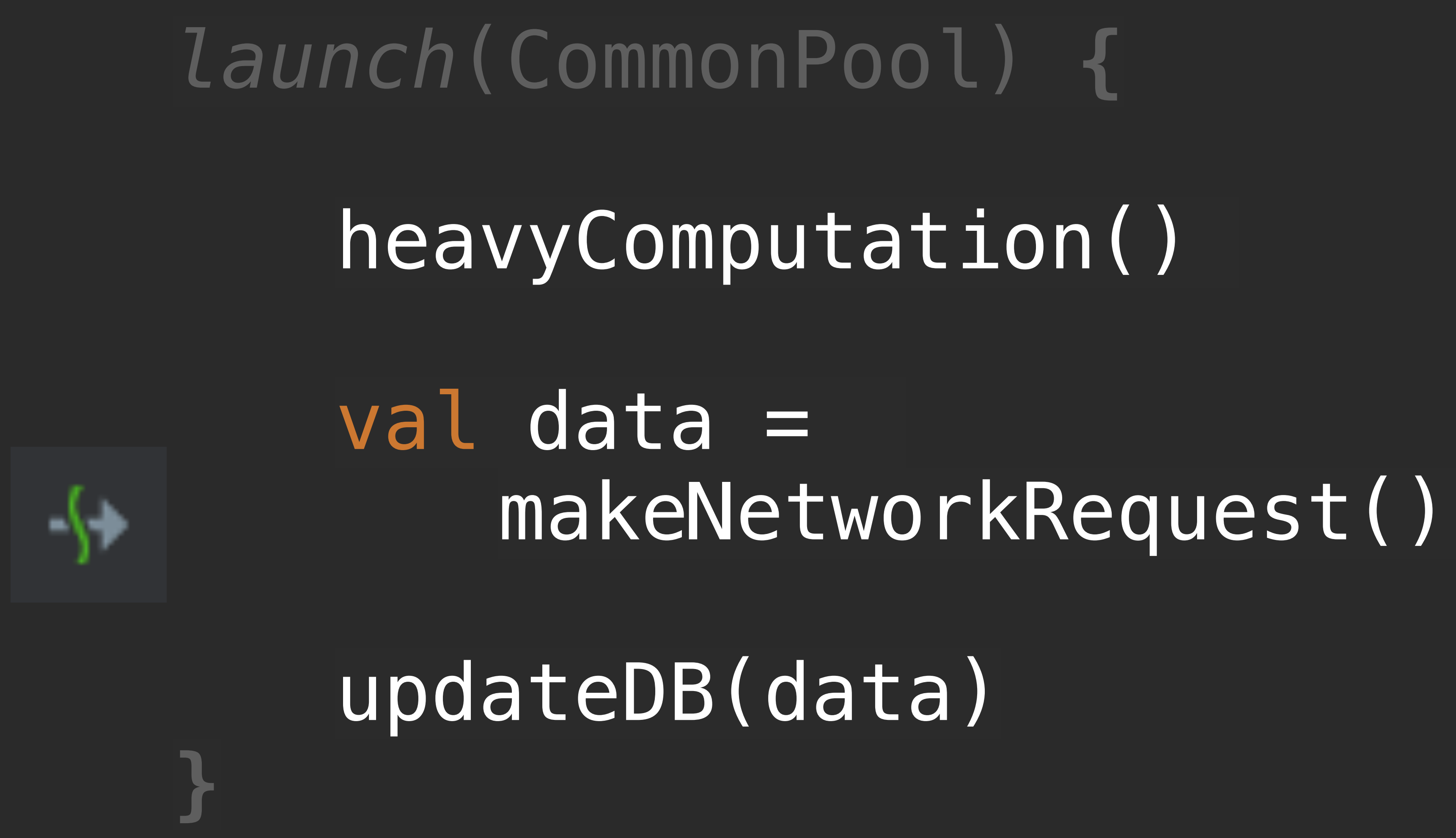
What is this?

Suspending Lambda

```
launch(CommonPool) {
    heavyComputation()
    val data =
        makeNetworkRequest()
    updateDB(data)
}
```

What is this?

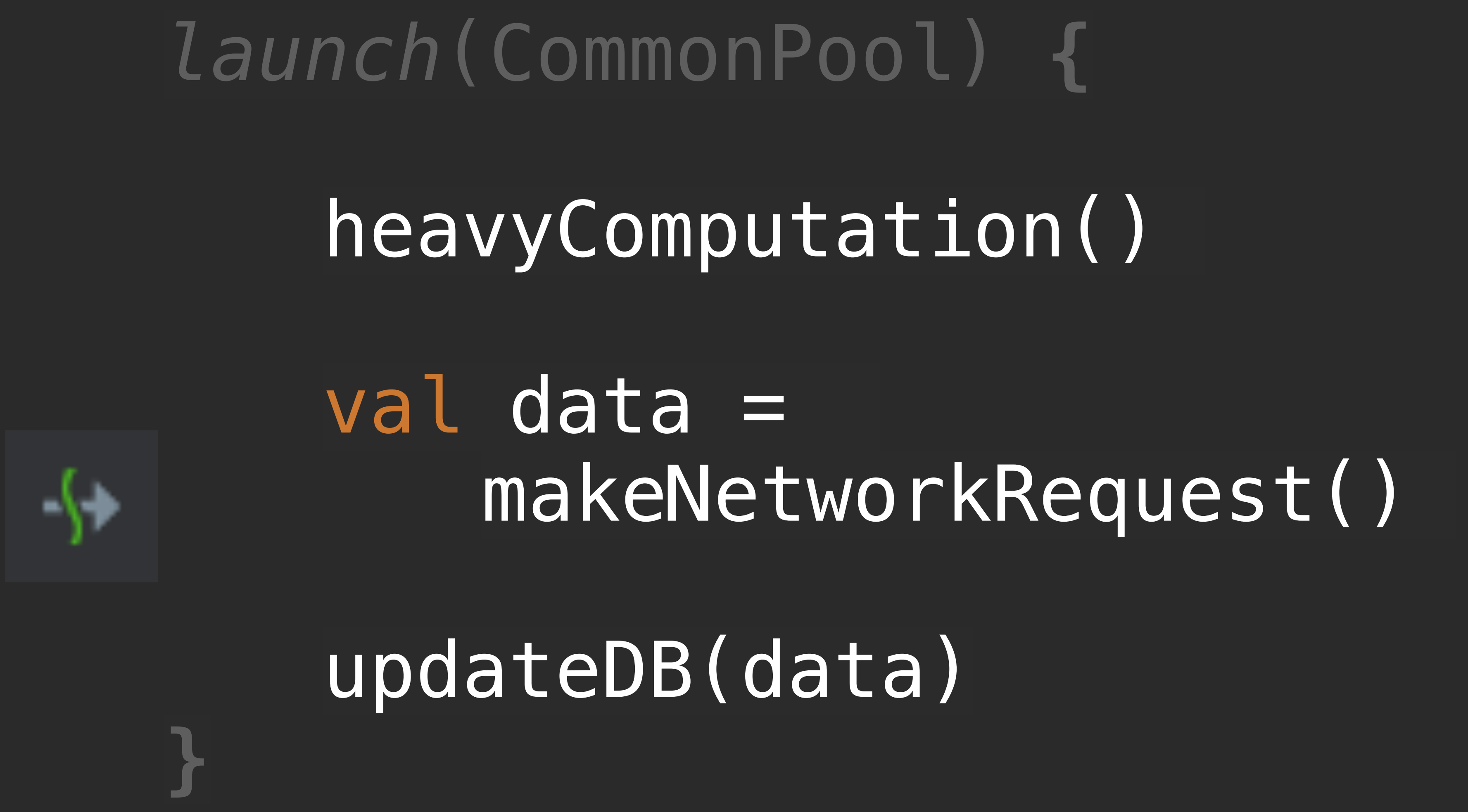
Suspending Lambda



```
launch(CommonPool) {
    heavyComputation()
    val data =
        makeNetworkRequest()
    updateDB(data)
}
```

What is this?

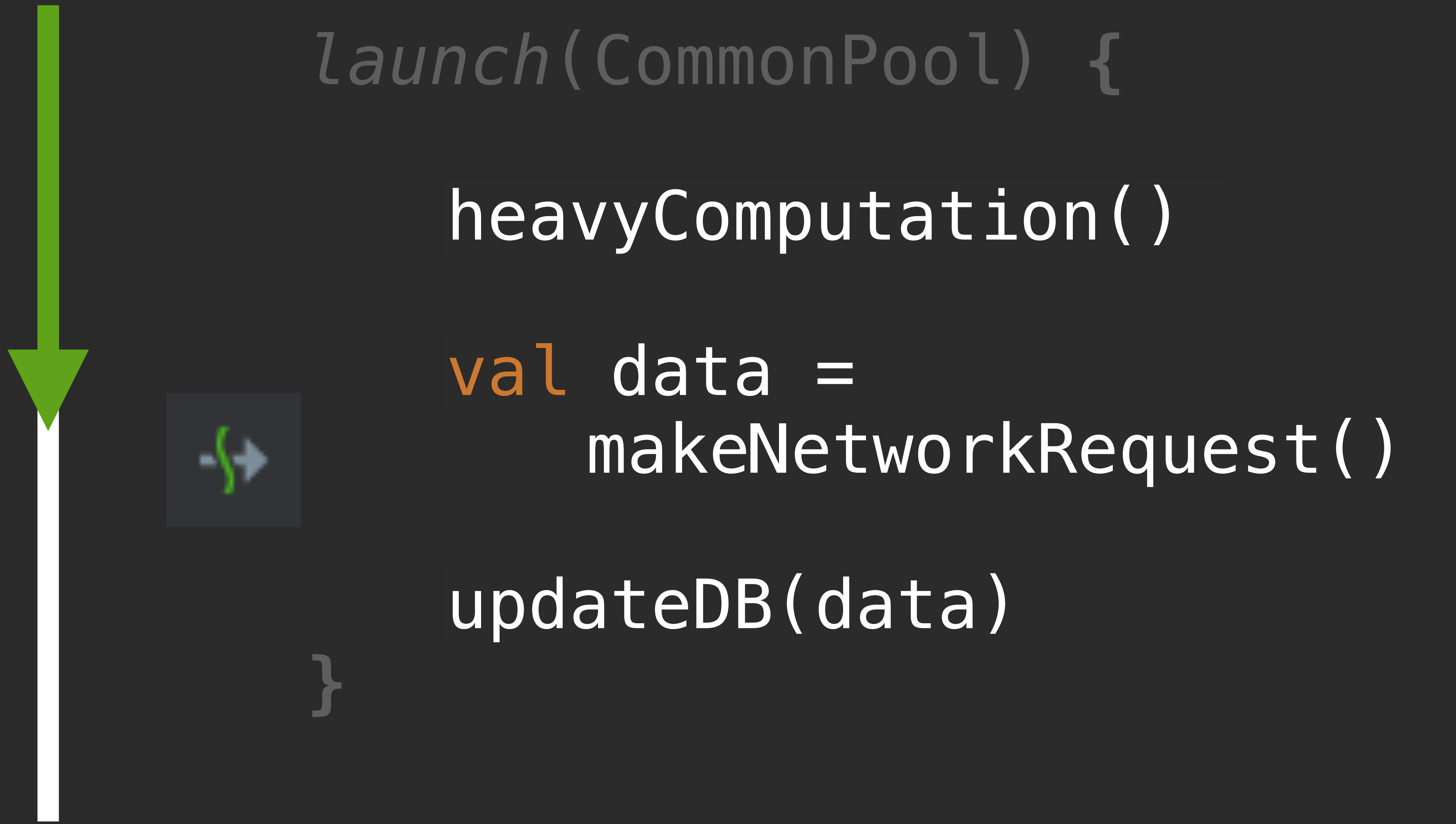
Suspending Lambda



```
launch(CommonPool) {
    heavyComputation()
    val data =
        makeNetworkRequest()
    updateDB(data)
}
```

What is this?

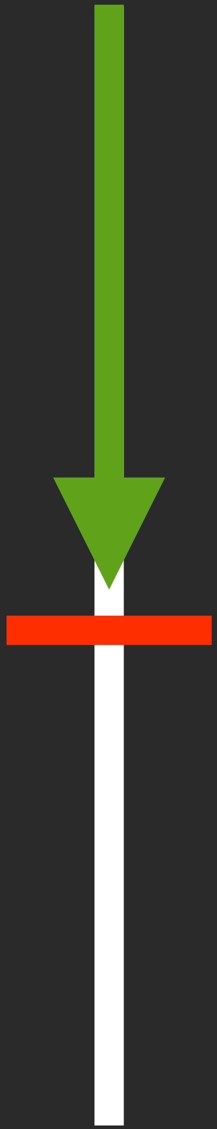
Suspending Lambda



```
launch(CommonPool) {
    heavyComputation()
    val data =
        makeNetworkRequest()
    updateDB(data)
}
```

What is this?

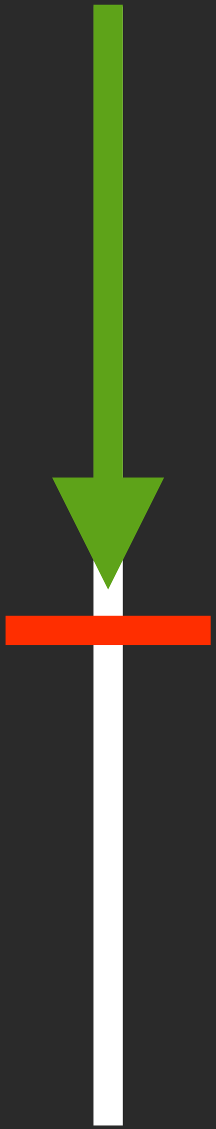
Suspending Lambda



```
launch(CommonPool) {
    heavyComputation()
    val data =
        makeNetworkRequest()
    updateDB(data)
}
```

What is this?

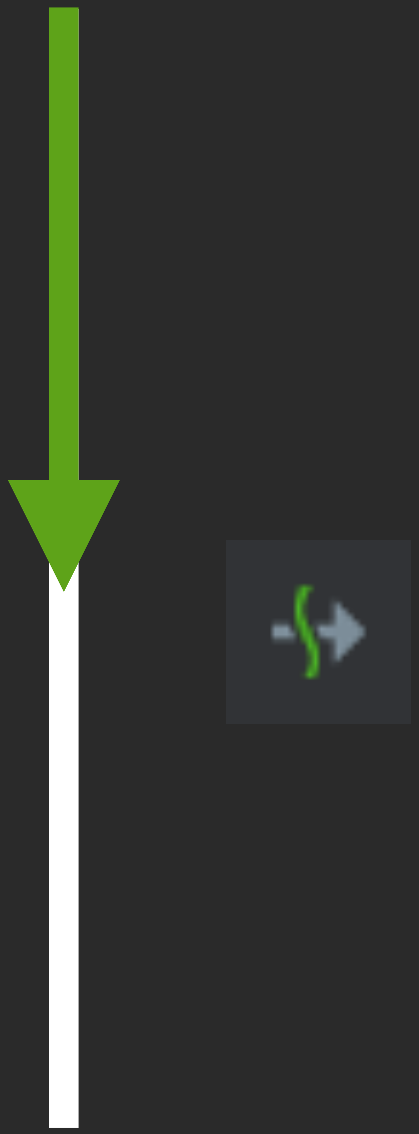
Suspending Lambda



```
launch(CommonPool) {
    heavyComputation()
    val data =
        makeNetworkRequest()
    updateDB(data)
}
```

What is this?

Suspending Lambda



```
launch(CommonPool) {
    heavyComputation()
    val data =
        makeNetworkRequest()
    updateDB(data)
}
```

What is this?

Suspending Lambda

```
launch(CommonPool) {

    heavyComputation()

    val data =
        makeNetworkRequest()

    updateDB(data)
}
```

COROUTINES RECAP COMPLETED

COROUTINES - RXJAVA COMPARISON

CANCEL EXECUTION

How to cancel an Observable?

With Disposables

How to cancel an Observable?

With Disposables

```
val disposable: Disposable =
     Observable.interval(1, TimeUnit.SECONDS)
               .subscribe()

disposable.dispose()
```

How to cancel an Observable?

With Disposables

```
val disposable: Disposable =
     Observable.interval(1, TimeUnit.SECONDS)
               .subscribe()

disposable.dispose()
```

How to cancel a Coroutine?

With the Coroutine Job (from the Coroutine Context)

How to cancel a Coroutine?

With the Coroutine Job (from the Coroutine Context)

```kotlin
val job = launch(CommonPool) {
    // my suspending block
}

job.cancel()
```

How to cancel a Coroutine?

With the Coroutine Job (from the Coroutine Context)

```
val parentJob = Job()

launch(parentJob + CommonPool) {
    // my suspending block
}

parentJob.cancel()
```

How to cancel a Coroutine?

With the Coroutine Job (from the Coroutine Context)

```kotlin
val parentJob = Job()

launch(CommonPool, parent = parentJob) {
    // my suspending block
}

parentJob.cancel()
```

CHANNELS

CHANNELS

- Transfer stream of values
- Similar to Reactive Streams **Publisher** or RxJava **Observable/Flowable**

CHANNELS

- Transfer stream of values
- Similar to Reactive Streams **Publisher** or RxJava **Observable/Flowable**

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}
```

CHANNELS

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}
```

CHANNELS

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}
```

CHANNELS

```kotlin
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}
```

CHANNELS

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}           )
```

CHANNELS

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}.subscribe()
```

CHANNELS

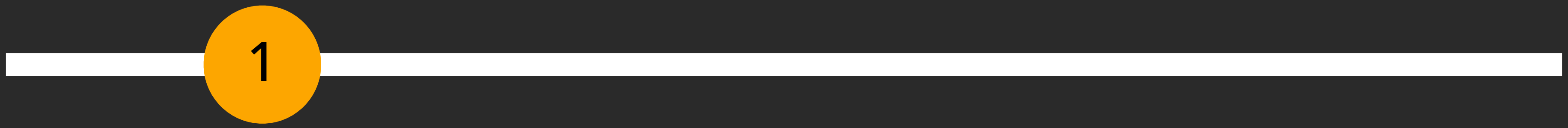
```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}.subscribe()
```

Observer Timeline

CHANNELS

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}.subscribe()
```

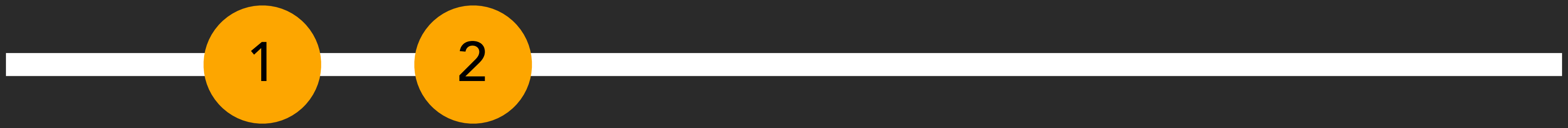
Observer Timeline



CHANNELS

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}.subscribe()
```

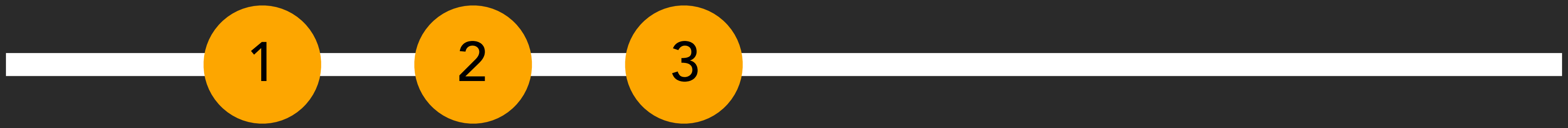
Observer Timeline



CHANNELS

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}.subscribe()
```

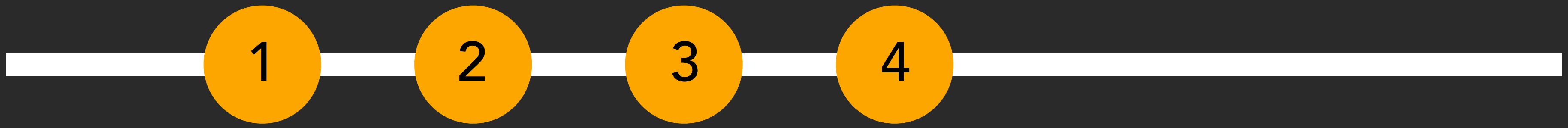
Observer Timeline



CHANNELS

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}.subscribe()
```

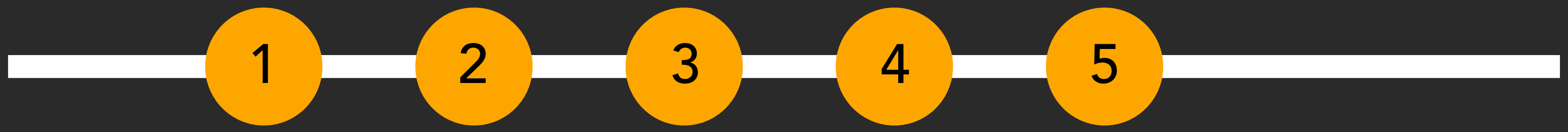
Observer Timeline



CHANNELS

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}.subscribe()
```

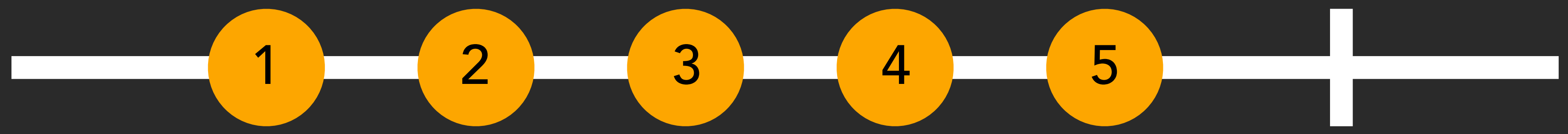
Observer Timeline



CHANNELS

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}.subscribe()
```

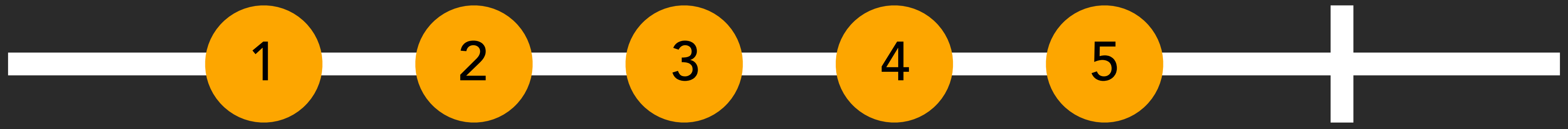
Observer Timeline



CHANNELS

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}.subscribe()
```

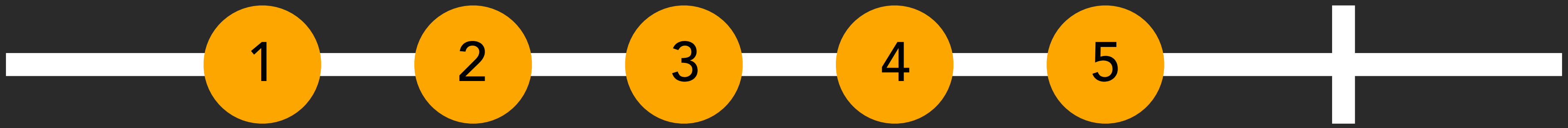
Observer Timeline



CHANNELS

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}.subscribe()
```

Observer Timeline

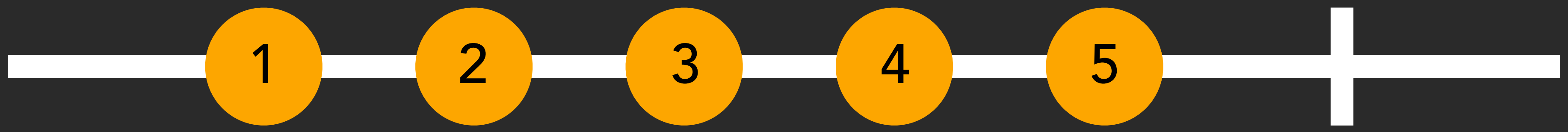


Observer 2 Timeline

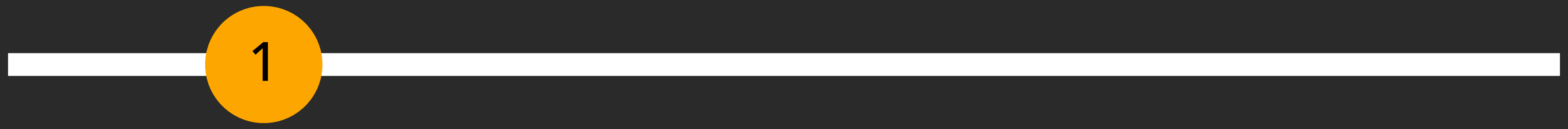
CHANNELS

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}.subscribe()
```

