PicPay

Jeziel Lago

Android Developer | PicPay

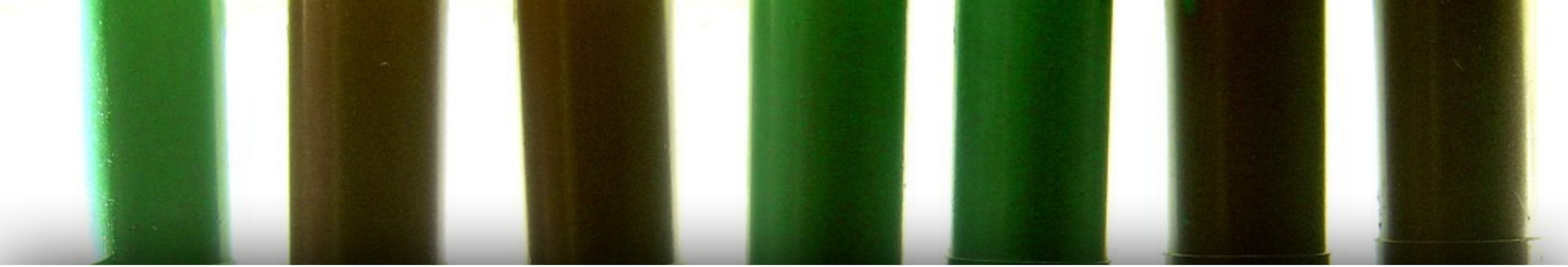
/jeziellago

@jeziellago

InsertKoinIO/**koin**

coil-kt/**coil**

PicPay



Kotlin Channels & Flow

Hello Streams!

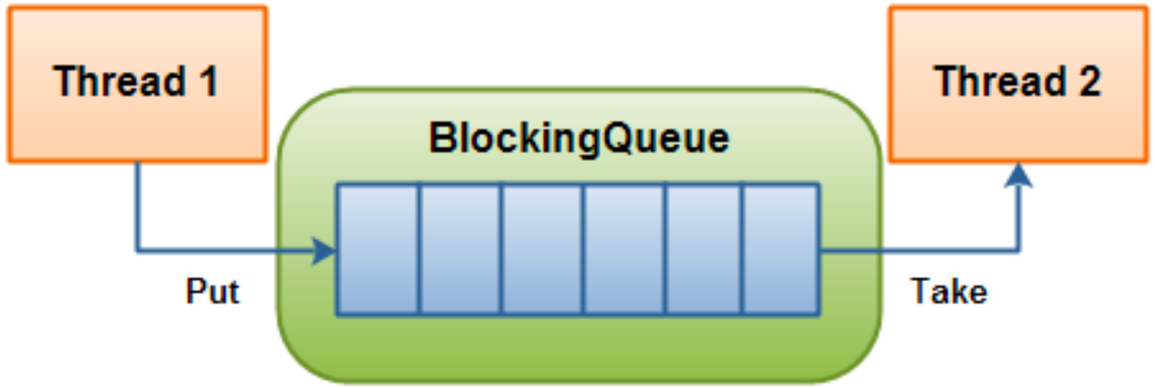
Antes de falar de Kotlin Channels,

vamos dar uma olhada em Java...🤔


A TRILHA É
ANDROID E KOTLIN
imgflip.com
MAS A SOLUÇÃO
VEIO DO JAVA

BlockingQueue

`java.util.concurrent.BlockingQueue`

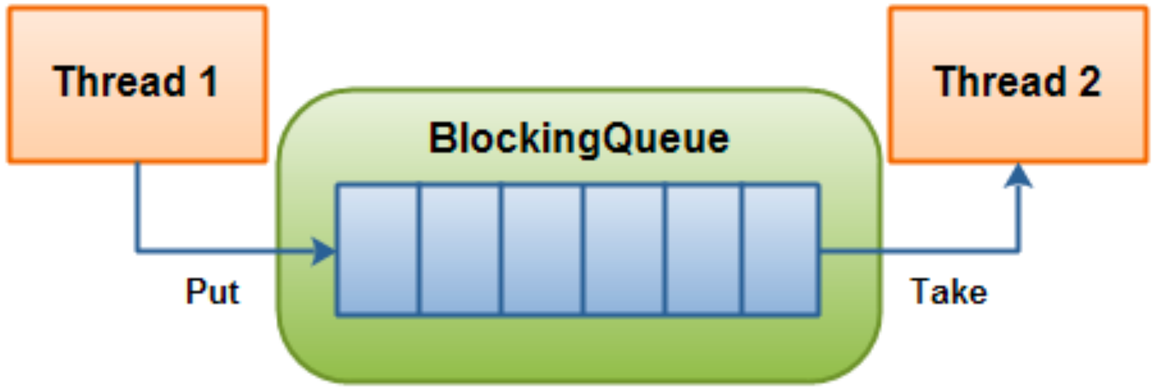


```java
public class BlockingQueueExample {

    public static void main(String[] args) throws Exception {

        BlockingQueue queue = new ArrayBlockingQueue(1024);

        Producer producer = new Producer(queue);
        Consumer consumer = new Consumer(queue);

        new Thread(producer).start();
        new Thread(consumer).start();

        Thread.sleep(4000);
    }
}
```

```java
public class Producer implements Runnable{

    protected BlockingQueue queue = null;

    public Producer(BlockingQueue queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            queue.put("1");
            queue.put("2");
            queue.put("3");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

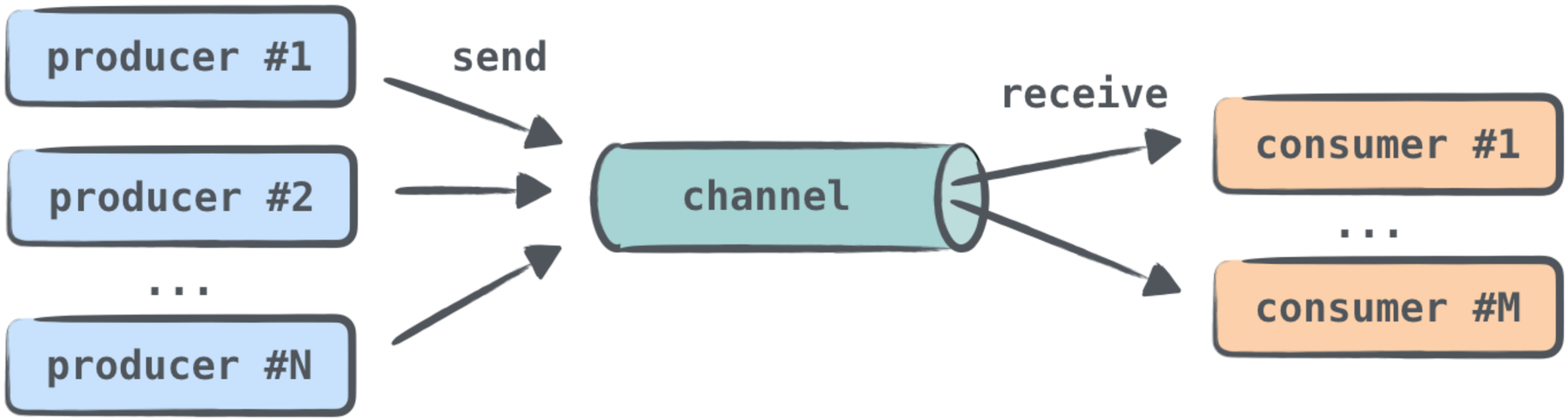
```java
public class Consumer implements Runnable{

    protected BlockingQueue queue = null;

    public Consumer(BlockingQueue queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            System.out.println(queue.take());
            System.out.println(queue.take());
            System.out.println(queue.take());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

