Lab 06 - Android Flow

Software Studio - 2022 Spring

In coroutines, a `flow` is a type that can emit multiple values sequentially, as opposed to suspend functions that return only a single value.

A flow is very similar to an Iterator that produces a sequence of values, but it uses `suspend` functions to produce and consume values asynchronously.

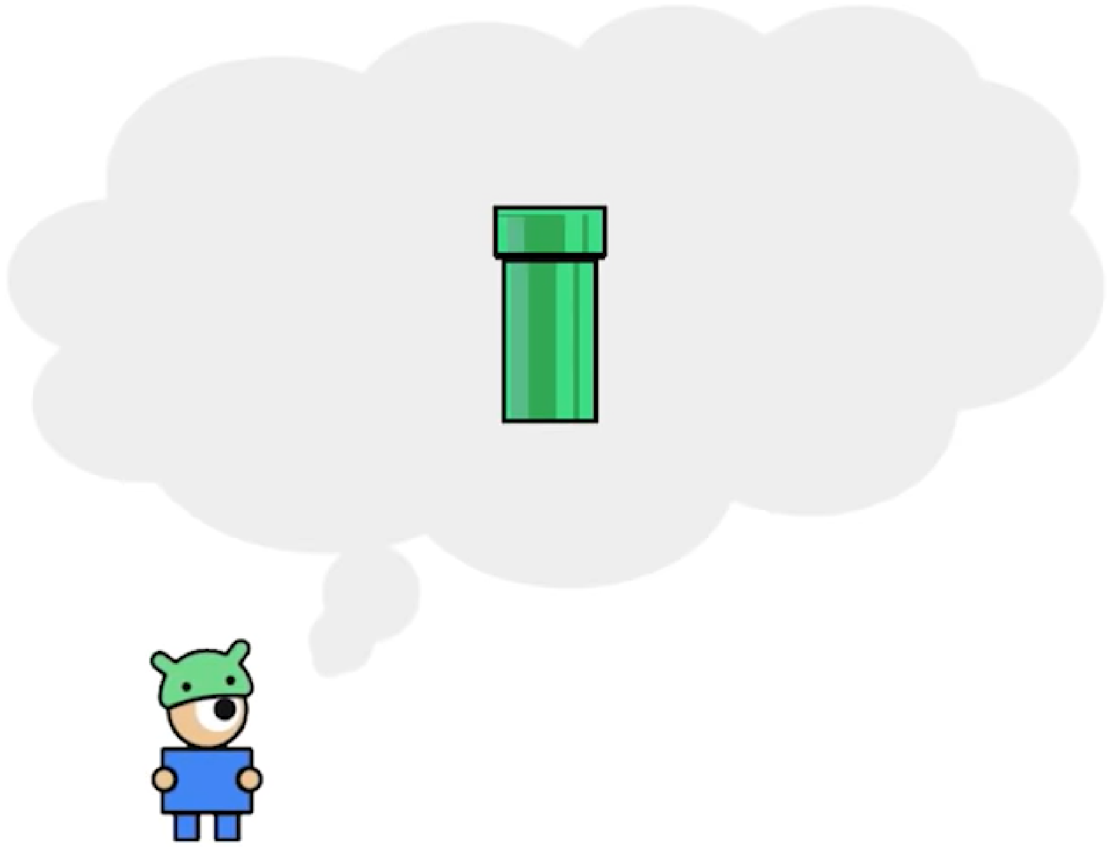
This means, for example, that the flow can safely make a network request to produce the next value without blocking the main thread.

Explain why we use Flow

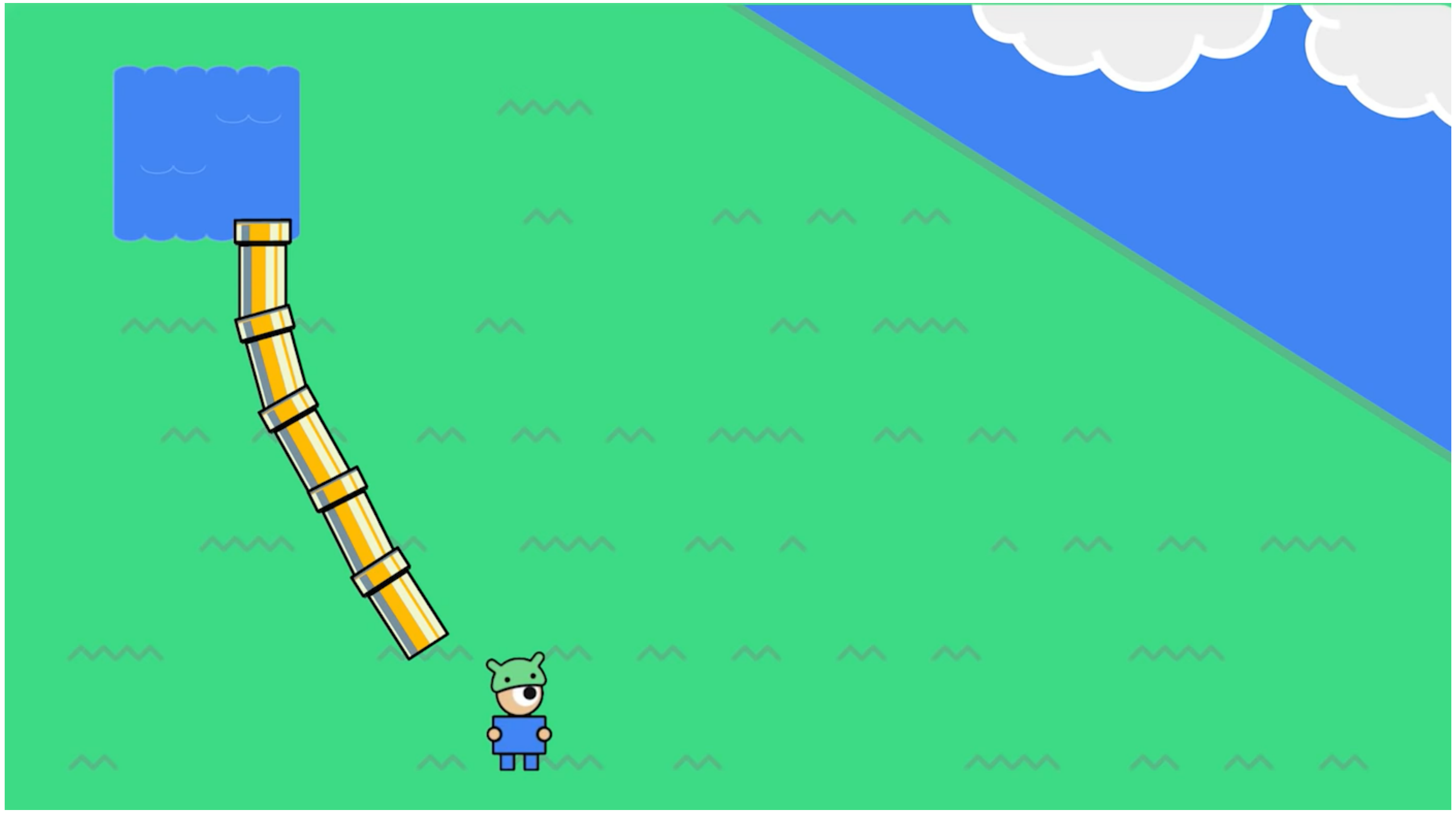
There is a developer called Pancoho.