Observer Timeline



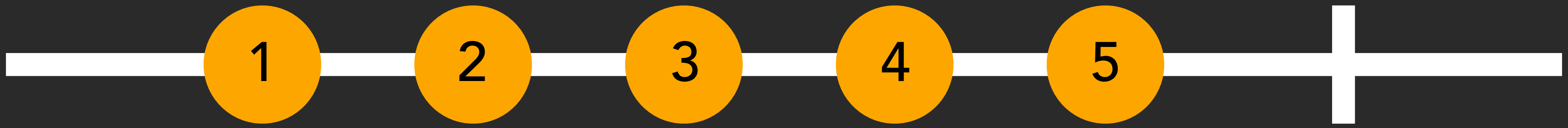
Observer 2 Timeline



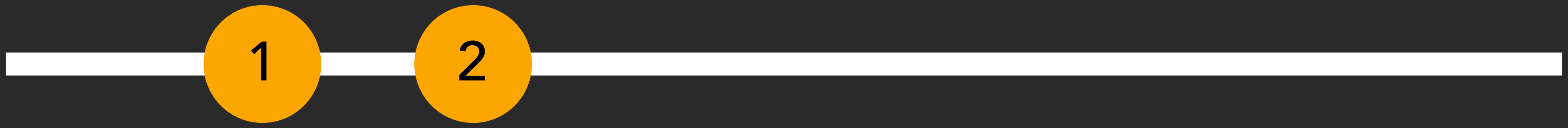
CHANNELS

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}.subscribe()
```

Observer Timeline



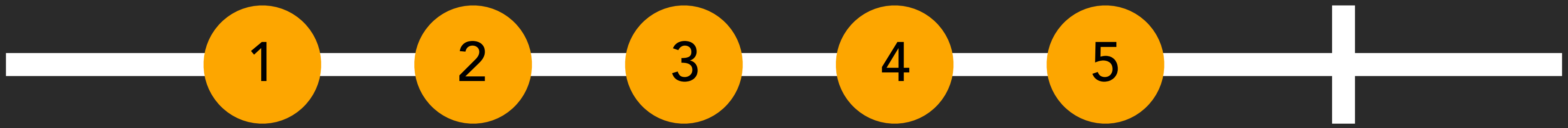
Observer 2 Timeline



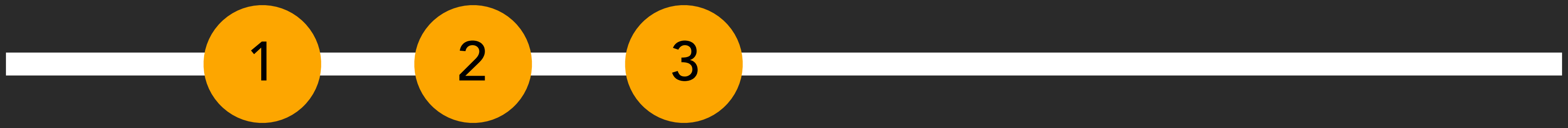
CHANNELS

```
Observable.create<Int> { emitter ->
  for (i in 1..5) {
    emitter.onNext(i)
  }
  emitter.onComplete()
}.subscribe()
```

Observer Timeline



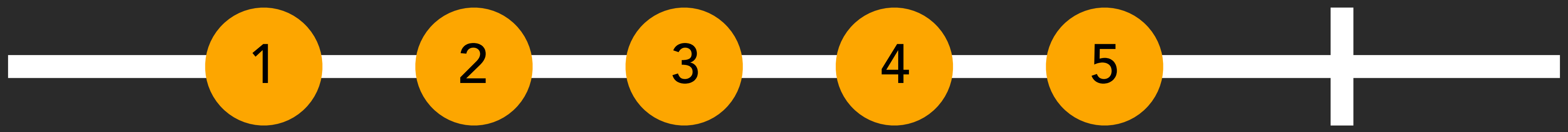
Observer 2 Timeline



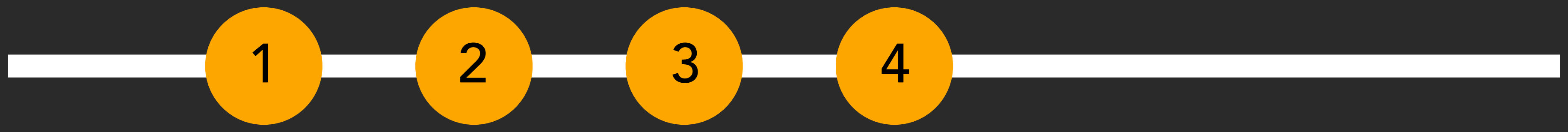
CHANNELS

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}.subscribe()
```

Observer Timeline



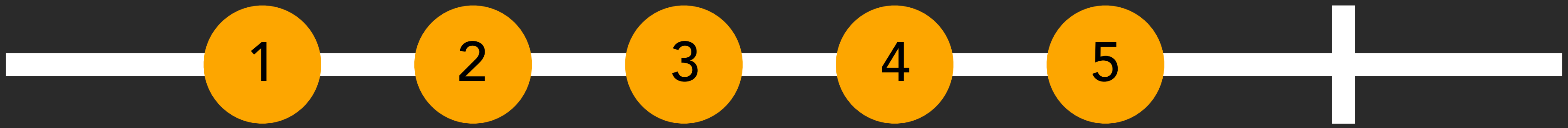
Observer 2 Timeline



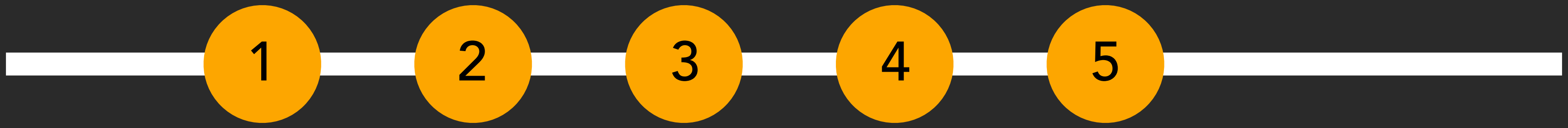
CHANNELS

```
Observable.create<Int> { emitter ->
  for (i in 1..5) {
    emitter.onNext(i)
  }
  emitter.onComplete()
}.subscribe()
```

Observer Timeline



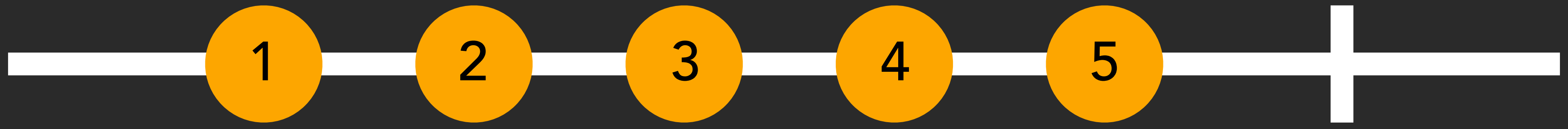
Observer 2 Timeline



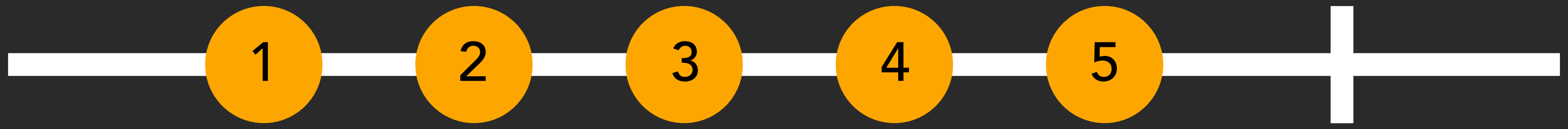
CHANNELS

```
Observable.create<Int> { emitter ->
    for (i in 1..5) {
        emitter.onNext(i)
    }
    emitter.onComplete()
}.subscribe()
```

Observer Timeline



Observer 2 Timeline



CHANNELS

- Transfer stream of values
- Can be shared between different Coroutines
- By default, channel capacity == 1

CHANNELS

- Transfer stream of values
- Can be shared between different Coroutines
- By default, channel capacity == 1

```
interface Channel<E> : SendChannel<E>, ReceiveChannel<E>
```

CHANNELS

```
interface Channel<E> : SendChannel<E>, ReceiveChannel<E>
```

CHANNELS

```
interface Channel<E> : SendChannel<E>, ReceiveChannel<E>



public interface SendChannel<in E> {

  public suspend fun send(element: E)
  public fun offer(element: E)
  public fun close(cause: Throwable? = null): Boolean
}

public interface ReceiveChannel<out E> {

  public suspend fun receive(): E
  public fun close(cause: Throwable? = null): Boolean
}
```

CHANNELS

```
interface Channel<E> : SendChannel<E>, ReceiveChannel<E>
```

```
public interface SendChannel<in E> {

   public suspend fun send(element: E)
   public fun offer(element: E)
   public fun close(cause: Throwable? = null): Boolean
}

public interface ReceiveChannel<out E> {

   public suspend fun receive(): E
   public fun close(cause: Throwable? = null): Boolean
}
```

CHANNELS

```
interface Channel<E> : SendChannel<E>, ReceiveChannel<E>


public interface SendChannel<in E> {

   public suspend fun send(element: E)
   public fun offer(element: E)
   public fun close(cause: Throwable? = null): Boolean
}

public interface ReceiveChannel<out E> {

   public suspend fun receive(): E
   public fun close(cause: Throwable? = null): Boolean
}
```

CHANNELS

```
interface Channel<E> : SendChannel<E>, ReceiveChannel<E>



public interface SendChannel<in E> {

  public suspend fun send(element: E)
  public fun offer(element: E)
  public fun close(cause: Throwable? = null): Boolean
}

public interface ReceiveChannel<out E> {

  public suspend fun receive(): E
  public fun close(cause: Throwable? = null): Boolean
}
```

CHANNELS

```kotlin
val channel = Channel<Int>()
```

Channel

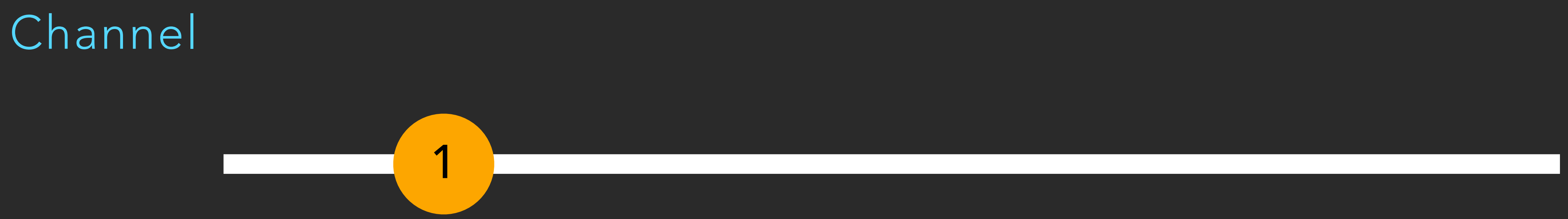
CHANNELS

```
val channel = Channel<Int>()

launch {
    channel.send(1)
}
```

Channel

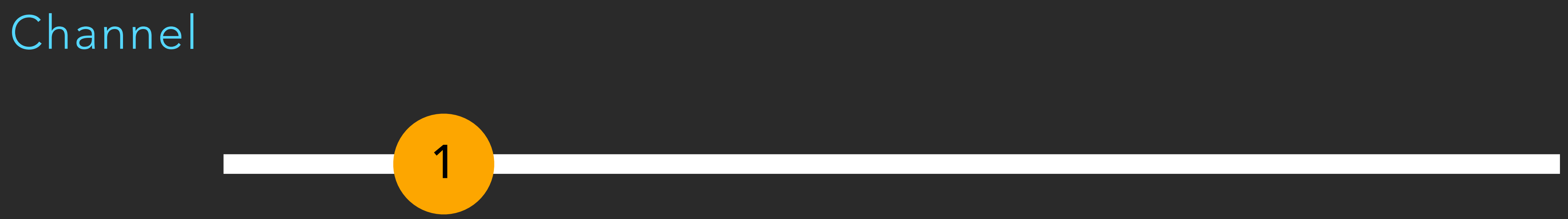
CHANNELS

```
val channel = Channel<Int>()

launch {
    channel.send(1)
}
```



CHANNELS

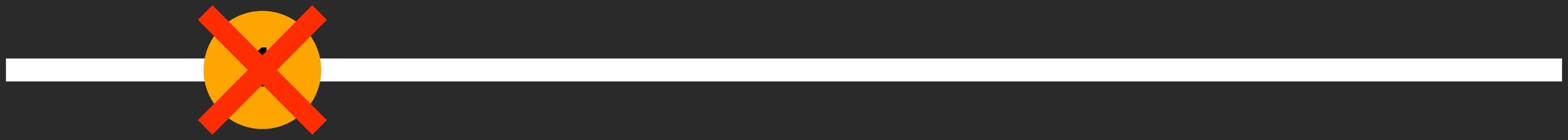
```
val channel = Channel<Int>()

launch {
    channel.send(1)
}

launch {
    val value = channel.receive()
}
```



CHANNELS

```
val channel = Channel<Int>()

launch {
    channel.send(1)
}

launch {
    val value = channel.receive()
}
```

Channel



CHANNELS

```
val channel = Channel<Int>()
```

Channel

Channel Capacity

1

CHANNELS

```
val channel = Channel<Int>()
launch {
    channel.send(1)
    channel.send(2)
}
```

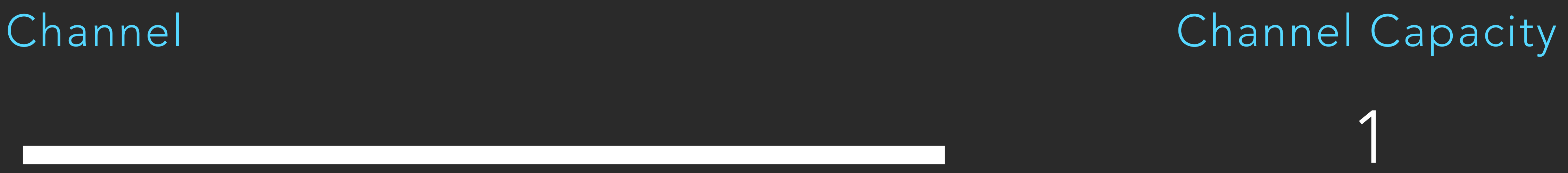
Channel

Channel Capacity

1

CHANNELS

```kotlin
val channel = Channel<Int>()
launch {
    channel.send(1)
    channel.send(2)
}
```



Channel

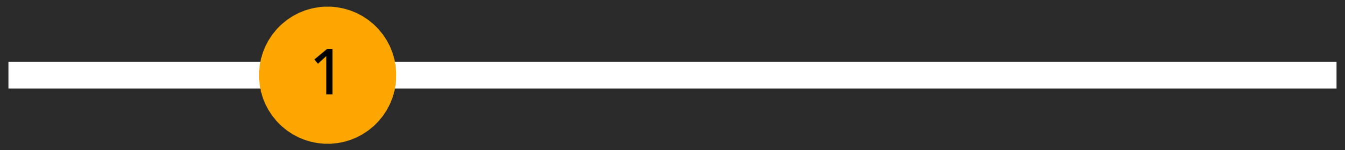
Channel Capacity

1

CHANNELS

```
val channel = Channel<Int>()
launch {
 ➡ channel.send(1)
    channel.send(2)
}
```

Channel



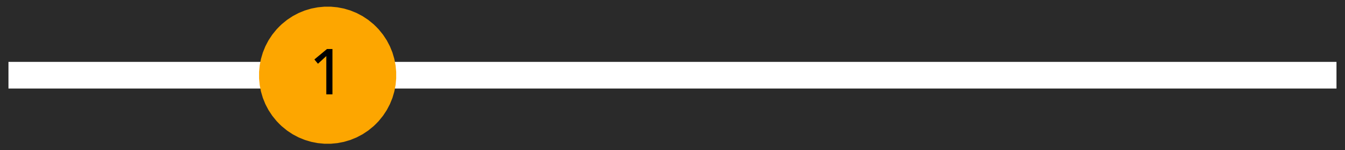
Channel Capacity

0

CHANNELS

```
val channel = Channel<Int>()

launch {
    channel.send(1)
    channel.send(2)
}
```

Channel



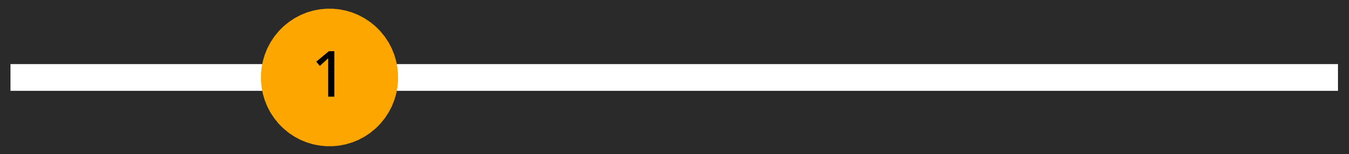
Channel Capacity

0

CHANNELS

```kotlin
val channel = Channel<Int>()

launch {
    channel.send(1)
    channel.send(2) // Suspended until the
}                   // channel is NOT full
```

Channel



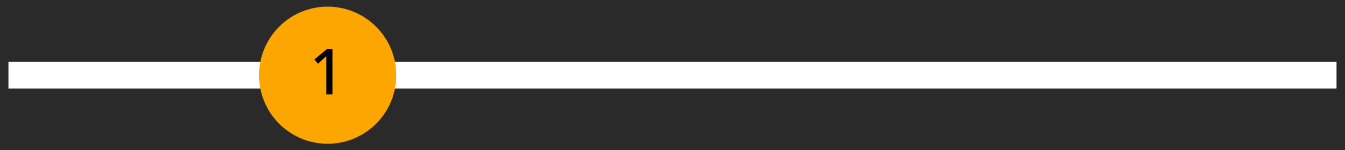
Channel Capacity

0

CHANNELS

```
val channel = Channel<Int>()

launch {
    channel.send(1)
    channel.send(2) // Suspended until the
}                   // channel is NOT full

launch {
    val value = channel.receive()
}
```

Channel

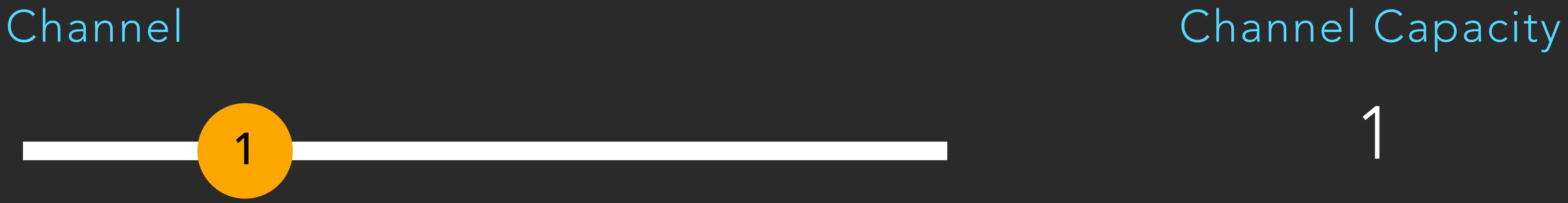


Channel Capacity

0

CHANNELS

```
val channel = Channel<Int>()

launch {
    channel.send(1)
    channel.send(2) // Suspended until the
}                   // channel is NOT full

launch {
    val value = channel.receive()
}
```



Channel

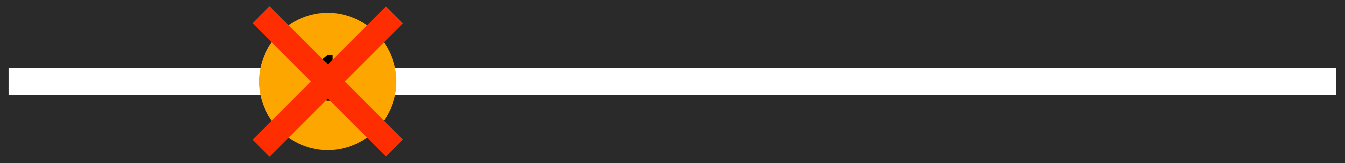
Channel Capacity

1

CHANNELS

```
val channel = Channel<Int>()

launch {
      channel.send(1)
      channel.send(2) // Suspended until the
}                   // channel is NOT full

launch {
     val value = channel.receive()
}
```

Channel



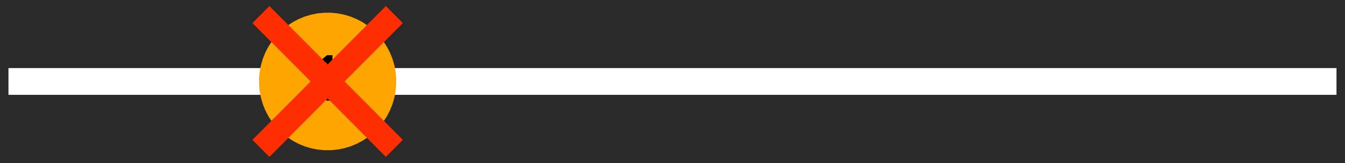
Channel Capacity

1

CHANNELS

```
val channel = Channel<Int>()

launch {
        channel.send(1)
        channel.send(2)
}

launch {
        val value = channel.receive()
}
```

Channel

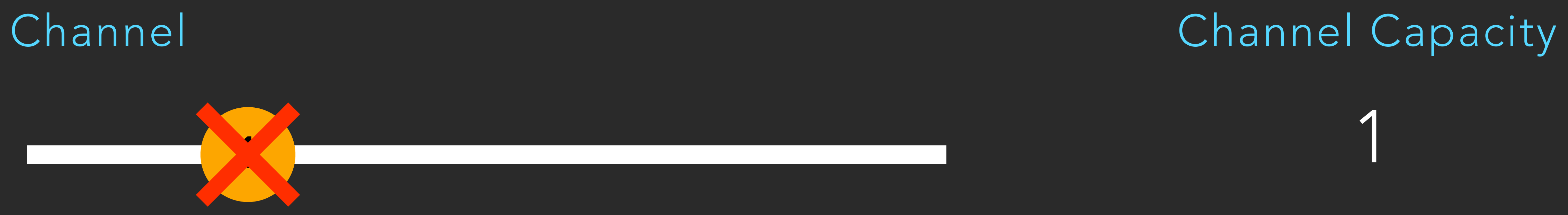


Channel Capacity

1

CHANNELS

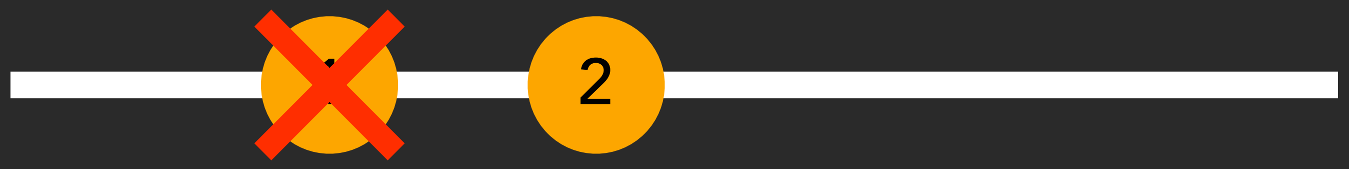
```
val channel = Channel<Int>()

launch {
    channel.send(1)
    channel.send(2)
}

launch {
    val value = channel.receive()
}
```



CHANNELS

```
val channel = Channel<Int>()

launch {
    channel.send(1)
➡   channel.send(2)
}

launch {
    val value = channel.receive()
}
```

Channel



Channel Capacity

0

CHANNELS

```
val channel = Channel<Int>()
launch {
    for (x in 1..5) channel.send(x)
}
```