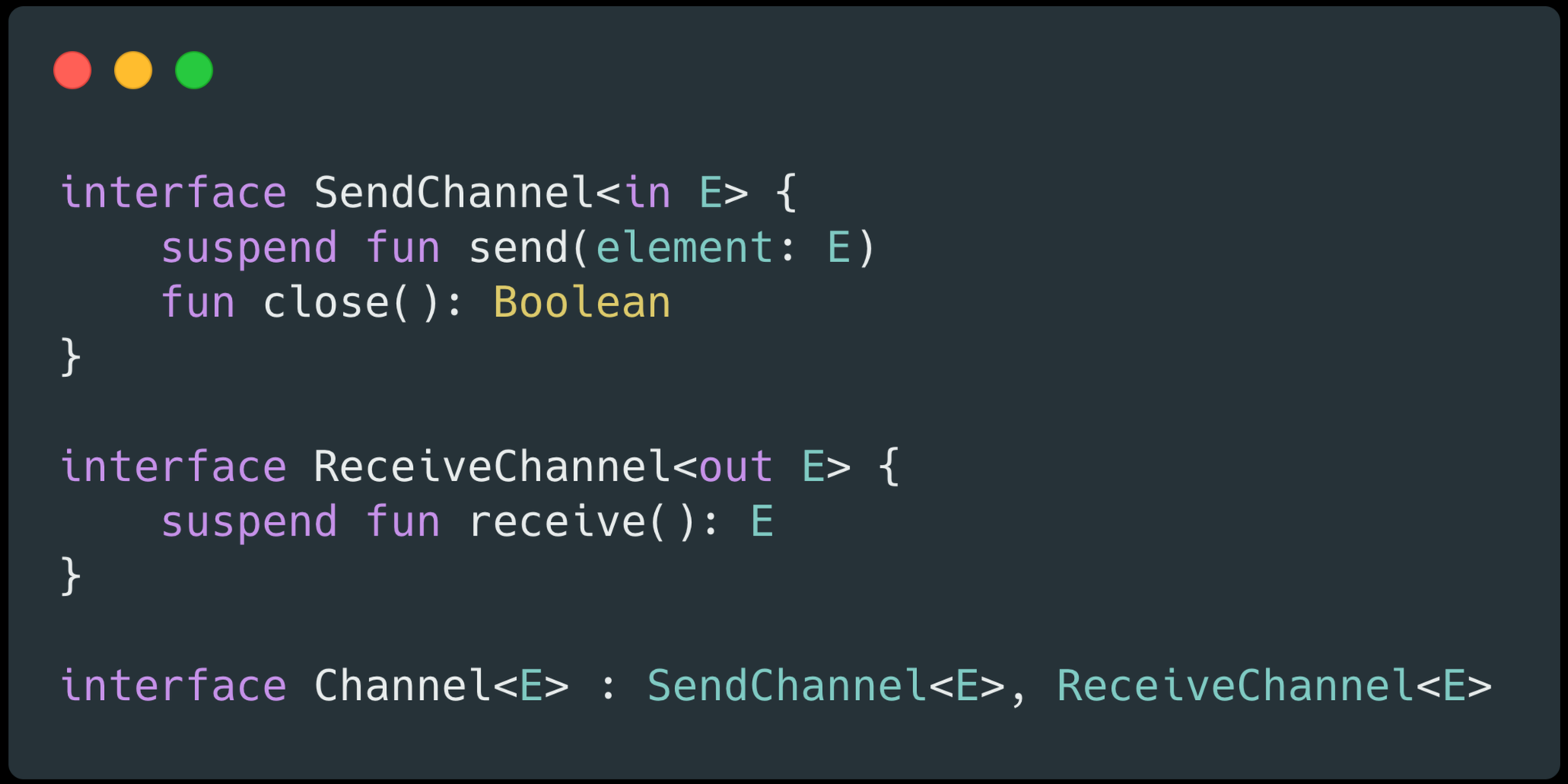
BlockingQueue



Channel

Channels


producer #1
producer #2
...
producer #N
send
channel
receive
consumer #1
...
consumer #M



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

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

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

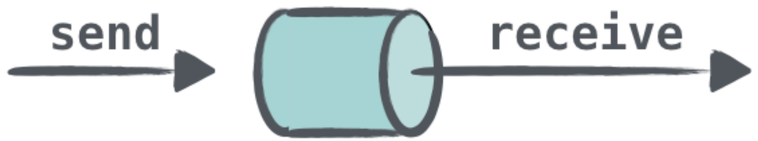
```kotlin
val channel = Channel<Int>()

launch {
    // send -> [1, 2, 3, 4, 5]
    for (i in 1..5) channel.send(i)
}

launch {
    // receive -> [1, 2, 3, 4, 5]
    for (i in 1..5) println(channel.receive())
}
```

Tipos de Channels

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

```
// Channel Factory
public fun <E> Channel(capacity: Int = RENDEZVOUS): Channel<E> =
    when (capacity) {
        RENDEZVOUS -> RendezvousChannel()
        UNLIMITED -> LinkedListChannel()
        CONFLATED -> ConflatedChannel()
        BUFFERED -> ArrayChannel(CHANNEL_DEFAULT_CAPACITY)
        else -> ArrayChannel(capacity)
    }
```

Rendezvous Channel



Channel sem *buffer*

Uma das funções (***send*** ou ***receive***) sempre fica **suspensa** até que a outra seja **chamada**.

```
// DEFAULT
Channel<Int>() == Channel<Int>(RENDEZVOUS)
```

Unlimited Channel

Channel com *buffer "ilimitado"*

Os *producers* podem **enviar** elementos para esse canal, e ele **crescerá infinitamente**.
*A chamada de **envio nunca será suspensa.***

Se não houver mais memória, você receberá uma OutOfMemoryException

Conflated Channel

Devolve sempre o elemento mais recente

Buffered Channel

Channel com *buffer de tamanho fixo*

O tamanho do *buffer* é **64** por padrão mas pode ser substituído pela *configuração* *DEFAULT_BUFFER_PROPERTY_NAME* via JVM

Os *producers* **podem enviar** elementos para este canal **até que o limite de tamanho seja atingido**.

Quando o ***buffer*** **estiver cheio**, a próxima chamada de **envio** será suspensa **até que apareça mais espaço livre**.

Na prática.... 👨🏻‍💻

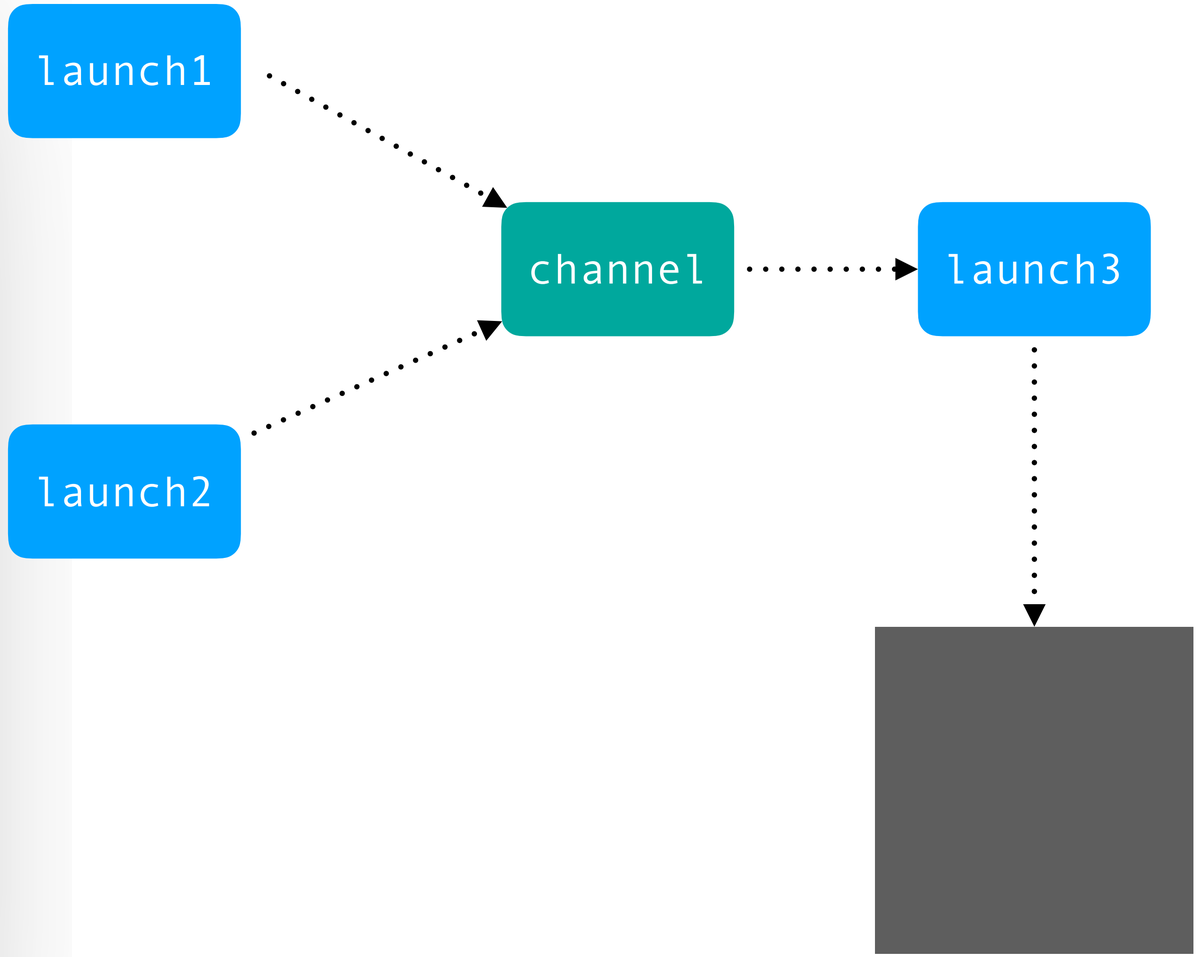
```
val rendezvousChannel = Channel<String>()
val bufferedChannel = Channel<String>(10)
val conflatedChannel = Channel<String>(CONFLATED)
val unlimitedChannel = Channel<String>(UNLIMITED)
```

