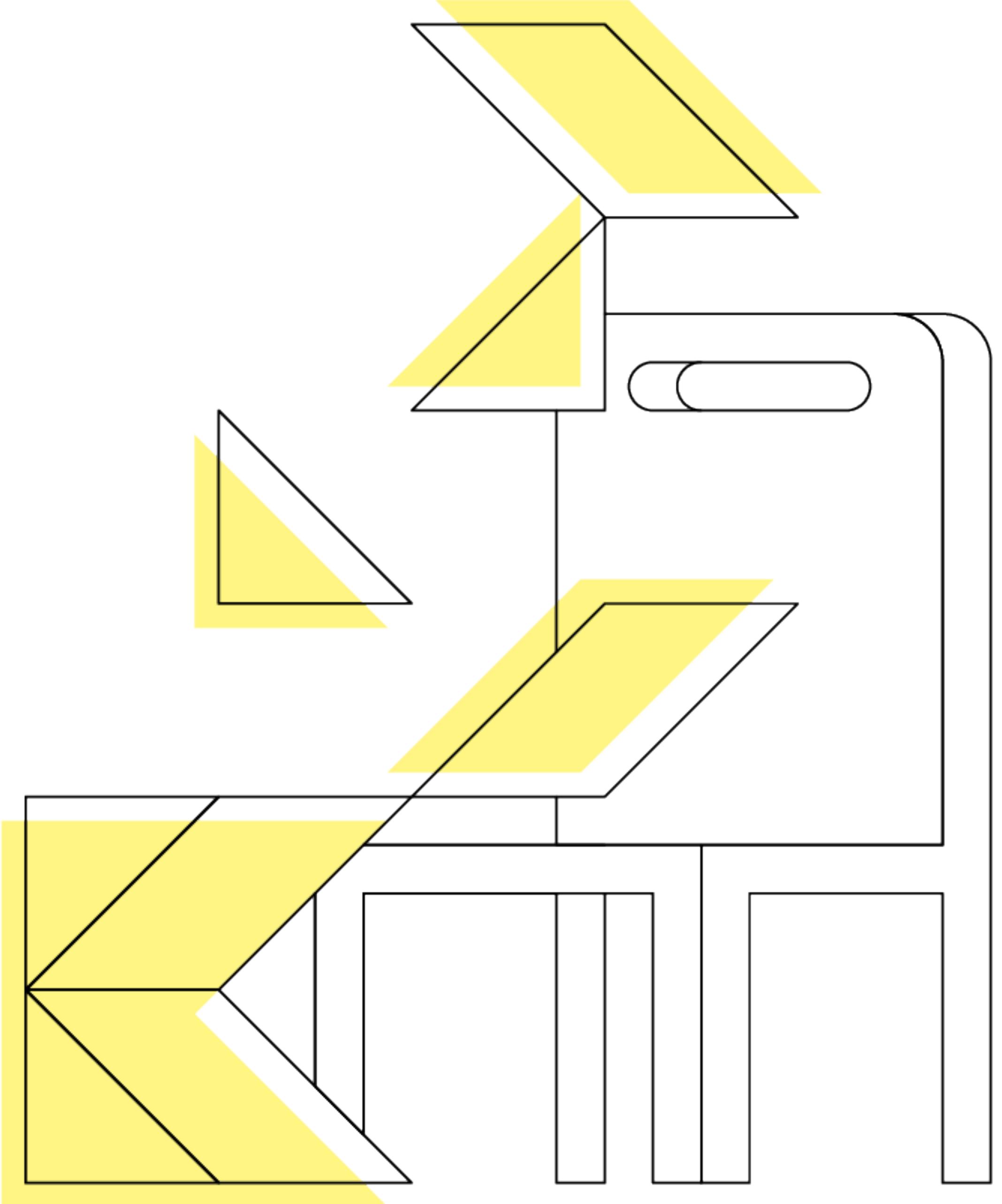


KOTLIN / Everywhere

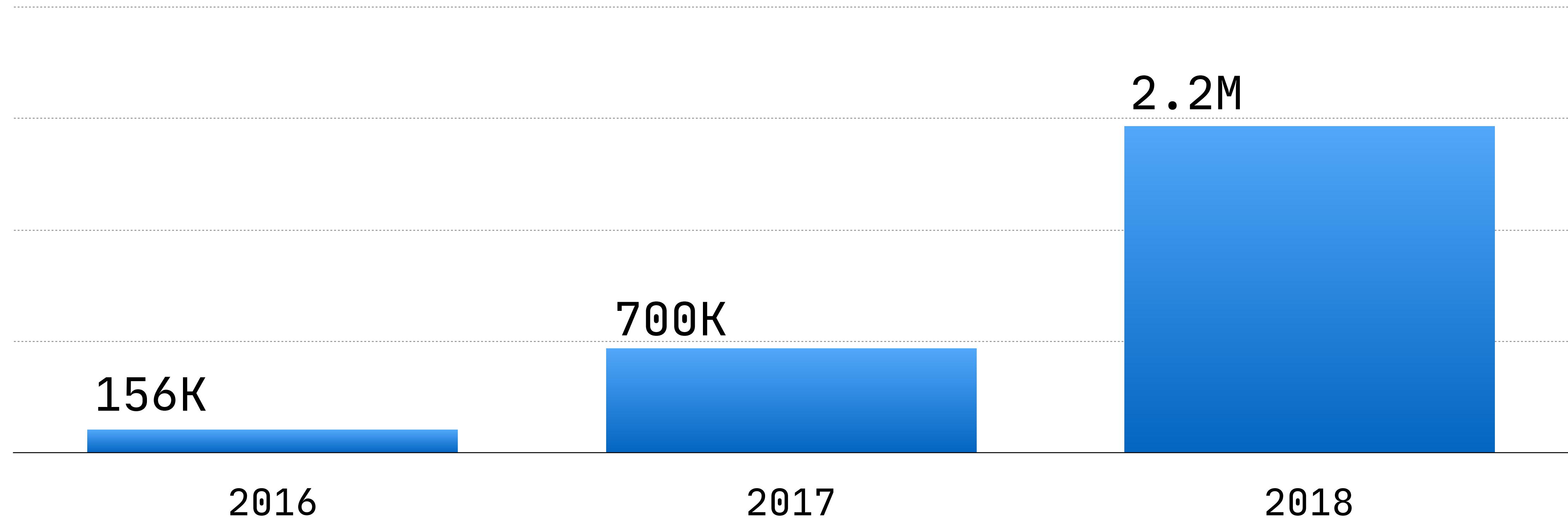
Beijing

What's new in Kotlin?

Svetlana Isakova
@sveta_isakova



Kotlin developers



Timeline

- 2010 Project started
- 2016 Kotlin 1.0
- 2017 Official on Android
- 2018 Kotlin 1.3





Kotlin evolution

- Kotlin 1.0 Kotlin gets stable!
- Kotlin 1.1 Coroutines (experimental)
- Kotlin 1.2 Multi-platform projects (experimental)
 Kotlin/Native (experimental)
- Kotlin 1.3 Coroutines get stable!
 Multi-platform projects can target Kotlin/Native (experimental)

Agenda

- Kotlin evolution
- “Experimental” features

Agenda: experimental features

- Inline Classes
- Contracts
- Channels
- Flows
- Multiplatform Projects

Principles of Pragmatic Evolution



Language design is cast in stone,
but this stone is reasonably soft,
and with some effort we can reshape
it later.

Kotlin Design Team

Principles of Pragmatic Evolution

- keeping the language modern
- comfortable updates
- feedback loop

KEEPing the language modern

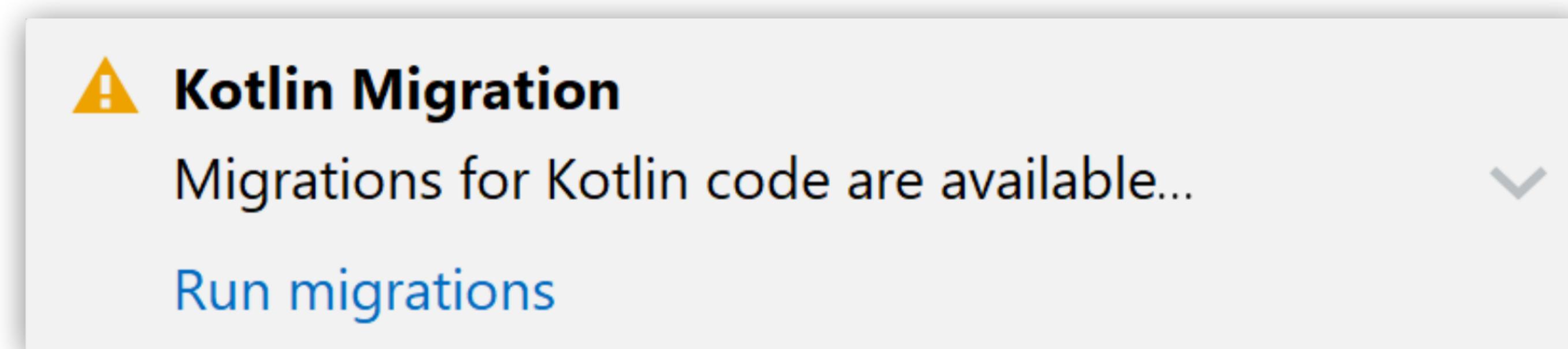
<https://github.com/Kotlin/KEEP>

KEEP = Kotlin Evolution
and Enhancement Process

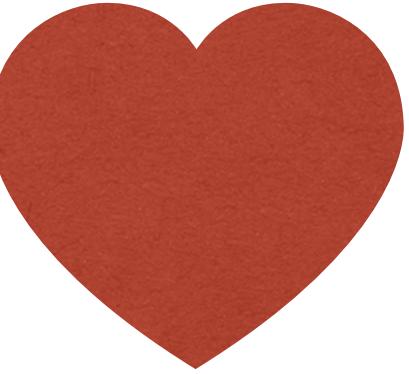
contains language proposals
and the corresponding discussions

Comfortable updates

- Deprecation warnings in advance
- Automatic migration



Feedback loop with the community

- Kotlin  the community
- Kotlin team listens to the community
- The community members influence the Kotlin evolution

Contributors from all over the world



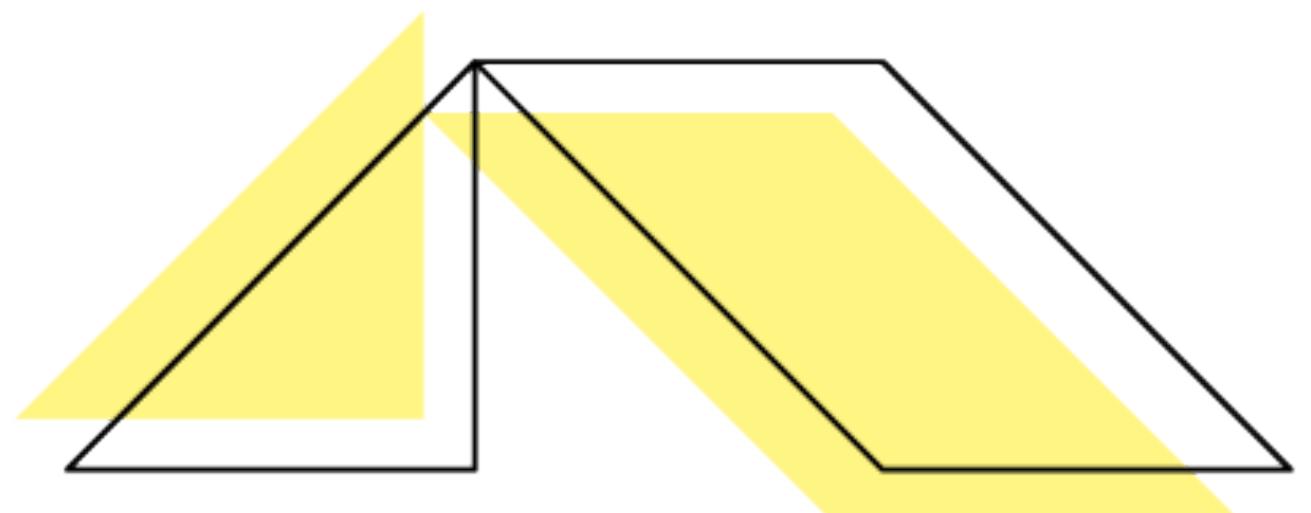
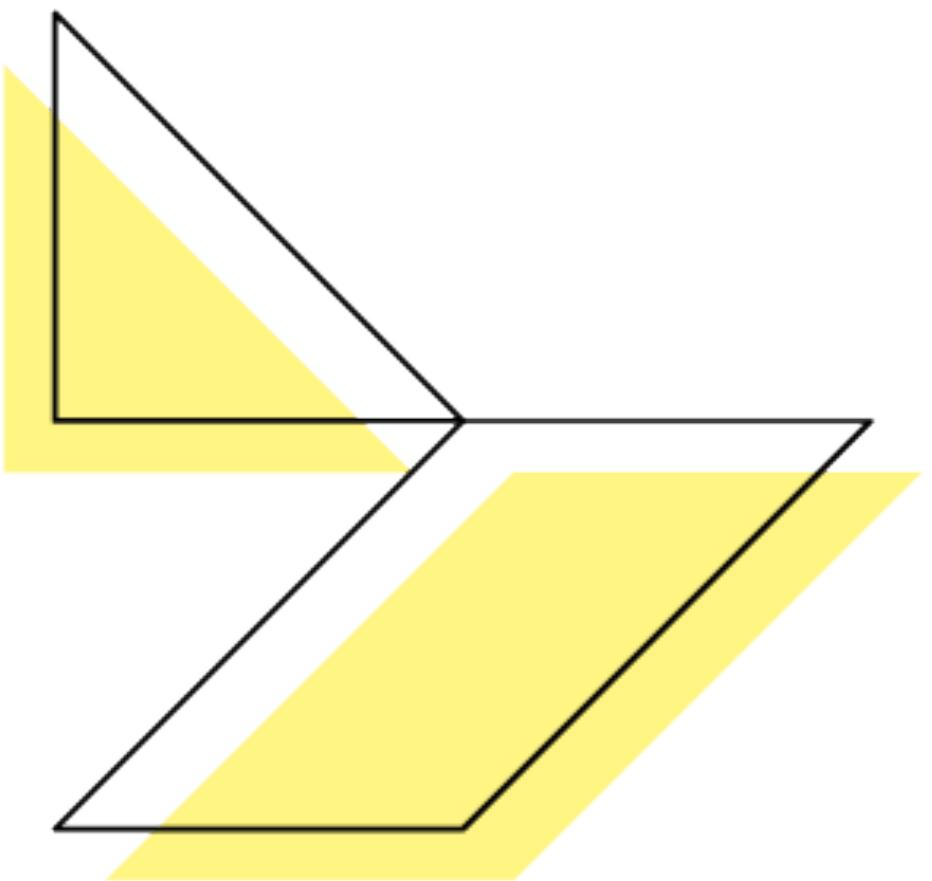
External Contributions

We want to especially thank [Toshiaki Kameyama](#) for his ongoing work on providing many useful intentions and inspections for IntelliJ IDEA.