CHANNELS

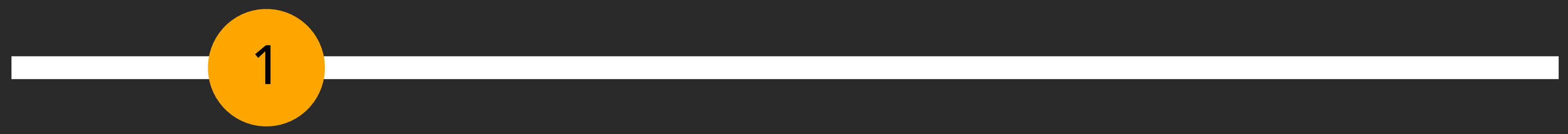
```
val channel = Channel<Int>()
launch {
    for (x in 1..5) channel.send(x)
}

launch {
    for (value in channel) {
        consumeValue(value)
    }
}
```

Channel

CHANNELS

```
val channel = Channel<Int>()
launch {
    for (x in 1..5) channel.send(x)
}

launch {
    for (value in channel) {
        consumeValue(value)
    }
}
```

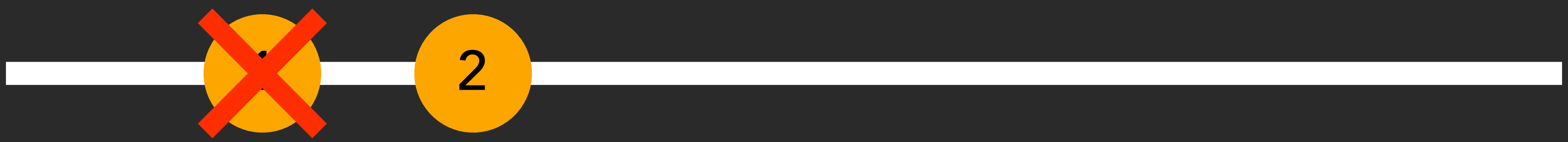
Channel



CHANNELS

```kotlin
val channel = Channel<Int>()
launch {
    for (x in 1..5) channel.send(x)
}

launch {
    for (value in channel) {
        consumeValue(value)
    }
}
```

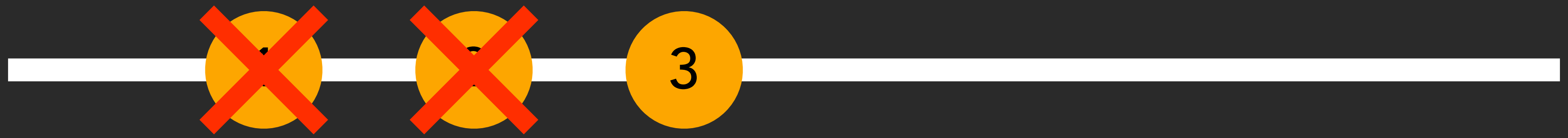
Channel



CHANNELS

```
val channel = Channel<Int>()
launch {
    for (x in 1..5) channel.send(x)
}

launch {
    for (value in channel) {
        consumeValue(value)
    }
}
```

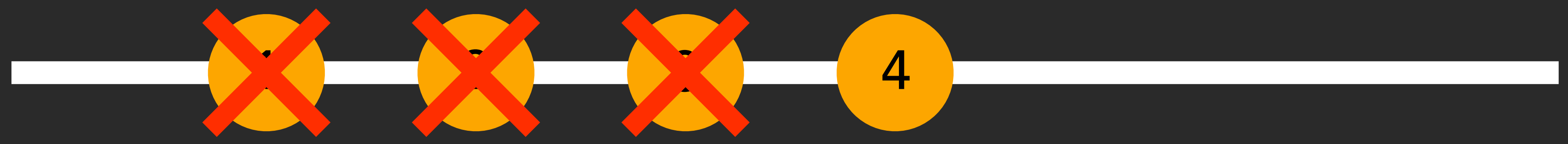
Channel



CHANNELS

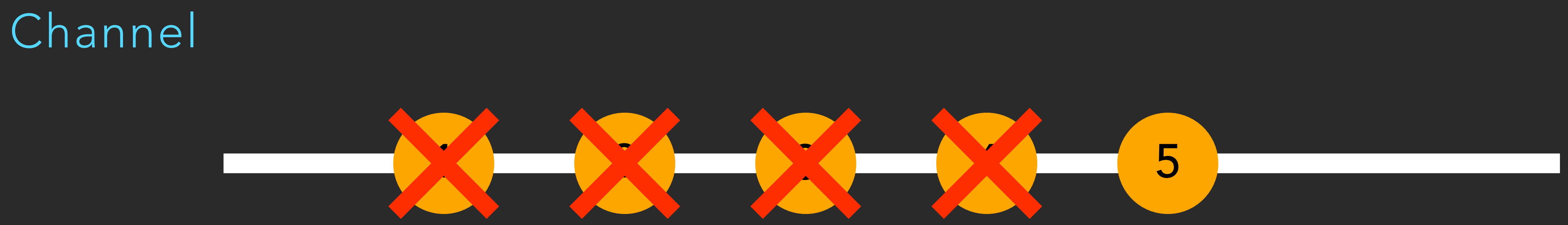
```
val channel = Channel<Int>()
launch {
    for (x in 1..5) channel.send(x)
}

launch {
    for (value in channel) {
        consumeValue(value)
    }
}
```

Channel



CHANNELS

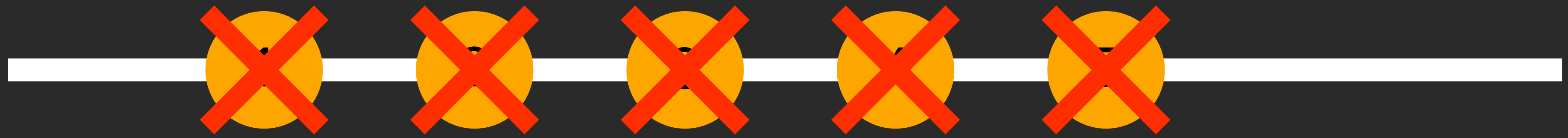
```
val channel = Channel<Int>()
launch {
    for (x in 1..5) channel.send(x)
}

launch {
    for (value in channel) {
        consumeValue(value)
    }
}
```



CHANNELS

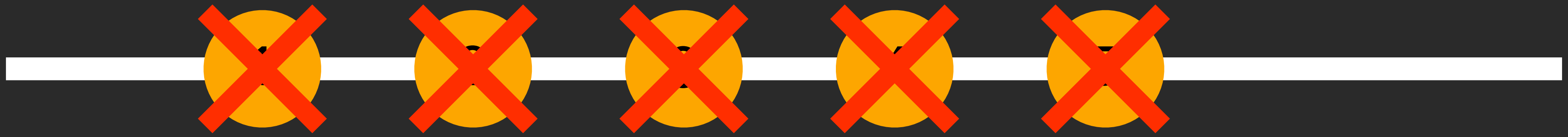
```
val channel = Channel<Int>()
launch {
    for (x in 1..5) channel.send(x)
}

launch {
    for (value in channel) {
        consumeValue(value)
    }
}
```

Channel



CHANNELS

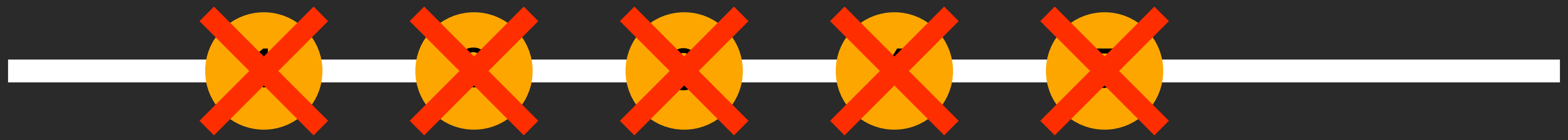
Channel



```
launch {
    consumeValue(channel.receive())
}
```

CHANNELS

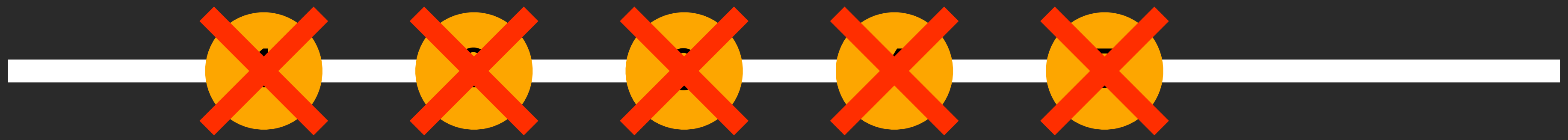
Channel



```
launch {
   consumeValue(channel.receive())
}
```

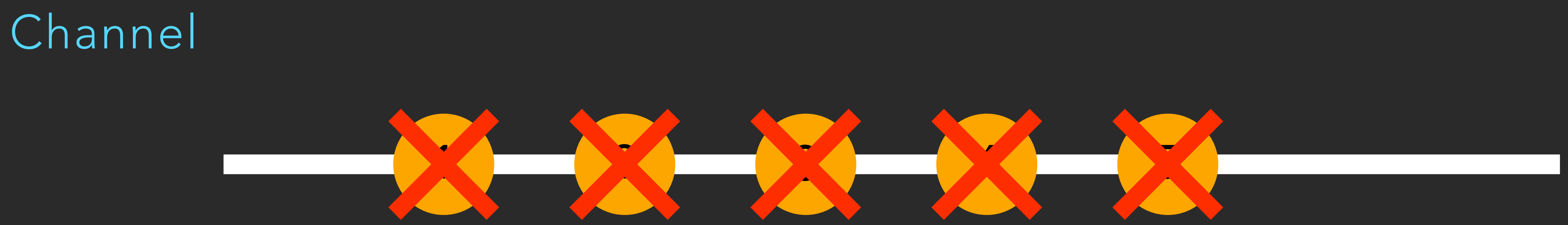
CHANNELS

Channel



```
launch {
    consumeValue(channel.receive())
}
```

CHANNELS



```
launch {
    consumeValue(channel.receive())
}
```

```
        // Suspended until the
        // channel is NOT empty
```

BUFFERED CHANNELS

- **Channels** take an optional **capacity** parameter
- Allow senders to send multiple elements before suspending

BUFFERED CHANNELS

- **Channels** take an optional **capacity** parameter
- Allow senders to send multiple elements before suspending

```
val channel = Channel<Int>(3)
```

CHANNELS

- We can also use **produce** that implements the ReceiveChannel interface
- Only the code inside ***produce*** can send elements to the channel
- Useful to create custom operators

CHANNELS

- We can also use **produce** that implements the ReceiveChannel interface
- Only the code inside ***produce*** can send elements to the channel
- Useful to create custom operators

```
val publisher = produce(capacity = 2) {
    for (x in 1..5) send(x)
}
```

CHANNELS

```kotlin
val publisher = produce(capacity = 2) {
    for (x in 1..5) send(x)
}
```

CHANNELS

```
val publisher = produce(capacity = 2) {
    for (x in 1..5) send(x)
}


launch {
    publisher.consumeEach {
        consumeValue(it)
    }
}
```

RACE CONDITION IN CHANNELS

```kotlin
val channel = Channel<Int>()
```

RACE CONDITION IN CHANNELS

```
val channel = Channel<Int>()


launch {
    val value1 = channel.receive()
}

launch {
    val value2 = channel.receive()
}
```

RACE CONDITION IN CHANNELS

```
val channel = Channel<Int>()

launch {
    val value1 = channel.receive()
}
launch {
    val value2 = channel.receive()
}

launch {
    channel.send(1)
}
```

BROADCAST CHANNEL

BROADCAST CHANNEL

- Similar to RxJava **Subjects**
- Rx Hot Observable behavior

BROADCAST CHANNEL

- Similar to RxJava **Subjects**
- Rx Hot Observable behavior

```
publishSubject.subscribe {
    consumeValue(it)
}
```

Observer 1 Timeline

```
publishSubject.subscribe {
    println(it)
}
```

Observer 2 Timeline

BROADCAST CHANNEL

- Similar to RxJava **Subjects**

- Rx Hot Observable behavior

```
publishSubject.subscribe {
    consumeValue(it)
}
```

```
publishSubject.subscribe {
    println(it)
}
```

```
publishSubject.onNext(3)
```

Observer 1 Timeline

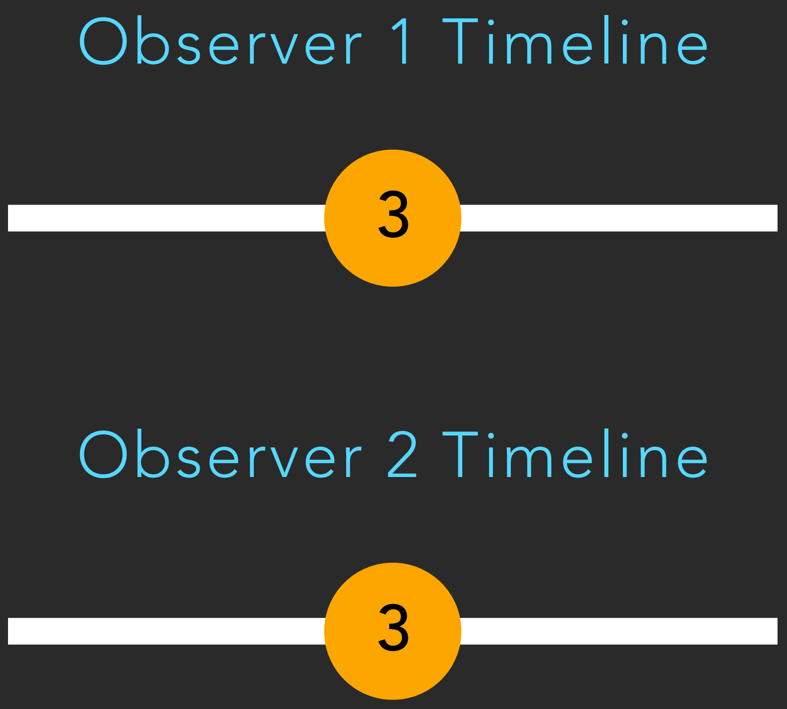
Observer 2 Timeline

BROADCAST CHANNEL

- Similar to RxJava **Subjects**

- Rx Hot Observable behavior

```
publishSubject.subscribe {
    consumeValue(it)
}


publishSubject.subscribe {
    println(it)
}


publishSubject.onNext(3)
```



BROADCAST CHANNEL

- BroadcastChannel implements the **SendChannel<E>** interface

- Emit the same item to multiple consumers that listen for the elements using **openSubscription()**

```
val channel = BroadcastChannel<Int>(2)
```

BROADCAST CHANNEL

```
val channel = BroadcastChannel<Int>(2)
```

```
val observer1Job = launch {
    channel.openSubscription().use { channel ->
        for (value in channel) {
            consumeValue(value)
        }
    }
}
```

BROADCAST CHANNEL

```
val channel = BroadcastChannel<Int>(2)

val observer1Job = launch {
    channel.openSubscription().use { channel ->
        for (value in channel) {
            consumeValue(value)
        }
    }
}
```

BROADCAST CHANNEL

```
val channel = BroadcastChannel<Int>(2)

val observer1Job = launch {
    channel.openSubscription().use { channel ->
        for (value in channel) {
            consumeValue(value)
        }
    }
}
```

BROADCAST CHANNEL

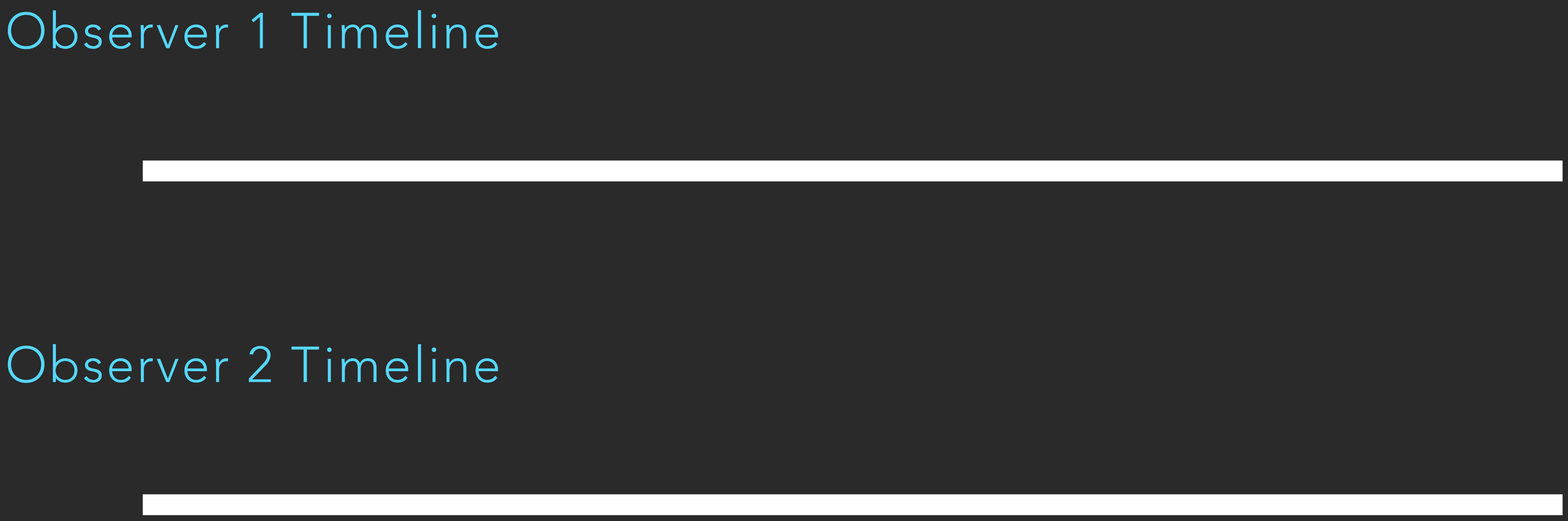
```
val observer2Job = launch {
    channel.consumeEach { value ->
      consumeValue(value)
    }
}
```

BROADCAST CHANNEL

```
val observer2Job = launch {
    channel.consumeEach { value ->
      consumeValue(value)
    }
}
```

Observer 2 Timeline

BROADCAST CHANNEL



Observer 1 Timeline

Observer 2 Timeline

Channel Capacity

2

BROADCAST CHANNEL

Observer 1 Timeline

Observer 2 Timeline

Channel Capacity

2

BROADCAST CHANNEL

Observer 1 Timeline

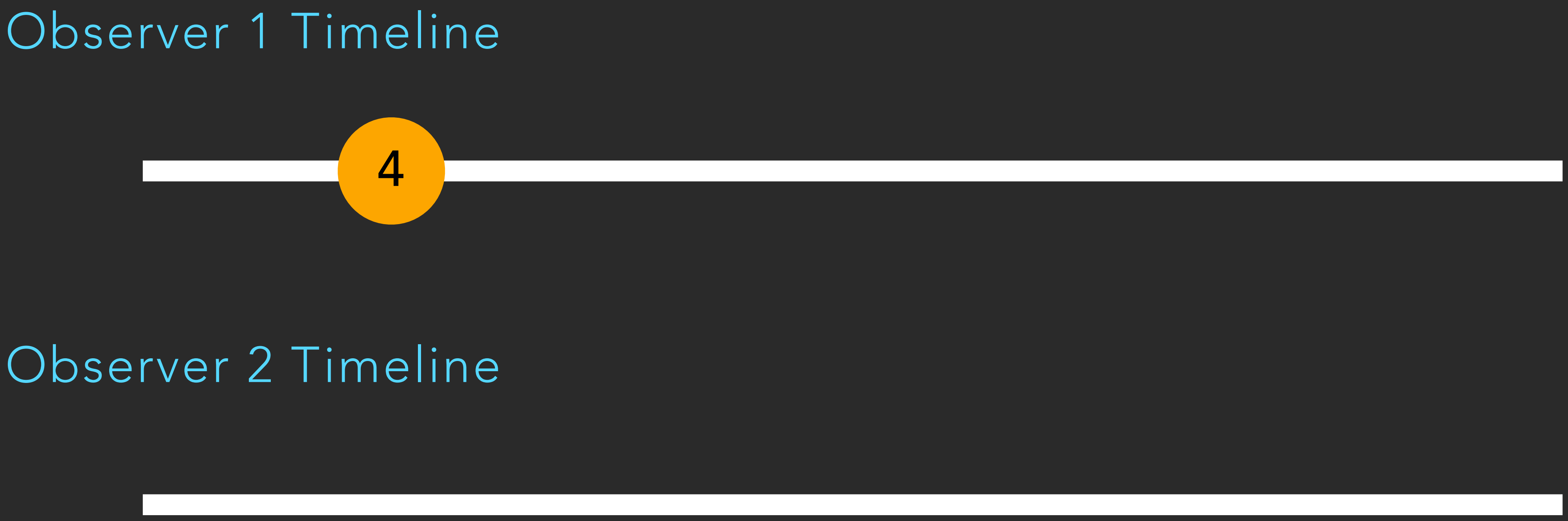
Observer 2 Timeline

```
channel.send(4)
```

Channel Capacity

1

BROADCAST CHANNEL



Observer 1 Timeline

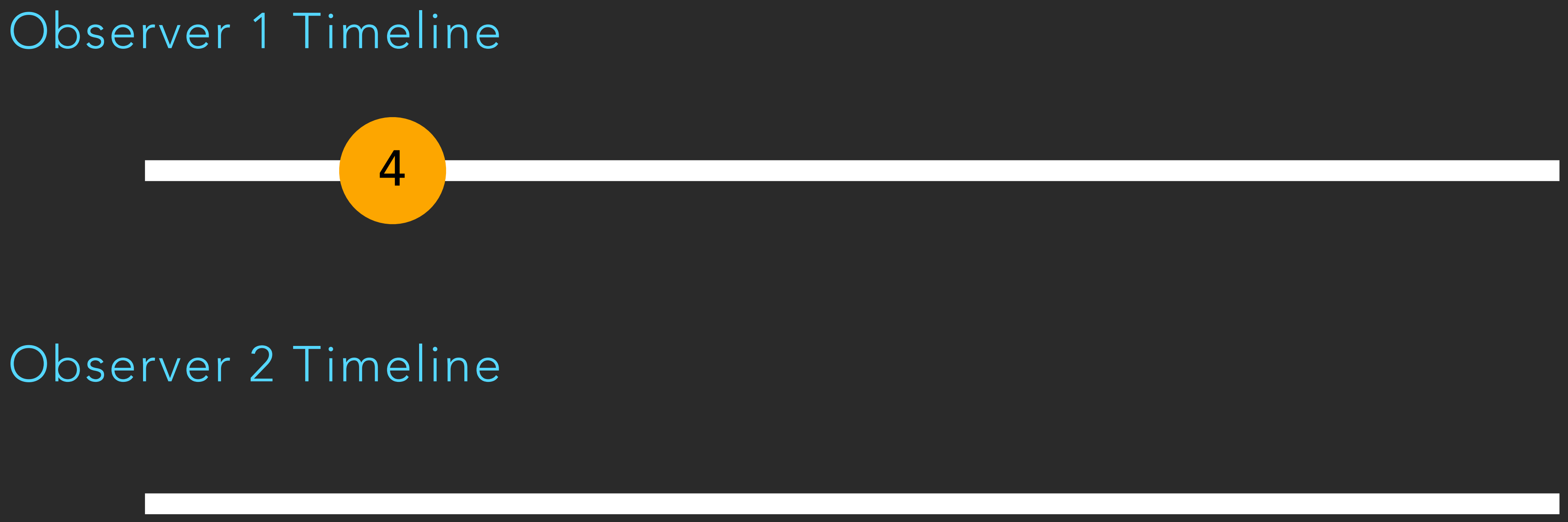
Observer 2 Timeline

```
channel.send(4)
```