Qual a saída esperada?


```
// RENDEZVOUS (DEFAULT)
val channel = Channel<String>()
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```

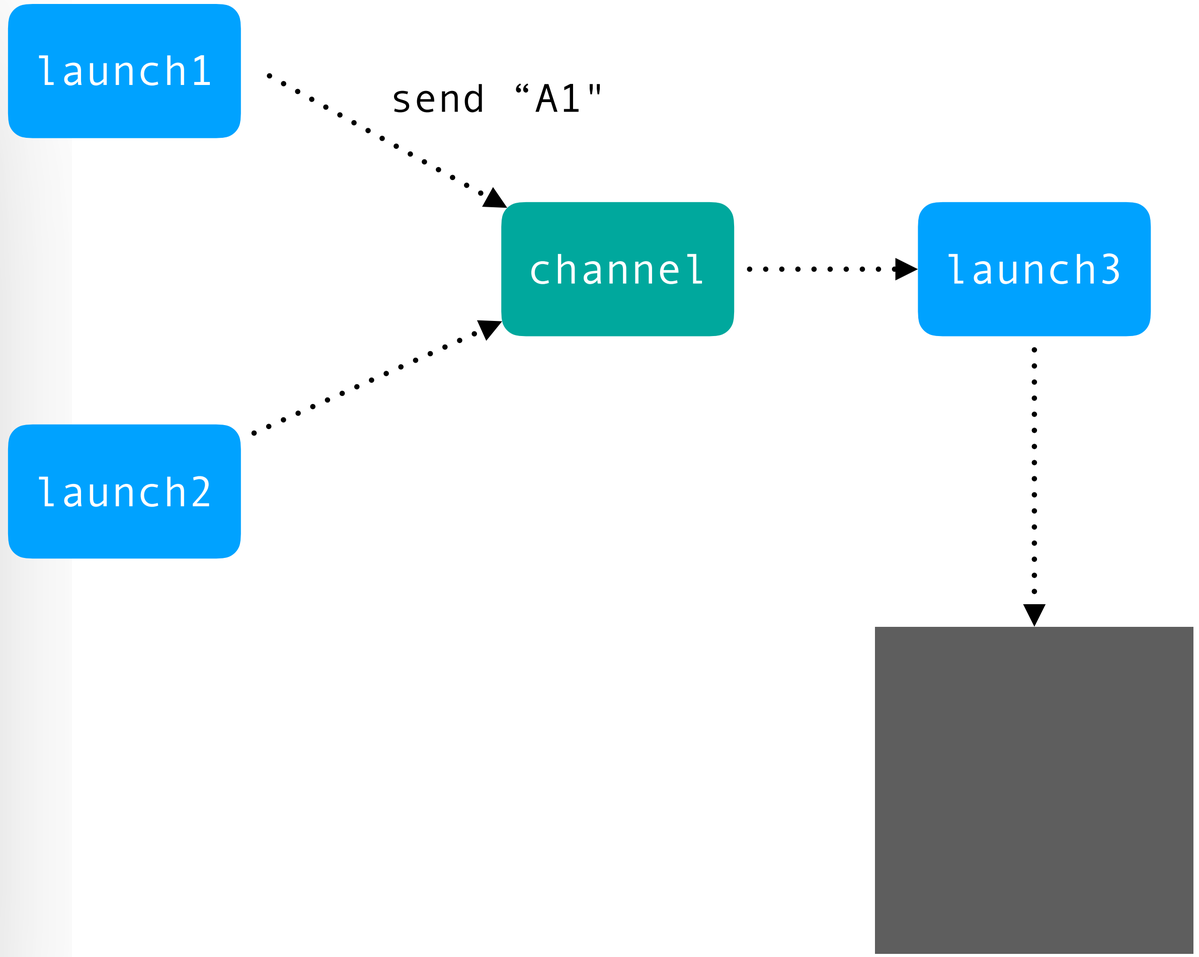
Qual a saída esperada?


```
// RENDEZVOUS (DEFAULT)
val channel = Channel<String>()
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