We'd like to thank all our external contributors whose pull requests were included in this release:

- * [Steven Schäfer](#)
- * [pyos](#)
- * [Ivan Gavrilovic](#)
- * [Mads Ager](#)
- * [Ting-Yuan Huang](#)

Experimental features



Experimental features

The goal: to let new features be tried
by early adopters as soon as possible

Experimental Language Features

- can be changed in the future
- you need to explicitly opt in at the call site to use experimental features

```
compileTestKotlin {  
    kotlinOptions {  
        freeCompilerArgs += "-Xinline-classes"  
    }  
}
```

Experimental API for Libraries

- can be publicly released as a part of the library
- may break at any moment

Experimental API

You can mark your new class or function as experimental

```
@Experimental  
annotation class ShinyNewAPI
```

```
@ShinyNewAPI  
class Foo {  
    ...  
}
```

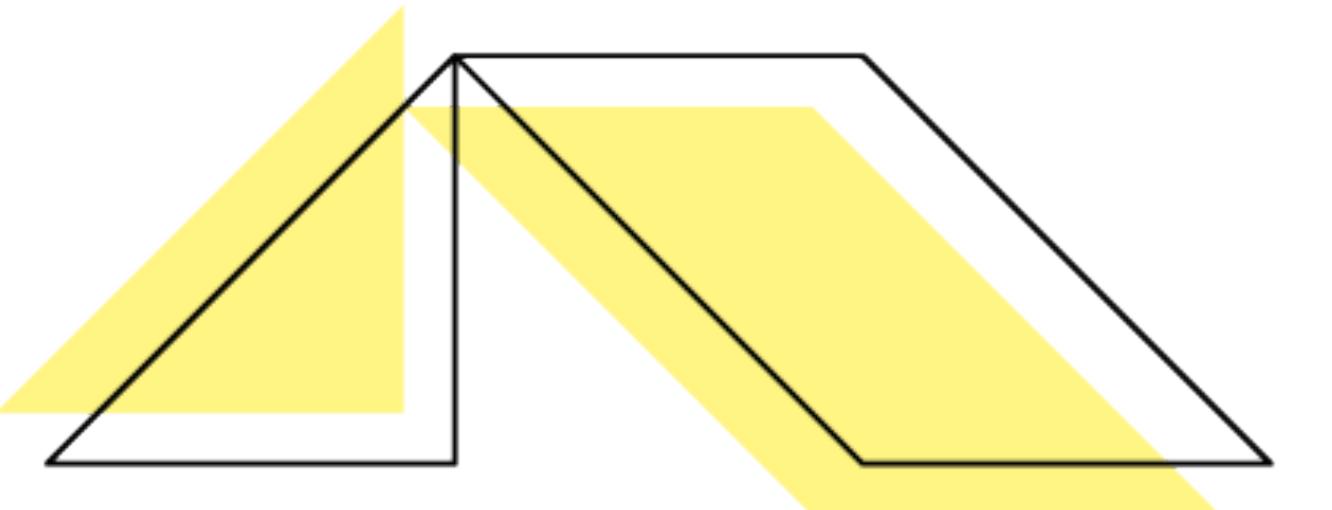
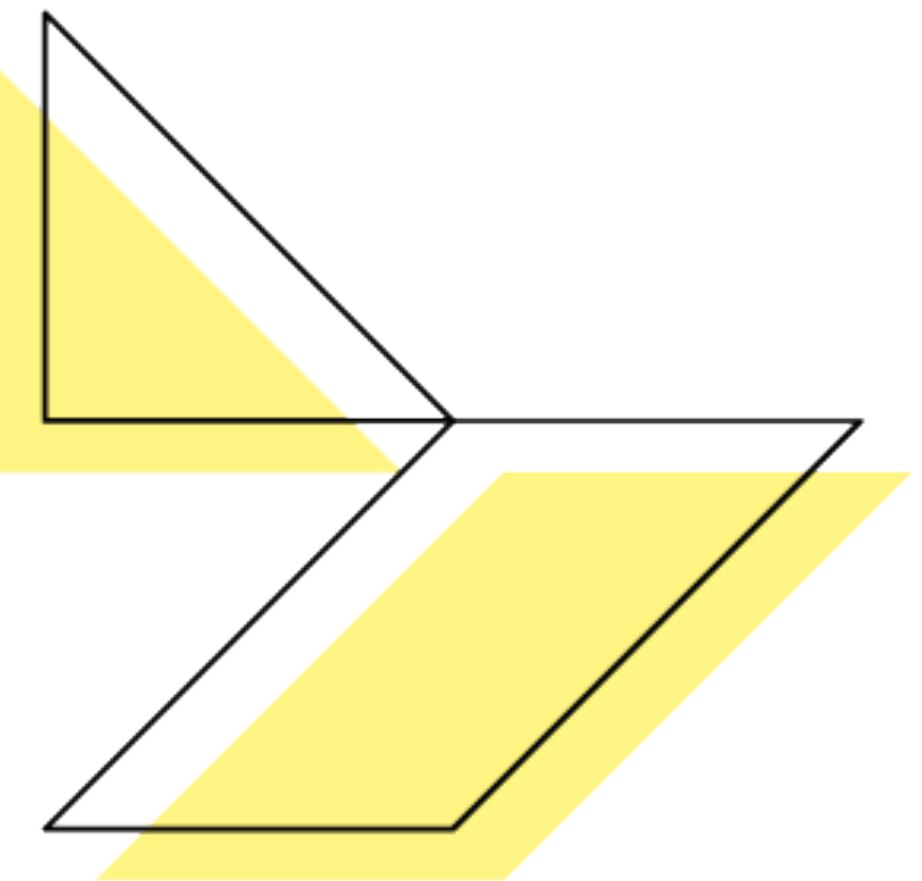
Using experimental API

```
@UseExperimental(ShinyNewAPI::class)
fun doSomethingImportant() {
    val foo = Foo()
    ...
}
```

Experimental: Summary

- feedback loop for new features and API

Inline classes



Inline classes

- can wrap values without additional overhead
- currently an experimental feature

Duration API

- KEEP: Duration and Time Measurement API
- status: experimental in Kotlin 1.3.50
- uses inline classes

Refining API: trying primitive args

```
fun greetAfterTimeout(millis: Long)
```

```
greetAfterTimeout(2) // meaning 2 seconds?
```

Refining API: trying primitive args

```
fun greetAfterTimeout(millis: Long)
```

```
greetAfterTimeout(2) // meaning 2 seconds?
```

Too easy to make a mistake

Refining API: trying overloads

```
fun greetAfterTimeoutMillis(millis: Long)
```

```
fun greetAfterTimeoutSeconds(seconds: Long)
```

```
greetAfterTimeoutSeconds(2)
```

Refining API: trying overloads

```
fun greetAfterTimeoutMillis(millis: Long)
```

```
fun greetAfterTimeoutSeconds(seconds: Long)
```

```
greetAfterTimeoutSeconds(2)
```

Too verbose

Refining API: trying class

```
class Duration(val millis: Long)
```

```
fun greetAfterTimeout(duration: Duration)
```

```
greetAfterTimeout(Duration(millis = 2000))
```

Refining API: trying class

```
class Duration(val millis: Long)
```

```
fun greetAfterTimeout(duration: Duration)
```

```
greetAfterTimeout(Duration(millis = 2000))
```

Extra object is allocated

Inline classes to the rescue

```
inline class Duration(val value: Long)
```

```
fun greetAfterTimeout(duration: Duration)
```

Under the hood:

```
fun greetAfterTimeout_(duration: Long)
```

Inline classes to the rescue

```
inline class Duration(val value: Long)
```

```
fun greetAfterTimeout(duration: Duration)
```

Under the hood:

```
fun greetAfterTimeout_(duration: Long)
```

No extra object is allocated!



Inline classes constraints

inline class can define only one val property

```
inline class Duration(val value: Long)
```

```
fun greetAfterTimeout(duration: Duration)
```

Creating Duration

```
val Int.seconds  
    get() = toDuration(DurationUnit.SECONDS)
```

```
fun greetAfterTimeout(duration: Duration)
```

```
greetAfterTimeout(2.seconds)
```

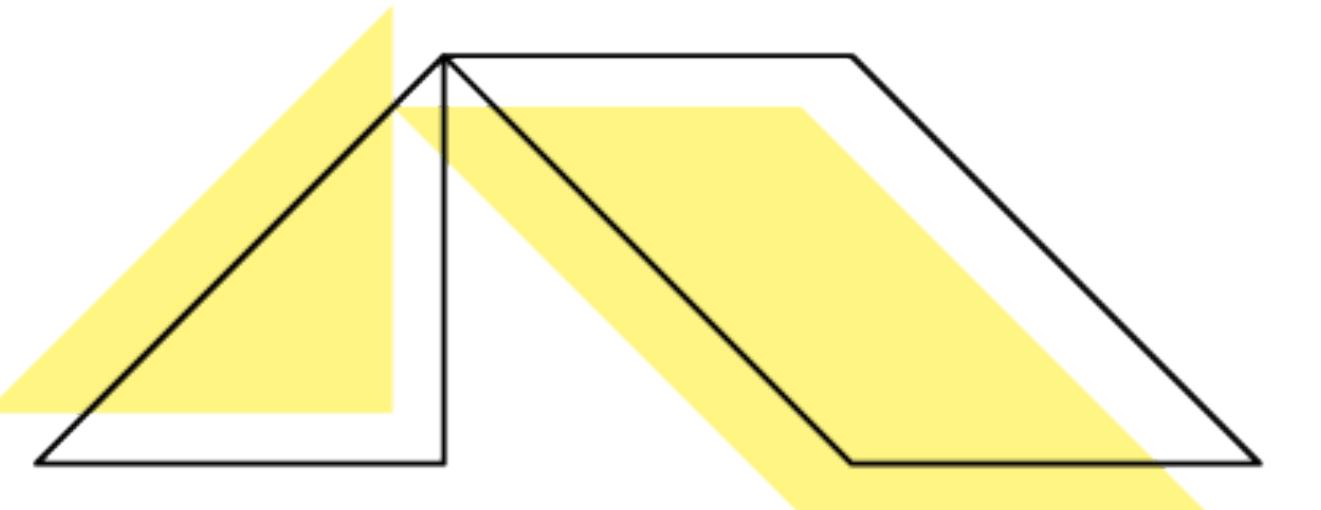
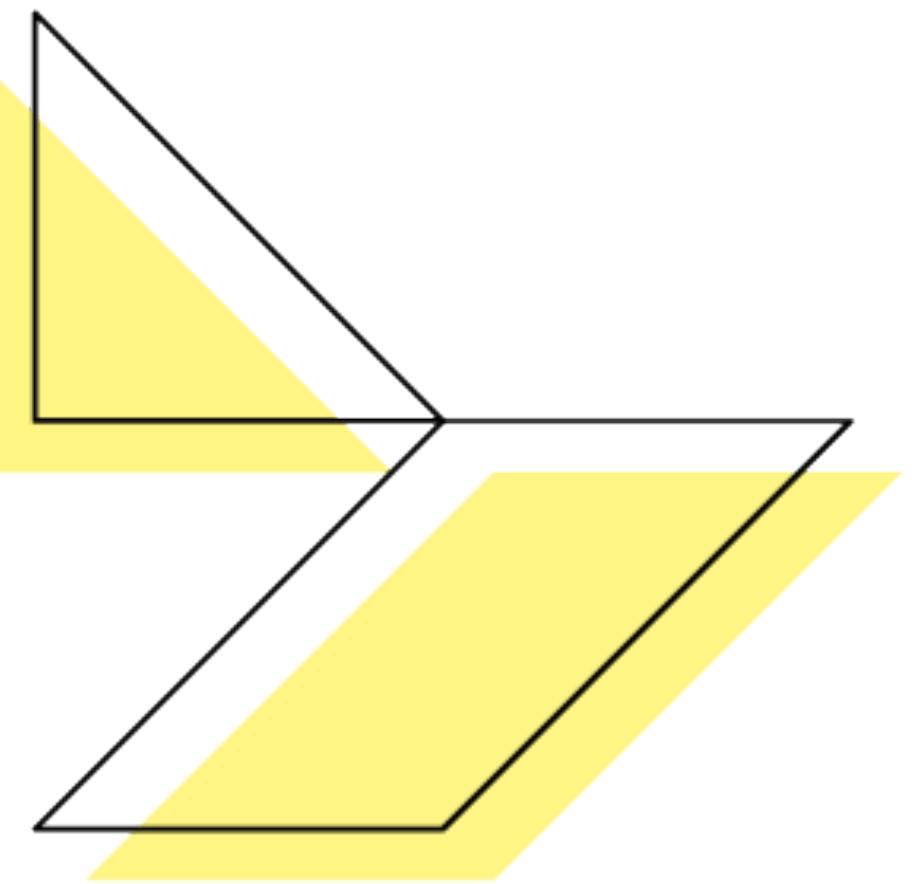
Explicit units in the code