Channel Capacity

1

BROADCAST CHANNEL



Observer 1 Timeline

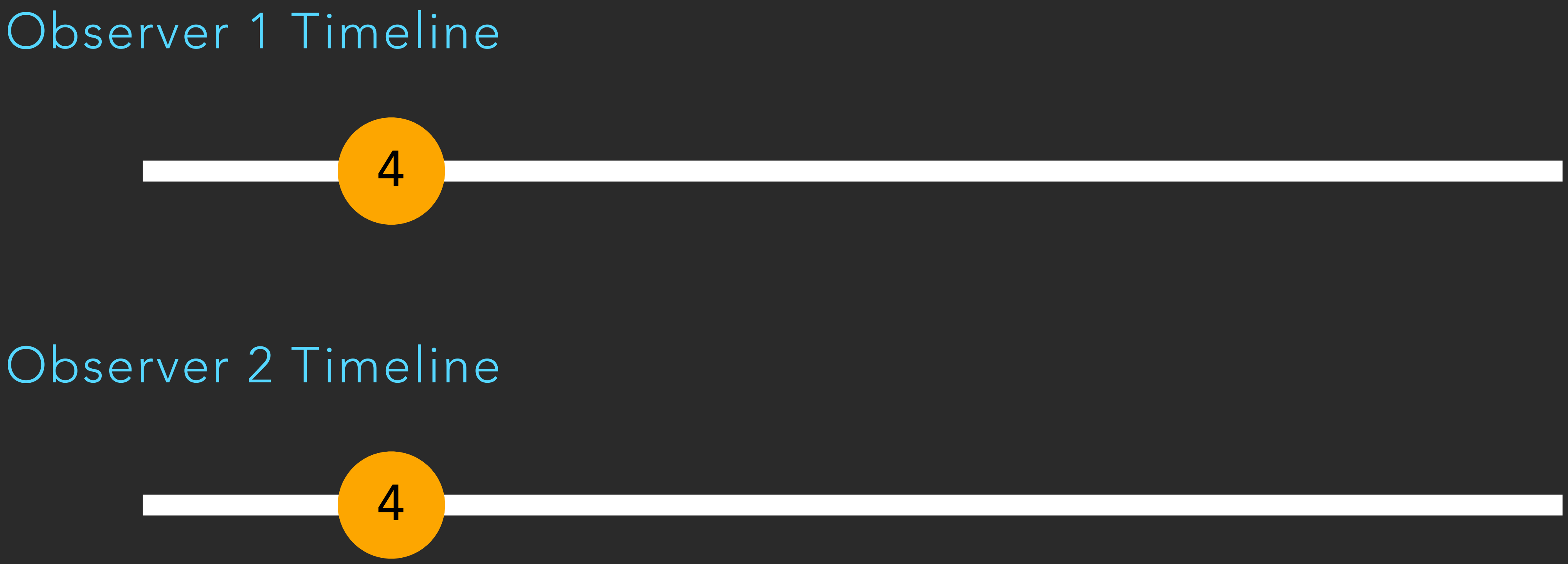
Observer 2 Timeline

```
channel.send(4)
```

Channel Capacity

1

BROADCAST CHANNEL



Observer 1 Timeline

Observer 2 Timeline

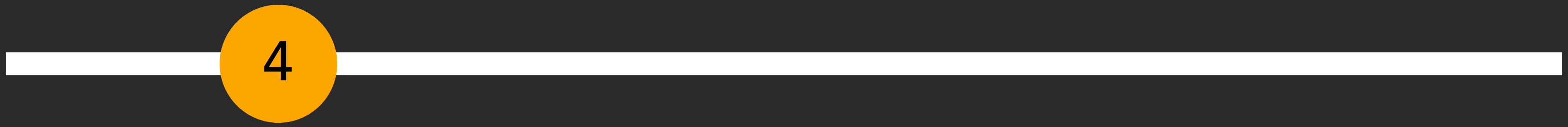
```
channel.send(4)
```

Channel Capacity

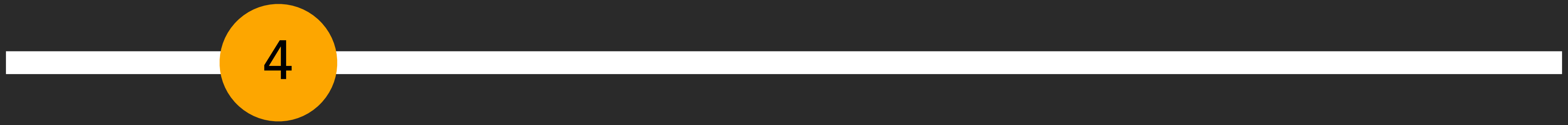
2

BROADCAST CHANNEL

Observer 1 Timeline



Observer 2 Timeline

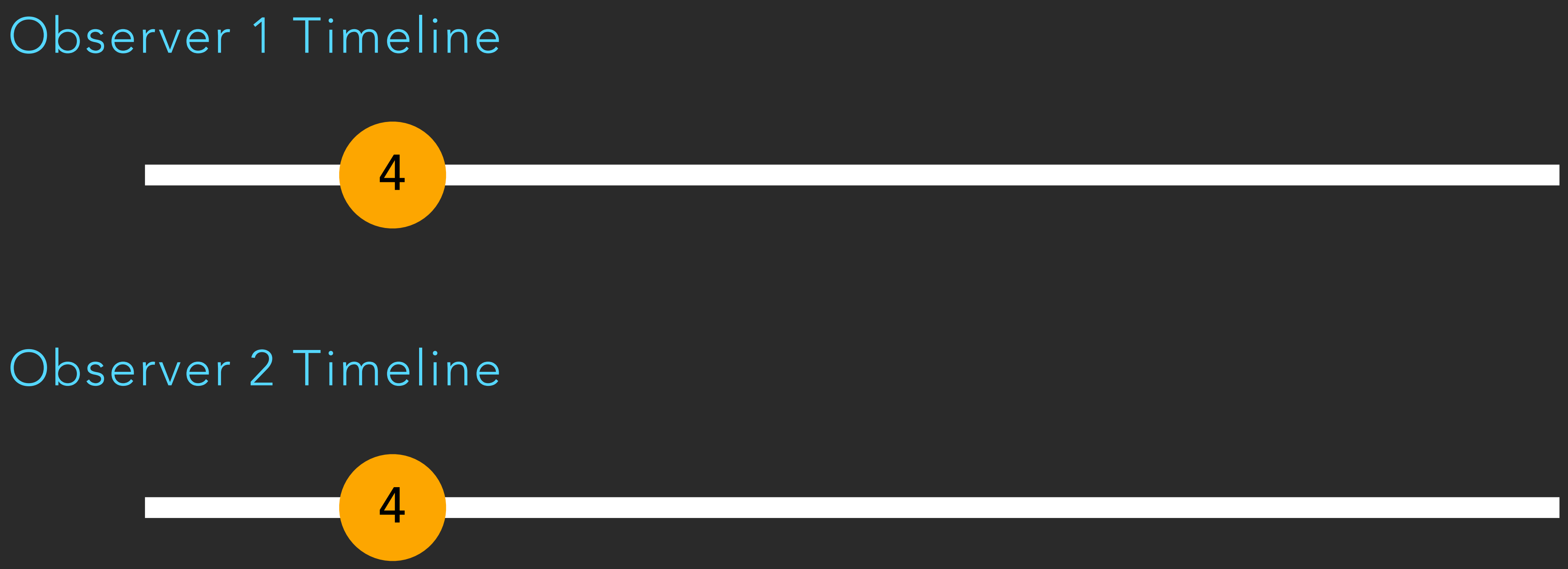


```
channel.send(4)
```

Channel Capacity

2

BROADCAST CHANNEL



Observer 1 Timeline

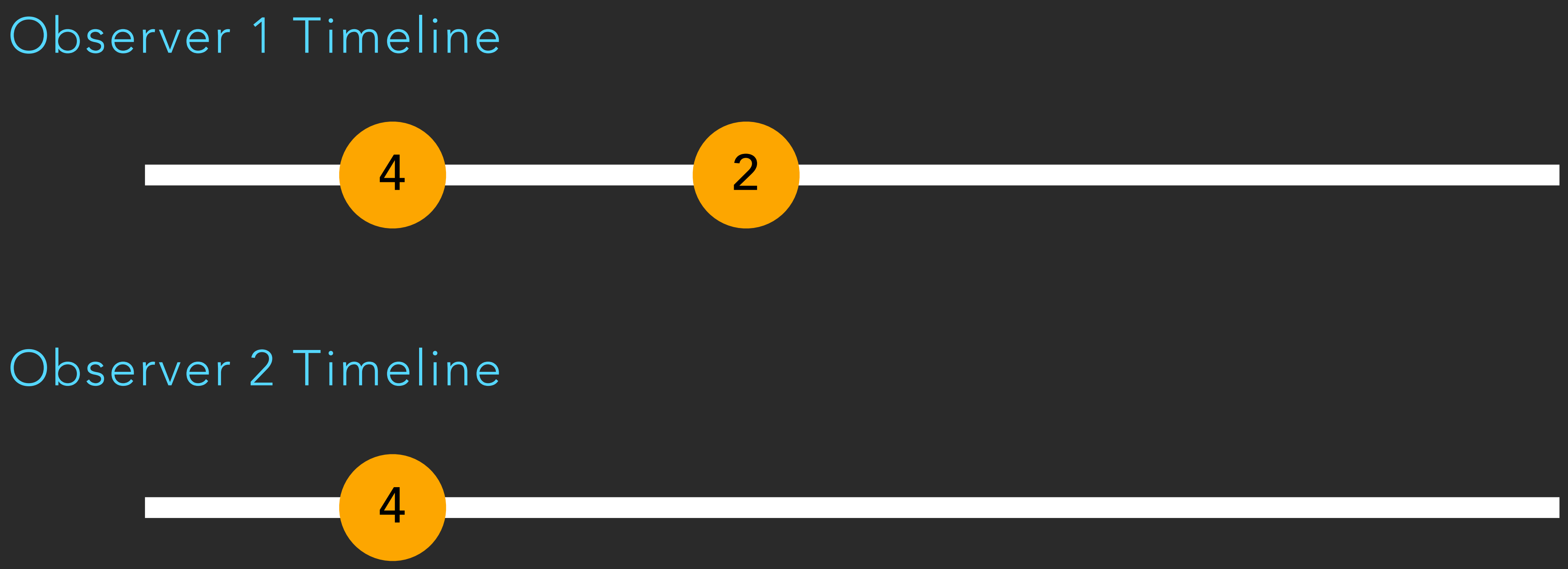
Observer 2 Timeline

```
channel.send(4)

channel.send(2)
```

Channel Capacity

1

BROADCAST CHANNEL



Observer 1 Timeline

Observer 2 Timeline

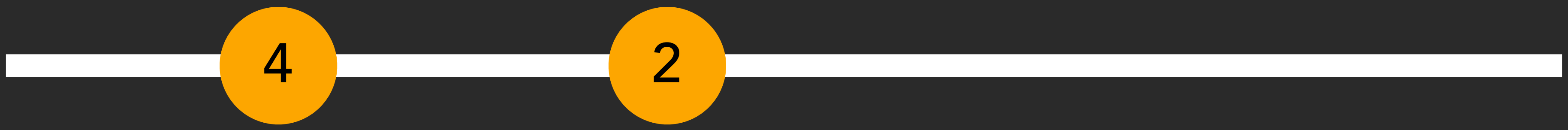
```
channel.send(4)

channel.send(2)
```

Channel Capacity

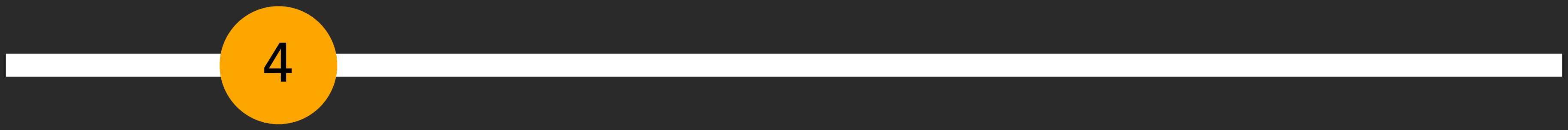
1

BROADCAST CHANNEL

Observer 1 Timeline



Observer 2 Timeline



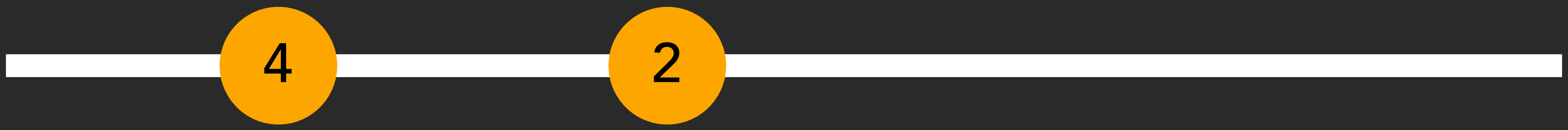
```
channel.send(4)

channel.send(2)
```

Channel Capacity

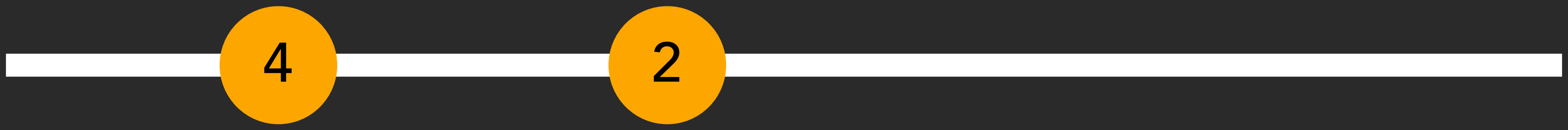
1

BROADCAST CHANNEL

Observer 1 Timeline



Observer 2 Timeline



```
channel.send(4)

channel.send(2)
```

Channel Capacity

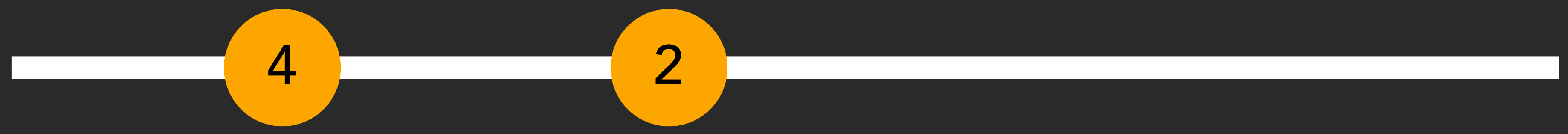
2

BROADCAST CHANNEL

- Special mention to **ConflatedBroadcastChannel**
- Conflated is a special type of capacity
- Behavior similar to Rx **BehaviorSubject**

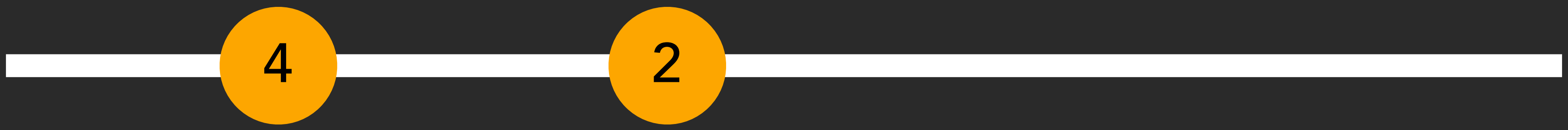
CONFLATED BROADCAST CHANNEL

Observer 1 Timeline



CONFLATED BROADCAST CHANNEL

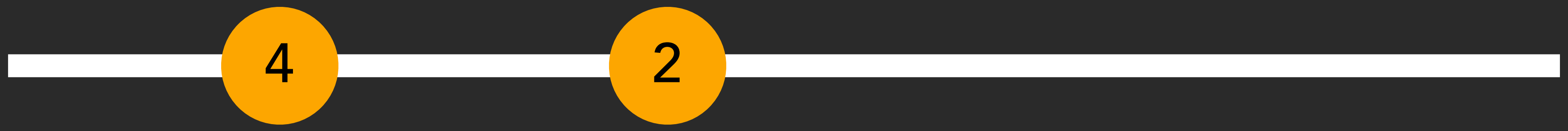
Observer 1 Timeline



```
// Closes Subscription
```

CONFLATED BROADCAST CHANNEL

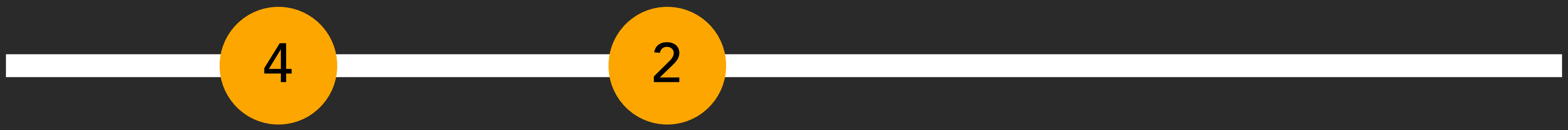
Observer 1 Timeline



```
// Closes Subscription

// Resubscribes to Broadcast Channel
```

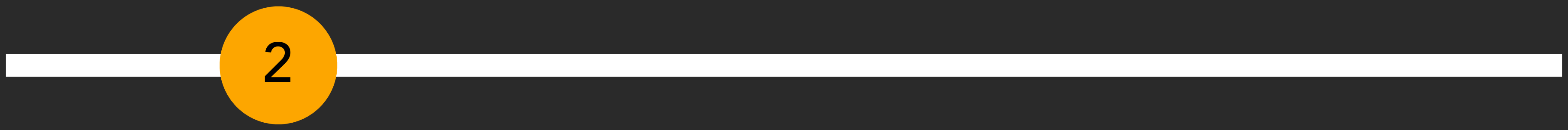
CONFLATED BROADCAST CHANNEL

Observer 1 Timeline



```
// Closes Subscription

// Resubscribes to Broadcast Channel
```

New Observer 1 Timeline



What about Rx back-pressure?

It’s supported by default

COMPARISON

Observable	Channel	Subject	Broadcast Channel
Cold	Hot	Hot	Hot
Unicast	Unicast	Broadcast	Broadcast

CHANNELS

- If we want a “Cold Observable” behavior, we can use **publish**

CHANNELS

- If we want a "Cold Observable" behavior, we can use **publish**

```
val publisher = publish {
    for (x in 1..5) send(x)
}
```

CHANNELS

- If we want a “Cold Observable” behavior, we can use **publish**

```
val publisher = publish {
    for (x in 1..5) send(x)
}
```

```
publisher.openSubscription().use { channel ->
        for (value in channel) {
            consumeValue(value)
        }
    }
```

INTEROP

org.jetbrains.kotlinx:**kotlinx-coroutines-rx2**:$kotlin_coroutines_version

RXJAVA -> COROUTINES

- OpenSubscription

```
Observable.interval(
     1, TimeUnit.SECONDS
)
.openSubscription().use { channel ->
    for (value in channel) {
        consumeValue(value)
    }
}
```

RXJAVA -> COROUTINES

- OpenSubscription

```
Observable.interval(
        1, TimeUnit.SECONDS
)
.openSubscription().use { channel ->
    for (value in channel) {
        consumeValue(value)
    }
}
```

RXJAVA -> COROUTINES

- OpenSubscription

```
Observable.interval(
        1, TimeUnit.SECONDS
)
.openSubscription().use { channel ->
    for (value in channel) {
        consumeValue(value)
    }
}
```

RXJAVA -> COROUTINES

- Await

RXJAVA -> COROUTINES

- Await

```
val value = Observable.interval(
     1, TimeUnit.SECONDS
)
.awaitFirstOrDefault(-1)
```

RXJAVA -> COROUTINES

- Await

```
val value = Observable.interval(
     1, TimeUnit.SECONDS
)
.awaitFirstOrDefault(-1)
```

COROUTINES -> RXJAVA

- Job.asCompletable

COROUTINES -> RXJAVA

- Job.asCompletable

```kotlin
val job = launch {
    heavyComputation()
}

job.asCompletable(CommonPool).subscribe({
    // Job completed
})
```

COROUTINES -> RXJAVA

- Job.asCompletable

```kotlin
val job = launch {
    heavyComputation()
}

job.asCompletable(CommonPool).subscribe({
    // Job completed
})
```

COROUTINES -> RXJAVA

- Job.asCompletable

```
val job = launch {
    heavyComputation()
}

job.asCompletable(CommonPool).subscribe({
    // Job completed
})
```

COROUTINES -> RXJAVA

- Deferred.asSingle

```
val deferred = async {
    heavyComputation()
}

deferred.asSingle(CommonPool).subscribe({
    // Job completed
}, {
    // Error happened
})
```

COROUTINES -> RXJAVA

- Deferred.asSingle

```kotlin
val deferred = async {
    heavyComputation()
}

deferred.asSingle(CommonPool).subscribe({
    // Job completed
}, {
    // Error happened
})
```

COROUTINES -> RXJAVA

- Deferred.asSingle

```
val deferred = async {
    heavyComputation()
}

deferred.asSingle(CommonPool).subscribe({
    // Job completed
}, {
    // Error happened
})
```

COROUTINES -> RXJAVA

- CoroutineBuilders
  - rxCompletable
  - rxMaybe
  - rxSingle
  - rxObservable
  - rxFlowable

COROUTINES -> RXJAVA

- CoroutineBuilders
  - rxCompletable
  - rxMaybe
  - rxSingle
  - rxObservable
  - rxFlowable

```
rxCompletable {
  // Suspending lambda
}.subscribe()
```

ACTORS

ACTORS

- Actor = Coroutine + Channel

ACTORS

- Actor = Coroutine + Channel

```
val actor = actor<Int>() {
    for (int in channel) {
         // iterate over received Integers
         // synchronously
    }
}
```

ACTORS

- Actor = Coroutine + Channel

```
val actor = actor<Int>() {
    for (int in channel) {
        // iterate over received Integers
        // synchronously
    }
}


launch {
    actor.send(2)
}
```

ACTORS

```
val userActionActor = actor<MainUserAction>(CommonPool) {
    for (msg in channel) { // iterate over incoming messages
        when (msg) {
            is MainUserAction.FirstAction -> {
                // Do something
            }
            is MainUserAction.SecondAction -> {
                // Do something
            }
        }
    }
}
```

ACTORS

```
val userActionActor = actor<MainUserAction>(CommonPool) {
    for (msg in channel) { // iterate over incoming messages
        when (msg) {
            is MainUserAction.FirstAction -> {
                // Do something
            }
            is MainUserAction.SecondAction -> {
                // Do something
            }
        }
    }
}
```

ACTORS

```
val userActionActor = actor<MainUserAction>(CommonPool) {
    for (msg in channel) { // iterate over incoming messages
        when (msg) {
            is MainUserAction.FirstAction -> {
                // Do something
            }
            is MainUserAction.SecondAction -> {
                // Do something
            }
        }
    }
}
```

ACTORS

```
val userActionActor = actor<MainUserAction>(CommonPool) {
    for (msg in channel) { // iterate over incoming messages
        when (msg) {
            is MainUserAction.FirstAction -> {
                // Do something
            }
            is MainUserAction.SecondAction -> {
                // Do something
            }
        }
    }
}
```

OPERATORS

OPERATORS

- Some operators are built into the language with Kotlin Collections

RxJava	Coroutines
map	map
filter	filter
skip	drop
reduce	reduce

OPERATORS

- Some others are easy to implement

```
fun range(
        context: CoroutineContext,
        start: Int,
        count: Int
) = publish(context) {
    for (x in start until start + count) send(x)
}
```

OPERATORS

- Some others require more work

- Completable.zip

```
suspend fun zip(block: () -> Unit, block2: () -> Unit) {

    val deferred1 = async { block() }
    val deferred2 = async { block2() }

    deferred1.await()
    deferred2.await()
}
```

COMPLEX OPERATORS

RxJava	Coroutines
timeout	withTimeoutOrNull
retry	repeat(times)
debounce	ReceiveChannel<T>.debounce()
groupBy	groupBy

THREADING

THREADING IN RX

THREADING IN RX

- Threading control Operators: **observeOn** and **subscribeOn**

THREADING IN RX

- Threading control Operators: **observeOn** and **subscribeOn**

- Scheduler is a tool that schedules actions to be performed

THREADING IN RX

- Threading control Operators: **observeOn** and **subscribeOn**

- Scheduler is a tool that schedules actions to be performed

- You can create your own Scheduler

THREADING

```
Single.zip(
        Single.just(3),
        Single.just(4),
        BiFunction<Int, Int, Int> { n1, n2 -> n1 + n2 }
)
        .subscribeOn(Schedulers.computation())
        .observeOn(Schedulers.io())
        .flatMap { n ->
            Single.just(n * n)
        }
        .observeOn(Schedulers.computation())
        .flatMap { n ->
            Single.just(n - 1)
        }
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({
            println("Finished with result $it")
        }, {
            println("Failed with error $it")
        })
```