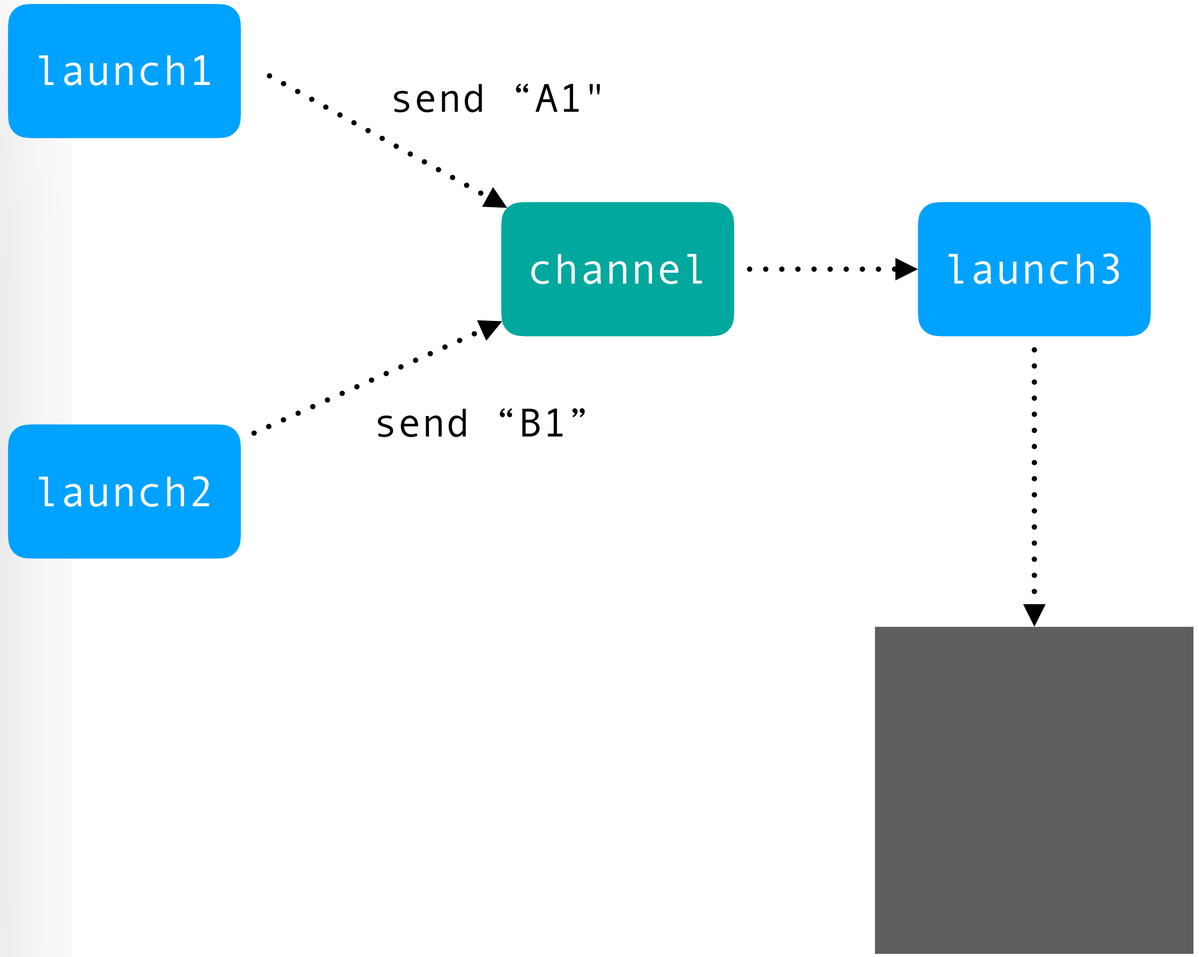
Qual a saída esperada?


```
// RENDEZVOUS (DEFAULT)
val channel = Channel<String>()
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



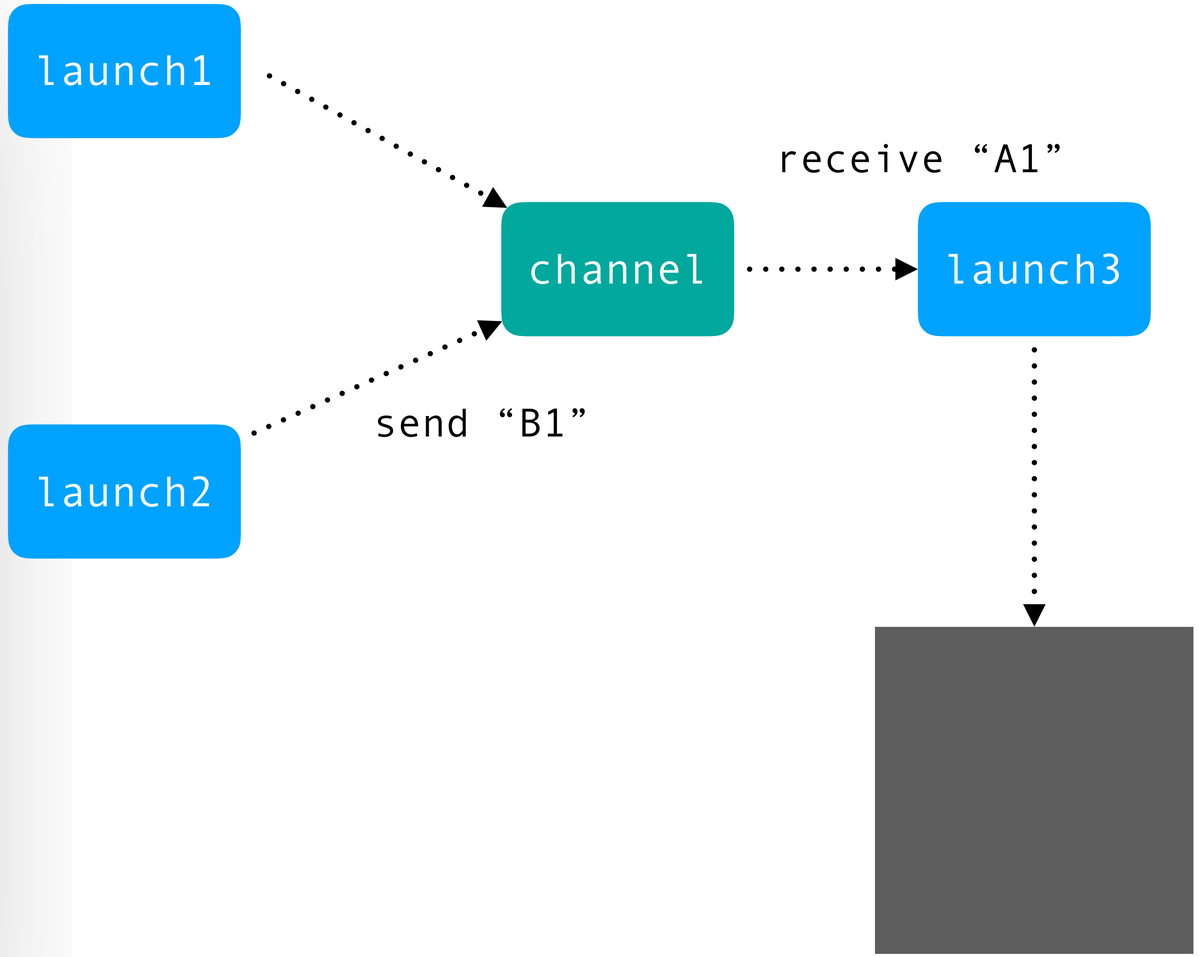
Qual a saída esperada?


```
// RENDEZVOUS (DEFAULT)
val channel = Channel<String>()
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



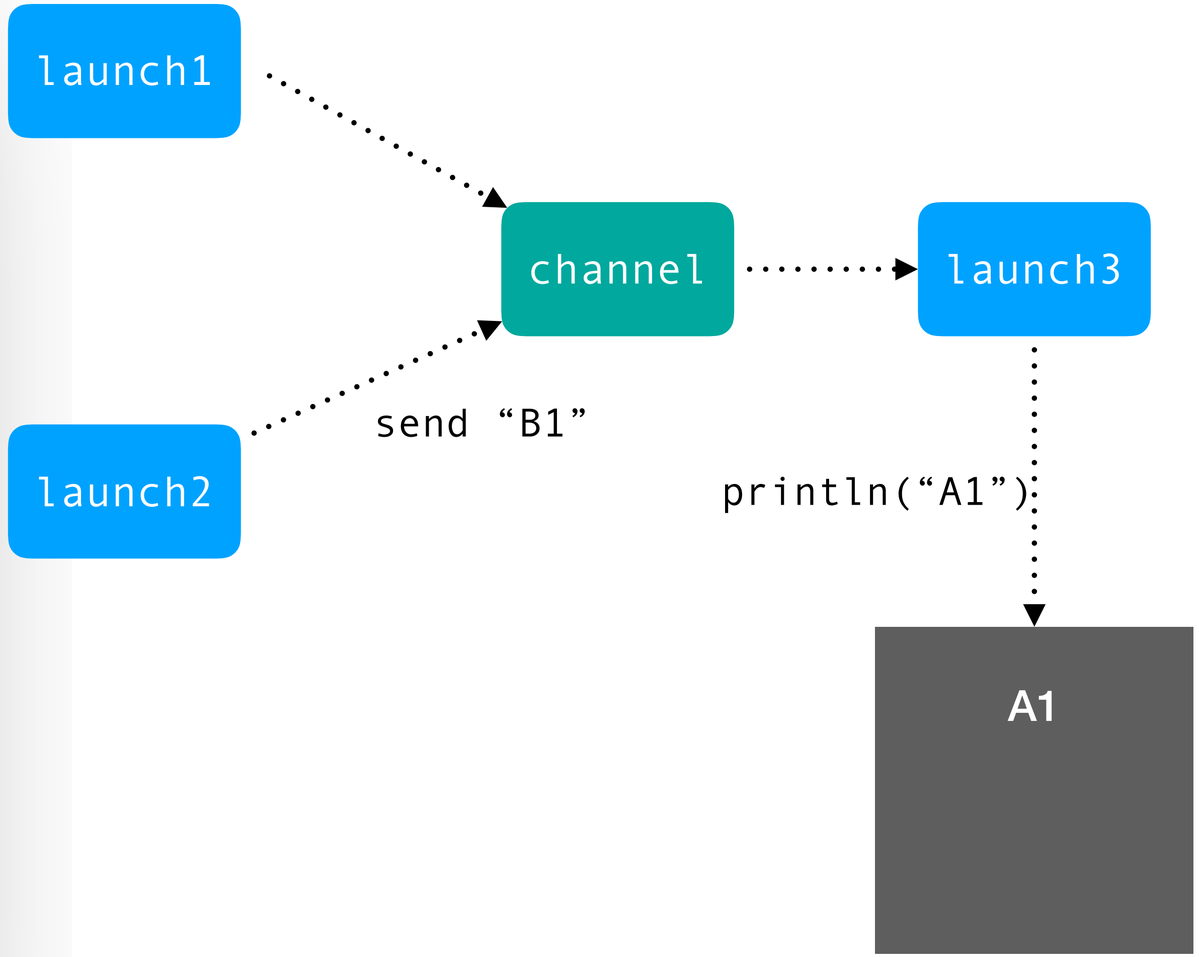
Qual a saída esperada?


```
// RENDEZVOUS (DEFAULT)
val channel = Channel<String>()
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



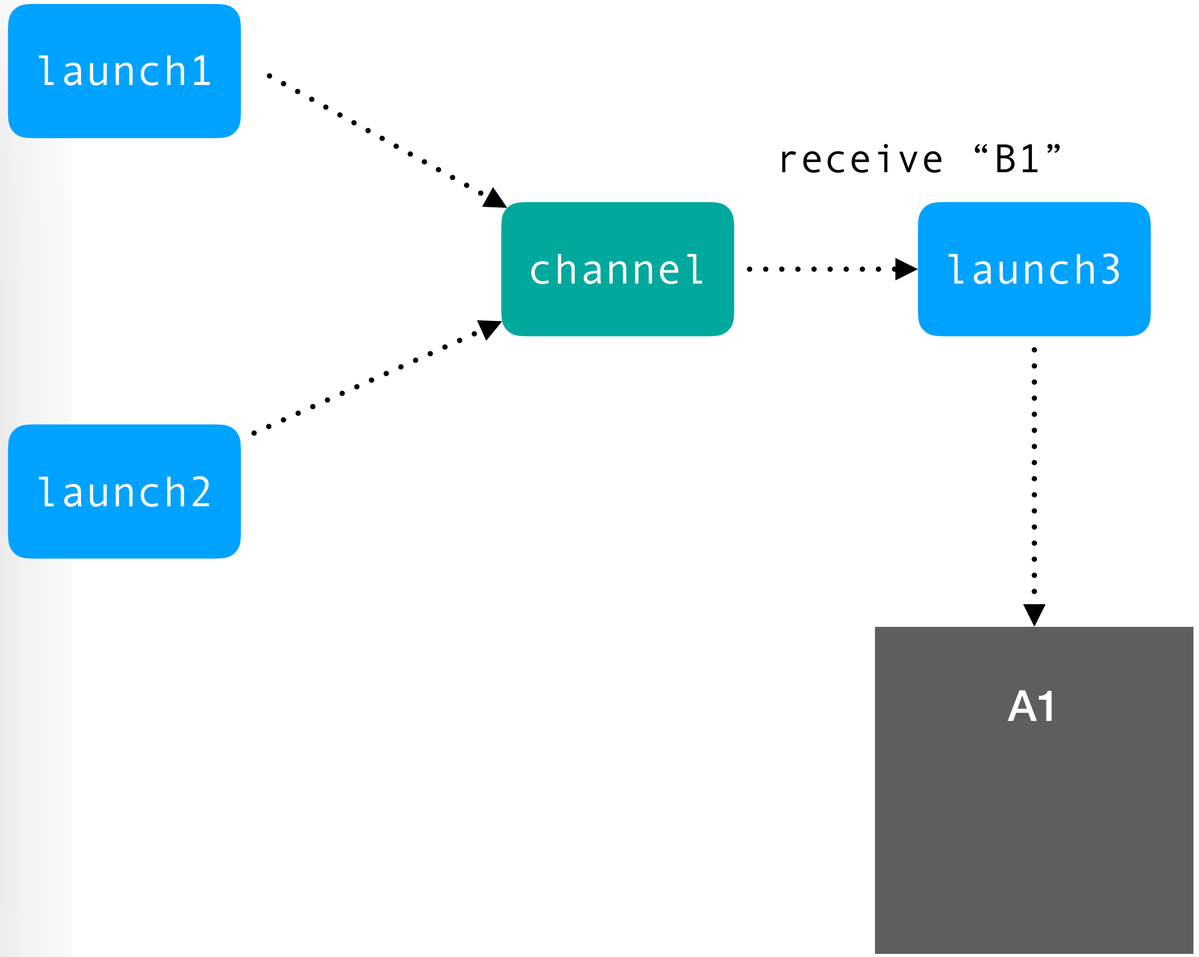
Qual a saída esperada?