Inline classes: summary

- KEEP: inline classes
- help to improve API and avoid extra allocations
- you can go and try it out

Contracts



Changes in standard library

```
fun String?.isNullOrEmpty(): Boolean {  
    return this == null || this.length == 0  
}
```

Changes in standard library

```
fun String?.isNullOrEmpty(): Boolean {  
    return this == null || this.length == 0  
}
```



```
fun String?.isNullOrEmpty(): Boolean {  
    contract {  
        returns(false) implies (this@isNullOrEmpty != null)  
    }  
  
    return this == null || this.length == 0  
}
```

Changes in standard library

```
fun String?.isNullOrEmpty(): Boolean {  
    return this == null || this.length == 0  
}
```



```
fun String?.isNullOrEmpty(): Boolean {
```

```
contract {  
    returns(false) implies (this@isNullOrEmpty != null)  
}
```

```
    return this == null || this.length == 0  
}
```



We know something about `isNullOrEmpty`,
which the compiler doesn't

```
val s: String? = ""
if (!s.isNullOrEmpty()) {
    s.first()
}
```

We know something about `isNullOrEmpty`,
which the compiler doesn't

```
val s: String? = ""
if (!s.isNullOrEmpty()) {
    s.first()
}
Compiler error:
Only safe (?.) or non-null asserted (!!.) calls
are allowed on a nullable receiver of type String?
```

We know something about `isNullOrEmpty`,
which the compiler doesn't

```
val s: String? = ""
if (!s.isNullOrEmpty()) {
    s.first()}
Compiler error:
Only safe (?.) or non-null asserted (!!.) calls
are allowed on a nullable receiver of type String?
```

```
if (s != null && s.isNotEmpty()) {
    s.first()
}
```

We know something about run,
that the compiler doesn't

```
val answer: Int  
run {  
    answer = 42  
}
```

println(answer)

We know something about run,
that the compiler doesn't

```
val answer: Int
run {
    answer = 42
} Compiler error:
Captured values initialization is forbidden
due to possible reassignment
println(answer)
```

Kotlin Contracts

...allow to share extra information
about code semantics with the compiler

Contract: if the function returns false,
the receiver is not-null

```
fun String?.isNullOrEmpty(): Boolean {  
    contract {  
        returns(false) implies (this@isNullOrEmpty != null)  
    }  
  
    return this == null || this.length == 0  
}
```

Contract: if the function returns false,
the receiver is not-null

```
fun String?.isNullOrEmpty(): Boolean {  
    contract {  
        returns(false) implies (this@isNullOrEmpty != null)  
    }  
  
    return this == null || this.length == 0  
}  
  
val s: String? = ""  
if (!s.isNullOrEmpty()) {  
    s.first()   
}
```

Contract: block lambda
will be always called once

```
inline fun <R> run(block: () -> R): R {  
    contract {  
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)  
    }  
    return block()  
}
```

Contract: block lambda
will be always called once

```
inline fun <R> run(block: () -> R): R {  
    contract {  
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)  
    }  
    return block()  
}
```

```
val answer: Int  
run {  
    answer = 42  
}
```



Why can't compiler just implicitly
infer such information?

Why can't compiler just implicitly infer such information?

Because then such implicitly inferred information:

- can be implicitly changed
- can accidentally break code depending on it

Why can't compiler just **implicitly** infer such information?

Because then such implicitly inferred information:

- can be **implicitly** changed
- can accidentally break code depending on it

Why can't compiler just **implicitly** infer such information?

Because then such implicitly inferred information:

- can be **implicitly** changed
- can accidentally break code depending on it

Contract = **explicit** statement about function behaviour

Kotlin Contract

extra information by developer
&
compiler uses this information for code analysis

experimental

Kotlin Contract

extra information by developer
&

compiler uses this information for code analysis

&

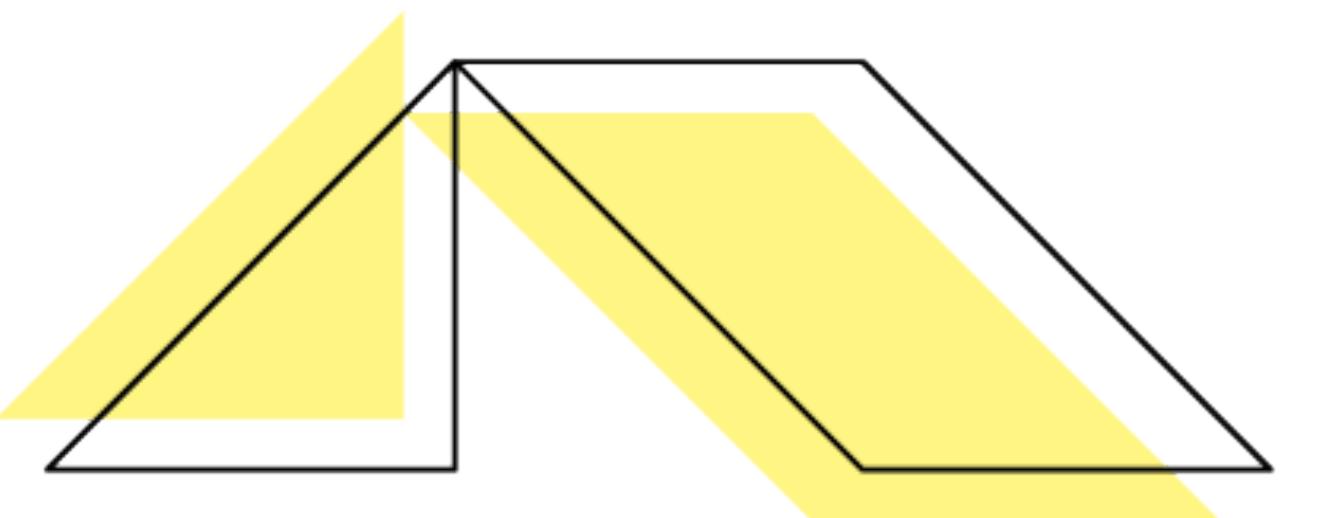
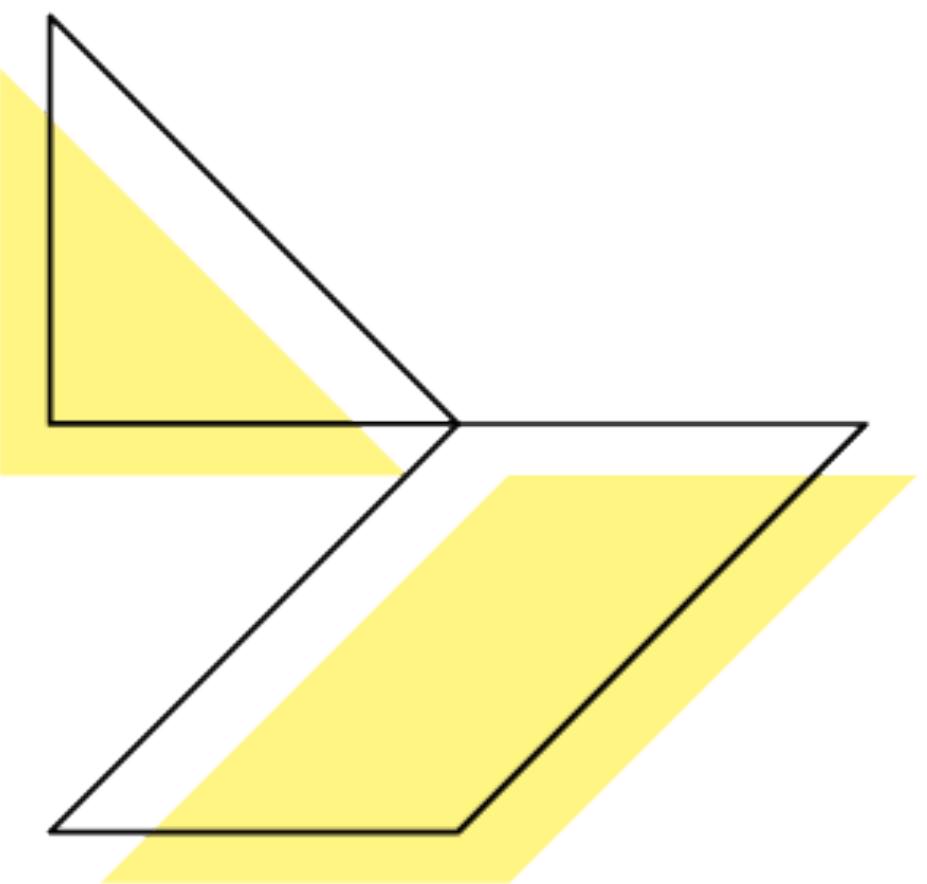
to be supported

checking that the information is correct
at compile time or runtime

Contracts: Summary

- handy functions (`run`, `isEmptyOrNull`) are even more useful
- contract DSL will change
- more use-cases in the future
- you can define contracts for your own functions

Channels



Channels

used for ~~synchronization~~
communication
between coroutines

“Share by communicating”

Shared
Mutable State



Share by
Communicating

Synchronization
Primitives

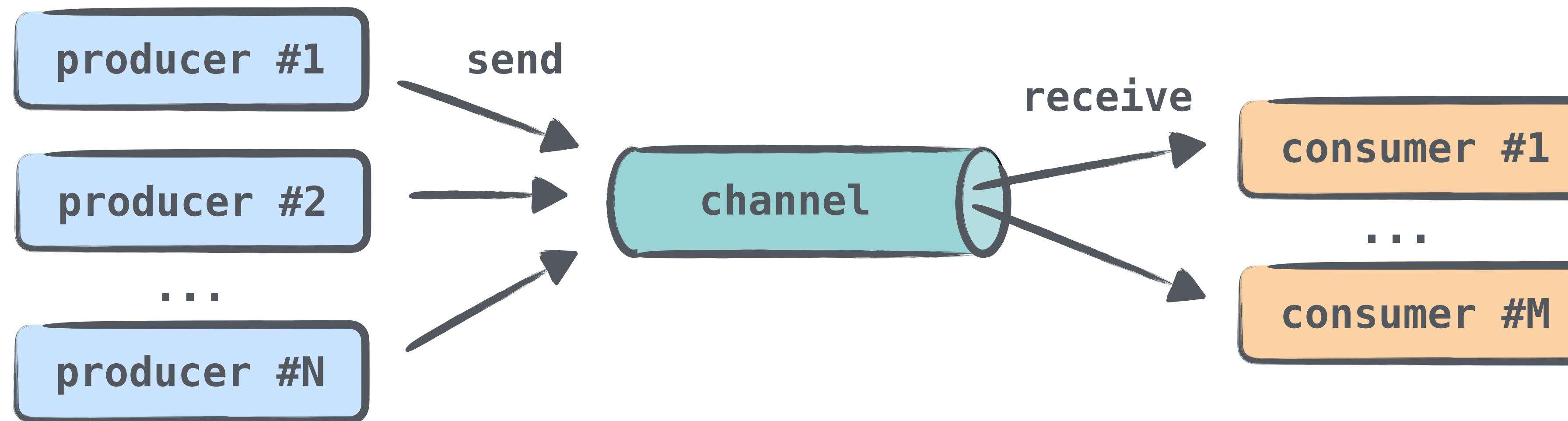


Communication
Primitives

Channel



Producer-consumer problem



Send & Receive “views” for the same channel

```
interface SendChannel<in E> {  
    suspend fun send(element: E)  
    fun close()  
}
```

```
interface ReceiveChannel<out E> {  
    suspend fun receive(): E  
}
```

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

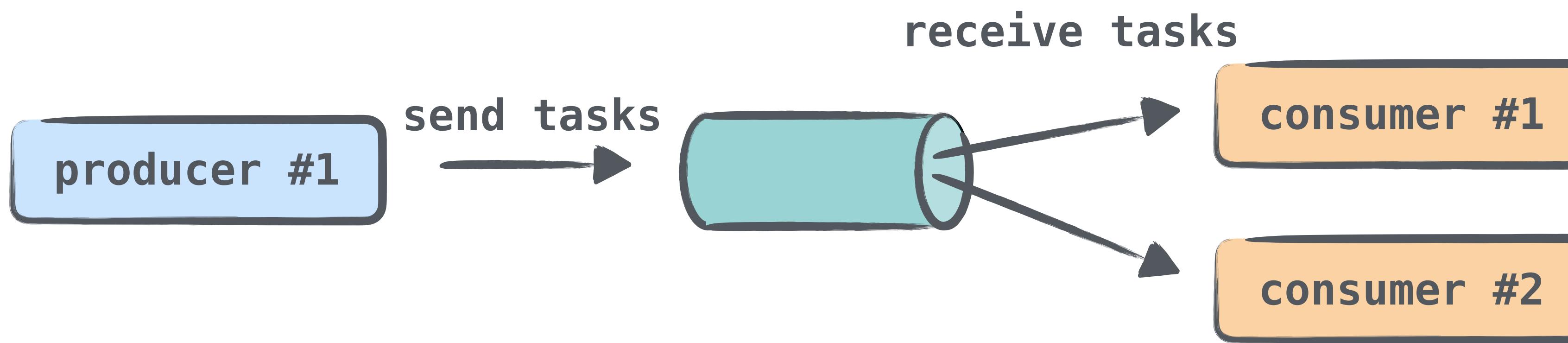
Send & Receive “views” for the same channel

```
interface SendChannel<in E> {  
    suspend fun send(element: E)  
    fun close()  
}
```

```
interface ReceiveChannel<out E> {  
    suspend fun receive(): E  
}
```