THREADING

```
Single.zip(
        Single.just(3),
        Single.just(4),
        BiFunction<Int, Int, Int> { n1, n2 -> n1 + n2 }
)
        .subscribeOn(Schedulers.computation())
        .observeOn(Schedulers.io())
        .flatMap { n ->
            Single.just(n * n)
        }
        .observeOn(Schedulers.computation())
        .flatMap { n ->
            Single.just(n - 1)
        }
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({
            println("Finished with result $it")
        }, {
            println("Failed with error $it")
        })
```

THREADING

```
Single.zip(
        Single.just(3),
        Single.just(4),
        BiFunction<Int, Int, Int> { n1, n2 -> n1 + n2 }
)
        .subscribeOn(Schedulers.computation())
        .observeOn(Schedulers.io())
        .flatMap { n ->
            Single.just(n * n)
        }
        .observeOn(Schedulers.computation())
        .flatMap { n ->
            Single.just(n - 1)
        }
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({
            println("Finished with result $it")
        }, {
            println("Failed with error $it")
        })
```

🟨 Computation

THREADING

```
Single.zip(
        Single.just(3),
        Single.just(4),
        BiFunction<Int, Int, Int> { n1, n2 -> n1 + n2 }
)
        .subscribeOn(Schedulers.computation())
        .observeOn(Schedulers.io())
        .flatMap { n ->
            Single.just(n * n)
        }
        .observeOn(Schedulers.computation())
        .flatMap { n ->
            Single.just(n - 1)
        }
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({
            println("Finished with result $it")
        }, {
            println("Failed with error $it")
        })
```

THREADING

```
Single.zip(
        Single.just(3),
        Single.just(4),
        BiFunction<Int, Int, Int> { n1, n2 -> n1 + n2 }
)
        .subscribeOn(Schedulers.computation())
        .observeOn(Schedulers.io())
        .flatMap { n ->
            Single.just(n * n)
        }
        .observeOn(Schedulers.computation())
        .flatMap { n ->
            Single.just(n - 1)
        }
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({
            println("Finished with result $it")
        }, {
            println("Failed with error $it")
        })
```

THREADING

🟨 Computation

```
Single.zip(
        Single.just(3),
        Single.just(4),
        BiFunction<Int, Int, Int> { n1, n2 -> n1 + n2 }
)
        .subscribeOn(Schedulers.computation())
        .observeOn(Schedulers.io())
        .flatMap { n ->
            Single.just(n * n)
        }
        .observeOn(Schedulers.computation())
        .flatMap { n ->
            Single.just(n - 1)
        }
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({
            println("Finished with result $it")
        }, {
            println("Failed with error $it")
        })
```

🟨 Computation

THREADING

```
Single.zip(
        Single.just(3),
        Single.just(4),
        BiFunction<Int, Int, Int> { n1, n2 -> n1 + n2 }
)
        .subscribeOn(Schedulers.computation())
        .observeOn(Schedulers.io())
        .flatMap { n ->
            Single.just(n * n)
        }
        .observeOn(Schedulers.computation())
        .flatMap { n ->
            Single.just(n - 1)
        }
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({
            println("Finished with result $it")
        }, {
            println("Failed with error $it")
        })
```

THREADING

🟨 Computation

🟦 Io

```
Single.zip(
        Single.just(3),
        Single.just(4),
        BiFunction<Int, Int, Int> { n1, n2 -> n1 + n2 }
)
        .subscribeOn(Schedulers.computation())
        .observeOn(Schedulers.io())
        .flatMap { n ->
            Single.just(n * n)
        }
        .observeOn(Schedulers.computation())
        .flatMap { n ->
            Single.just(n - 1)
        }
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({
            println("Finished with result $it")
        }, {
            println("Failed with error $it")
        })
```

THREADING

🟨 Computation

🟦 Io

```
Single.zip(
        Single.just(3),
        Single.just(4),
        BiFunction<Int, Int, Int> { n1, n2 -> n1 + n2 }
)
        .subscribeOn(Schedulers.computation())
        .observeOn(Schedulers.io())
        .flatMap { n ->
            Single.just(n * n)
        }
        .observeOn(Schedulers.computation())
        .flatMap { n ->
            Single.just(n - 1)
        }
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({
            println("Finished with result $it")
        }, {
            println("Failed with error $it")
        })
```

THREADING

🟨 Computation

🟦 Io

```
Single.zip(
        Single.just(3),
        Single.just(4),
        BiFunction<Int, Int, Int> { n1, n2 -> n1 + n2 }
)
        .subscribeOn(Schedulers.computation())
        .observeOn(Schedulers.io())
        .flatMap { n ->
            Single.just(n * n)
        }
        .observeOn(Schedulers.computation())
        .flatMap { n ->
            Single.just(n - 1)
        }
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({
            println("Finished with result $it")
        }, {
            println("Failed with error $it")
        })
```

THREADING

🟨 Computation

🟦 Io

```
Single.zip(
        Single.just(3),
        Single.just(4),
        BiFunction<Int, Int, Int> { n1, n2 -> n1 + n2 }
)
        .subscribeOn(Schedulers.computation())
        .observeOn(Schedulers.io())
        .flatMap { n ->
            Single.just(n * n)
        }
        .observeOn(Schedulers.computation())
        .flatMap { n ->
            Single.just(n - 1)
        }
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({
            println("Finished with result $it")
        }, {
            println("Failed with error $it")
        })
```

THREADING

🟨 Computation

🟦 Io

🟥 Main Thread

```
Single.zip(
        Single.just(3),
        Single.just(4),
        BiFunction<Int, Int, Int> { n1, n2 -> n1 + n2 }
)
        .subscribeOn(Schedulers.computation())
        .observeOn(Schedulers.io())
        .flatMap { n ->
            Single.just(n * n)
        }
        .observeOn(Schedulers.computation())
        .flatMap { n ->
            Single.just(n - 1)
        }
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({
            println("Finished with result $it")
        }, {
            println("Failed with error $it")
        })
```

THREADING IN COROUTINES

THREADING IN COROUTINES

- It is defined in the CoroutineContext with a value for the key **ContinuationInterceptor**

THREADING IN COROUTINES

- It is defined in the CoroutineContext with a value for the key **ContinuationInterceptor**
- Specified with a CoroutineDispatcher
    - Specific Thread
    - Thread Pool

THREADING IN COROUTINES

- Some values:
  - CommonPool
  - UI (Android)
  - Unconfined

THREADING IN COROUTINES

THREADING IN COROUTINES

- Create your own ThreadPoolDispatcher

THREADING IN COROUTINES

- Create your own ThreadPoolDispatcher
    - NewSingleThreadContext

```
val coroutineDispatcher = newSingleThreadContext("Name")
```

THREADING IN COROUTINES

- Create your own ThreadPoolDispatcher
  - NewSingleThreadContext

```
val coroutineDispatcher = newSingleThreadContext("Name")
```

  - NewFixedThreadPoolContext

```
val coroutineDispatcher = newFixedThreadPoolContext(4, "Name")
```

THREADING

```
launch(CommonPool) {
    val deferred1 = async(coroutineContext) { 3 }
    val deferred2 = async(coroutineContext) { 4 }

    var result = deferred1.await() + deferred2.await()

    launch(newSingleThreadContext("CustomThread")) {
        result = result * result - 1
    }.join()

    withContext(UI) {
        println("Finished with result $result")
    }
}
```

THREADING

```
launch(CommonPool) {
    val deferred1 = async(coroutineContext) { 3 }
    val deferred2 = async(coroutineContext) { 4 }

    var result = deferred1.await() + deferred2.await()

    launch(newSingleThreadContext("CustomThread")) {
        result = result * result - 1
    }.join()

    withContext(UI) {
        println("Finished with result $result")
    }
}
```

THREADING

```
launch(CommonPool) {
    val deferred1 = async(coroutineContext) { 3 }
    val deferred2 = async(coroutineContext) { 4 }

    var result = deferred1.await() + deferred2.await()

    launch(newSingleThreadContext("CustomThread")) {
        result = result * result - 1
    }.join()

    withContext(UI) {
        println("Finished with result $result")
    }
}
```

🟨 CommonPool

THREADING

```
launch(CommonPool) {
    val deferred1 = async(coroutineContext) { 3 }
    val deferred2 = async(coroutineContext) { 4 }

    var result = deferred1.await() + deferred2.await()

    launch(newSingleThreadContext("CustomThread")) {
        result = result * result - 1
    }.join()

    withContext(UI) {
        println("Finished with result $result")
    }
}
```

THREADING

🟨 CommonPool

```
launch(CommonPool) {
    val deferred1 = async(coroutineContext) { 3 }
    val deferred2 = async(coroutineContext) { 4 }

    var result = deferred1.await() + deferred2.await()

    launch(newSingleThreadContext("CustomThread")) {
        result = result * result - 1
    }.join()

    withContext(UI) {
        println("Finished with result $result")
    }
}
```

THREADING

🟨 CommonPool

🟦 CustomThread

```
launch(CommonPool) {
    val deferred1 = async(coroutineContext) { 3 }
    val deferred2 = async(coroutineContext) { 4 }

    var result = deferred1.await() + deferred2.await()

    launch(newSingleThreadContext("CustomThread")) {
        result = result * result - 1
    }.join()

    withContext(UI) {
        println("Finished with result $result")
    }
}
```

🟨 CommonPool

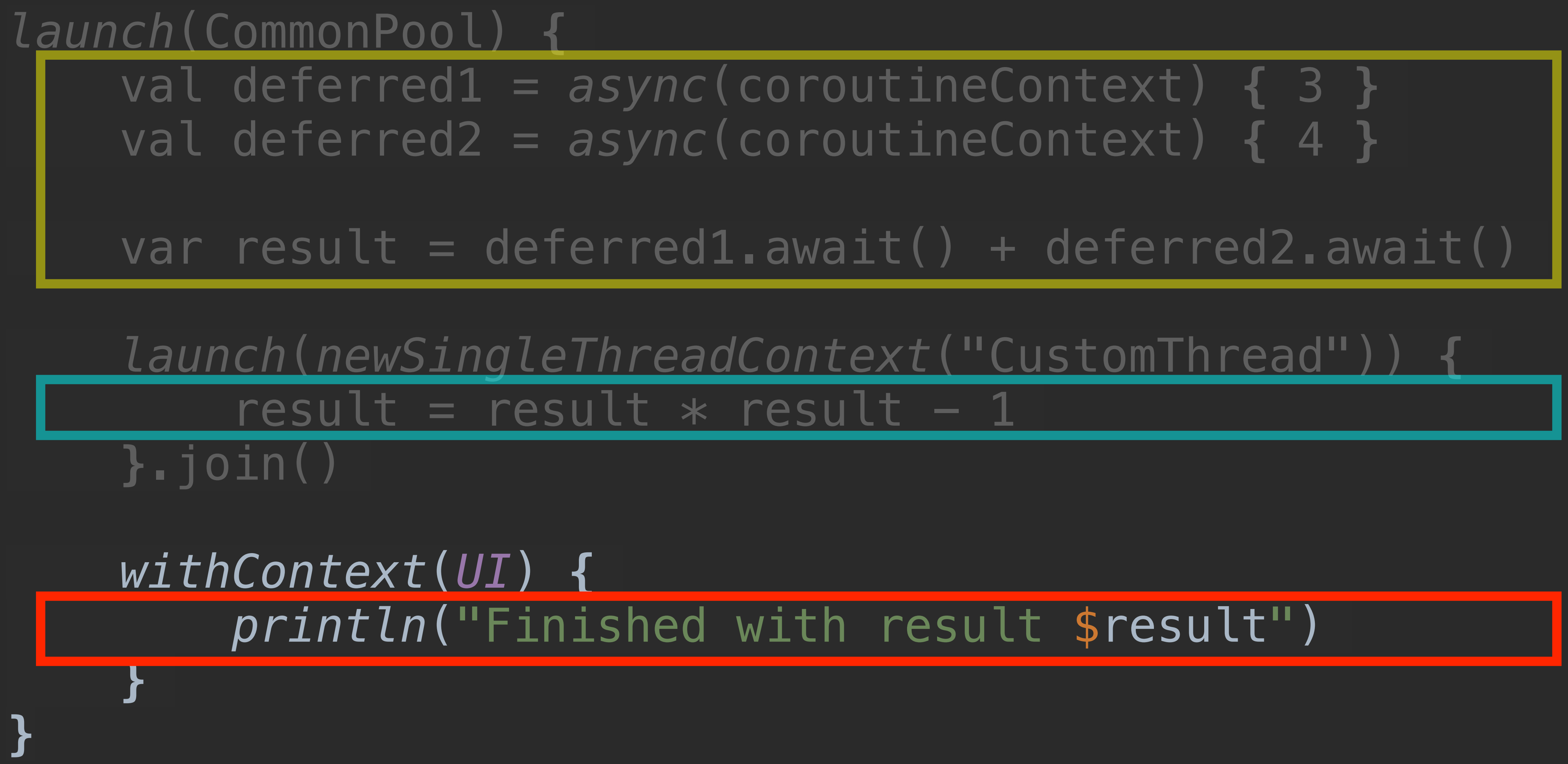
🟦 CustomThread

THREADING

```
launch(CommonPool) {
    val deferred1 = async(coroutineContext) { 3 }
    val deferred2 = async(coroutineContext) { 4 }

    var result = deferred1.await() + deferred2.await()

    launch(newSingleThreadContext("CustomThread")) {
        result = result * result - 1
    }.join()

    withContext(UI) {
        println("Finished with result $result")
    }
}
```

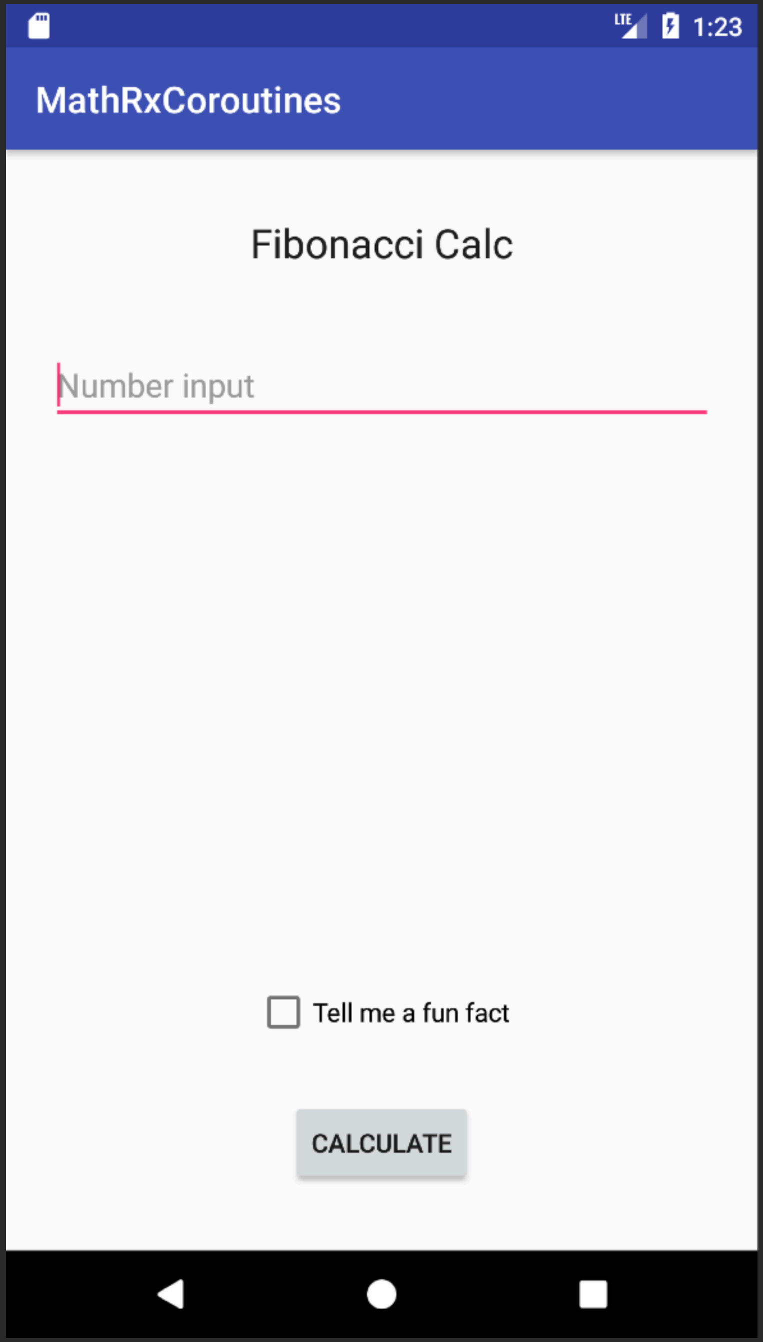
THREADING

🟨 CommonPool
🟦 CustomThread
🟥 UI

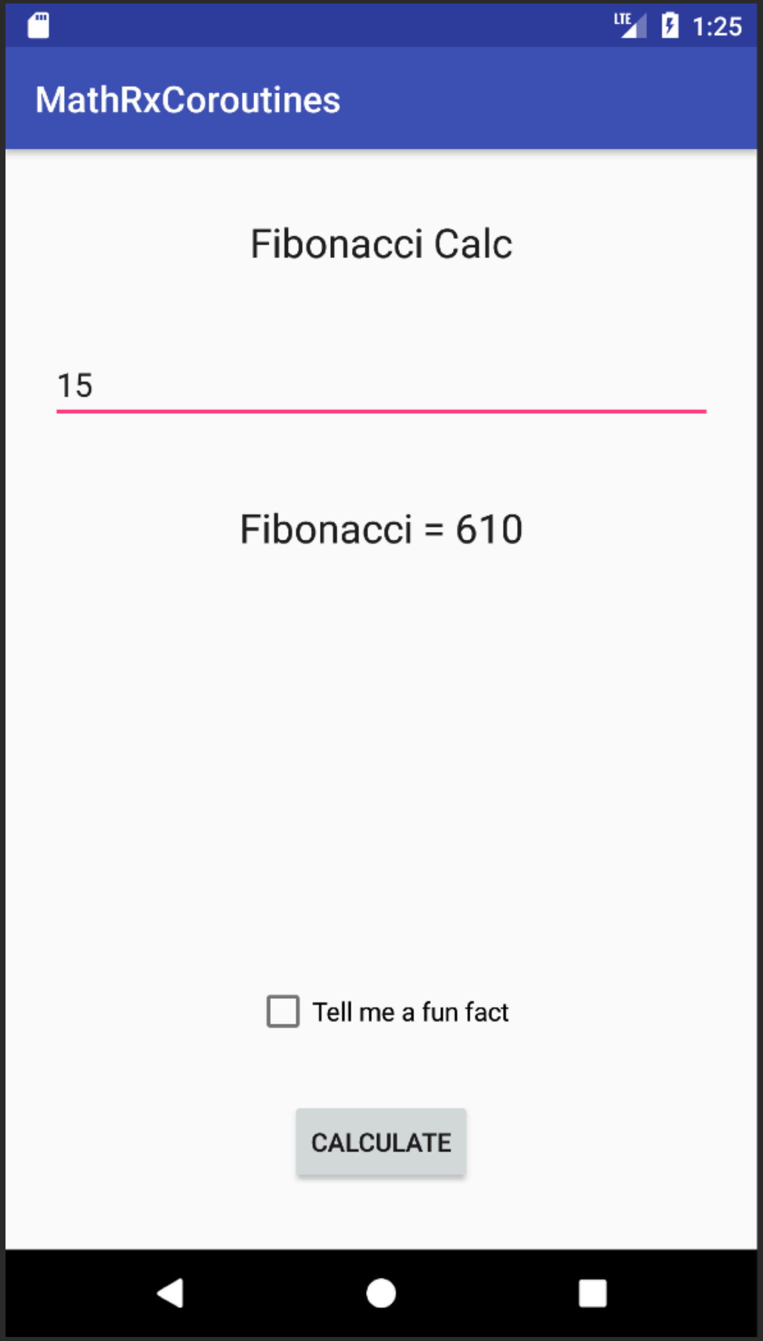


```
launch(CommonPool) {
    val deferred1 = async(coroutineContext) { 3 }
    val deferred2 = async(coroutineContext) { 4 }

    var result = deferred1.await() + deferred2.await()

    launch(newSingleThreadContext("CustomThread")) {
        result = result * result - 1
    }.join()

    withContext(UI) {
        println("Finished with result $result")
    }
}
```

MATH APP
MVI ARCHITECTURE



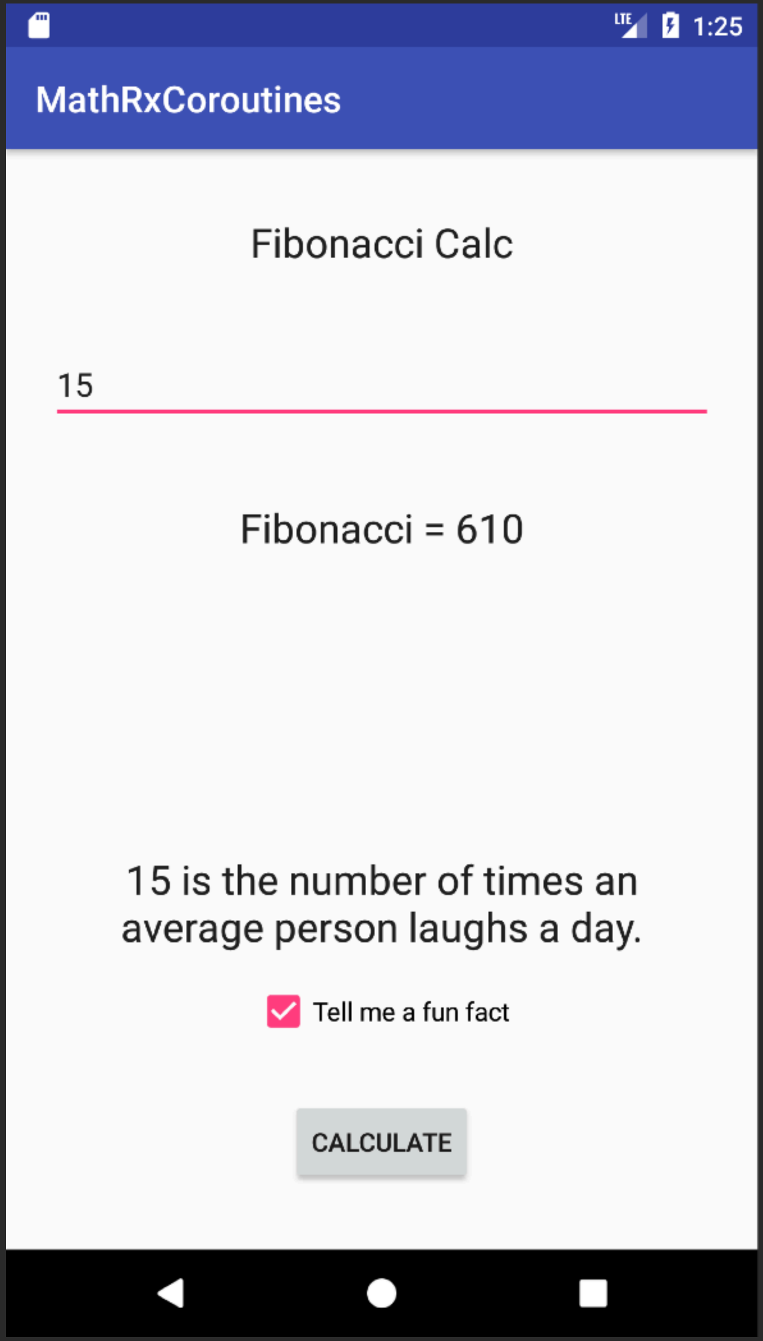
□ Tell me a fun fact



15

Fibonacci = 610

□ Tell me a fun fact



15

Fibonacci = 610

15 is the number of times an average person laughs a day.