Explain why we use Flow

Explain why we use Flow

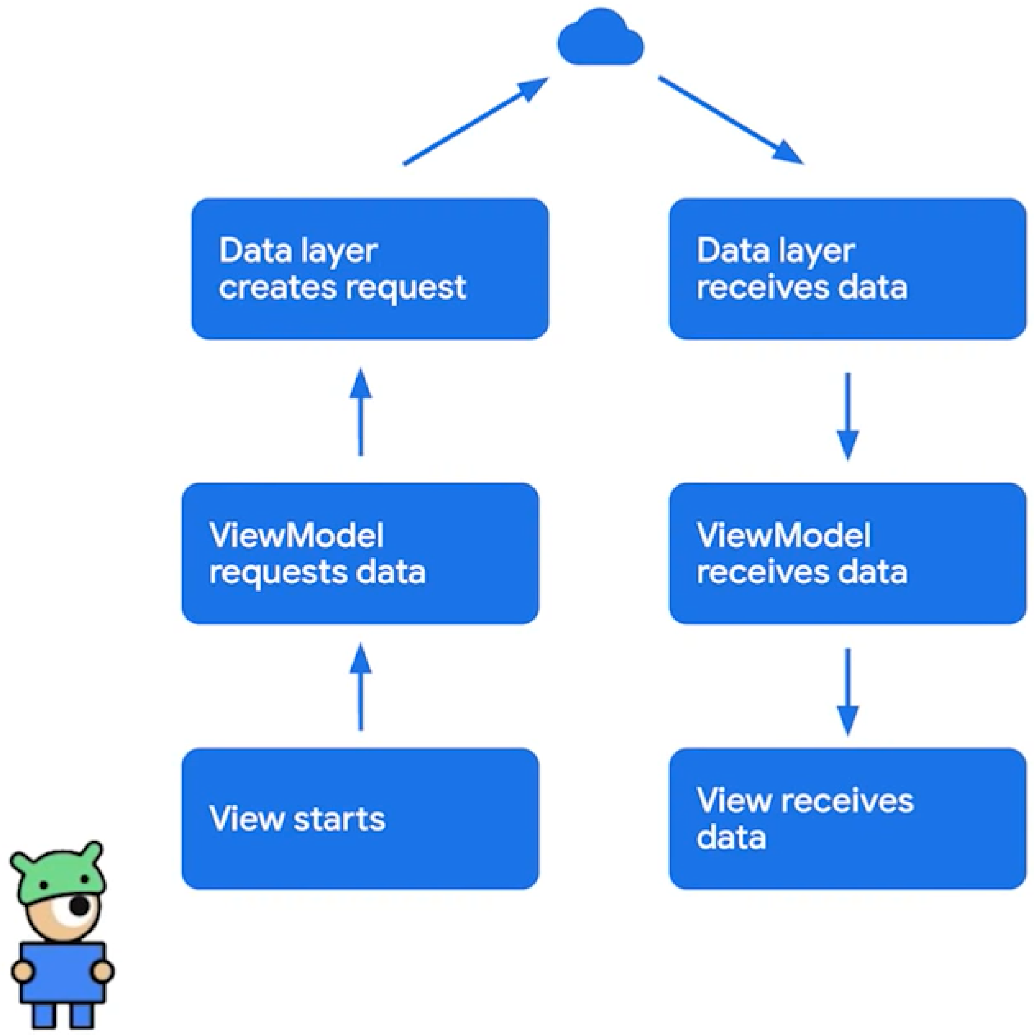


Explain why we use Flow



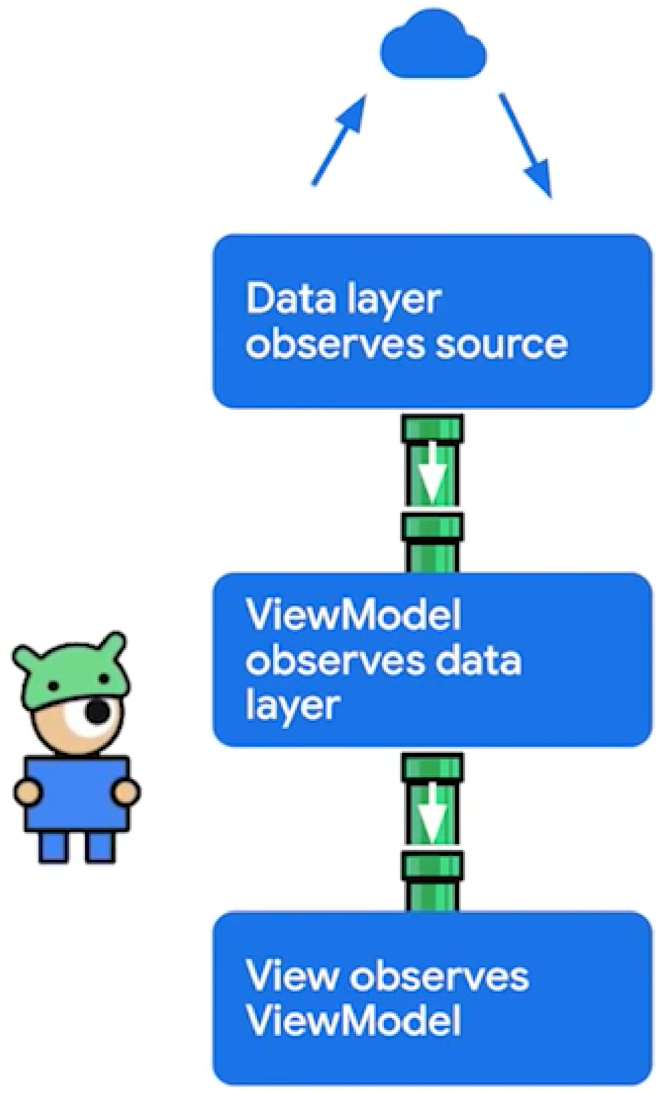
Without Flow

- You can do it with lots of `suspend` functions.
- However, after doing that for a while, it takes a lot of time for developers to invest in the infrastructures.