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

Producer-consumer problem



Producer-consumer solution: producer

```
val channel = Channel<Task>()
async {
    channel.send(Task("task1"))
    channel.send(Task("task2"))
    channel.close()
}
```

producer

Producer-consumer solution: consumers

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

...

```
async { worker(channel) }
```

consumer #1

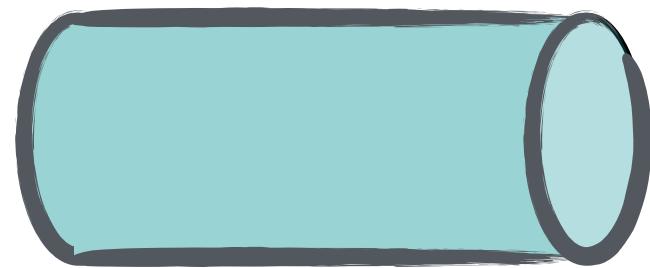
```
async { worker(channel) }
```

consumer #2

```
suspend fun worker(channel: Channel<Task>) {  
    ↳    val task = channel.receive()  
        processTask(task)  
}
```

Producer-consumer solution

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



Producer-consumer solution

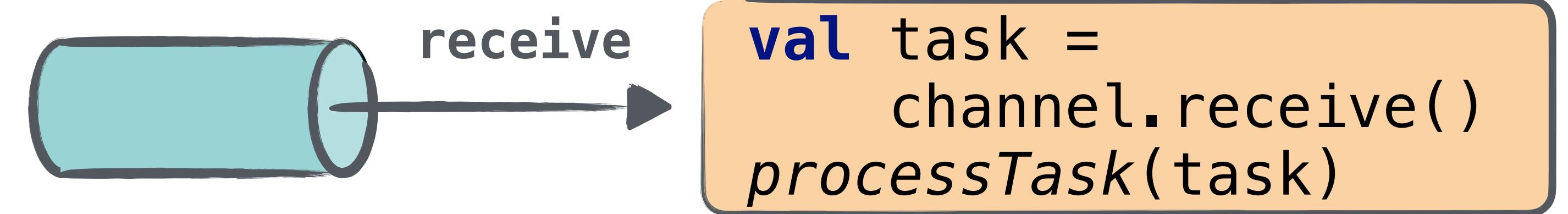
```
val channel = Channel<Task>()
```



```
val task =  
    channel.receive()  
processTask(task)
```

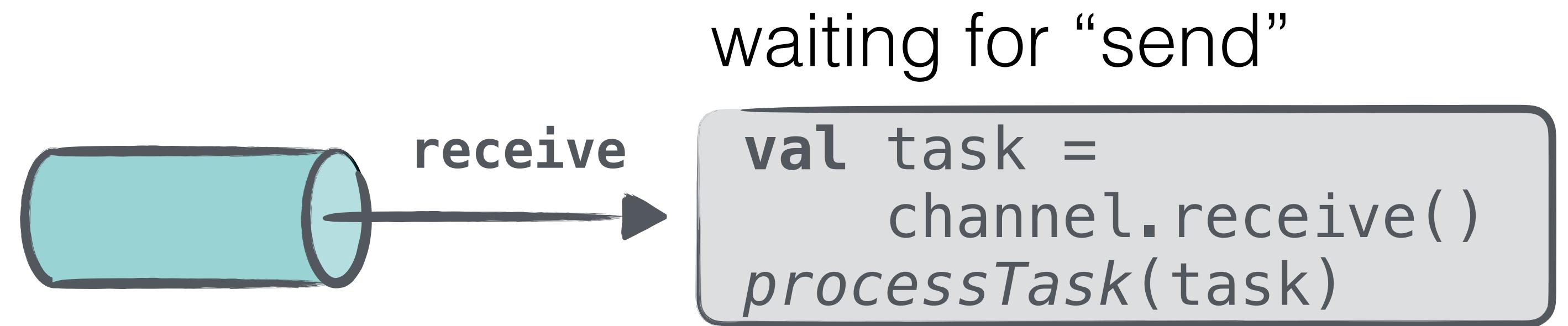
Producer-consumer solution

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



Producer-consumer solution

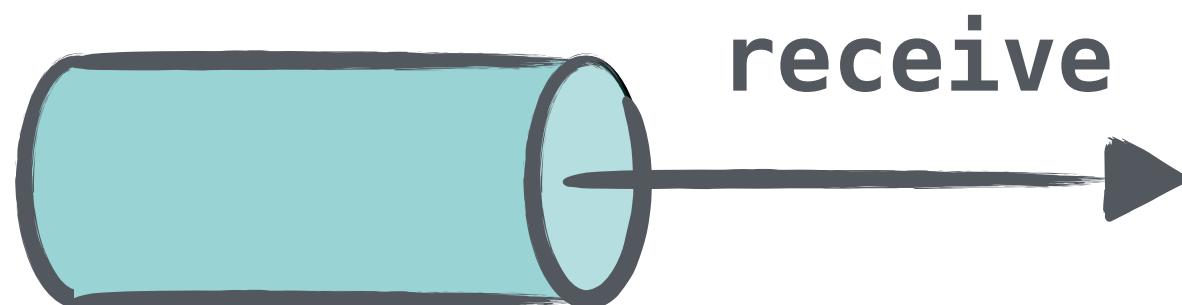
```
val channel = Channel<Task>()
```



Producer-consumer solution

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

```
channel.send(task1)  
channel.send(task2)  
channel.close()
```



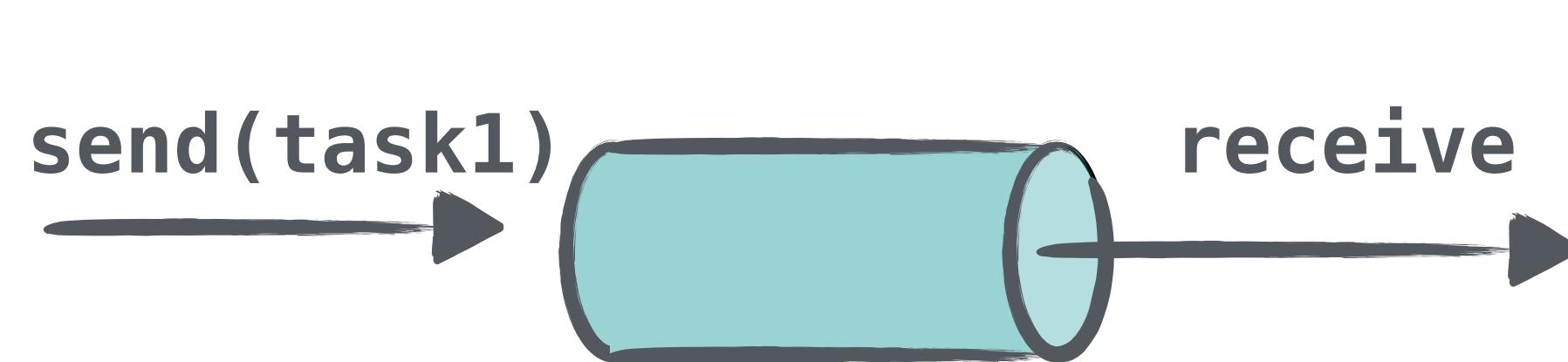
waiting for “send”

```
val task =  
    channel.receive()  
processTask(task)
```

Producer-consumer solution

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

```
channel.send(task1)  
channel.send(task2)  
channel.close()
```

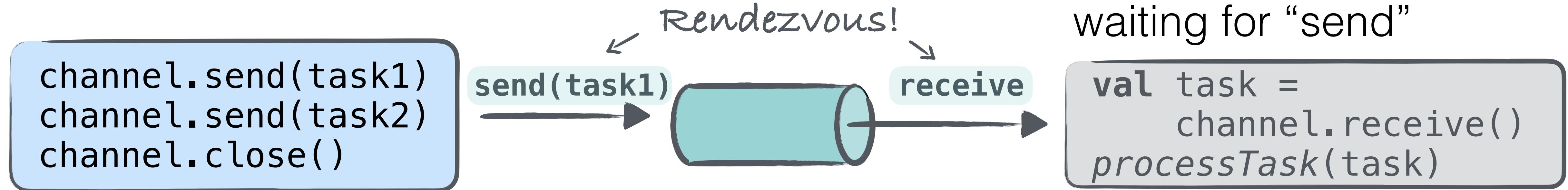


waiting for “send”

```
val task =  
    channel.receive()  
processTask(task)
```

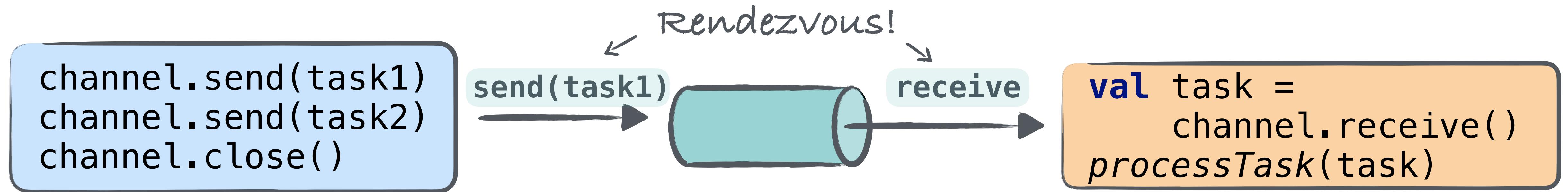
Producer-consumer solution

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



Producer-consumer solution

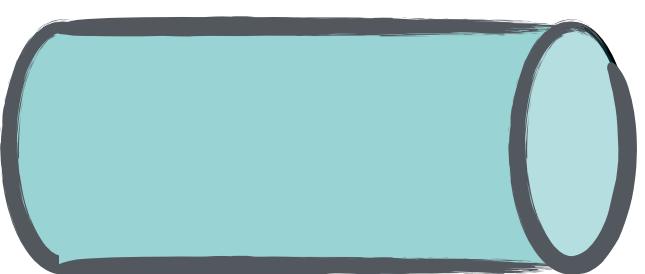
```
val channel = Channel<Task>()
```



Producer-consumer solution

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

```
channel.send(task1)  
channel.send(task2)  
channel.close()
```



processing task1 

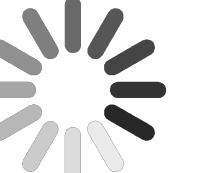
```
val task =  
    channel.receive()  
processTask(task)
```

Producer-consumer solution

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

```
send(task2)
```

```
channel.send(task1)  
channel.send(task2)  
channel.close()
```

processing task1 

```
val task =  
    channel.receive()  
processTask(task)
```

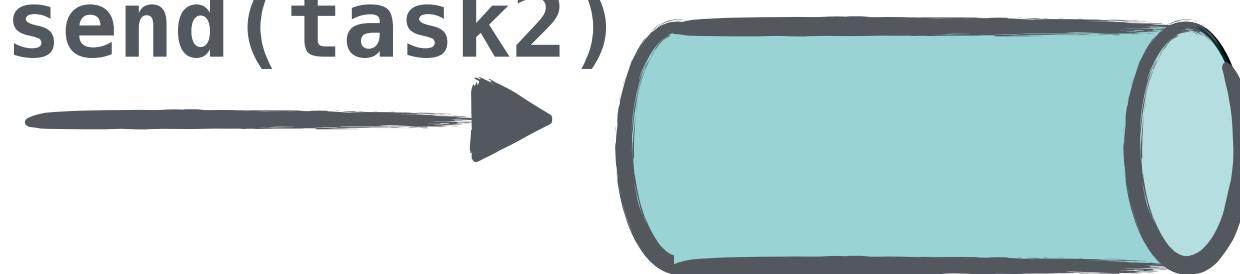
Producer-consumer solution

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

waiting for “receive”

```
channel.send(task1)  
channel.send(task2)  
channel.close()
```

send(task2)