☑ Tell me a fun fact

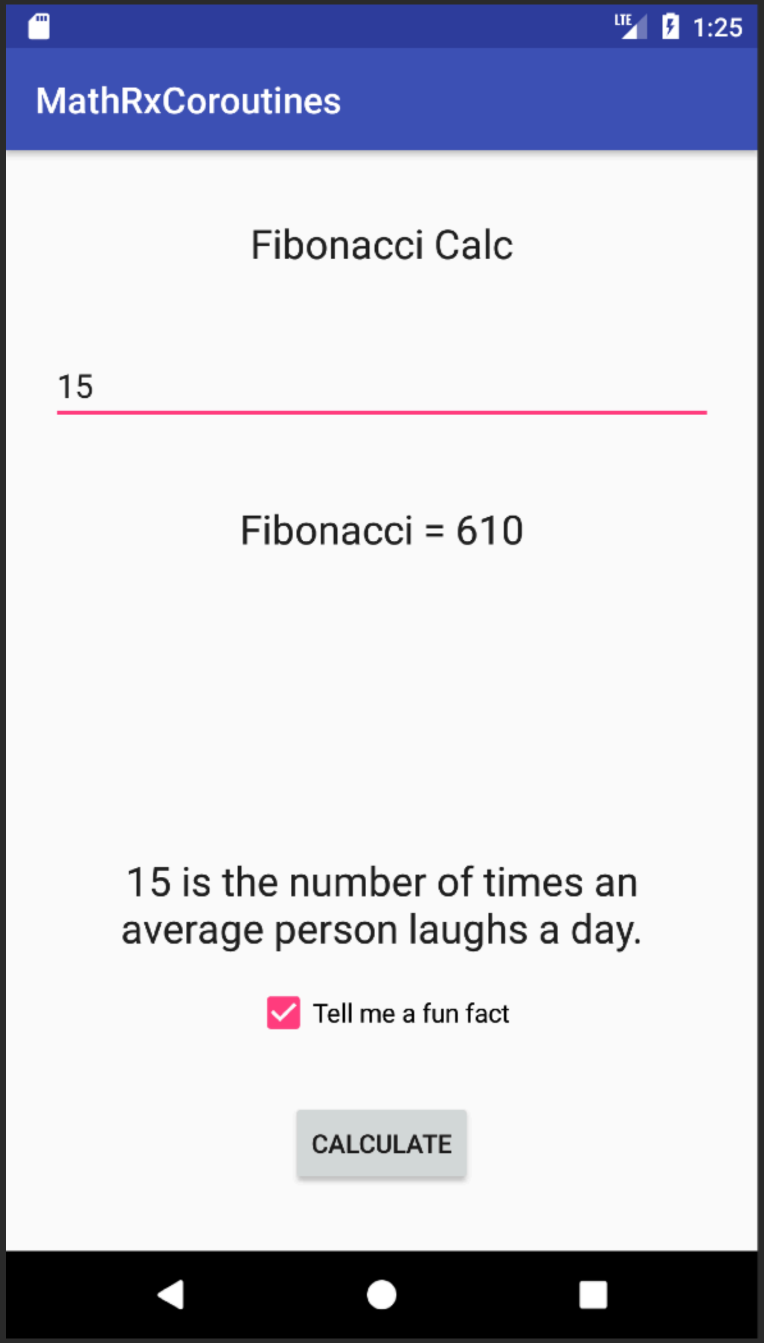
MATH APP

- Three Projects. Implemented with:
  - Coroutines
  - RxJava
  - Coroutines/RxJava Interop

MATH APP

- MVI
- Architecture Components ViewModels
    - Survives Configuration Changes

RESTORE STATE



MathRxCoroutines

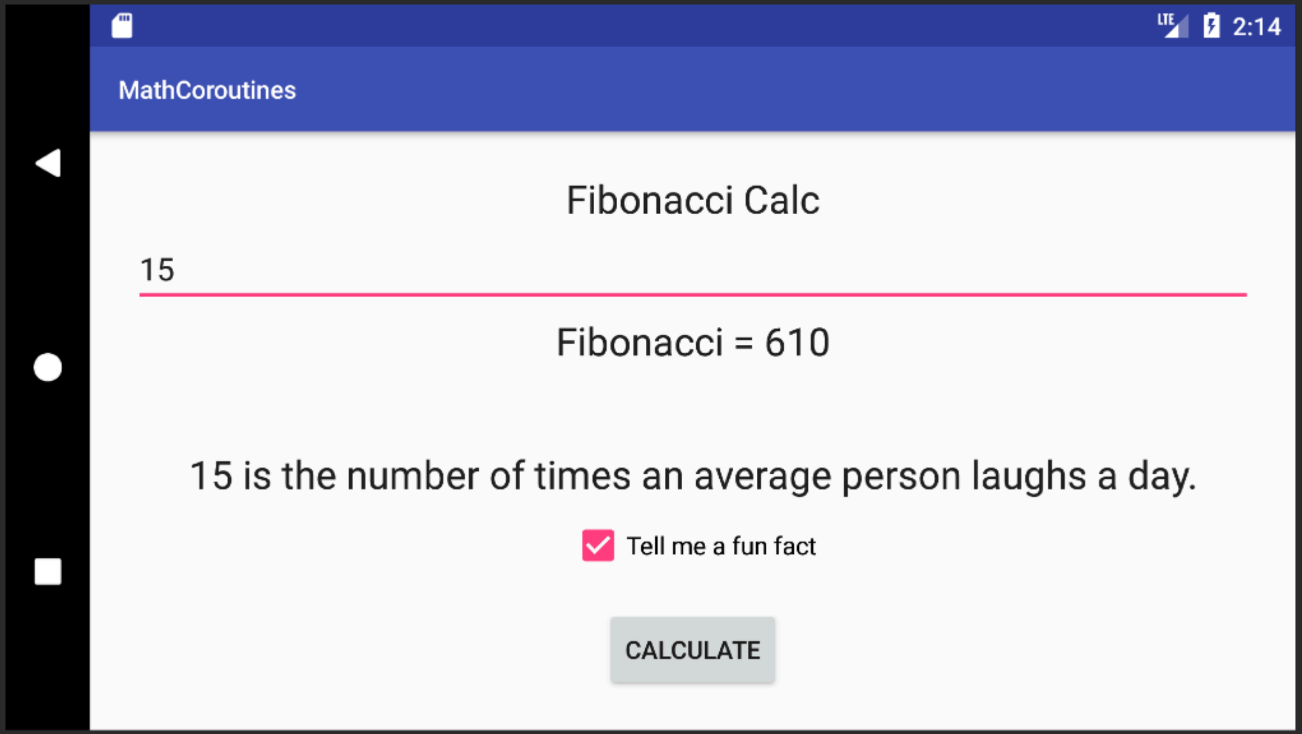
Fibonacci Calc

15

Fibonacci = 610

15 is the number of times an average person laughs a day.

☑ Tell me a fun fact



MathCoroutines

Fibonacci Calc

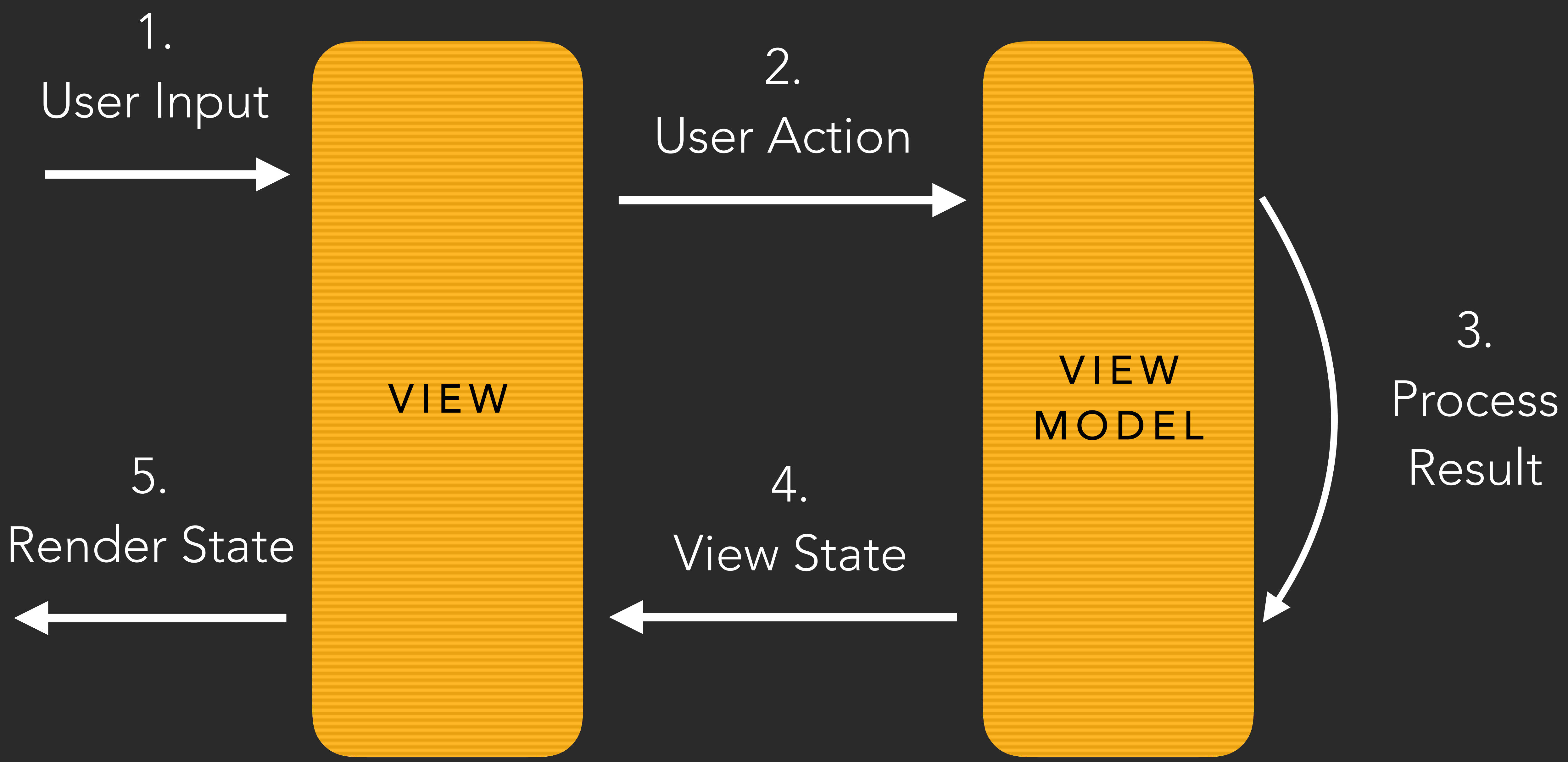
15

Fibonacci = 610

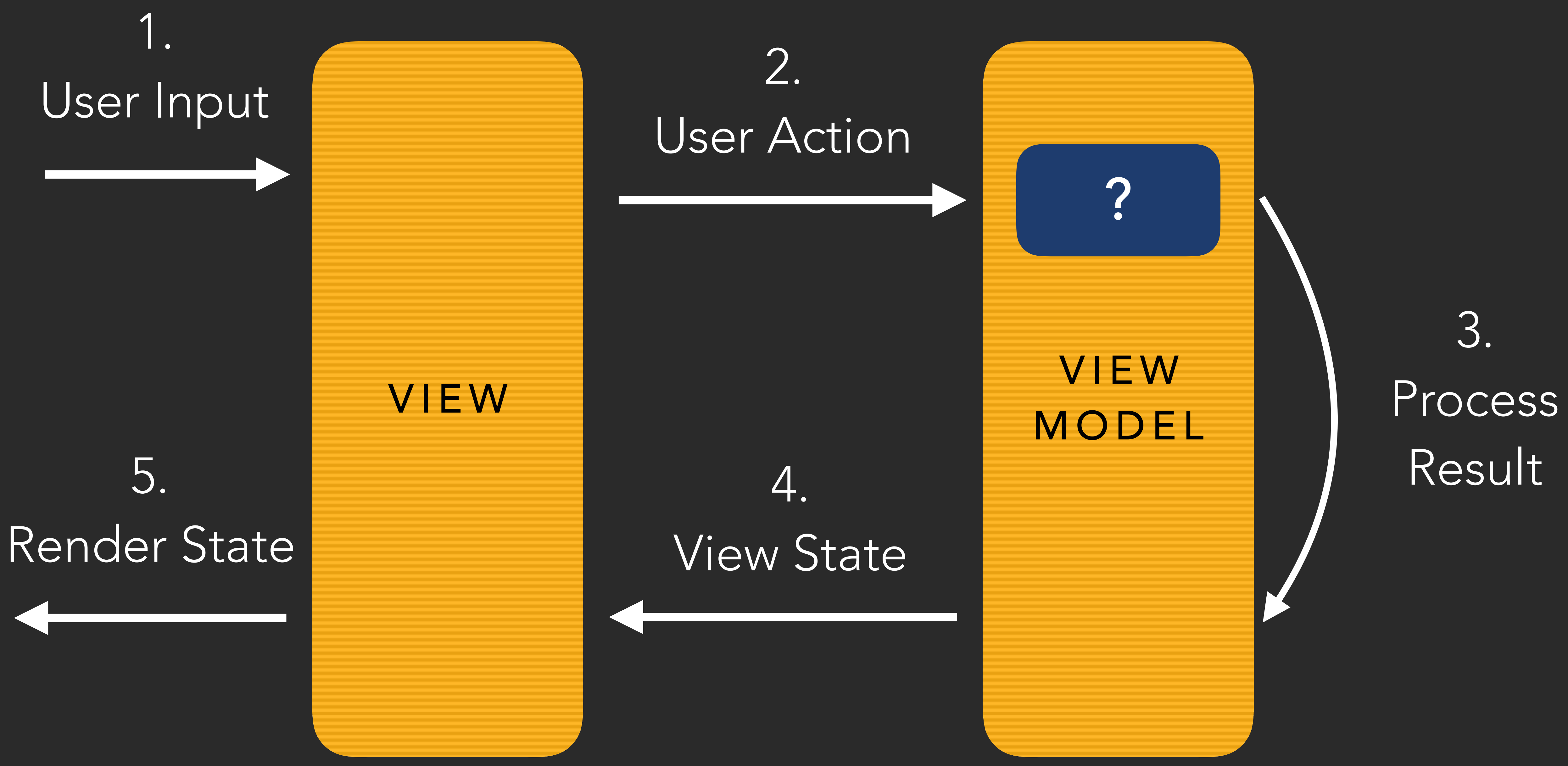
15 is the number of times an average person laughs a day.