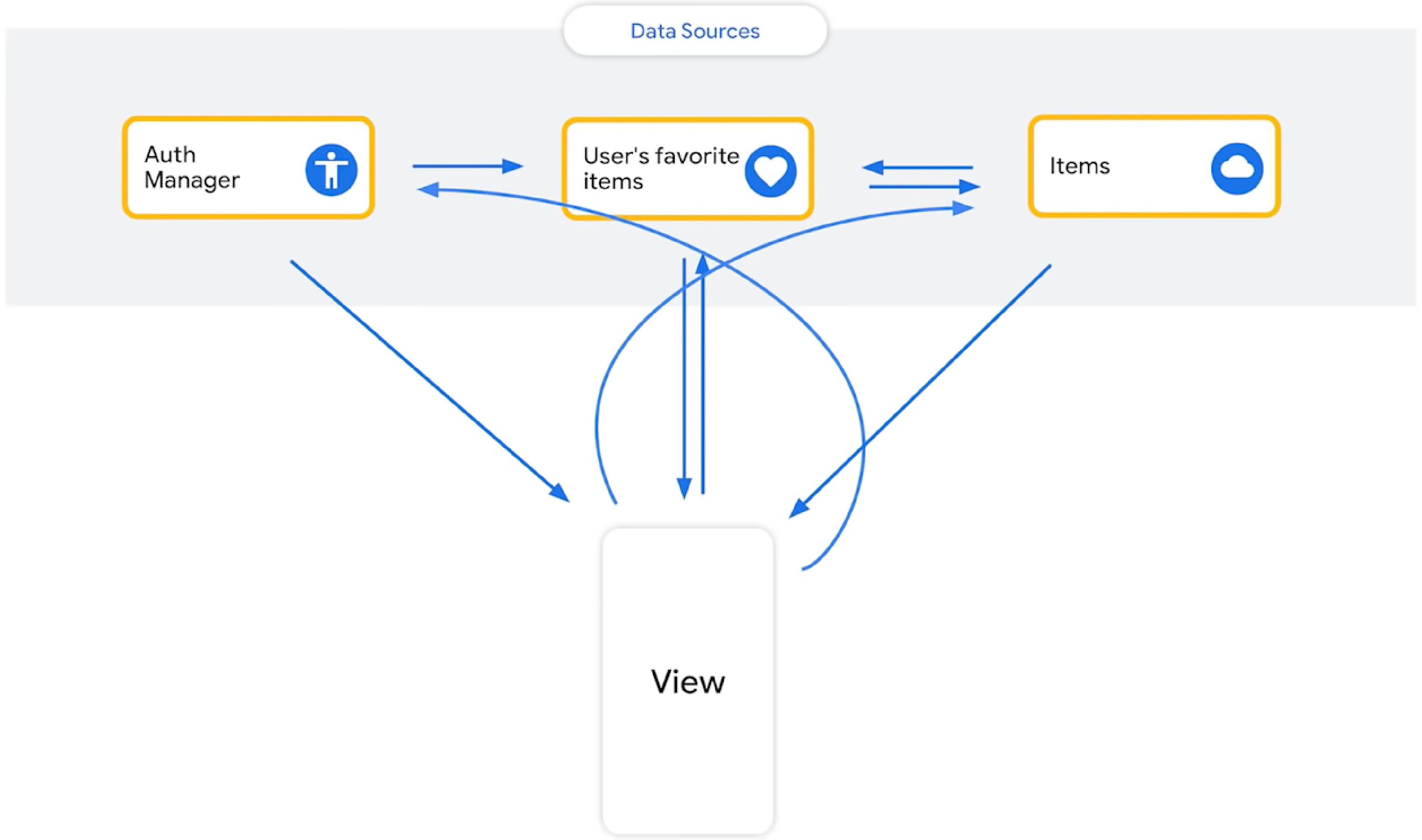


With Flow

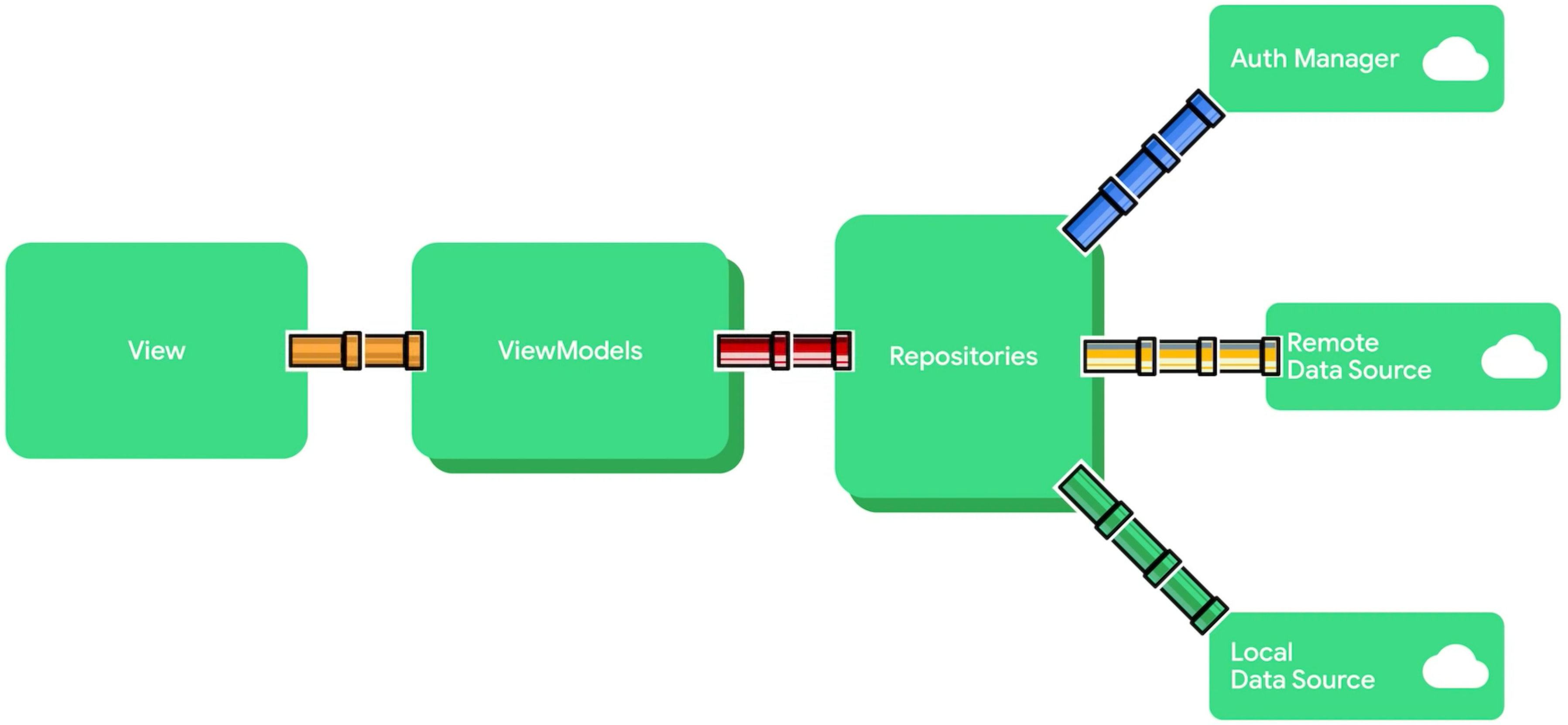
- Instead of requesting data, we **observe** it.
- Observing it like installing tubes for water.
- **Any updates to the source data will flow down to the view automatically**, you don't have to walk to the lake!!



Without Flow



With Flow


View
ViewModels
Repositories
Auth Manager
Remote Data Source
Local Data Source

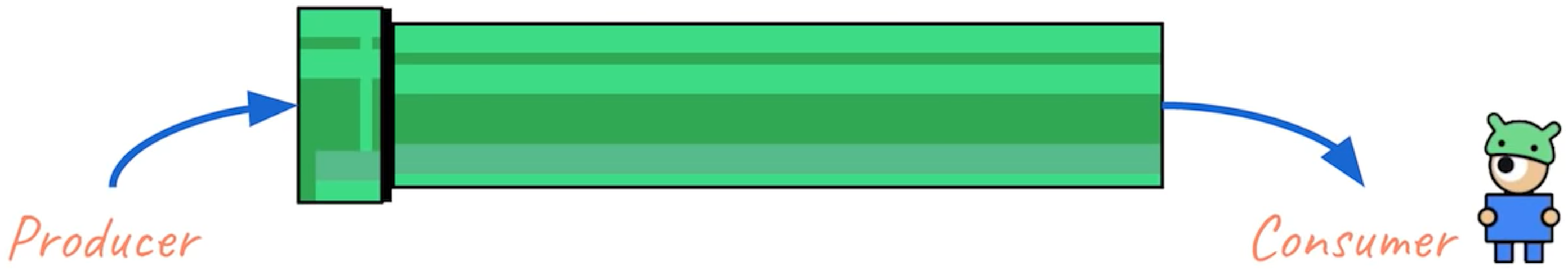
Kotlin Flow<T>

Producers emits data to the flow.

- Data Sources or Repositories

Consumers collects data from the flow.