```
// RENDEZVOUS (DEFAULT)
val channel = Channel<String>()
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



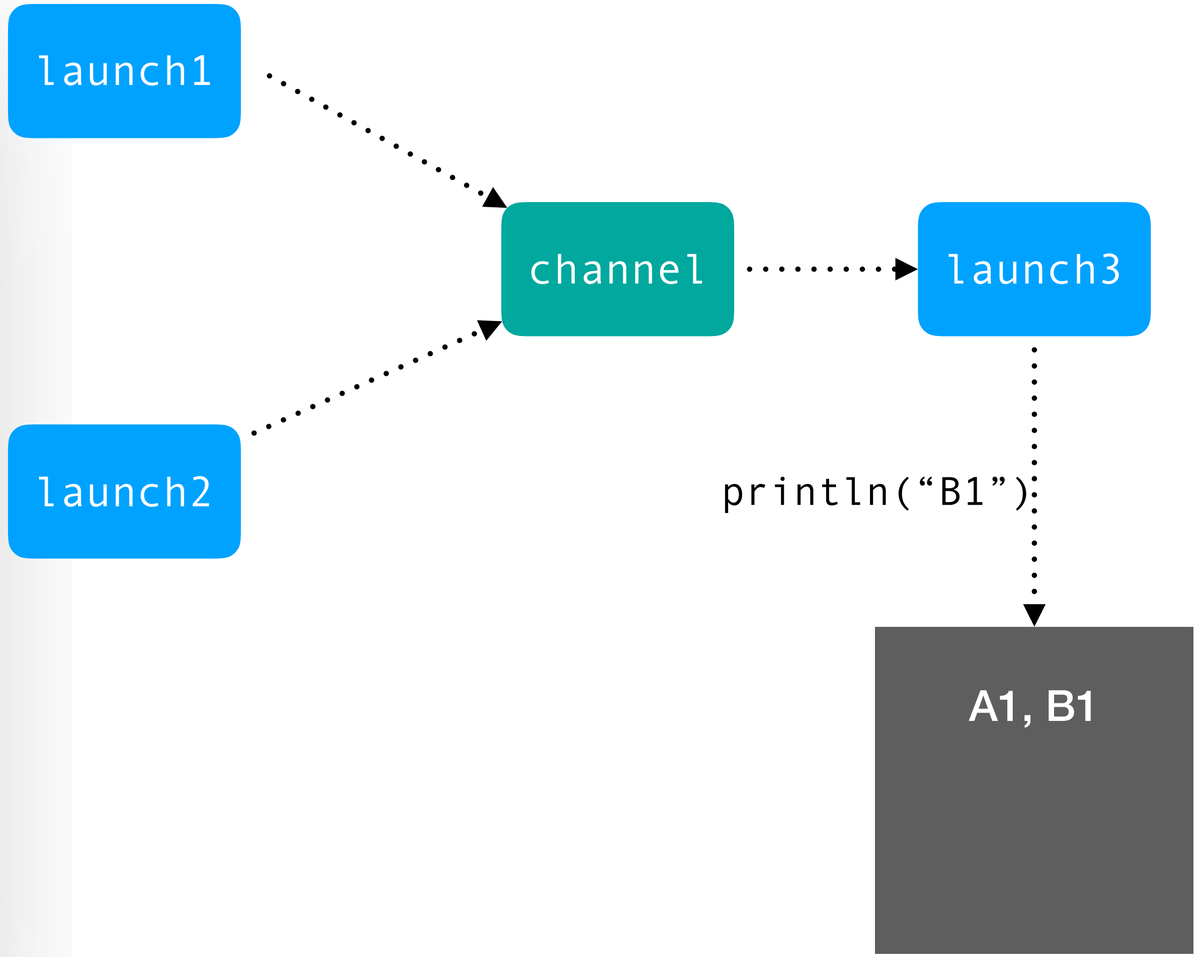
Qual a saída esperada?


```
// RENDEZVOUS (DEFAULT)
val channel = Channel<String>()
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



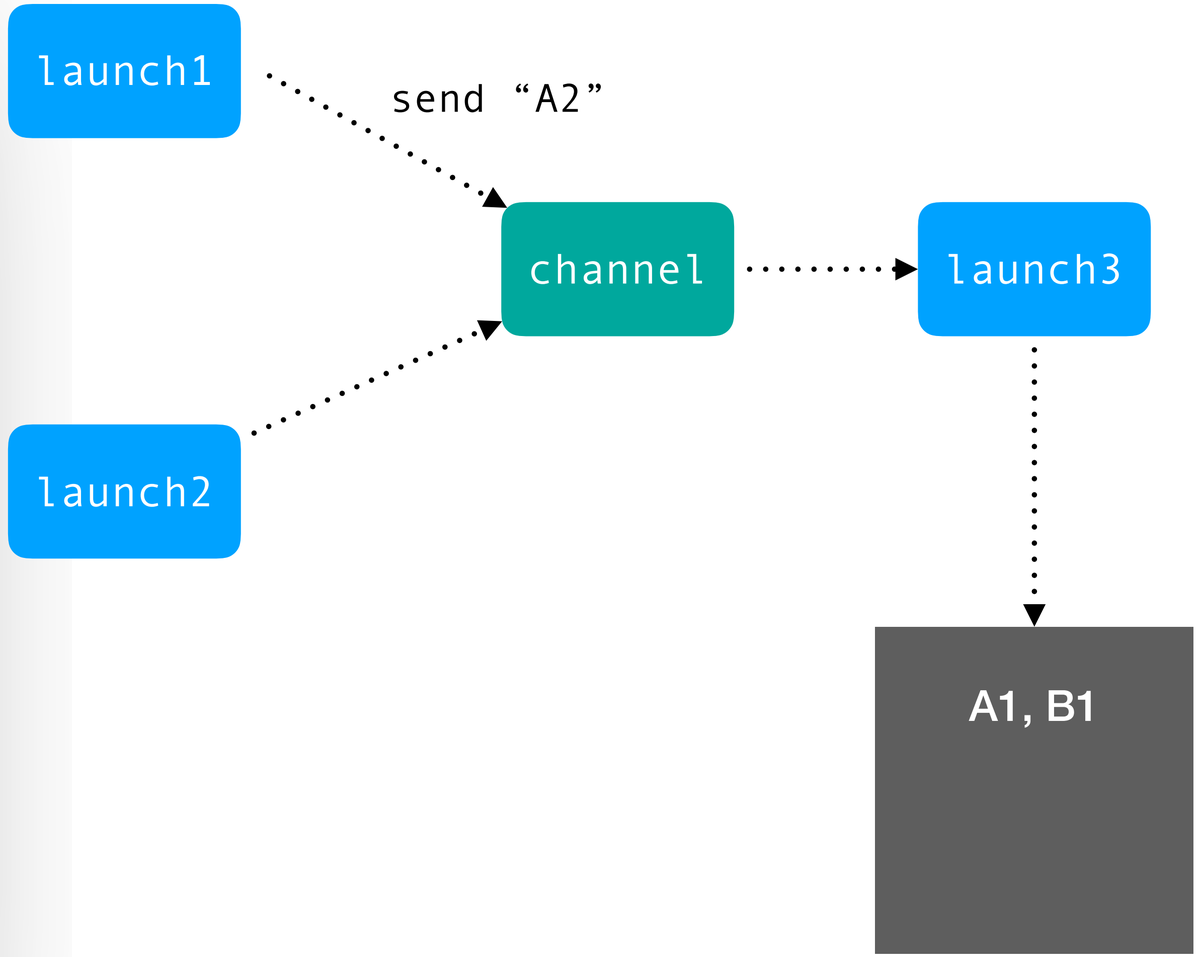
Qual a saída esperada?