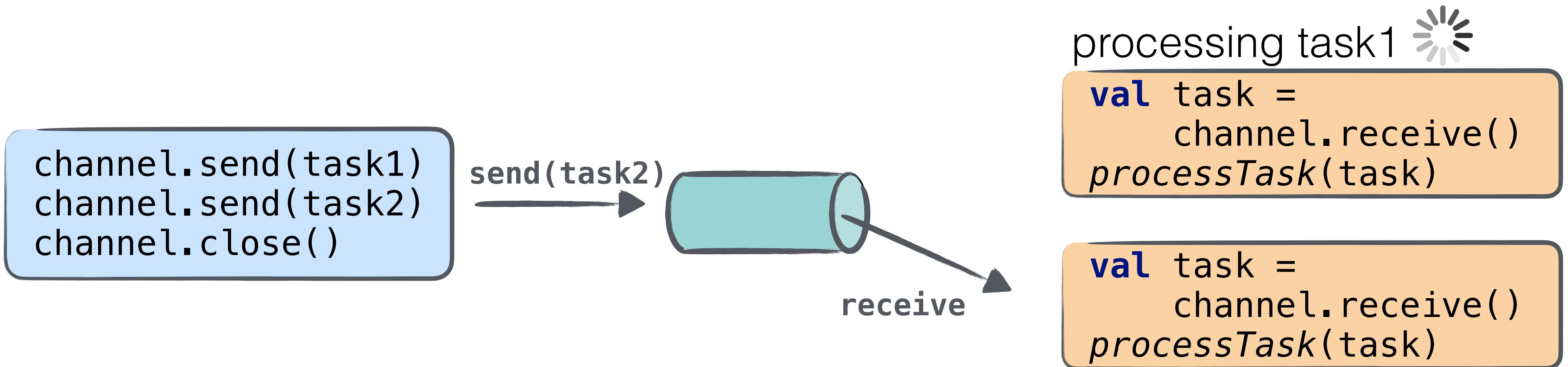


processing task1

```
val task =  
    channel.receive()  
processTask(task)
```

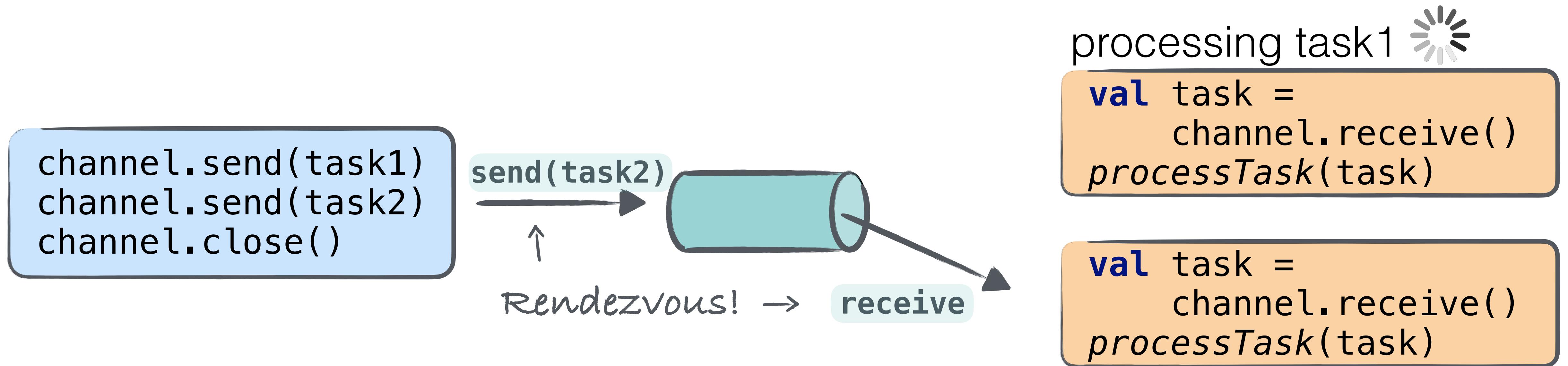
Producer-consumer solution

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



Producer-consumer solution

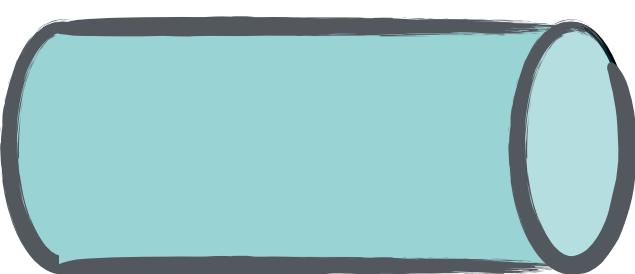
```
val channel = Channel<Task>()
```

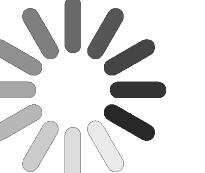


Producer-consumer solution

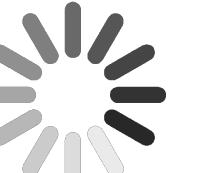
```
val channel = Channel<Task>()
```

```
channel.send(task1)  
channel.send(task2)  
channel.close()
```



processing task1 

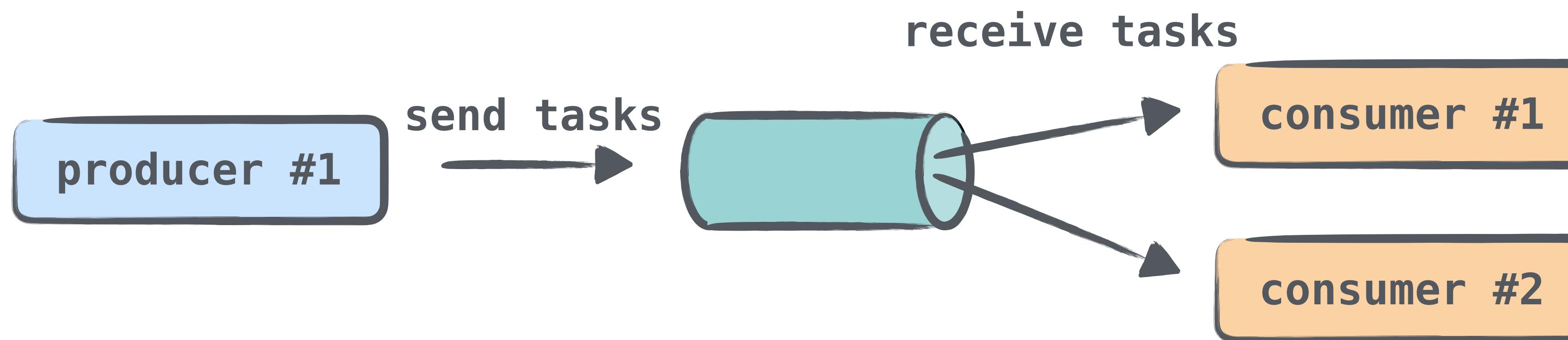
```
val task =  
    channel.receive()  
processTask(task)
```

processing task2 

```
val task =  
    channel.receive()  
processTask(task)
```

Producer-consumer solution: many tasks

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



Producer-consumer solution: many tasks

```
val channel = Channel<Task>()
async {
    for (i in 1..N) {
        channel.send(Task("task$i"))
    }
    channel.close()
}
```

producer

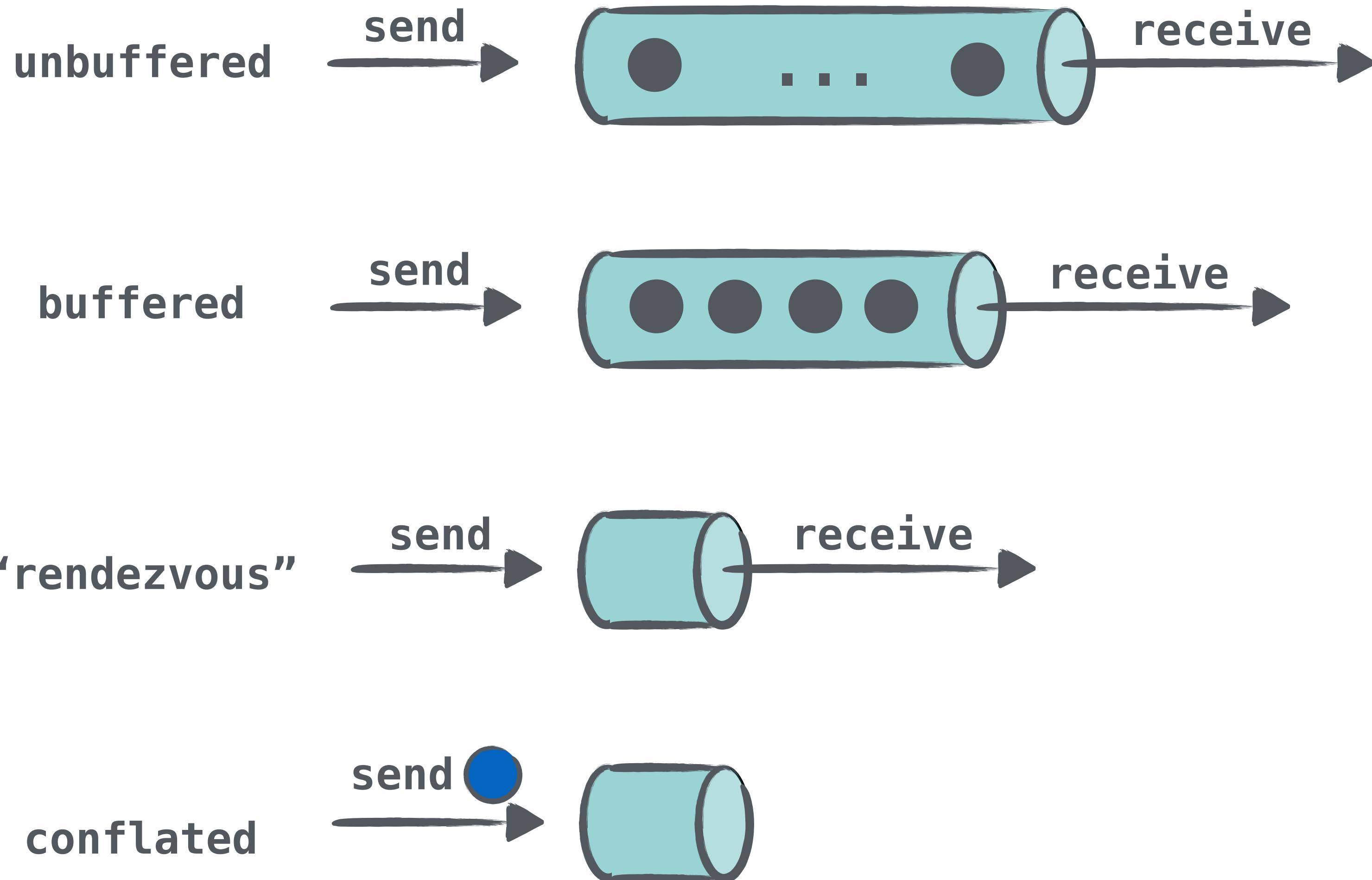
Producer-consumer solution: many tasks

```
val channel = Channel<Task>()  
...  
async { worker(channel) }           consumer #1  
async { worker(channel) }           consumer #2
```

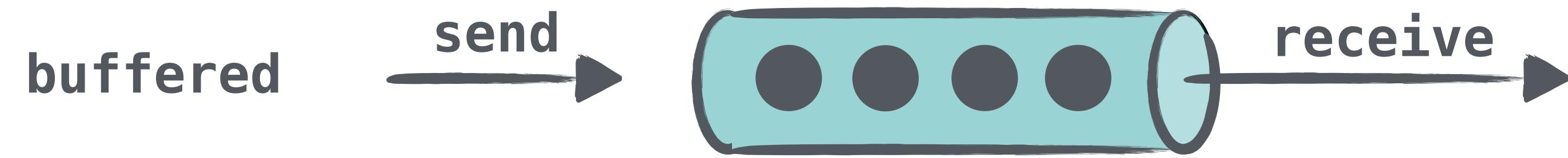
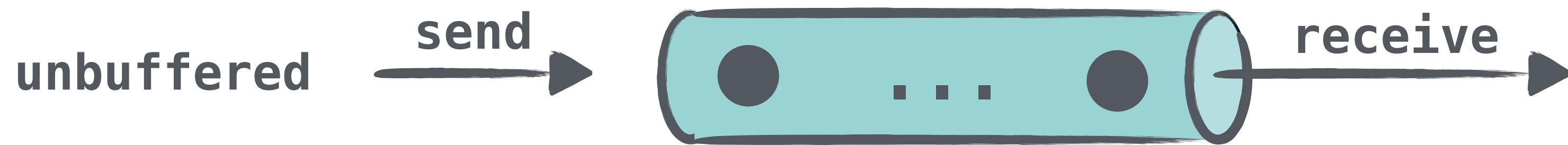
```
suspend fun worker(channel: Channel<Task>) {  
    →    for (task in channel) {  
        processTask(task)  
    }  
}
```