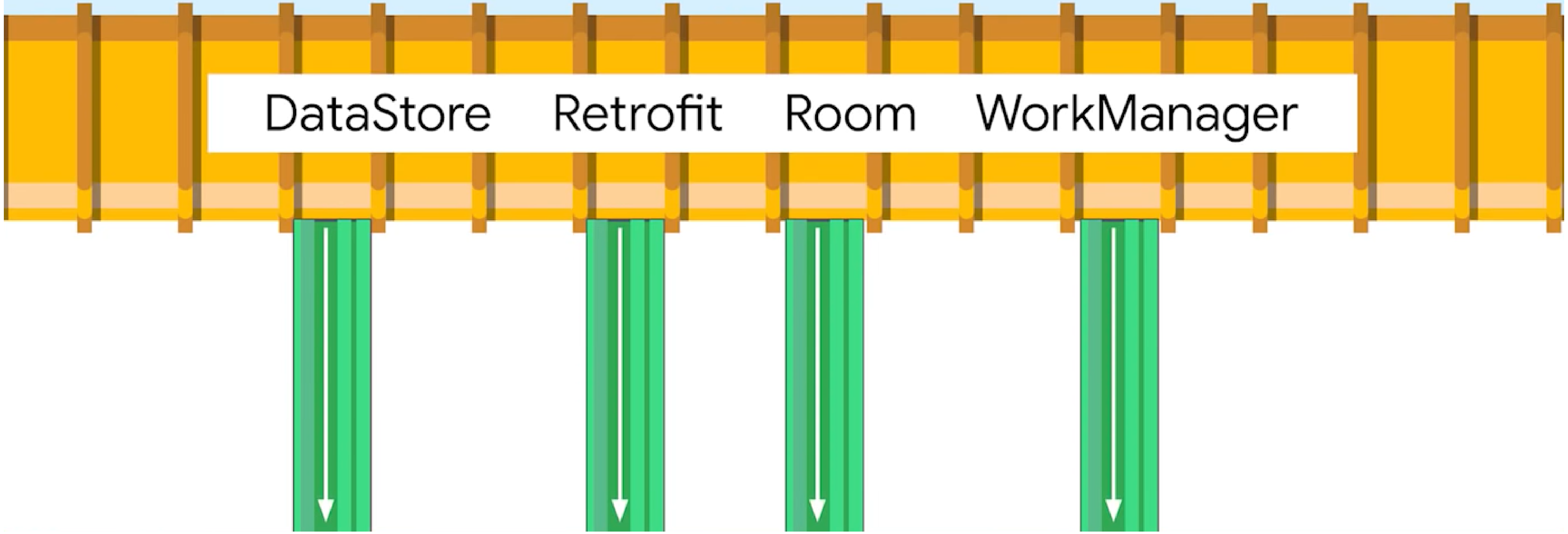
- UI layer



Creating Flows

- Most of the times, you don't need to create a flow. The libraries in your data sources are already integreated with coroutines and flows.

You don't need to create Flows most of the time



Room DAO

The `Room` library acts as a producer and **emit** the content of the query every time.

```
@Dao
interface ScheduleDao {

    @Query("SELECT * FROM schedule ORDER BY arrival_time ASC")
    fun getAll(): Flow<List<Schedule>>
}
```

Creating Flows By Yourself!!

```kotlin
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking
import kotlinx.coroutines.delay

fun getSequence(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(1000)
        emit(i)
    }
}
fun main() = runBlocking {
    getSequence()
        .collect { value ->
            println(value)
        }
}
```

Creating Flows By Yourself!!

`flow()` is a flow builder. It would create a `Flow<T>`.

`emit()` send the result into flow. And we know that flow can emit multiple times and values.

`collect()` receive the values from `emit()`. Every time `emit()` is called, the block in `collect()` will be executed.

Flow Builder

1. `flow()`
2. `asFlow()`
3. `flowOf()`
4. ...

`flow()`

```kotlin
fun <T> flow(
    block: suspend FlowCollector<T>.() -> Unit
): Flow<T>
```

It creates a **cold flow**.

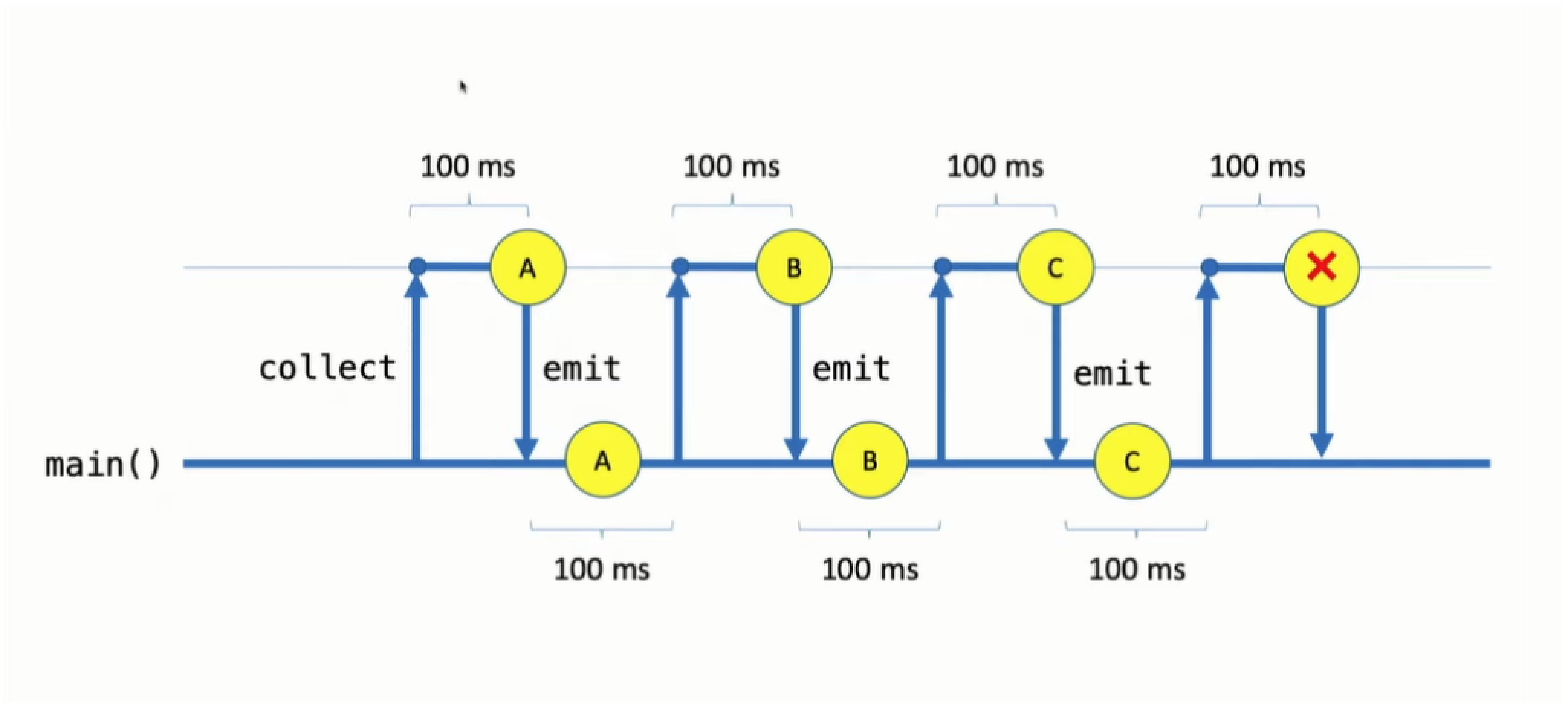
The flow being *cold* means that the block is called every time a terminal operator is applied to the resulting flow.

Example of `flow()`

```kotlin
fun getSequence(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100)
        println("Emit $i")
        emit(i)
    }
}
fun main() = runBlocking {
    val f = getSequence()
    println("Start to collect")
    f.collect { value ->
        delay(1000)
        println("Collected $value")
    }
}
```

Result of `flow()`

```
Start to collect
Emit 1
Collected 1
Emit 2
Collected 2
Emit 3
Collected 3
```


100 ms
100 ms
100 ms
100 ms
A
B
C
collect
emit
emit
emit
main()
A
B
C
100 ms
100 ms
100 ms

`asFlow()`

```
fun IntRange.asFlow(): Flow<Int>
```

Creates a flow that produces values from the range.