```
// RENDEZVOUS (DEFAULT)
val channel = Channel<String>()
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



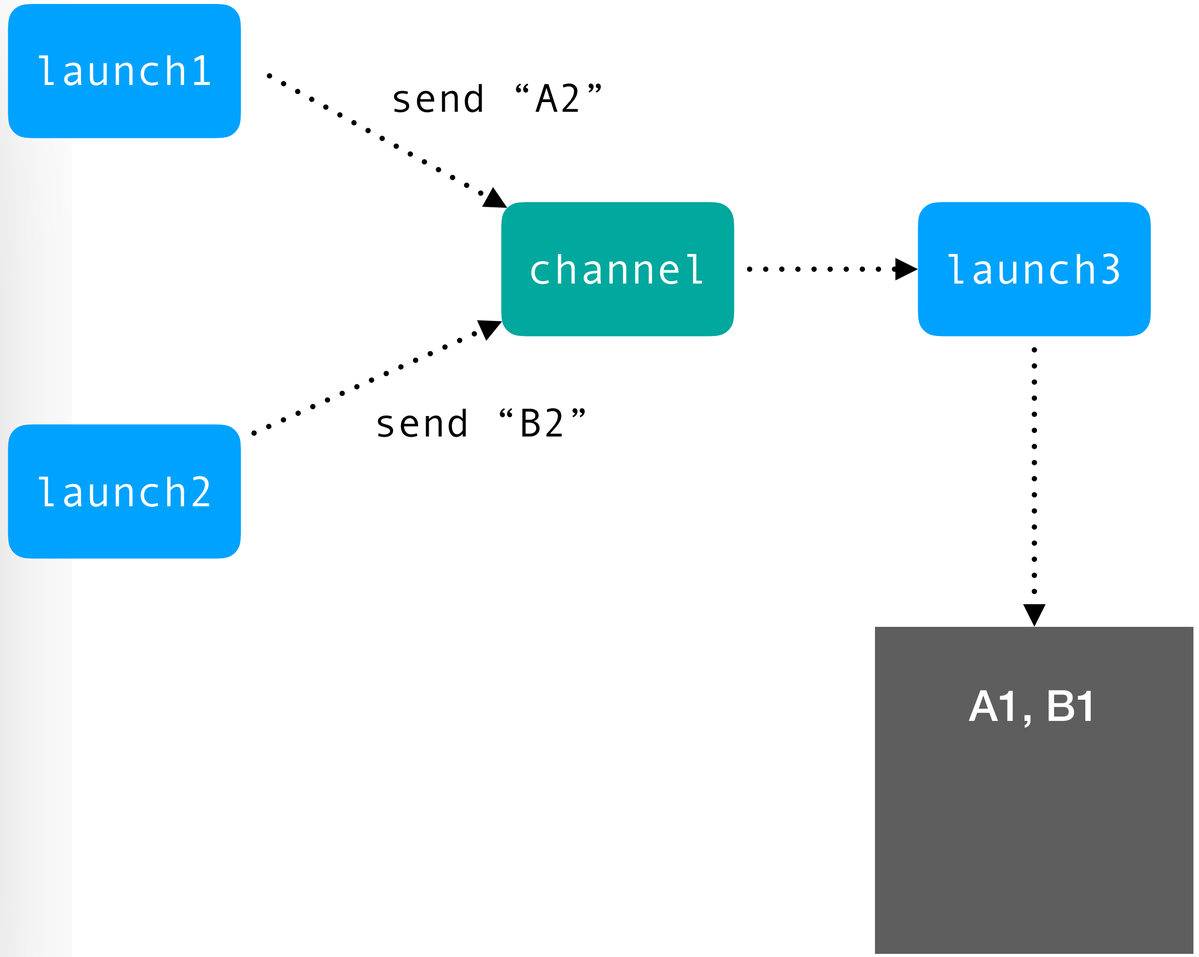
Qual a saída esperada?


```
// RENDEZVOUS (DEFAULT)
val channel = Channel<String>()
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



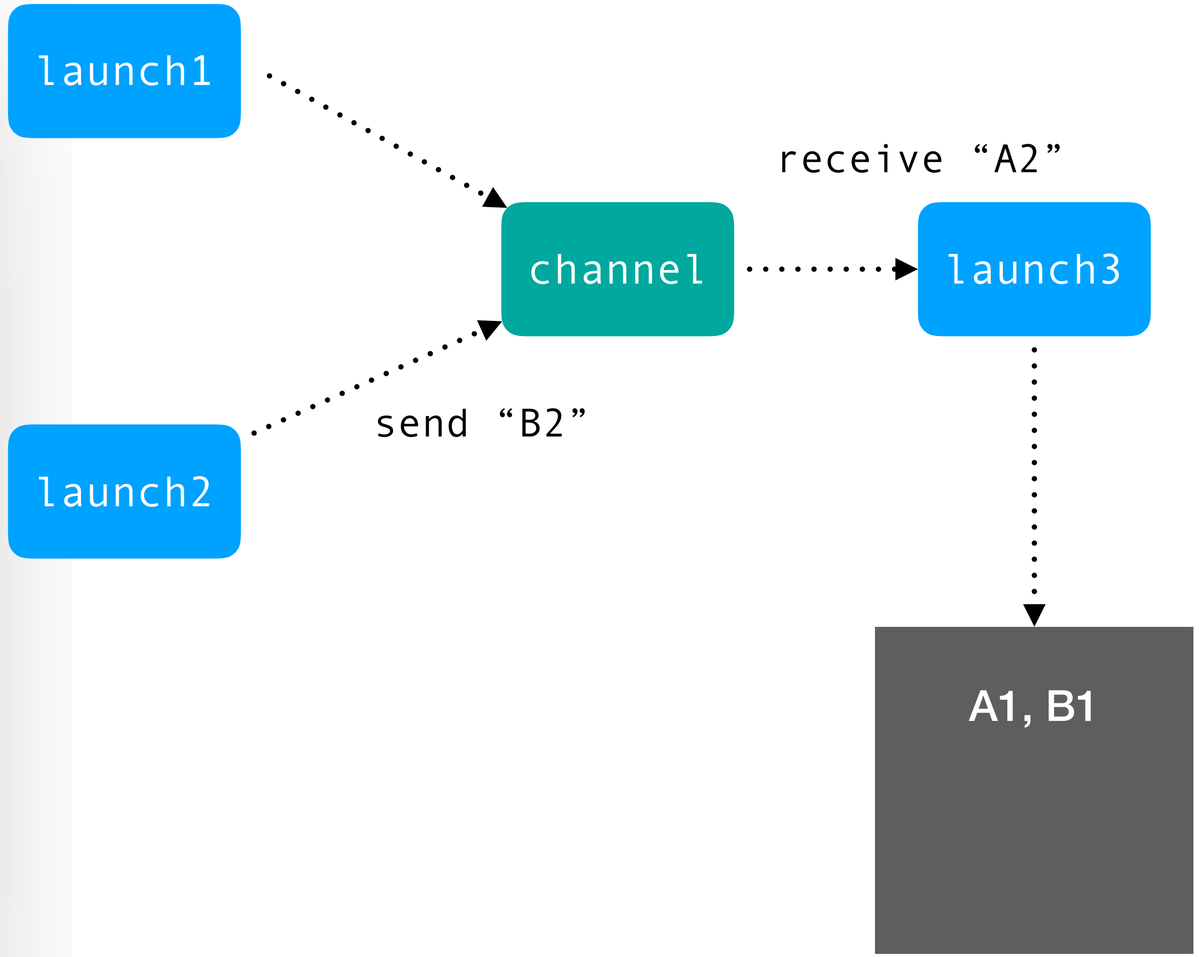
Qual a saída esperada?