☑ Tell me a fun fact

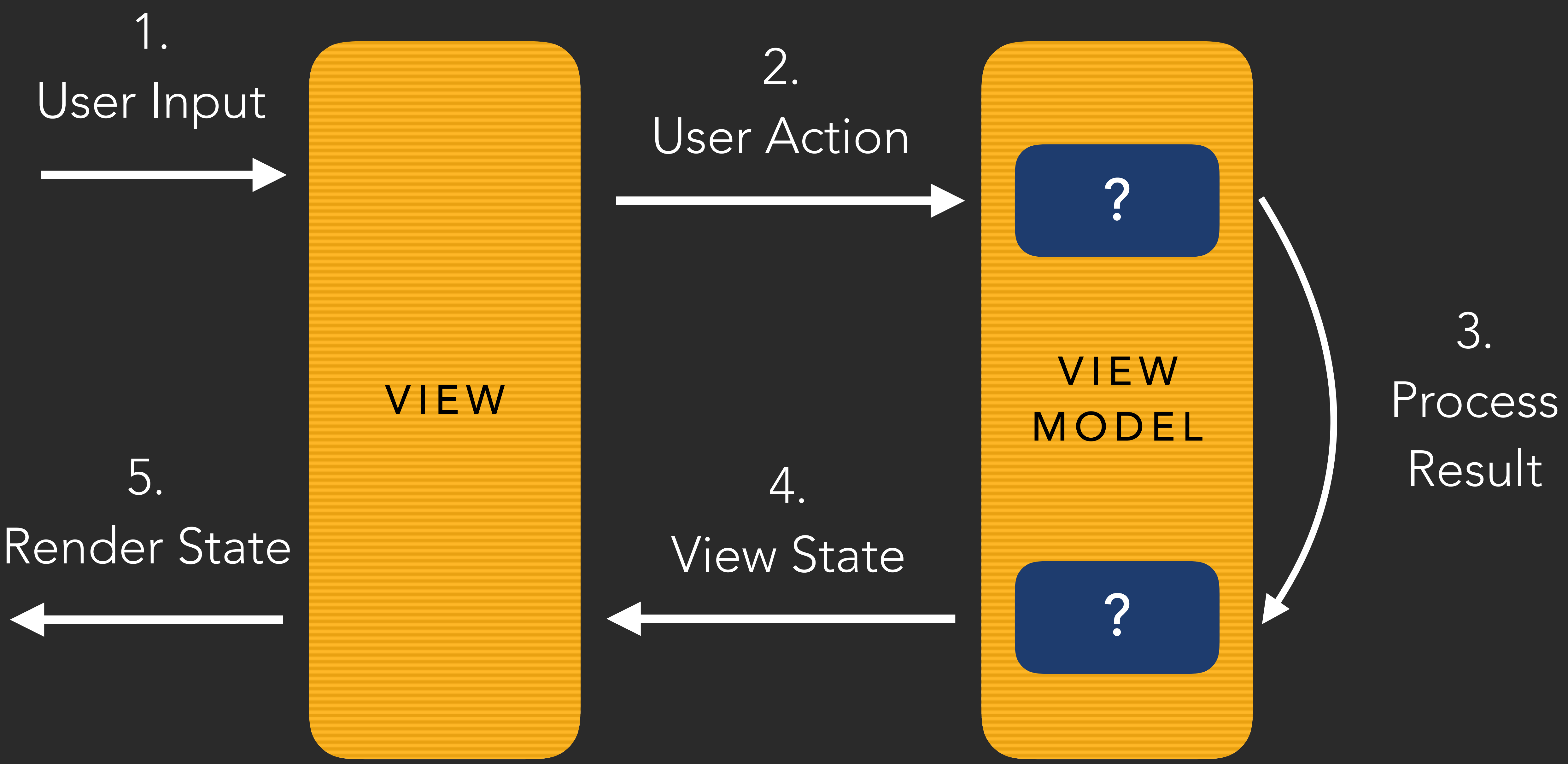
MVI ARCHITECTURE



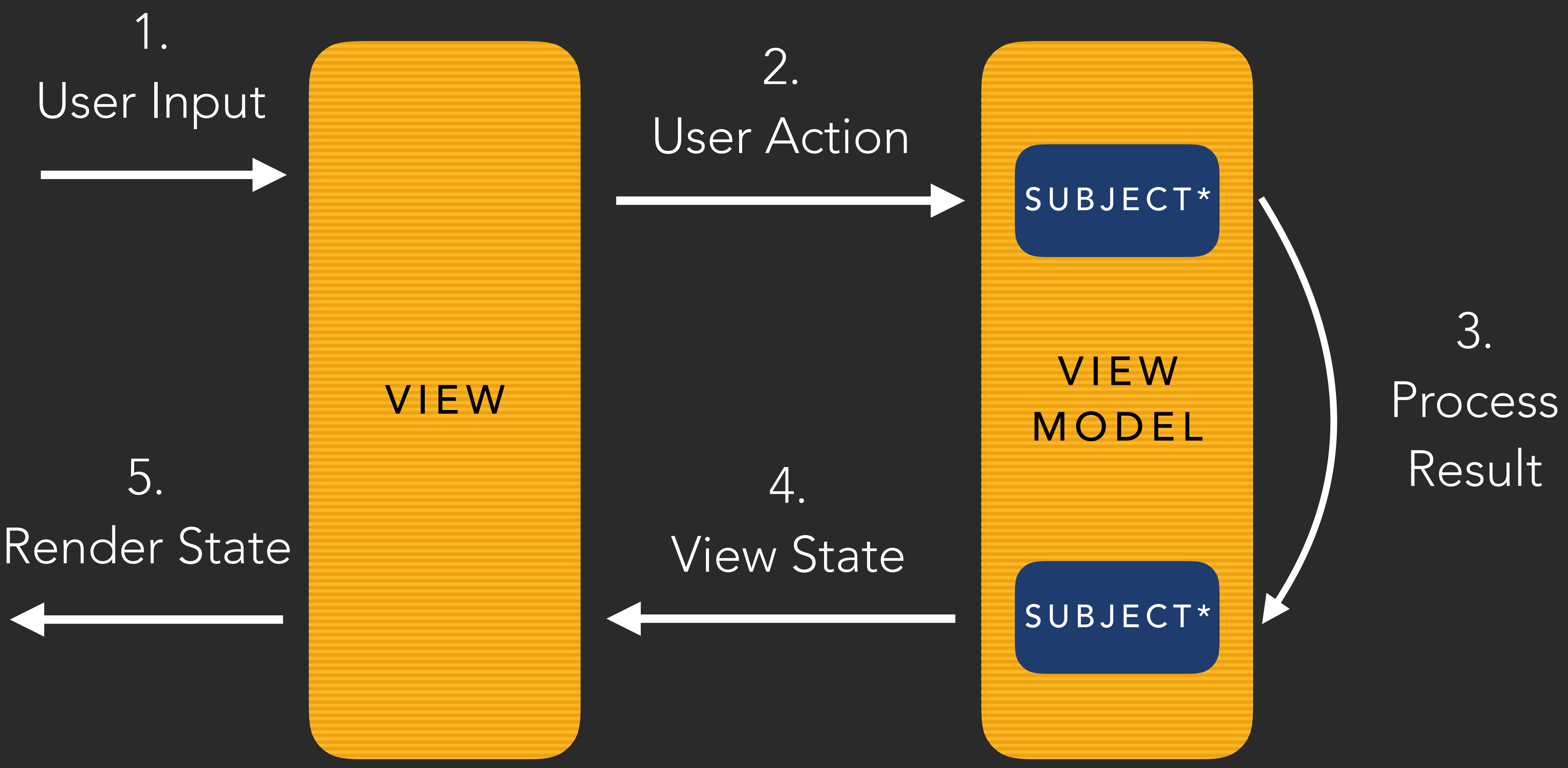
MVI ARCHITECTURE



MVI ARCHITECTURE

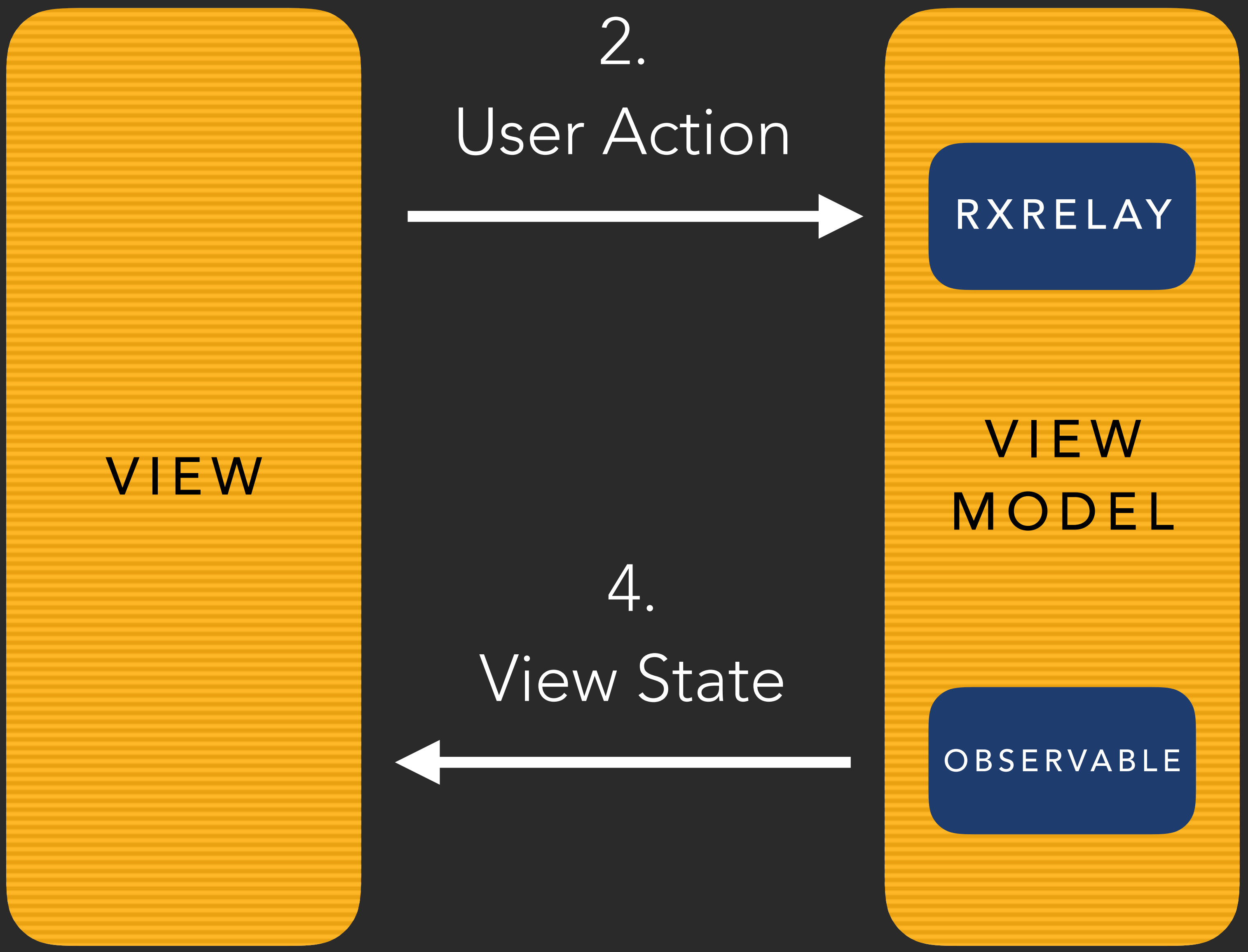


RX ARCHITECTURE



* Not optimal

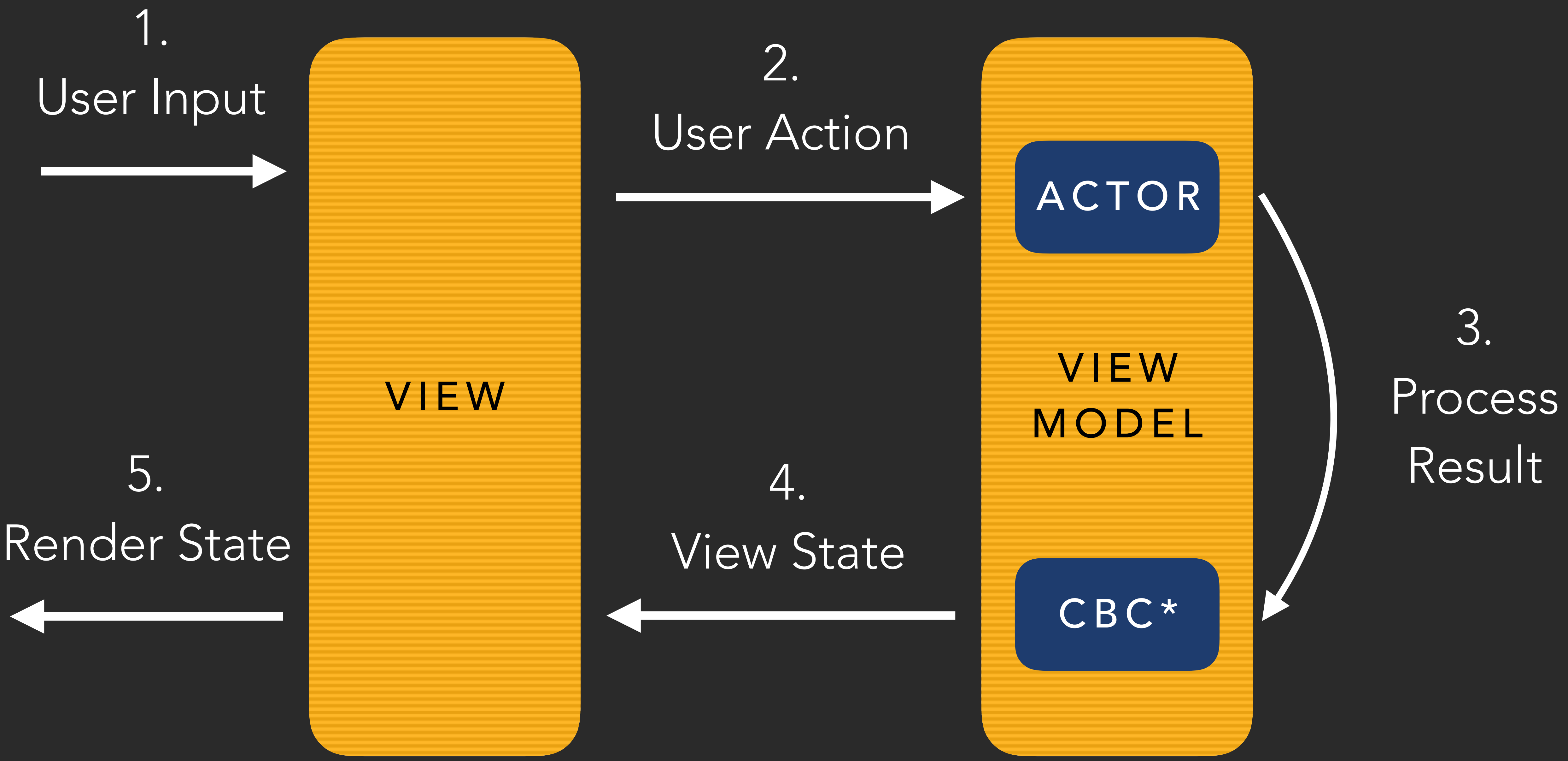
RX ARCHITECTURE



RxRelay to avoid the View calling onComplete()

Exposes an Observable but implemented with a Subject so View cannot call onNext()

COROUTINES ARCHITECTURE



* ConflatedBroadcastChannel

COMMON IMPLEMENTATION

```
sealed class MainUserAction {

    class Calculate(
       val number: Long
    ) : MainUserAction()

    class FunFactEnabled(
       val enabled: Boolean
    ) : MainUserAction()
}
```

COMMON IMPLEMENTATION

```kotlin
sealed class MainViewState {

    object Loading : MainViewState()

    class Rendered(
       val fibonacciNumber: Long,
       val funFact: String
    ) : MainViewState()

    object WrongInputError : MainViewState()

    object RequestError : MainViewState()
}
```

VIEW MODEL

- Extends from ViewModel in Architecture Components
- Receives User Actions
- Processes User Action
- Notifies View with the ViewState

RX VIEWMODEL

RX VIEWMODEL

- Receives User Actions

```
val userActionSubject: PublishSubject<MainUserAction>
```

RX VIEWMODEL

- Receives User Actions

```
val userActionSubject: PublishSubject<MainUserAction>
```

- Notifies View with the ViewState

```
val viewStateSubject: BehaviorSubject<MainViewState>
```

RX VIEWMODEL

```
init {
    userActionSubject
            .subscribeOn(Schedulers.computation())
            .subscribe({
                when (it) {
                    is MainUserAction.Calculate -> {
                        if (it.number <= 0) {
                            viewStateSubject.
                                 onNext(MainViewState.WrongInputError)
                        } else {
                            viewStateSubject.onNext(MainViewState.Loading)
                            processCalculation(it)
                        }
                    }
                    is MainUserAction.FunFactEnabled -> {
                        askForFunFact = it.enabled
                    }
                }
            })
}
```

RX VIEWMODEL

```
init {
    userActionSubject
            .subscribeOn(Schedulers.computation())
            .subscribe({
                when (it) {
                    is MainUserAction.Calculate -> {
                        if (it.number <= 0) {
                            viewStateSubject.
                                 onNext(MainViewState.WrongInputError)
                        } else {
                            viewStateSubject.onNext(MainViewState.Loading)
                            processCalculation(it)
                        }
                    }
                    is MainUserAction.FunFactEnabled -> {
                        askForFunFact = it.enabled
                    }
                }
            })
}
```

RX VIEWMODEL

```
init {
    userActionSubject
            .subscribeOn(Schedulers.computation())
            .subscribe({
                when (it) {
                    is MainUserAction.Calculate -> {
                        if (it.number <= 0) {
                            viewStateSubject.
                                  onNext(MainViewState.WrongInputError)
                        } else {
                            viewStateSubject.onNext(MainViewState.Loading)
                            processCalculation(it)
                        }
                    }
                    is MainUserAction.FunFactEnabled -> {
                        askForFunFact = it.enabled
                    }
                }
            })
}
```

RX VIEWMODEL

```
init {
    userActionSubject
            .subscribeOn(Schedulers.computation())
            .subscribe({
                when (it) {
                    is MainUserAction.Calculate -> {
                        if (it.number <= 0) {
                            viewStateSubject.
                                 onNext(MainViewState.WrongInputError)
                        } else {
                            viewStateSubject.onNext(MainViewState.Loading)
                            processCalculation(it)
                        }
                    }
                    is MainUserAction.FunFactEnabled -> {
                        askForFunFact = it.enabled
                    }
                }
            })
}
```

RX VIEWMODEL

```
init {
    userActionSubject
            .subscribeOn(Schedulers.computation())
            .subscribe({
                when (it) {
                    is MainUserAction.Calculate -> {
                        if (it.number <= 0) {
                            viewStateSubject.
                                  onNext(MainViewState.WrongInputError)
                        } else {
                            viewStateSubject.onNext(MainViewState.Loading)
                            processCalculation(it)
                        }
                    }
                    is MainUserAction.FunFactEnabled -> {
                        askForFunFact = it.enabled
                    }
                }
            })
}
```

RX VIEWMODEL

- Clean up when it’s not longer needed

```
override fun onCleared() {
    userActionSubject.onComplete()
    viewStateSubject.onComplete()
    super.onCleared()
}
```

COROUTINES VIEWMODEL

COROUTINES VIEWMODEL

- Receives User Actions

```
val userActionActor = actor<MainUserAction>(CommonPool, parent = parentJob)
```

COROUTINES VIEWMODEL

- Receives User Actions

```
val userActionActor = actor<MainUserAction>(CommonPool, parent = parentJob)
```

- Notifies View with the ViewState

```
val viewStateChannel = ConflatedBroadcastChannel<MainViewState>()
```

COROUTINES VIEWMODEL

```
private val parentJob = Job()

val userActionActor = actor<MainUserAction>(
        CommonPool,
        parent = parentJob
) {
    for (msg in channel) { // iterate over incoming messages
        when (msg) {
            is MainUserAction.Calculate -> {
                if (msg.number <= 0) {
                    viewStateChannel.offer(MainViewState.WrongInputError)
                } else {
                    viewStateChannel.offer(MainViewState.Loading)
                    processCalculation(msg)
                }
            }
            is MainUserAction.FunFactEnabled -> {
                askForFunFact = msg.enabled
            }
        }
    }
}
```

COROUTINES VIEWMODEL

```
private val parentJob = Job()

val userActionActor = actor<MainUserAction>(
        CommonPool,
        parent = parentJob
) {
    for (msg in channel) { // iterate over incoming messages
        when (msg) {
            is MainUserAction.Calculate -> {
                if (msg.number <= 0) {
                    viewStateChannel.offer(MainViewState.WrongInputError)
                } else {
                    viewStateChannel.offer(MainViewState.Loading)
                    processCalculation(msg)
                }
            }
            is MainUserAction.FunFactEnabled -> {
                askForFunFact = msg.enabled
            }
        }
    }
}
```

COROUTINES VIEWMODEL

```
private val parentJob = Job()

val userActionActor = actor<MainUserAction>(
        CommonPool,
        parent = parentJob
) {
    for (msg in channel) { // iterate over incoming messages
        when (msg) {
            is MainUserAction.Calculate -> {
                if (msg.number <= 0) {
                    viewStateChannel.offer(MainViewState.WrongInputError)
                } else {
                    viewStateChannel.offer(MainViewState.Loading)
                    processCalculation(msg)
                }
            }
            is MainUserAction.FunFactEnabled -> {
                askForFunFact = msg.enabled
            }
        }
    }
}
```

COROUTINES VIEWMODEL

```
private val parentJob = Job()

val userActionActor = actor<MainUserAction>(
        CommonPool,
        parent = parentJob
) {
    for (msg in channel) { // iterate over incoming messages
        when (msg) {
            is MainUserAction.Calculate -> {
                if (msg.number <= 0) {
                    viewStateChannel.offer(MainViewState.WrongInputError)
                } else {
                    viewStateChannel.offer(MainViewState.Loading)
                    processCalculation(msg)
                }
            }
            is MainUserAction.FunFactEnabled -> {
                askForFunFact = msg.enabled
            }
        }
    }
}
```

COROUTINES VIEWMODEL

```
private val parentJob = Job()

val userActionActor = actor<MainUserAction>(
        CommonPool,
        parent = parentJob
) {
    for (msg in channel) { // iterate over incoming messages
        when (msg) {
            is MainUserAction.Calculate -> {
                if (msg.number <= 0) {
                    viewStateChannel.offer(MainViewState.WrongInputError)
                } else {
                    viewStateChannel.offer(MainViewState.Loading)
                    processCalculation(msg)
                }
            }
            is MainUserAction.FunFactEnabled -> {
                askForFunFact = msg.enabled
            }
        }
    }
}
```

COROUTINES VIEWMODEL

- Clean up when it’s not longer needed

```
override fun onCleared() {
    viewStateChannel.close()
    parentJob.cancel()          // Cancels the Actor
    super.onCleared()
}
```

VIEW

- Listens for User Events
- Talks to ViewModel to process Input
- Renders ViewState

RXJAVA VIEW

- Listens for User Events
- Talks to ViewModel to process Input

```
private fun setupViews() {
    calcButton.setOnClickListener {
        viewModel.userActionSubject.onNext(
                MainUserAction.Calculate(
                        input.text.toString().toLong()))
    }

    funFact.setOnCheckedChangeListener { _, isChecked ->
        viewModel.userActionSubject.onNext(
                MainUserAction.FunFactEnabled(isChecked))
    }
}
```

RXJAVA VIEW

- Listens for User Events
- Talks to ViewModel to process Input

```
private fun setupViews() {
    calcButton.setOnClickListener {
        viewModel.userActionSubject.onNext(
                MainUserAction.Calculate(
                        input.text.toString().toLong()))
    }

    funFact.setOnCheckedChangeListener { _, isChecked ->
        viewModel.userActionSubject.onNext(
                MainUserAction.FunFactEnabled(isChecked))
    }
}
```

RXJAVA VIEW

- Listens for User Events

- Talks to ViewModel to process Input

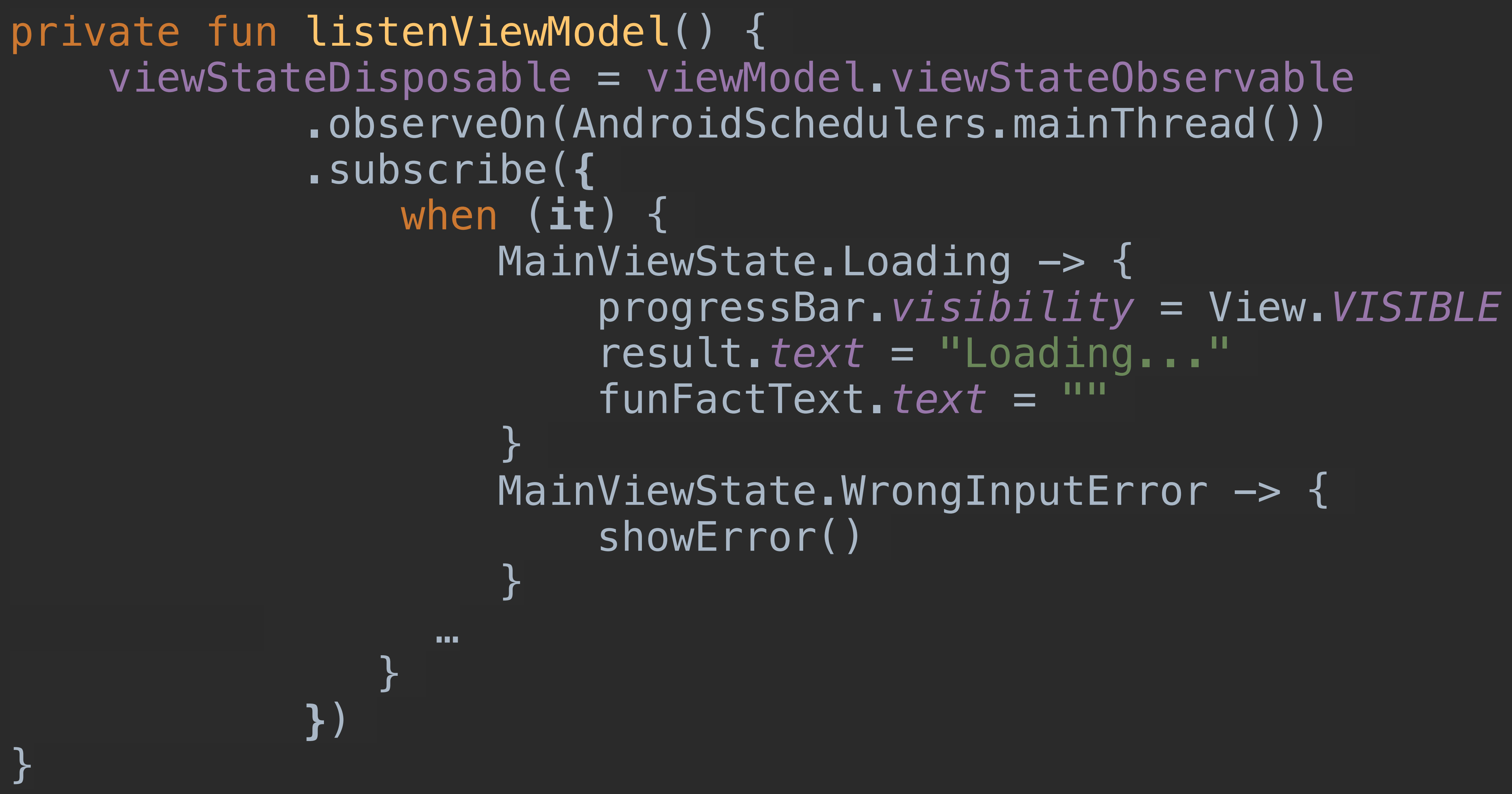
```
private fun setupViews() {
    calcButton.setOnClickListener {
        viewModel.userActionSubject.onNext(
                MainUserAction.Calculate(
                        input.text.toString().toLong()))
    }

    funFact.setOnCheckedChangeListener { _, isChecked ->
        viewModel.userActionSubject.onNext(
                MainUserAction.FunFactEnabled(isChecked))
    }
}
```

RXJAVA VIEW

```
private fun listenViewModel() {
    viewStateDisposable = viewModel.viewStateObservable
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe({
                when (it) {
                    MainViewState.Loading -> {
                        progressBar.visibility = View.VISIBLE
                        result.text = "Loading..."
                        funFactText.text = ""
                    }
                    MainViewState.WrongInputError -> {
                        showError()
                    }

                 …
               }
            })
}
```

RXJAVA VIEW

- Renders ViewState



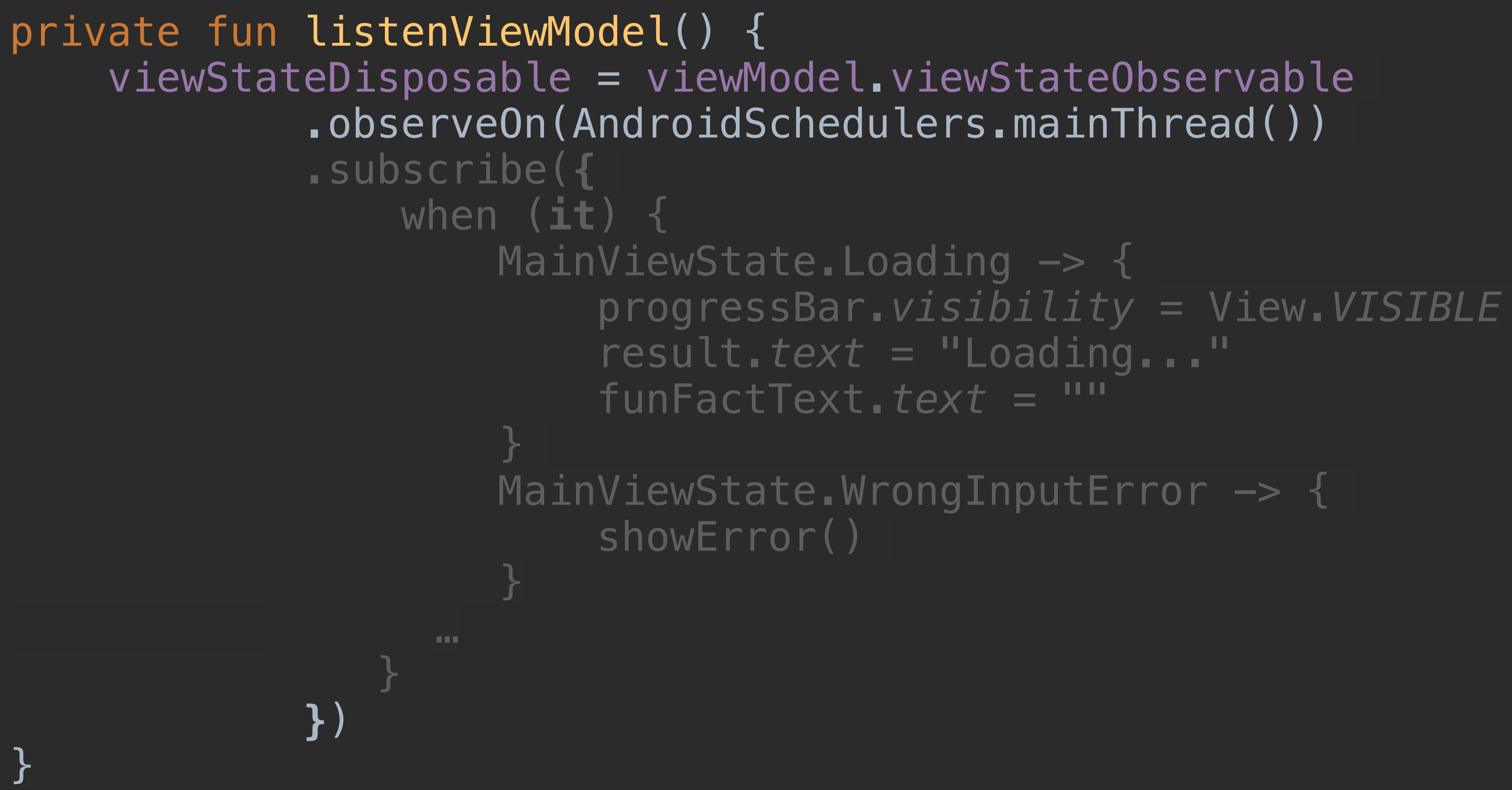
```
private fun listenViewModel() {
    viewStateDisposable = viewModel.viewStateObservable
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe({
                when (it) {
                    MainViewState.Loading -> {
                        progressBar.visibility = View.VISIBLE
                        result.text = "Loading..."
                        funFactText.text = ""
                    }
                    MainViewState.WrongInputError -> {
                        showError()
                    }

                 …
               }
            })
}
```

RXJAVA VIEW

```
private fun listenViewModel() {
    viewStateDisposable = viewModel.viewStateObservable
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe({
                when (it) {
                    MainViewState.Loading -> {
                        progressBar.visibility = View.VISIBLE
                        result.text = "Loading..."
                        funFactText.text = ""
                    }
                    MainViewState.WrongInputError -> {
                        showError()
                    }

                 …
               }
            })
}
```