Example of `asFlow()`

```kotlin
fun main() = runBlocking {
    val f = (1..3).asFlow()
    println("1. Start to collect")
    f.collect { value ->
        println("Collected $value")
    }
    println("2. Start to collect")
    f.collect { value ->
        println("Collected $value")
    }
}
```

Result of `asFlow()`

```
1. Start to collect
Collected 1
Collected 2
Collected 3
2. Start to collect
Collected 1
Collected 2
Collected 3
```

`flowOf()`

```kotlin
fun <T> flowOf(vararg elements: T): Flow<T>
```

Creates a flow that produces values from the specified `vararg`-arguments.

Example of `asFlow()`

```kotlin
fun main() = runBlocking {
    val f = flowOf(1, 2, 3)
    println("1. Start to collect")
    f.collect { value ->
        println("Collected $value")
    }
    println("2. Start to collect")
    f.collect { value ->
        println("Collected $value")
    }
}
```

Result of `asFlow()`

```
1. Start to collect
Collected 1
Collected 2
Collected 3
2. Start to collect
Collected 1
Collected 2
Collected 3
```

Operators

Flow is sequential.
Before calling `collect()`, we can use different function operators to deal with or transform the value in flow.

`map()`

```
fun main() = runBlocking {
    (1..3).asFlow()
        .map { "Hello $it" }
        .collect { println(it) }
}
```

Result :

```
Hello 1
Hello 2
Hello 3
```

`filter()`

```kotlin
fun main() = runBlocking {
    (1..10).asFlow()
        .filter { it % 2 == 0 }
        .collect { println(it) }
}
```

Result :

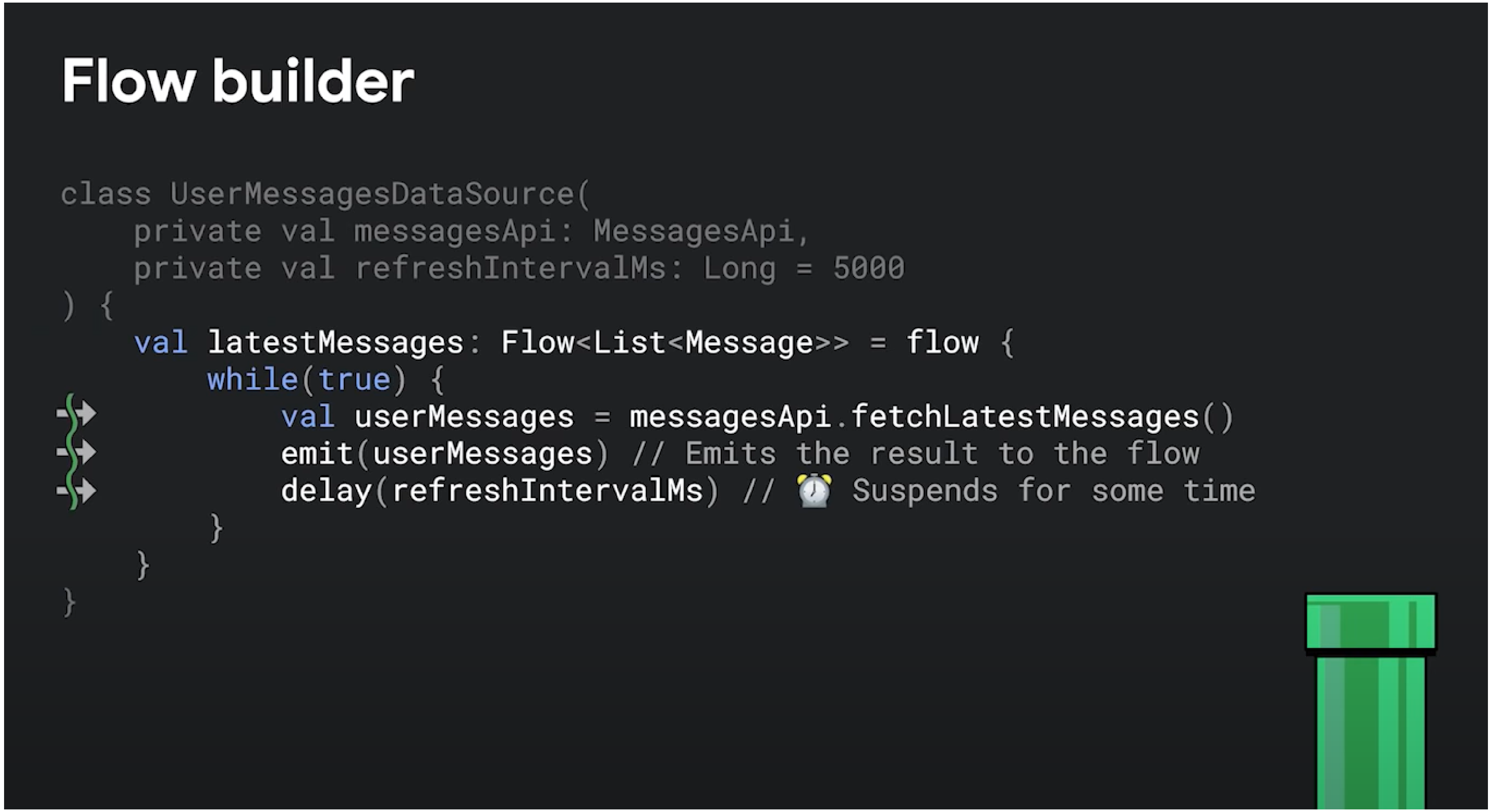
```
2
4
6
8
10
```

`transform()`

`transform()` is a flexible function that may transform emitted element, skip it or emit it multiple times.

```
fun main() = runBlocking {
    (1..10).asFlow()
        .transform {
            if (it % 2 == 0) {
                emit(it)
                emit(it)
            }
        }
        .collect { println(it) }
}
```