```
// RENDEZVOUS (DEFAULT)
val channel = Channel<String>()
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



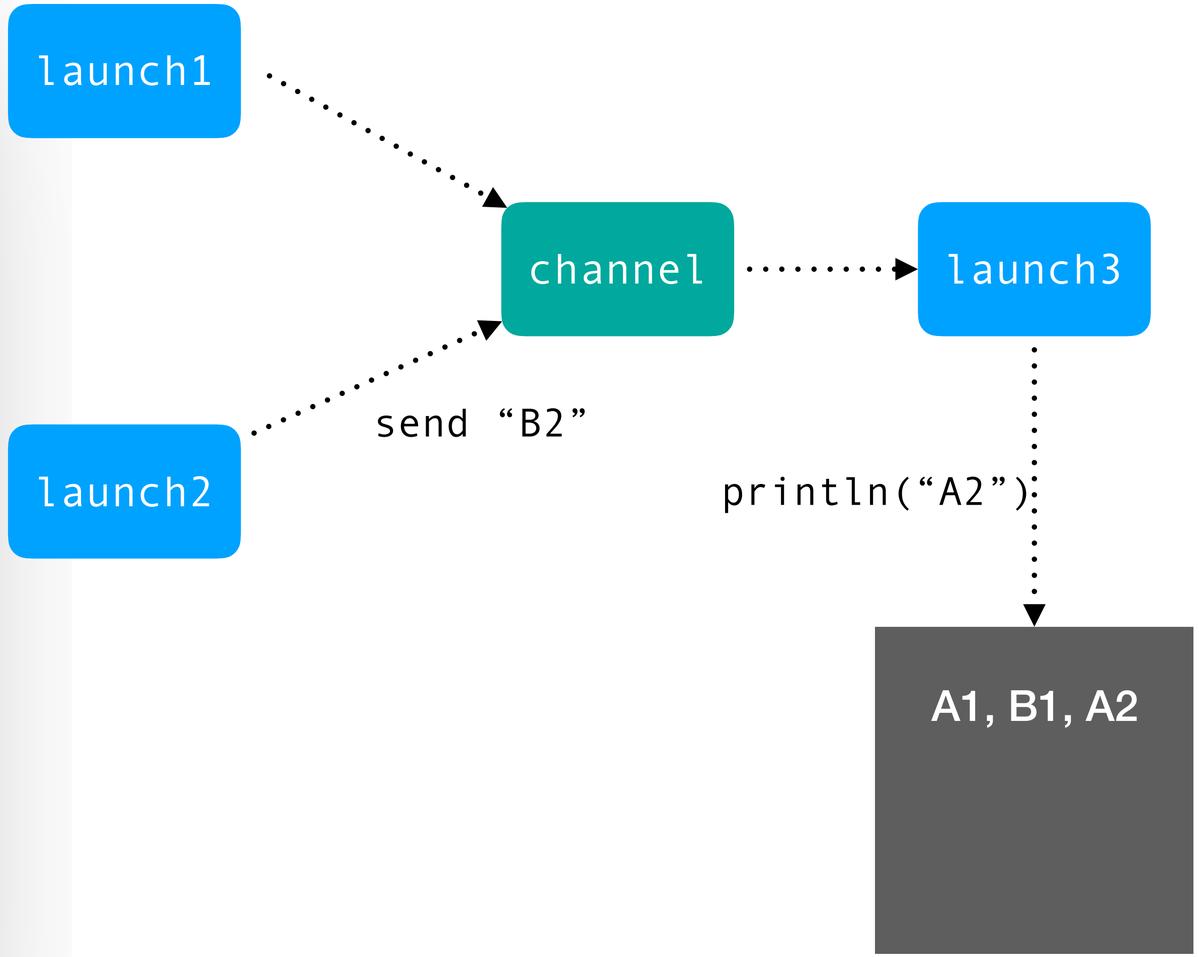
Qual a saída esperada?


```
// RENDEZVOUS (DEFAULT)
val channel = Channel<String>()
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



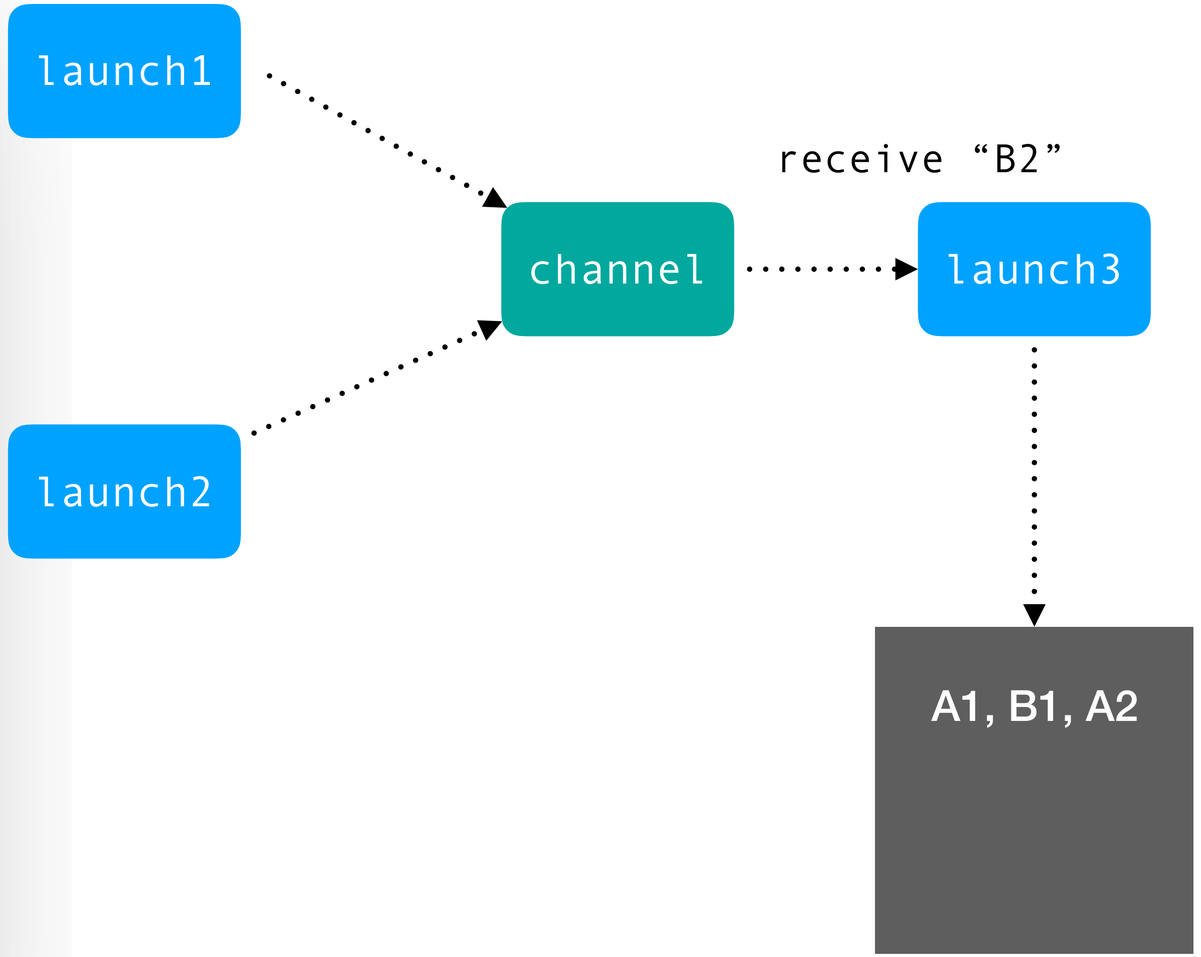
Qual a saída esperada?