calls receive while iterating

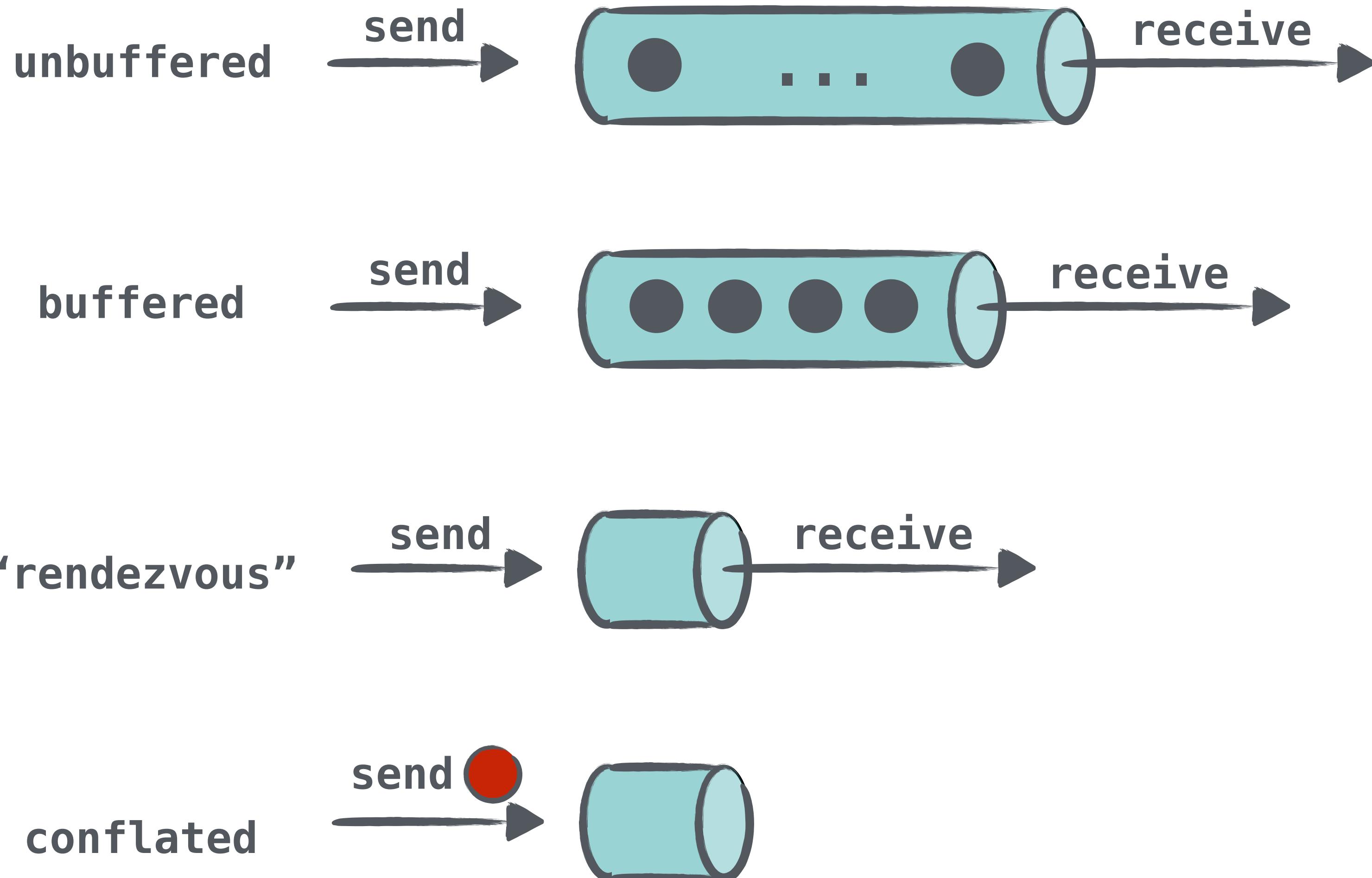
Types of Channels



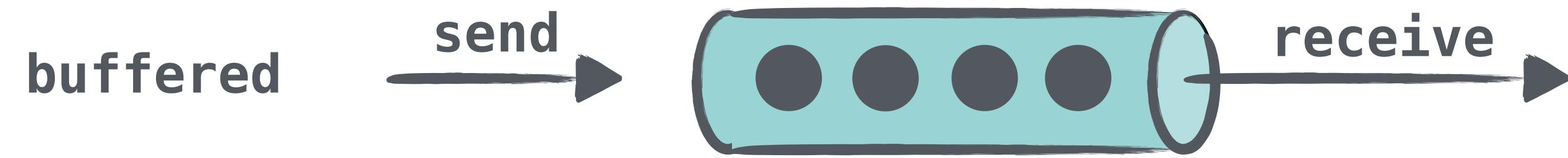
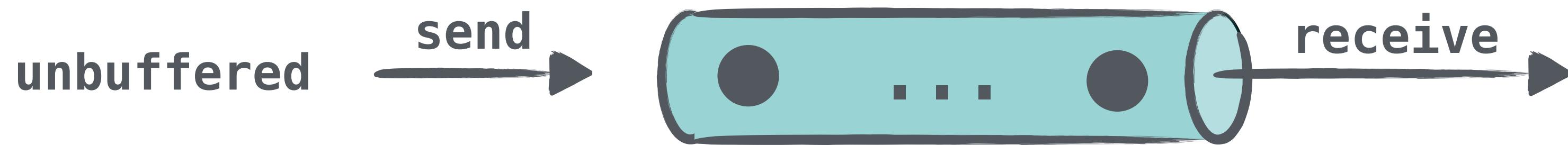
Types of Channels



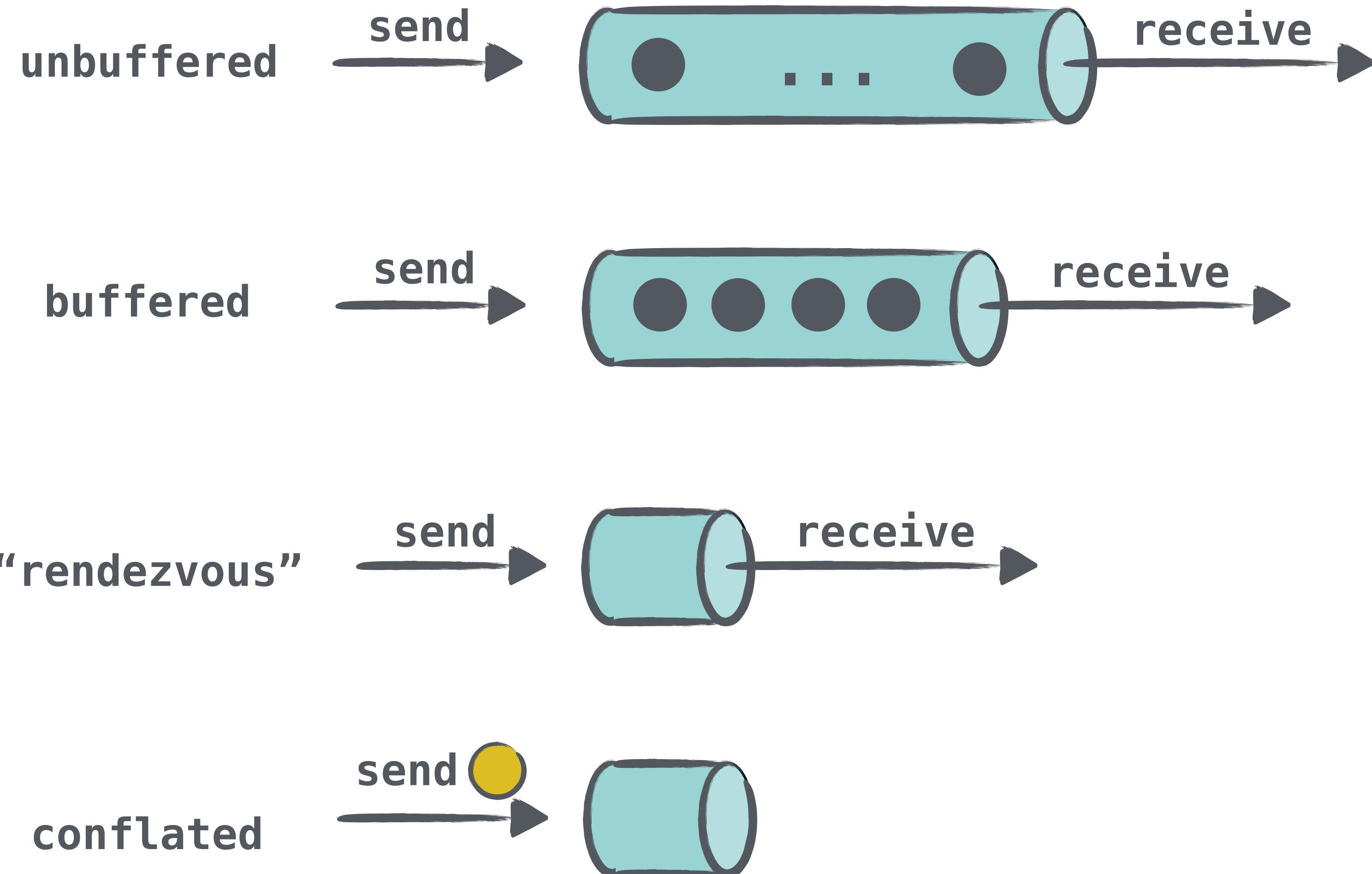
Types of Channels



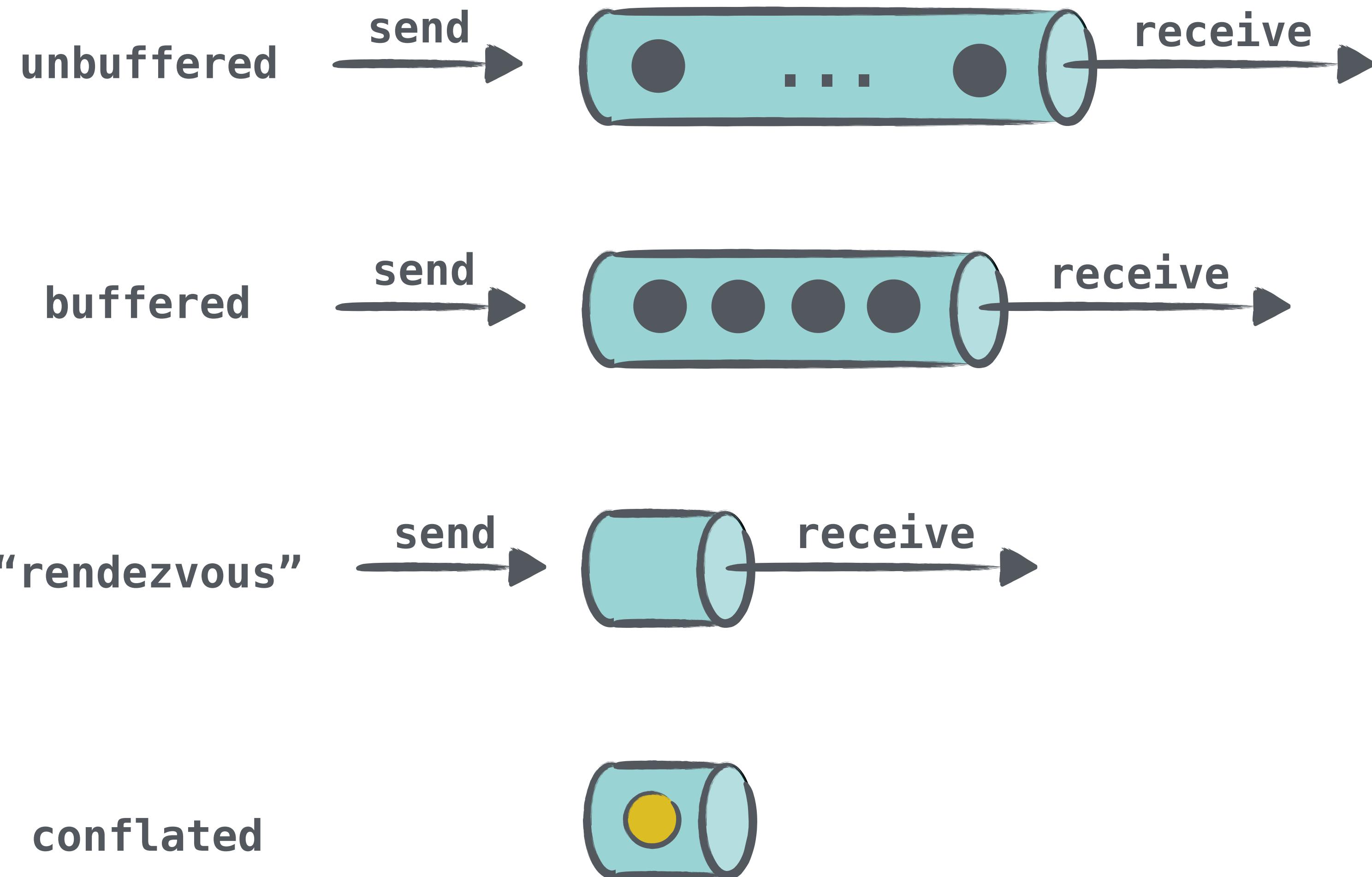
Types of Channels



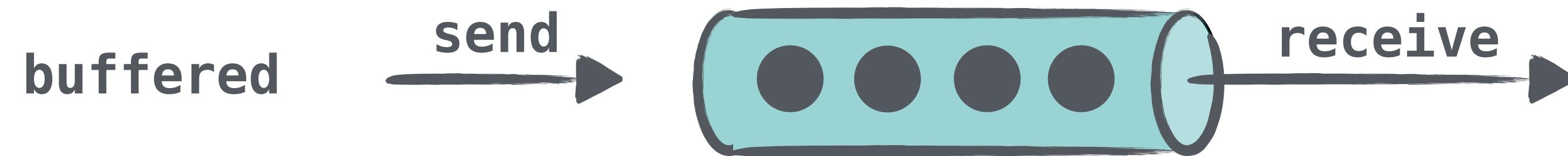
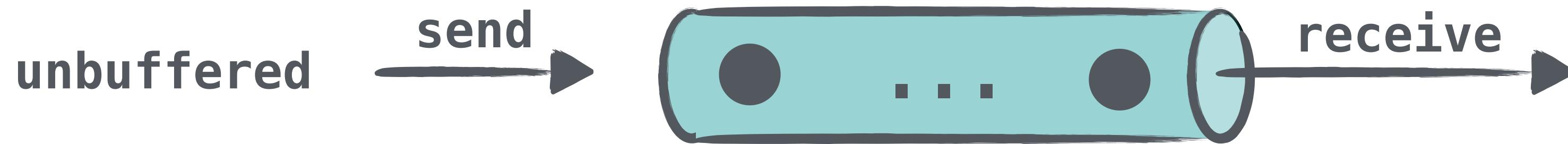
Types of Channels