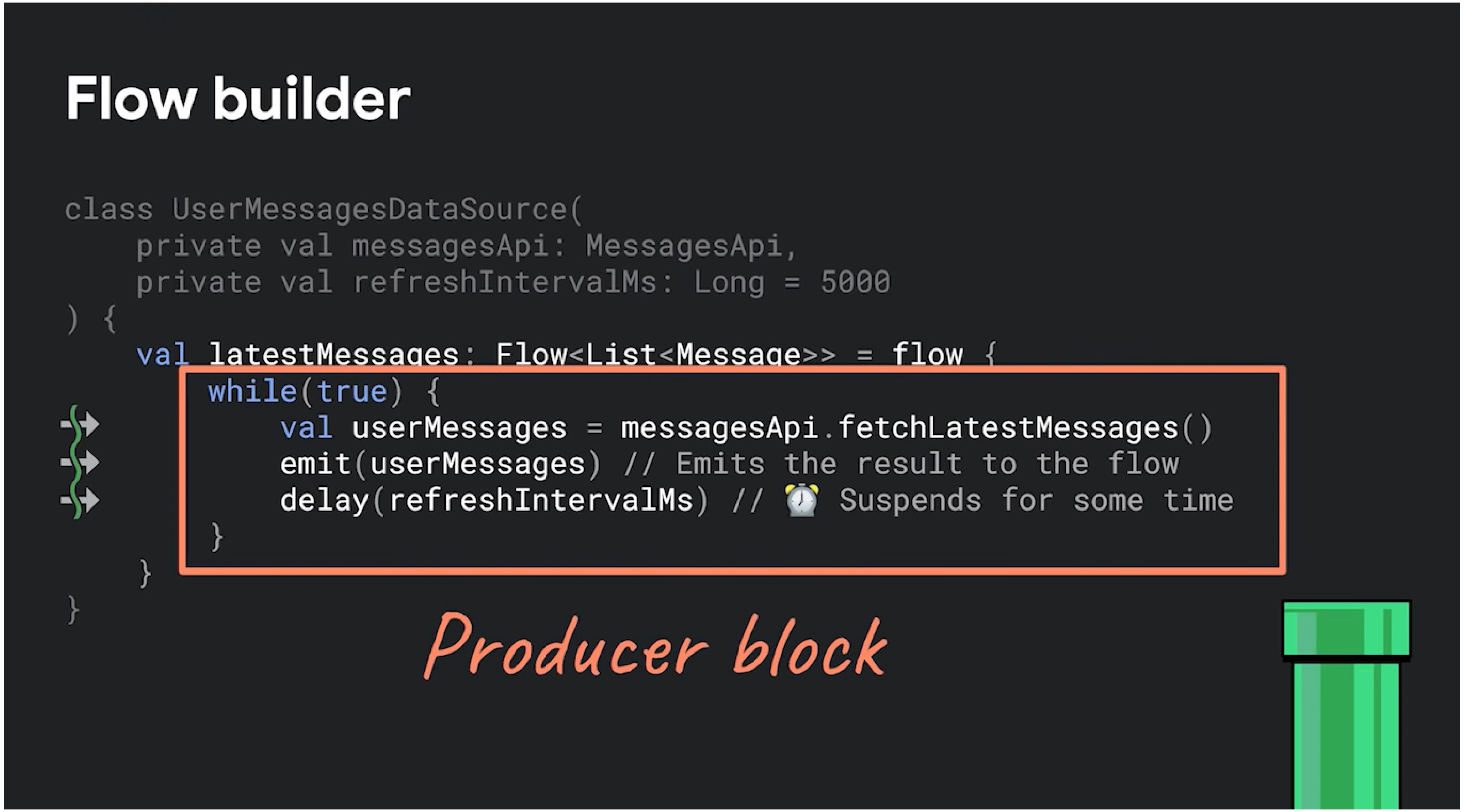
Advanced example



Flow builder

```kotlin
class UserMessagesDataSource(
    private val messagesApi: MessagesApi,
    private val refreshIntervalMs: Long = 5000
) {
    val latestMessages: Flow<List<Message>> = flow {
        while(true) {
            val userMessages = messagesApi.fetchLatestMessages()
            emit(userMessages) // Emits the result to the flow
            delay(refreshIntervalMs) // ⏰ Suspends for some time
        }
    }
}
```

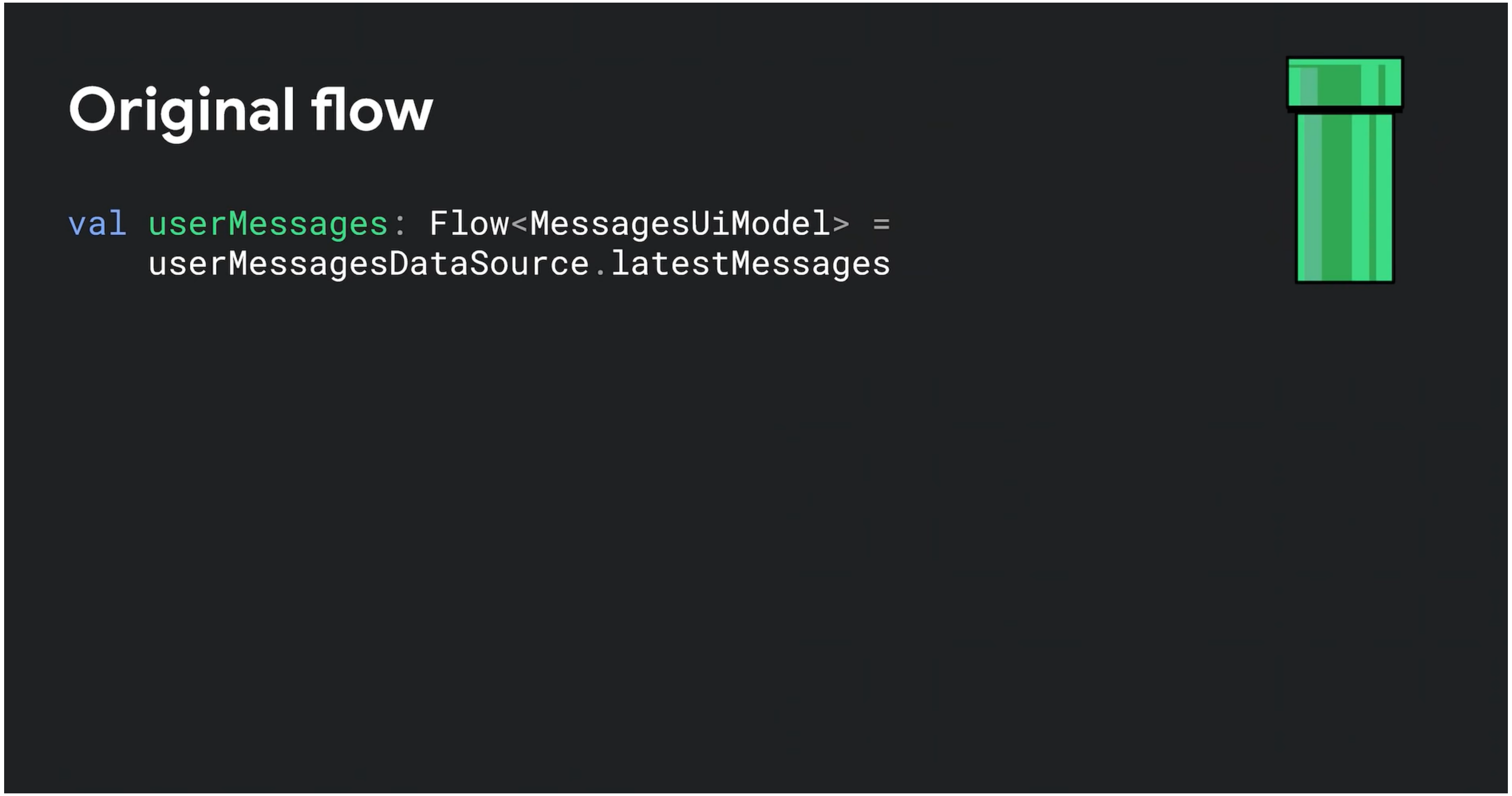
Advanced example



Flow builder

```
class UserMessagesDataSource(
    private val messagesApi: MessagesApi,
    private val refreshIntervalMs: Long = 5000
) {
    val latestMessages: Flow<List<Message>> = flow {
        while(true) {
            val userMessages = messagesApi.fetchLatestMessages()
            emit(userMessages) // Emits the result to the flow
            delay(refreshIntervalMs) // ⏰ Suspends for some time
        }
    }
}
```