RXJAVA VIEW

- Renders ViewState



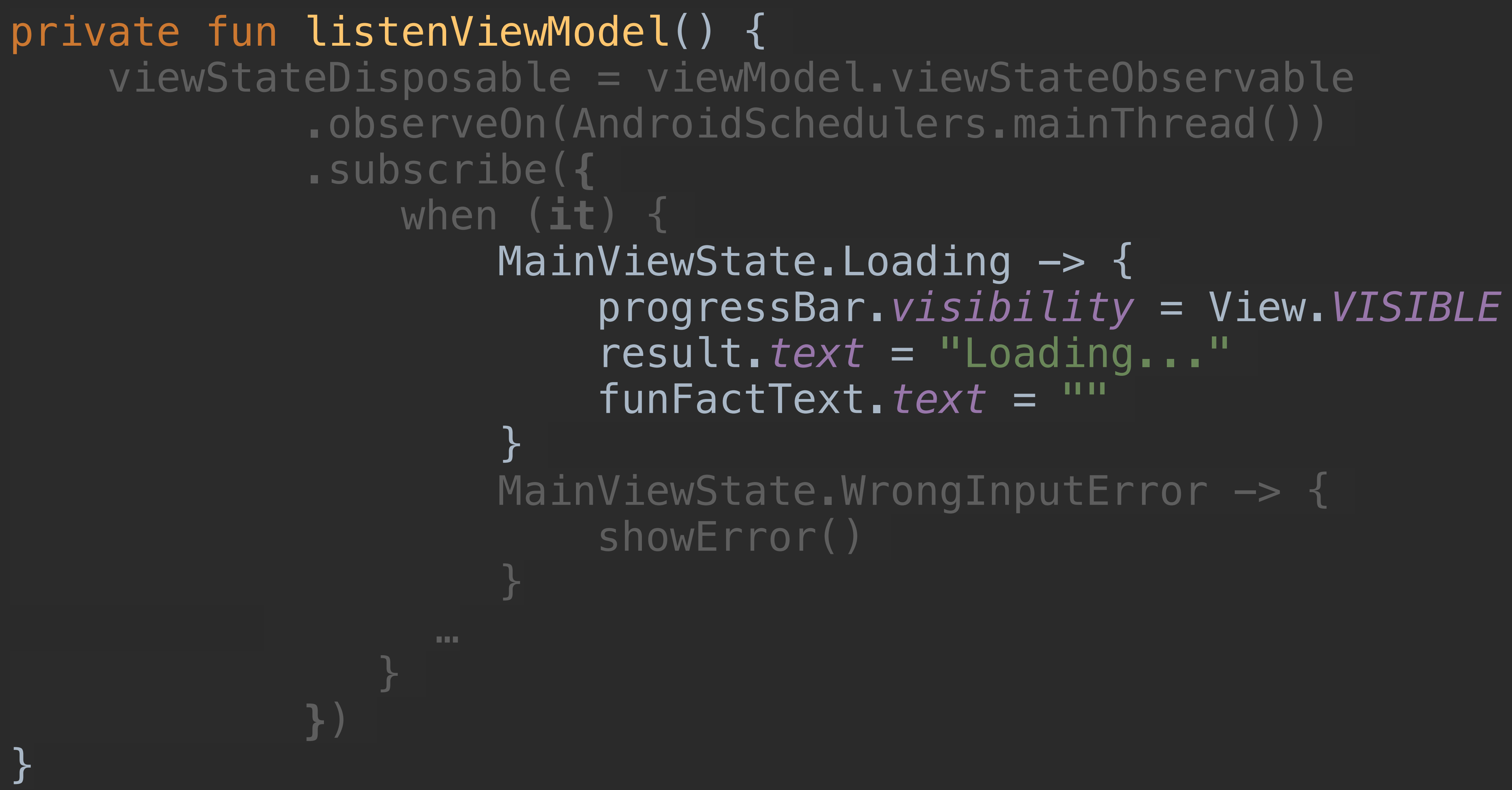
```kotlin
private fun listenViewModel() {
    viewStateDisposable = viewModel.viewStateObservable
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe({
                when (it) {
                    MainViewState.Loading -> {
                        progressBar.visibility = View.VISIBLE
                        result.text = "Loading..."
                        funFactText.text = ""
                    }
                    MainViewState.WrongInputError -> {
                        showError()
                    }

                 …
               }
            })
}
```

RXJAVA VIEW

```
private fun listenViewModel() {
    viewStateDisposable = viewModel.viewStateObservable
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe({
                when (it) {
                    MainViewState.Loading -> {
                        progressBar.visibility = View.VISIBLE
                        result.text = "Loading..."
                        funFactText.text = ""
                    }
                    MainViewState.WrongInputError -> {
                        showError()
                    }

                  …
                }
            })
}
```

RXJAVA VIEW

- Renders ViewState



```
private fun listenViewModel() {
    viewStateDisposable = viewModel.viewStateObservable
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe({
                when (it) {
                    MainViewState.Loading -> {
                        progressBar.visibility = View.VISIBLE
                        result.text = "Loading..."
                        funFactText.text = ""
                    }
                    MainViewState.WrongInputError -> {
                        showError()
                    }

                 …
               }
            })
}
```

RXJAVA VIEW

```
override fun onStart() {
    super.onStart()
    listenViewModel() // Registers to ViewState Observable
}

override fun onStop() {
    if (viewStateDisposable?.isDisposed == false) {
        viewStateDisposable?.dispose()
    }
    super.onStop()
}
```

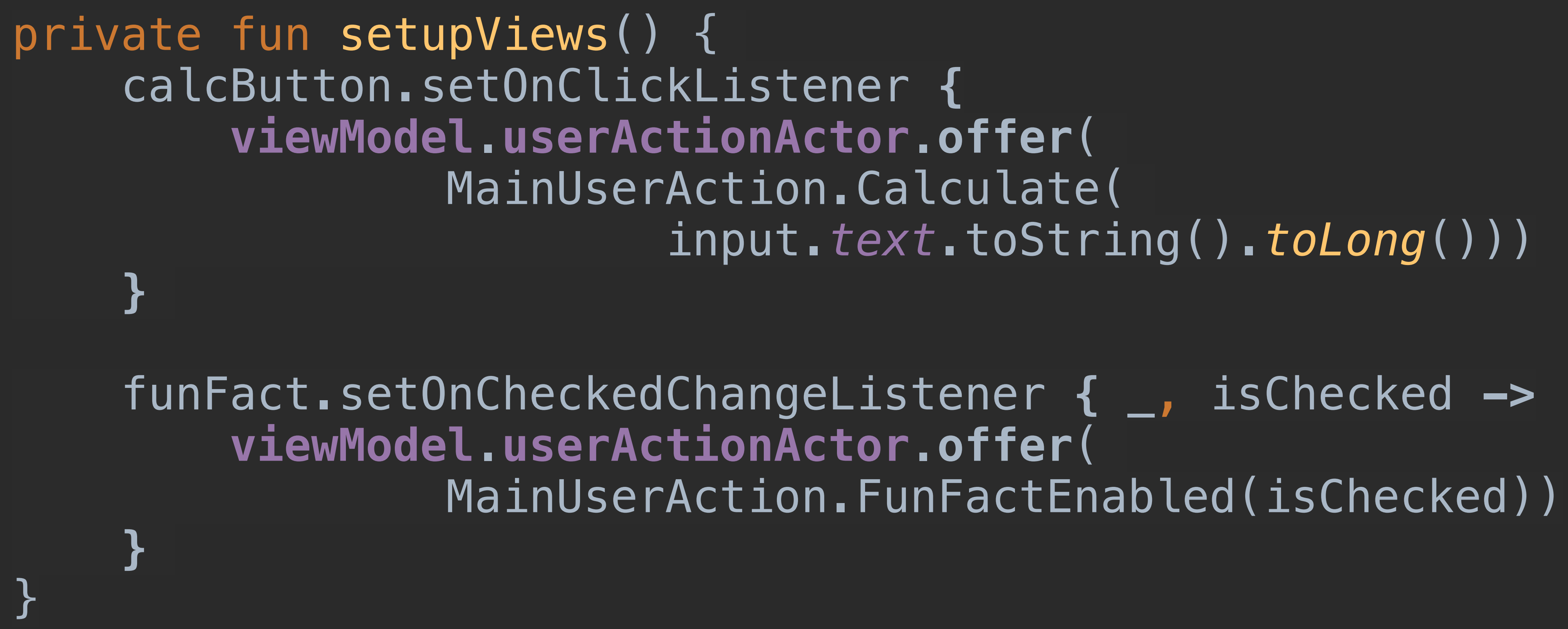
RXJAVA VIEW

- View Lifecycle

```
override fun onStart() {
    super.onStart()
    listenViewModel() // Registers to ViewState Observable
}

override fun onStop() {
    if (viewStateDisposable?.isDisposed == false) {
        viewStateDisposable?.dispose()
    }
    super.onStop()
}
```

COROUTINES VIEW

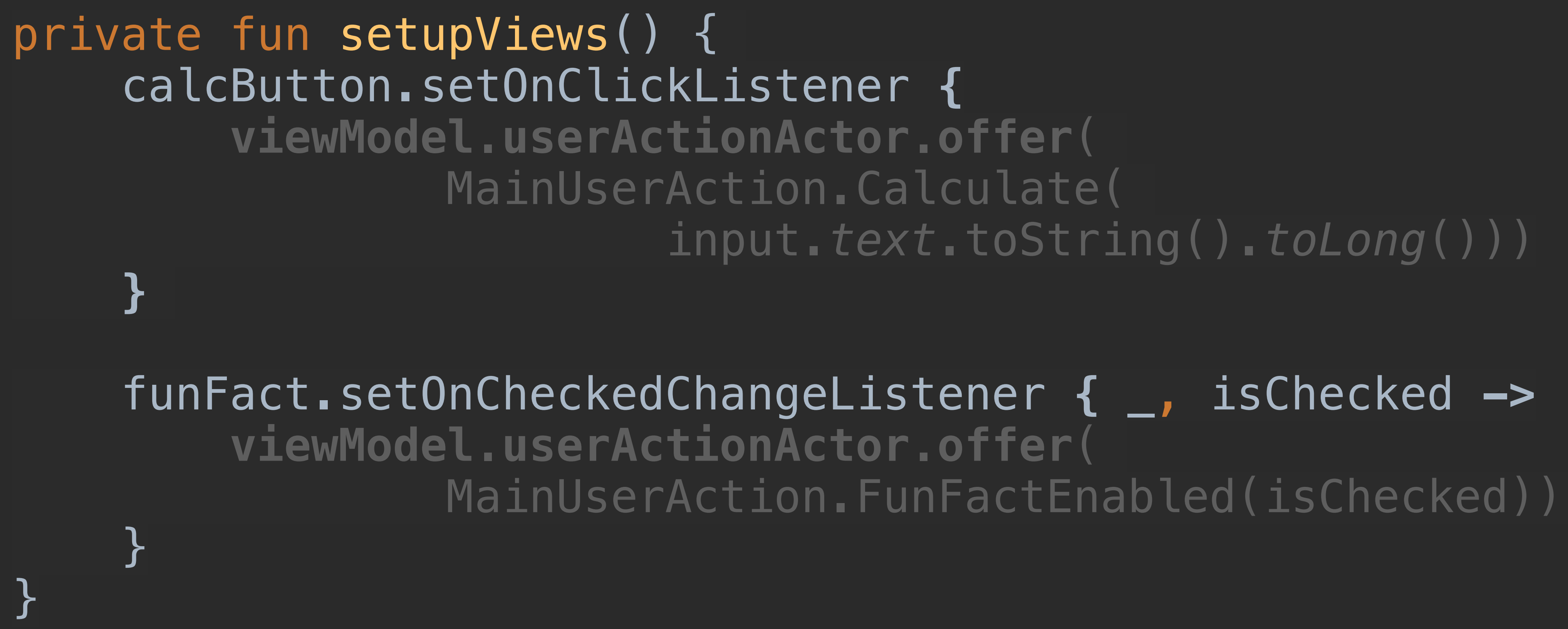
- Listens for User Events
- Talks to ViewModel to process Input



```
private fun setupViews() {
    calcButton.setOnClickListener {
        viewModel.userActionActor.offer(
                MainUserAction.Calculate(
                        input.text.toString().toLong()))
    }

    funFact.setOnCheckedChangeListener { _, isChecked ->
        viewModel.userActionActor.offer(
                MainUserAction.FunFactEnabled(isChecked))
    }
}
```

COROUTINES VIEW

- Listens for User Events
- Talks to ViewModel to process Input

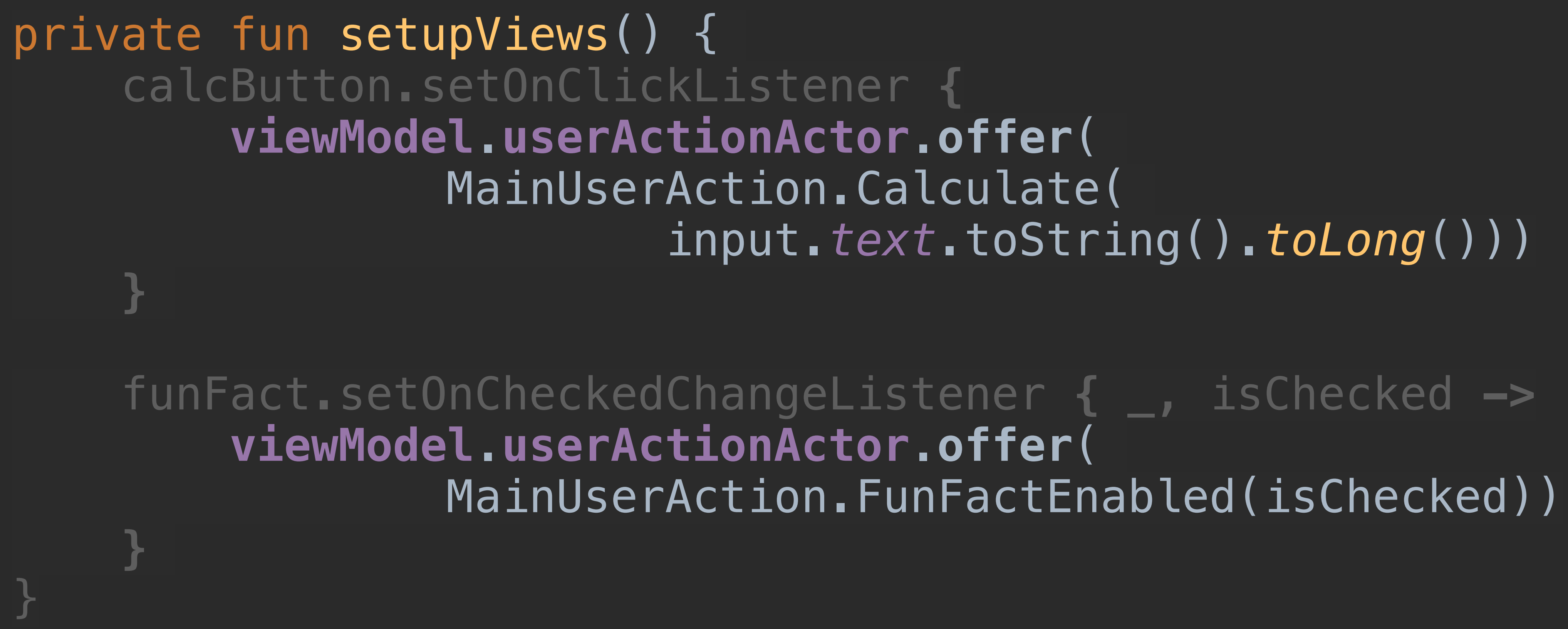


```kotlin
private fun setupViews() {
    calcButton.setOnClickListener {
        viewModel.userActionActor.offer(
                MainUserAction.Calculate(
                        input.text.toString().toLong()))
    }

    funFact.setOnCheckedChangeListener { _, isChecked ->
        viewModel.userActionActor.offer(
                MainUserAction.FunFactEnabled(isChecked))
    }
}
```

COROUTINES VIEW

- Listens for User Events

- Talks to ViewModel to process Input



```
private fun setupViews() {
    calcButton.setOnClickListener {
        viewModel.userActionActor.offer(
                MainUserAction.Calculate(
                        input.text.toString().toLong()))
    }

    funFact.setOnCheckedChangeListener { _, isChecked ->
        viewModel.userActionActor.offer(
                MainUserAction.FunFactEnabled(isChecked))
    }
}
```

COROUTINES VIEW

```kotlin
private val parentJob = Job()

private fun listenViewModel() {
    // Launch on the CommonPool to not block the MainThread
    launch(parentJob + CommonPool) {
        viewModel.viewStateChannel.consumeEach {
            withContext(UI) {
                when (it) {
                    MainViewState.Loading -> {
                        progressBar.visibility = View.VISIBLE
                        result.text = "Loading..."
                        funFactText.text = ""
                    }
                  …
                }
            }
        }
    }
}
```

COROUTINES VIEW

- Renders ViewState

```
private val parentJob = Job()

private fun listenViewModel() {
    // Launch on the CommonPool to not block the MainThread
    launch(parentJob + CommonPool) {
        viewModel.viewStateChannel.consumeEach {
            withContext(UI) {
                when (it) {
                    MainViewState.Loading -> {
                        progressBar.visibility = View.VISIBLE
                        result.text = "Loading..."
                        funFactText.text = ""
                    }
                  …
               }
            }
        }
    }
}
```

COROUTINES VIEW

```
private val parentJob = Job()

private fun listenViewModel() {
    // Launch on the CommonPool to not block the MainThread
    launch(parentJob + CommonPool) {
        viewModel.viewStateChannel.consumeEach {
            withContext(UI) {
                when (it) {
                    MainViewState.Loading -> {
                        progressBar.visibility = View.VISIBLE
                        result.text = "Loading..."
                        funFactText.text = ""
                    }
                  …
               }
            }
        }
    }
}
```

COROUTINES VIEW

- Renders ViewState

```kotlin
private val parentJob = Job()

private fun listenViewModel() {
    // Launch on the CommonPool to not block the MainThread
    launch(parentJob + CommonPool) {
        viewModel.viewStateChannel.consumeEach {
            withContext(UI) {
                when (it) {
                    MainViewState.Loading -> {
                        progressBar.visibility = View.VISIBLE
                        result.text = "Loading..."
                        funFactText.text = ""
                    }
                  …
                }
            }
        }
    }
}
```

COROUTINES VIEW

```
private val parentJob = Job()

private fun listenViewModel() {
    // Launch on the CommonPool to not block the MainThread
    launch(parentJob + CommonPool) {
        viewModel.viewStateChannel.consumeEach {
            withContext(UI) {
                when (it) {
                    MainViewState.Loading -> {
                        progressBar.visibility = View.VISIBLE
                        result.text = "Loading..."
                        funFactText.text = ""
                    }
                  …
                }
            }
        }
    }
}
```

COROUTINES VIEW

- Renders ViewState

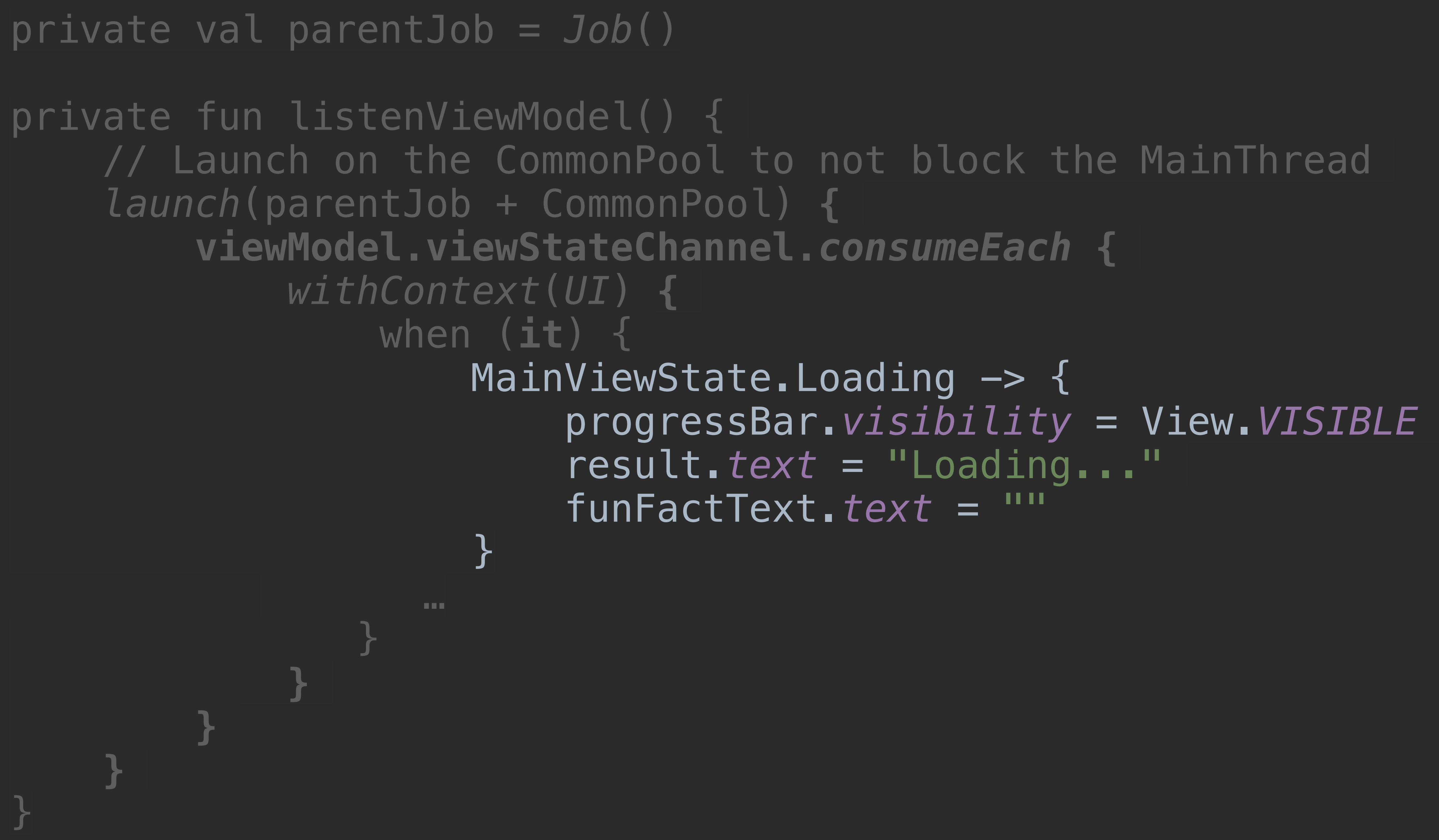
```
private val parentJob = Job()

private fun listenViewModel() {
    // Launch on the CommonPool to not block the MainThread
    launch(parentJob + CommonPool) {
        viewModel.viewStateChannel.consumeEach {
            withContext(UI) {
                when (it) {
                    MainViewState.Loading -> {
                        progressBar.visibility = View.VISIBLE
                        result.text = "Loading..."
                        funFactText.text = ""
                    }
                …
                }
            }
        }
    }
}
```

COROUTINES VIEW

```
private val parentJob = Job()

private fun listenViewModel() {
    // Launch on the CommonPool to not block the MainThread
    launch(parentJob + CommonPool) {
        viewModel.viewStateChannel.consumeEach {
            withContext(UI) {
                when (it) {
                    MainViewState.Loading -> {
                        progressBar.visibility = View.VISIBLE
                        result.text = "Loading..."
                        funFactText.text = ""
                    }
                  …
                }
            }
        }
    }
}
```

COROUTINES VIEW

- Renders ViewState



```
private val parentJob = Job()

private fun listenViewModel() {
    // Launch on the CommonPool to not block the MainThread
    launch(parentJob + CommonPool) {
        viewModel.viewStateChannel.consumeEach {
            withContext(UI) {
                when (it) {
                    MainViewState.Loading -> {
                        progressBar.visibility = View.VISIBLE
                        result.text = "Loading..."
                        funFactText.text = ""
                    }

                  …
                }
            }
        }
    }
}
```

COROUTINES VIEW

```
override fun onStart() {
    super.onStart()
    listenViewModel()    // Listens to ViewStateChannel
}

override fun onStop() {
    parentJob.cancel()
    super.onStop()
}
```

COROUTINES VIEW

- View Lifecycle

```
override fun onStart() {
    super.onStart()
    listenViewModel()    // Listens to ViewStateChannel
}

override fun onStop() {
    parentJob.cancel()
    super.onStop()
}
```

SHOW ME SOME NUMBERS

PERFORMANCE

PERFORMANCE

- Roughly the same
- Not difficult enough to compare

SIZE

Measure	Coroutines	RxJava	Interop
APK Size	2.4MB	2.9MB	3.1MB
Method Count	29,131	37,271	39,590

GITHUB LINKS

Coroutines

manuelvicnt/**MathCoroutines**

RxJava

manuelvicnt/**MathRxJava**

Coroutines and RxJava

manuelvicnt/**MathRxCoroutines**

CONCLUSION

CONCLUSION

- Both libraries provide a way to do Asynchronous Programming
- If you are a RxJava expert, no need to switch
- If you struggle with RxJava, Coroutines is another option you can try
- For new Android developers, Coroutines has a lower learning curve

ANY QUESTIONS?

COROUTINES AND RXJAVA

AN ASYNCHRONICITY COMPARISON

MANUEL VICENTE VIVO

@MANUELVICNT