Types of Channels



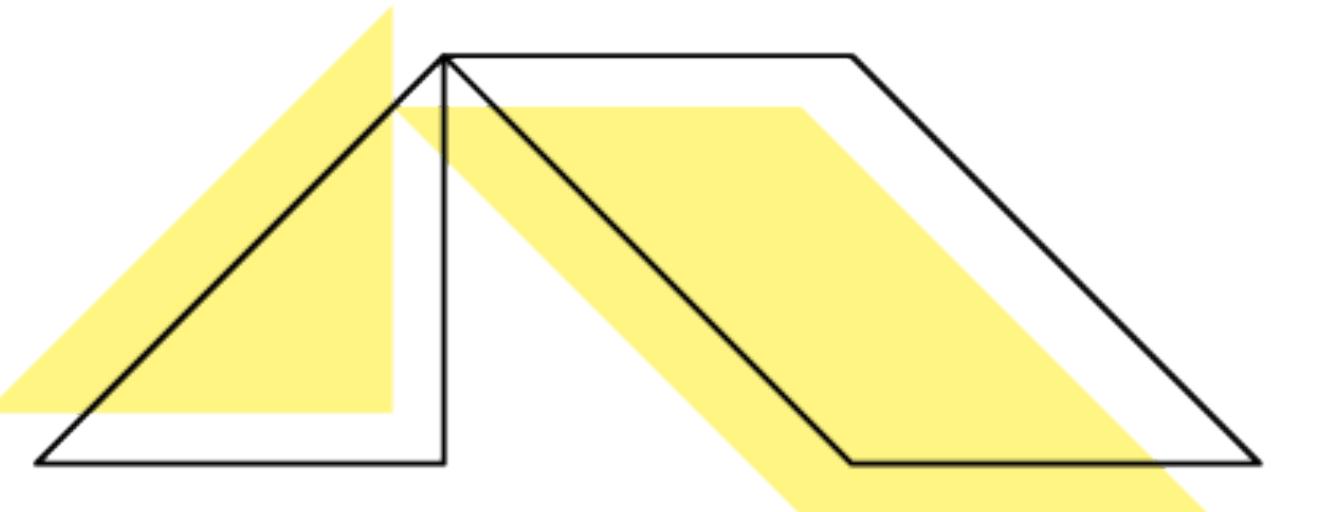
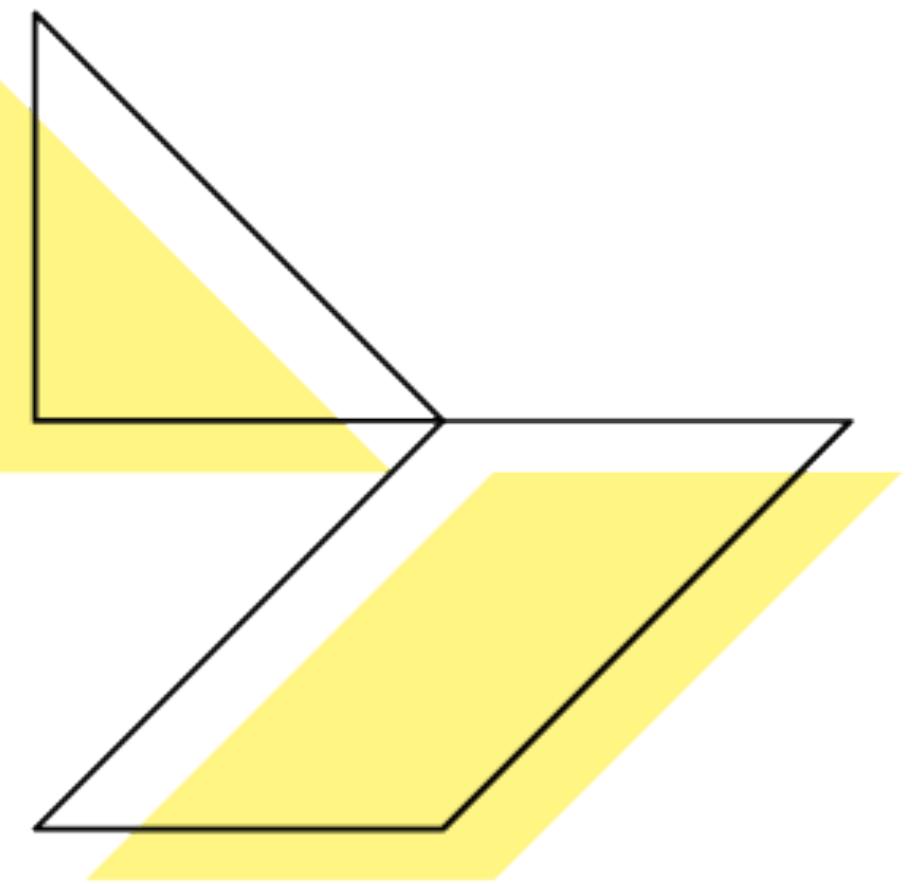
Types of Channels



Channels: summary

- used for communication between different coroutines
- stable part of the `kotlinx.coroutines` library

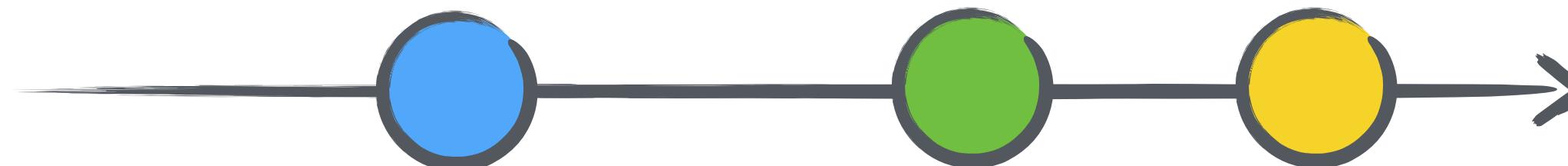
Flows



Flow

- suspend-based reactive stream

```
flow { emit(value) }  
    .map { transform(it) }  
    .filter { condition(it) }  
    .catch { exception → log(exception) }  
    .collect { process(it) }
```



Integration with RxJava

Use extension functions:

- `flow.asPublisher()`
- `publisher.asFlow()`

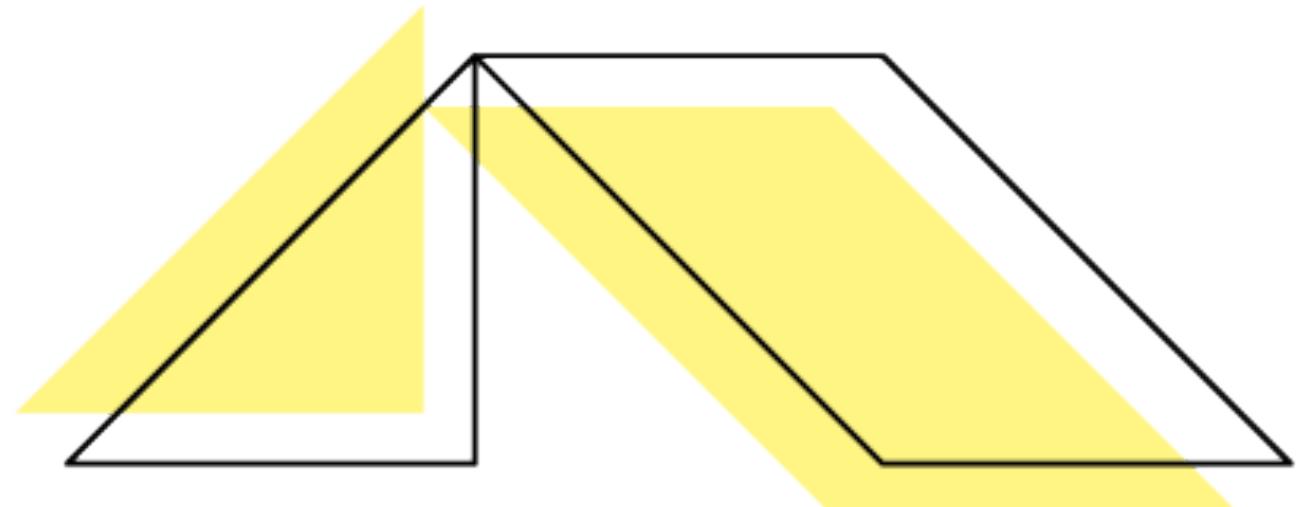
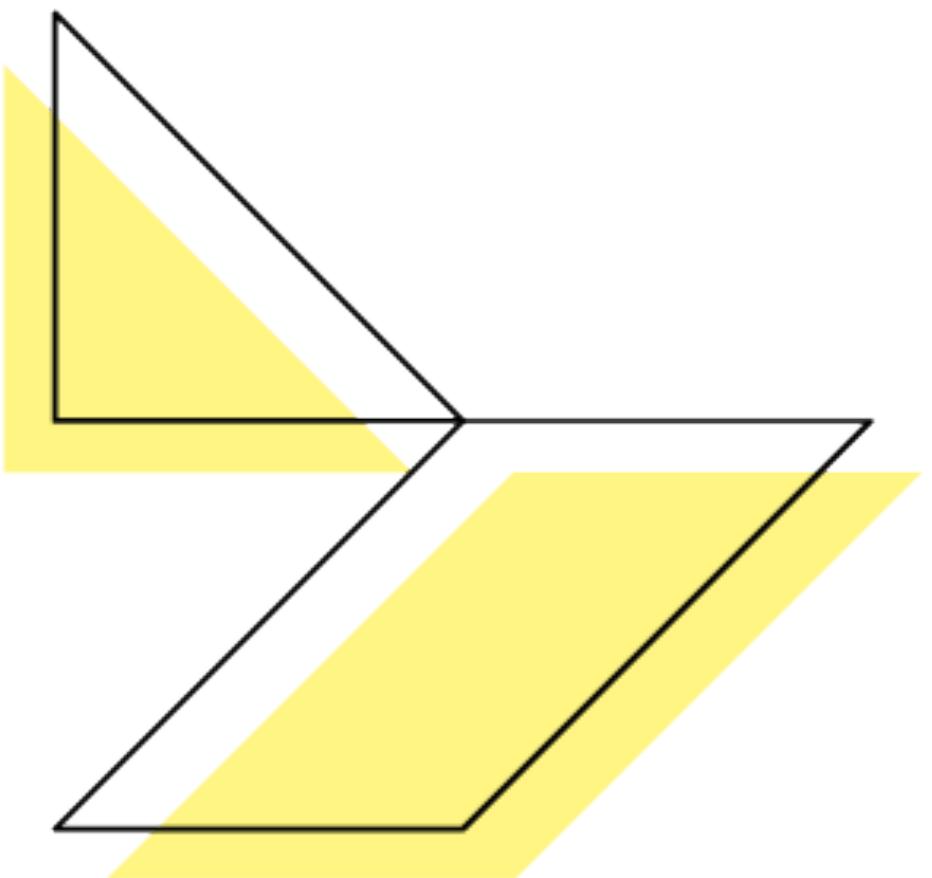
Backpressure

- Backpressure happens automatically thanks to suspension mechanism

Flows: summary

- bring reactive streams to coroutines library
- currently in an experimental state
- will get stable soon

Multi-platform Projects



expect / actual in standard library

Before:

```
fun Char.isUpperCase(): Boolean =  
    java.lang.Character.isUpperCase(this)
```

expect / actual in standard library

Before:

```
fun Char.isUpperCase(): Boolean =  
    java.lang.Character.isUpperCase(this)
```



Now:

```
expect fun Char.isUpperCase(): Boolean  
public actual fun Char.isUpperCase(): Boolean =  
    java.lang.Character.isUpperCase(this)
```

expect / actual in standard library

Before:

```
fun Char.isUpperCase(): Boolean =  
    java.lang.Character.isUpperCase(this)
```



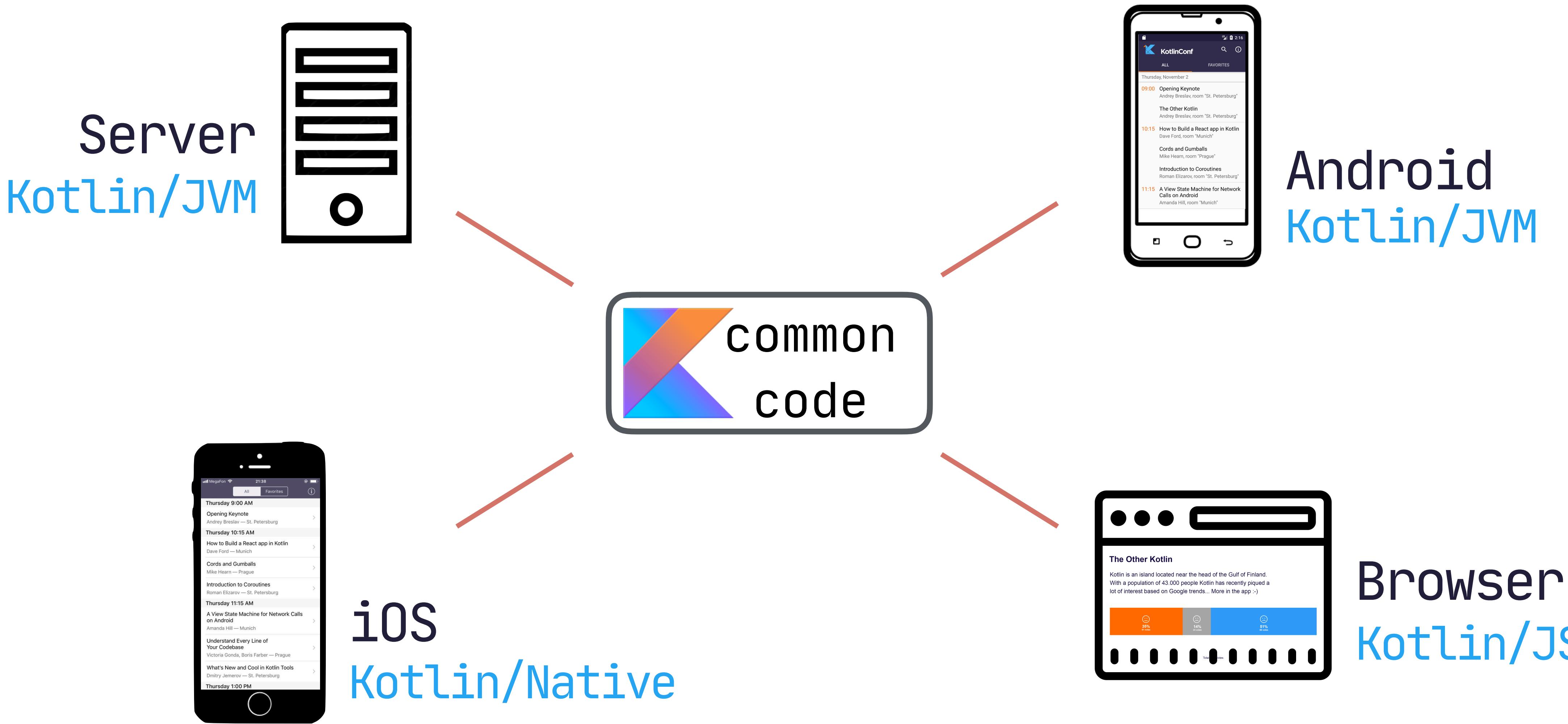
Now:

```
expect fun Char.isUpperCase(): Boolean
```

```
public actual fun Char.isUpperCase(): Boolean =  
    java.lang.Character.isUpperCase(this)
```



Multi-platform projects



Sharing common code

- Sharing business logic
- Keeping UI platform-dependent
- The shared part might vary

Common code

- you define **expect** declarations in the common code and use them
- you provide different **actual** implementations for different platforms

Time measurement example

Expected platform-specific API:

```
expect fun measureTime(action: () → Unit): Duration
```

Expected API can be used in the common code:

```
measureTime {  
    // operation  
}
```