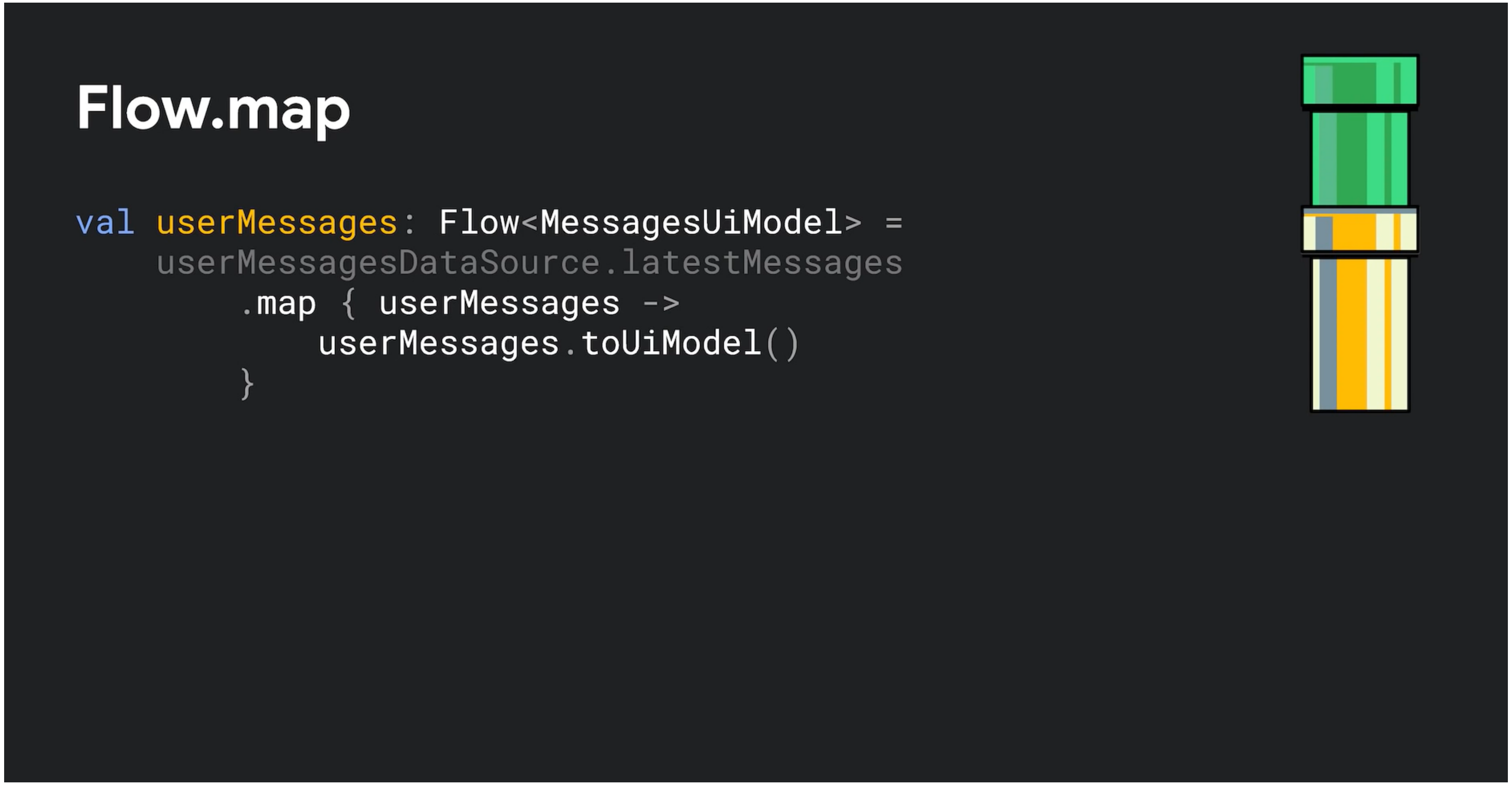
Advanced example



Original flow

```
val userMessages: Flow<MessagesUiModel> =
    userMessagesDataSource.latestMessages
```

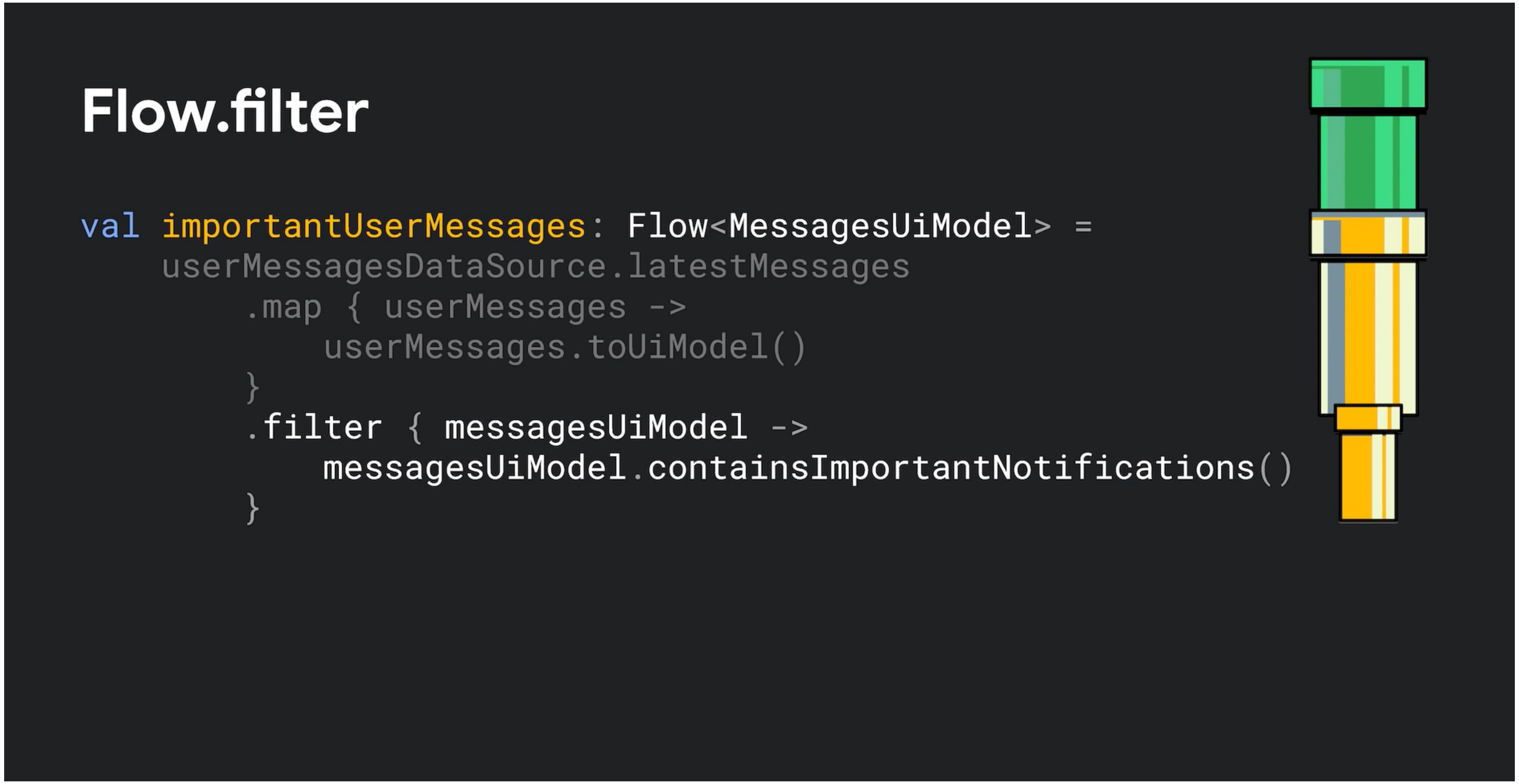
Advanced example



Flow.map

```
val userMessages: Flow<MessagesUiModel> =
    userMessagesDataSource.latestMessages
        .map { userMessages ->
            userMessages.toUiModel()
        }
```