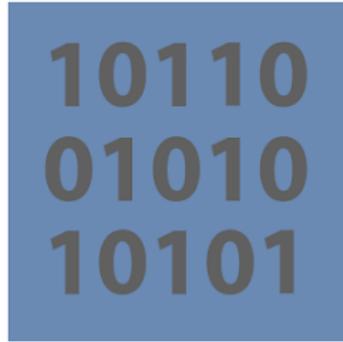
Platform-specific Implementations

```
expect fun measureTime(action: () → Unit): Duration
```



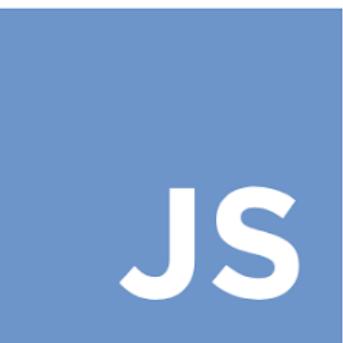
Kotlin/JVM

```
actual fun measureTime(action: () → Unit): Duration {  
    // implementation using System.nanoTime()  
}
```



Kotlin/Native

```
actual fun measureTime(action: () → Unit): Duration {  
    // implementation using std::chrono::high_resolution_clock  
}
```



Kotlin/JS

```
actual fun measureTime(action: () → Unit): Duration {  
    // implementation using window.performance.now()  
}
```

Common code

- can use the standard library
- can define `expect` declarations and use them
- can use other multi-platform libraries

Multi-platform libraries

- Standard library
- Ktor HTTP client
- kotlinx.serialization
- kotlinx.coroutines
- ... and more

Many apps already in production



ANA REDMOND

Founder & CTO, infinut, <http://infinut.com>

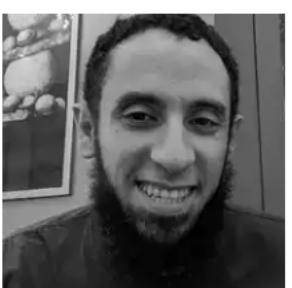
Going Native: How I used Kotlin Native to Port 6 years of Android Game Code to iOS in 6 months



ALEC STRONG

Android on @CashApp

Shipping a Mobile Multiplatform Project on iOS & Android



AHMED EL-HELW

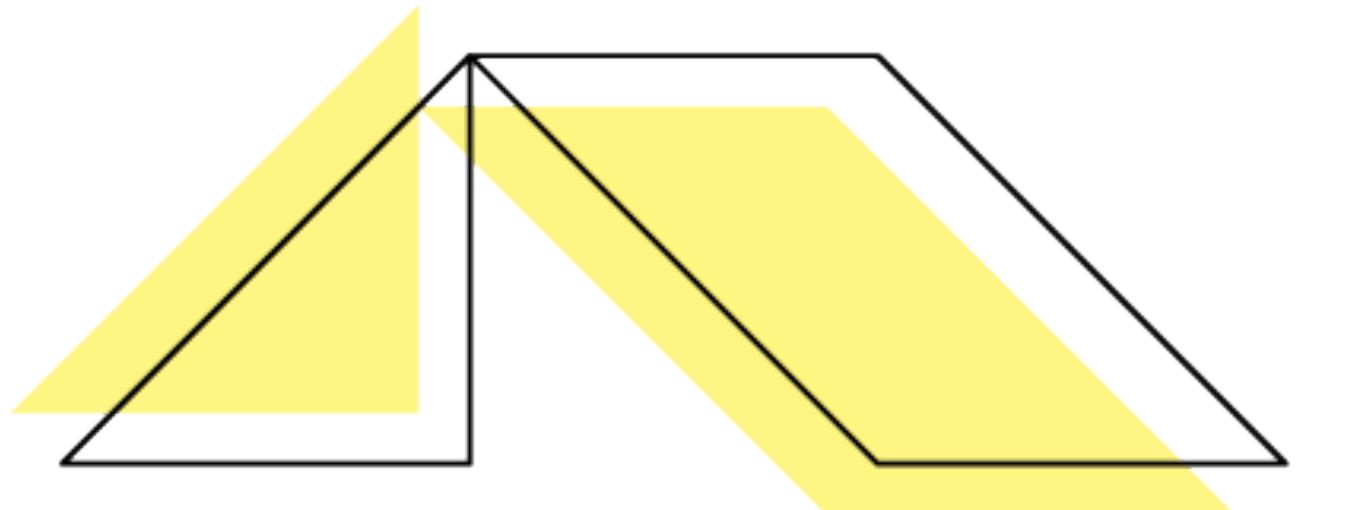
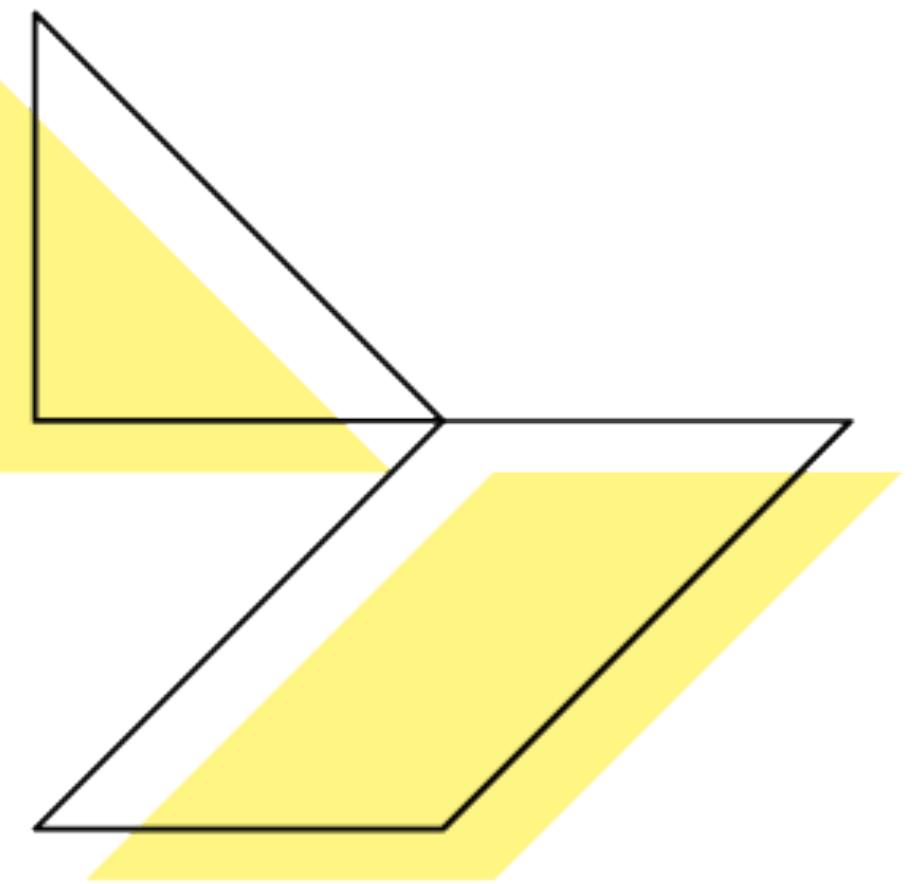
Android Lead at Careem

Your Multiplatform Kaptain has Arrived: iOS release is powered by Kotlin Multiplatform

Multi-platform projects: summary

- a modern approach to multi-platform development
- you can easily tune what parts you want to be shared
- you can go and try it out

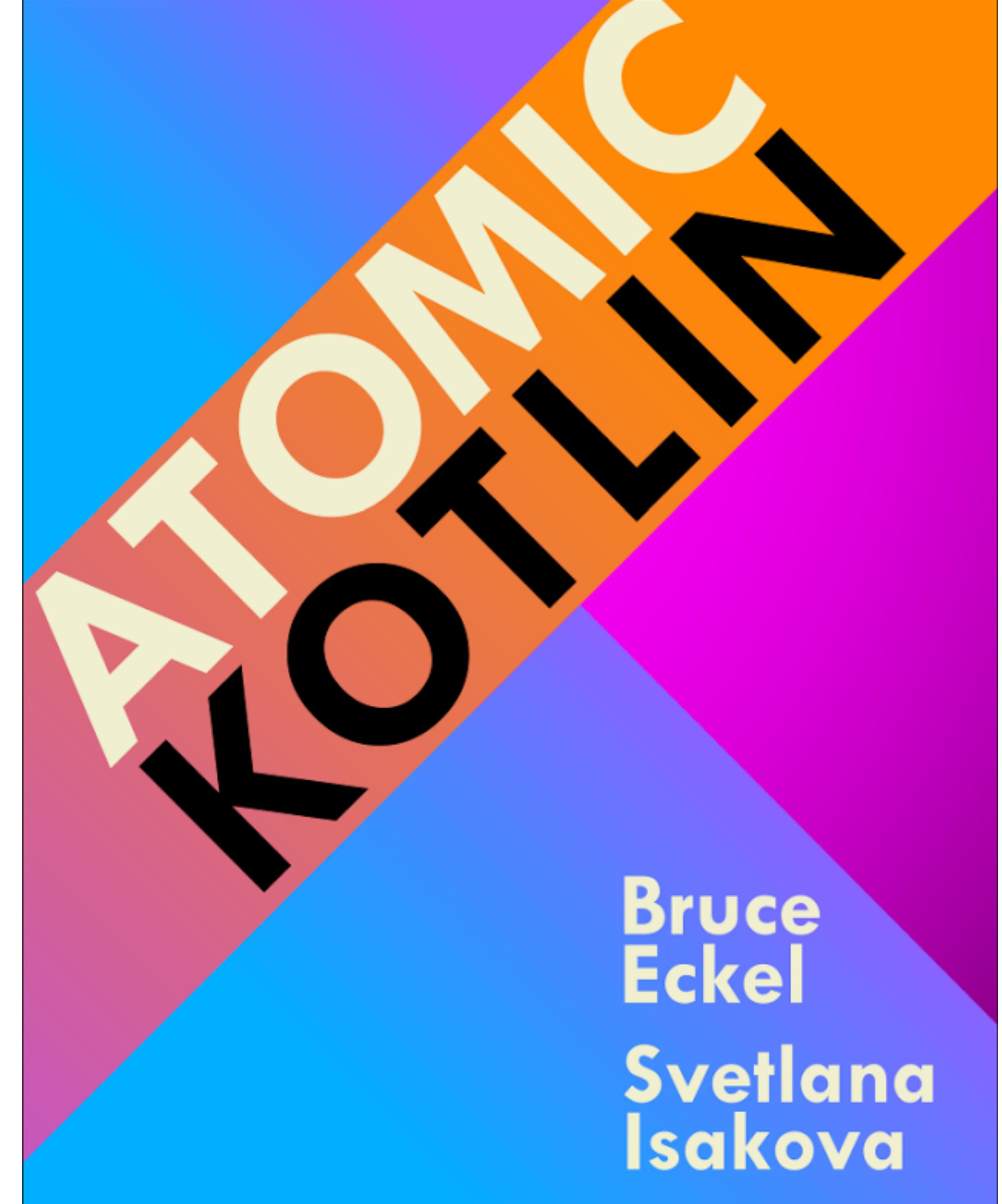
More about Kotlin



Kotlin IN ACTION

Dmitry Jemerov
Svetlana Isakova

FOREWORD BY Andrey Breslav



Bruce
Eckel
Svetlana
Isakova

Kotlin course at Coursera

The screenshot shows the Coursera website interface for the "Kotlin for Java Developers" course.

Header: The top navigation bar includes the Coursera logo, a search bar with placeholder text "What do you want to learn?", a search icon, and a user profile icon labeled "Sveta".

Course Information: The main title "Kotlin for Java Developers" is displayed prominently, along with the provider "by JetBrains".

Week Overview: On the left sidebar, there is a list of weeks: Week 1, Week 2, Week 3, Week 4, Week 5, Grades, Discussion Forums, Messages, and Course Info. Week 1 is currently selected and expanded.

Week 1 Content: The expanded Week 1 section contains a welcome message from the instructor: "Welcome to Kotlin for Java Developers! You're joining thousands of learners currently enrolled in the course. I'm excited to have you in the class." Below the message is a "More" link.

Timeline: A horizontal timeline at the bottom of the page shows the progression through the four weeks. The "START" point is marked with a green dot. The first week, "WEEK 1", is shown with an open dot, indicating it is currently active. The subsequent weeks, "WEEK 2", "WEEK 3", and "WEEK 4", are marked with closed dots and a lock icon, indicating they are locked or completed.

Week 1 Video: A video player for "WEEK 1" is visible, titled "Video: Nullable types". It includes a "Start" button, a duration indicator of "10 min", and a note: "You're doing great! Pick up where you left off and you'll be done in no time."

Hands-on lab “Intro to coroutines & channels”

<http://kotl.in/hands-on>

The screenshot shows a web browser window with the title 'Welcome to Kotlin hands-on'. The URL in the address bar is https://play.kotlinlang.org/hands-on/Introduction%20to%20Coroutines%20and%20Channels/01_Introduction. The page is titled 'Introduction' and features the 'Kotlin' logo. A navigation bar at the top includes 'Playground', 'Hands-on' (which is highlighted in blue), 'Examples', and 'Koans'. On the left, a sidebar lists eight steps: 1. Introduction (highlighted with a blue circle), 2. Blocking request, 3. Using callbacks, 4. Using suspend functions, 5. Concurrency, 6. Structured concurrency, 7. Showing progress, and 8. Channels. The main content area contains the following text:

Introduction

In this hands-on tutorial, you'll get familiar with the concept of coroutines. Coroutines help you to gain all the benefits of asynchronous and non-blocking behavior without lack of readability. You'll see how to use coroutines to perform network requests without blocking the underlying thread and without using callbacks.

You'll learn:

- why and how to use suspend functions to perform networks requests
- how to send requests concurrently using coroutines
- how to share information between different coroutines using channels

We'll also discuss how coroutines are different from other existing solutions.

You're expected to be familiar with the basic Kotlin syntax, but the prior knowledge of coroutines is not required.

A 'Next' button is located in the bottom right corner of the main content area.



Have a nice Kotlin!