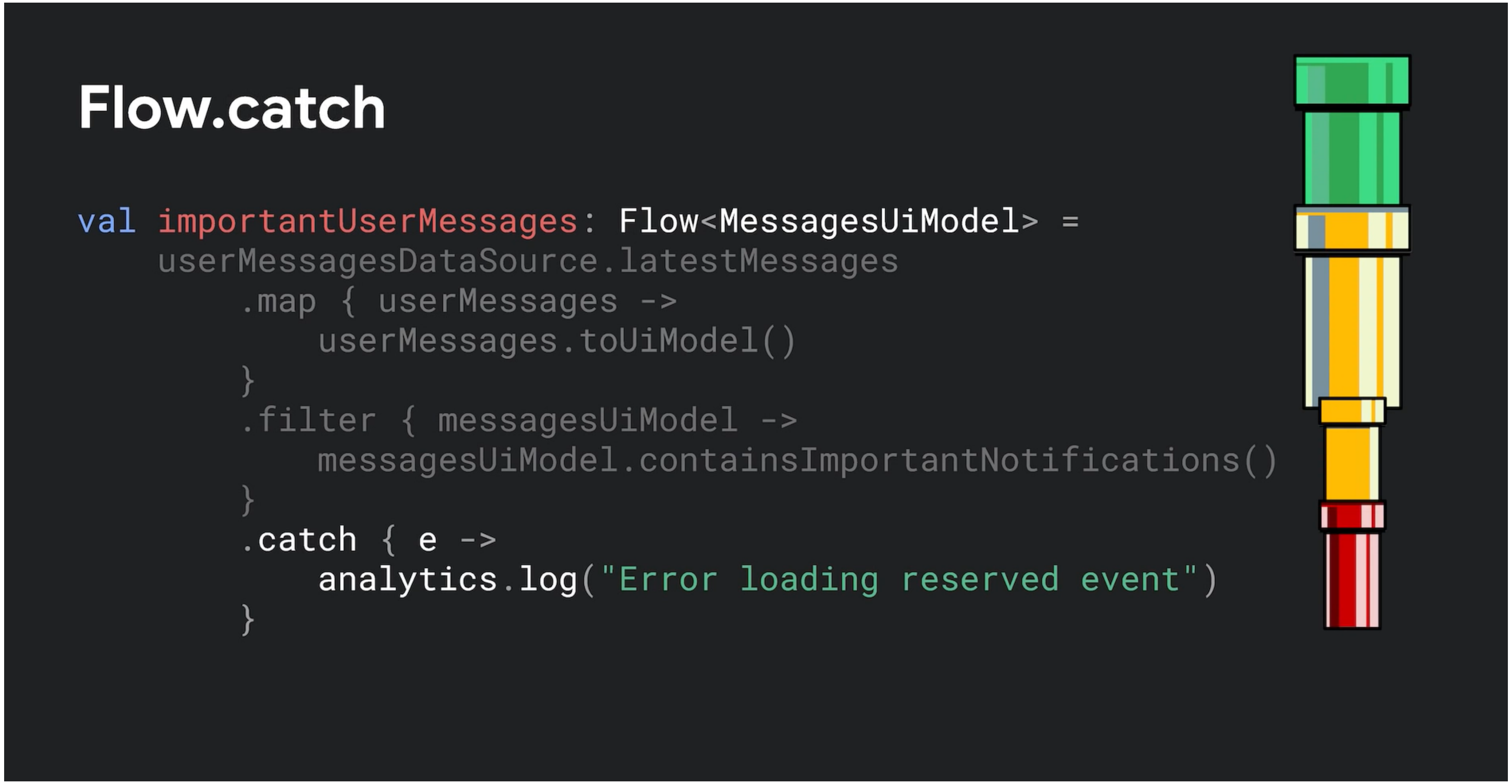
Advanced example



Flow.filter

```
val importantUserMessages: Flow<MessagesUiModel> =
    userMessagesDataSource.latestMessages
        .map { userMessages ->
            userMessages.toUiModel()
        }
        .filter { messagesUiModel ->
            messagesUiModel.containsImportantNotifications()
        }
```

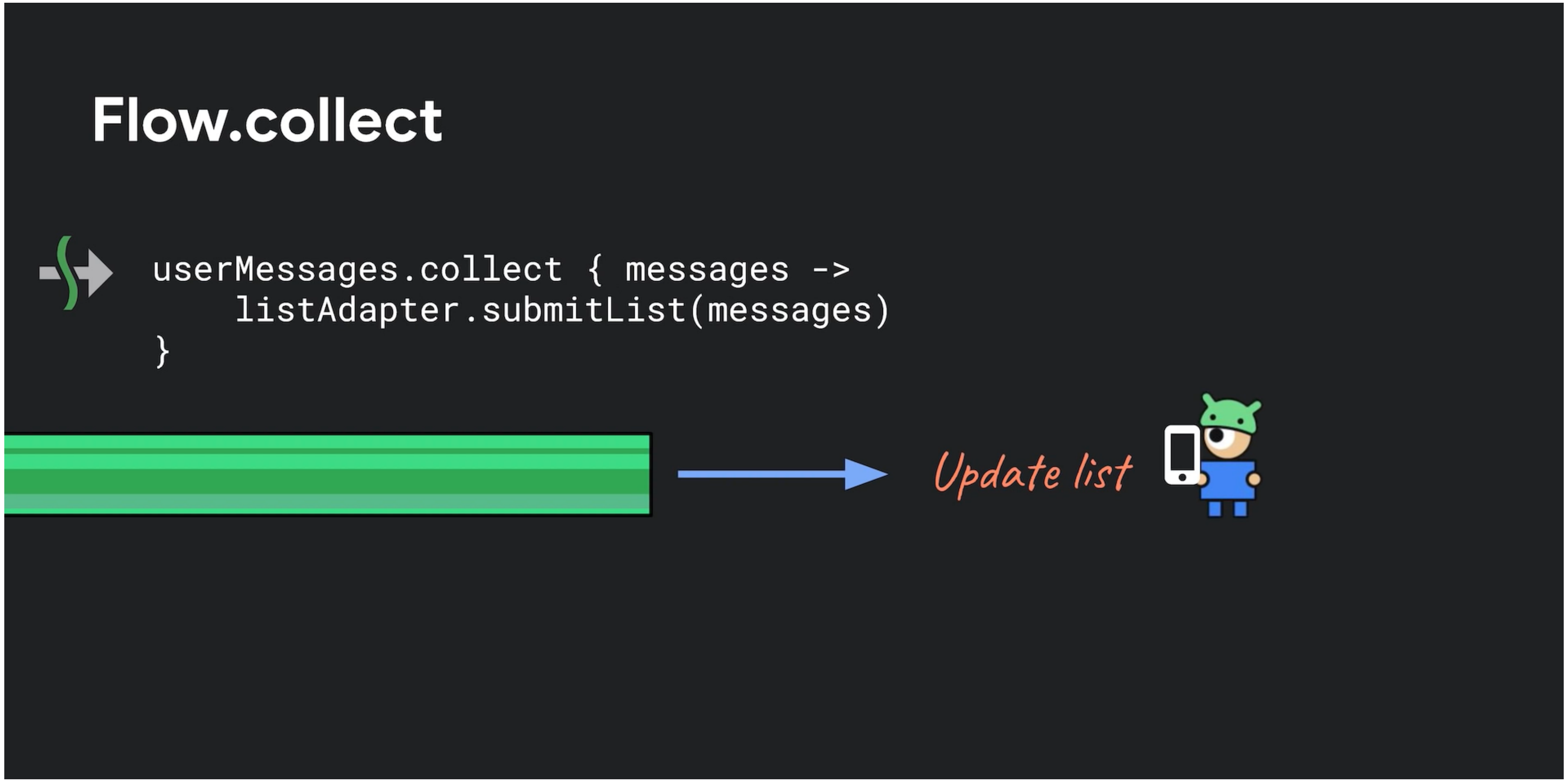
Advanced example



Flow.catch

```kotlin
val importantUserMessages: Flow<MessagesUiModel> =
    userMessagesDataSource.latestMessages
        .map { userMessages ->
            userMessages.toUiModel()
        }
        .filter { messagesUiModel ->
            messagesUiModel.containsImportantNotifications()
        }
        .catch { e ->
            analytics.log("Error loading reserved event")
        }
```

Advanced example



Flow.collect

```
userMessages.collect { messages ->
    listAdapter.submitList(messages)
}
```

Reference

Kotlin flows on Android

Asynchronous Flow

Kotlin Coroutine Flow