```
// RENDEZVOUS (DEFAULT)
val channel = Channel<String>()
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



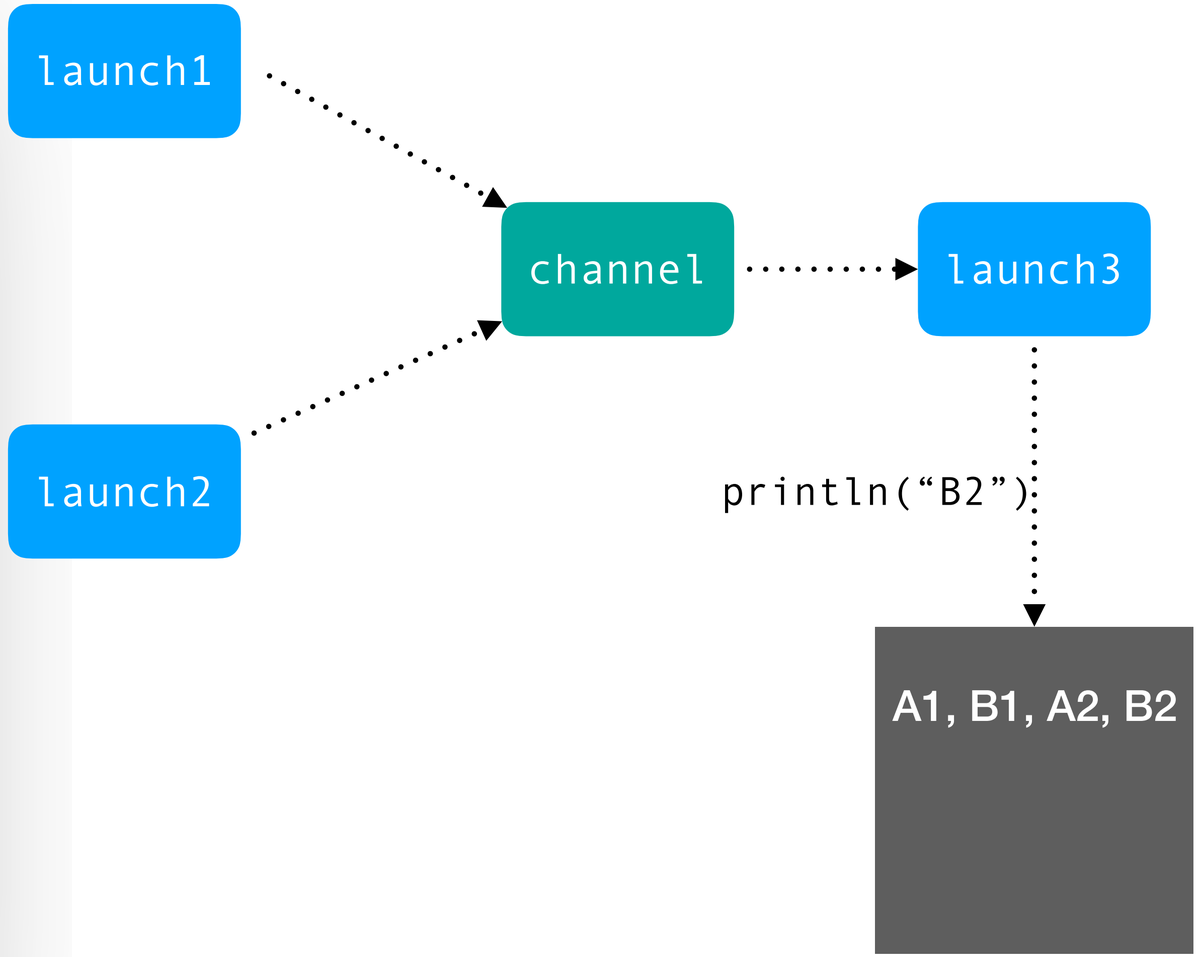
Qual a saída esperada?


```
// RENDEZVOUS (DEFAULT)
val channel = Channel<String>()
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



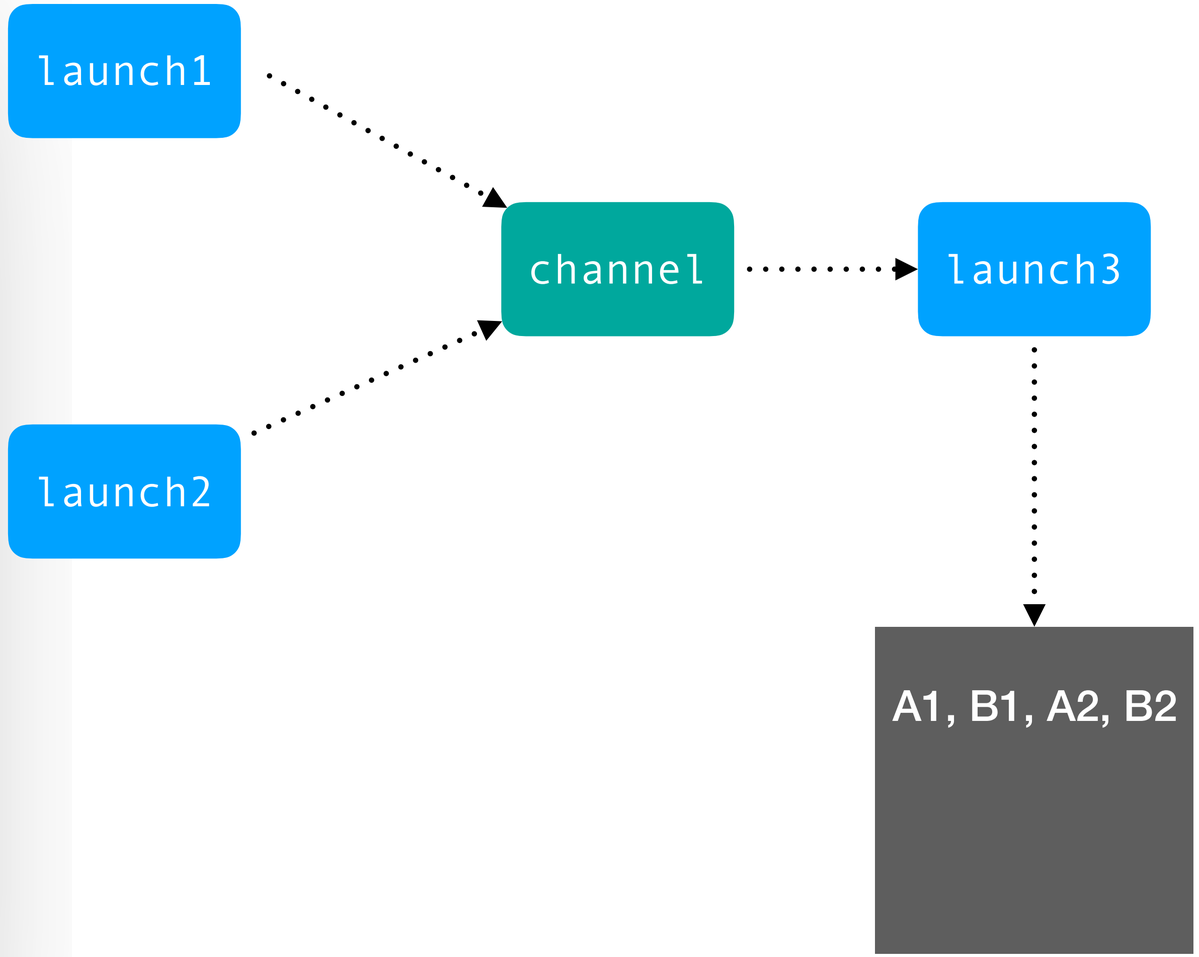
Qual a saída esperada?


```
// RENDEZVOUS (DEFAULT)
val channel = Channel<String>()
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



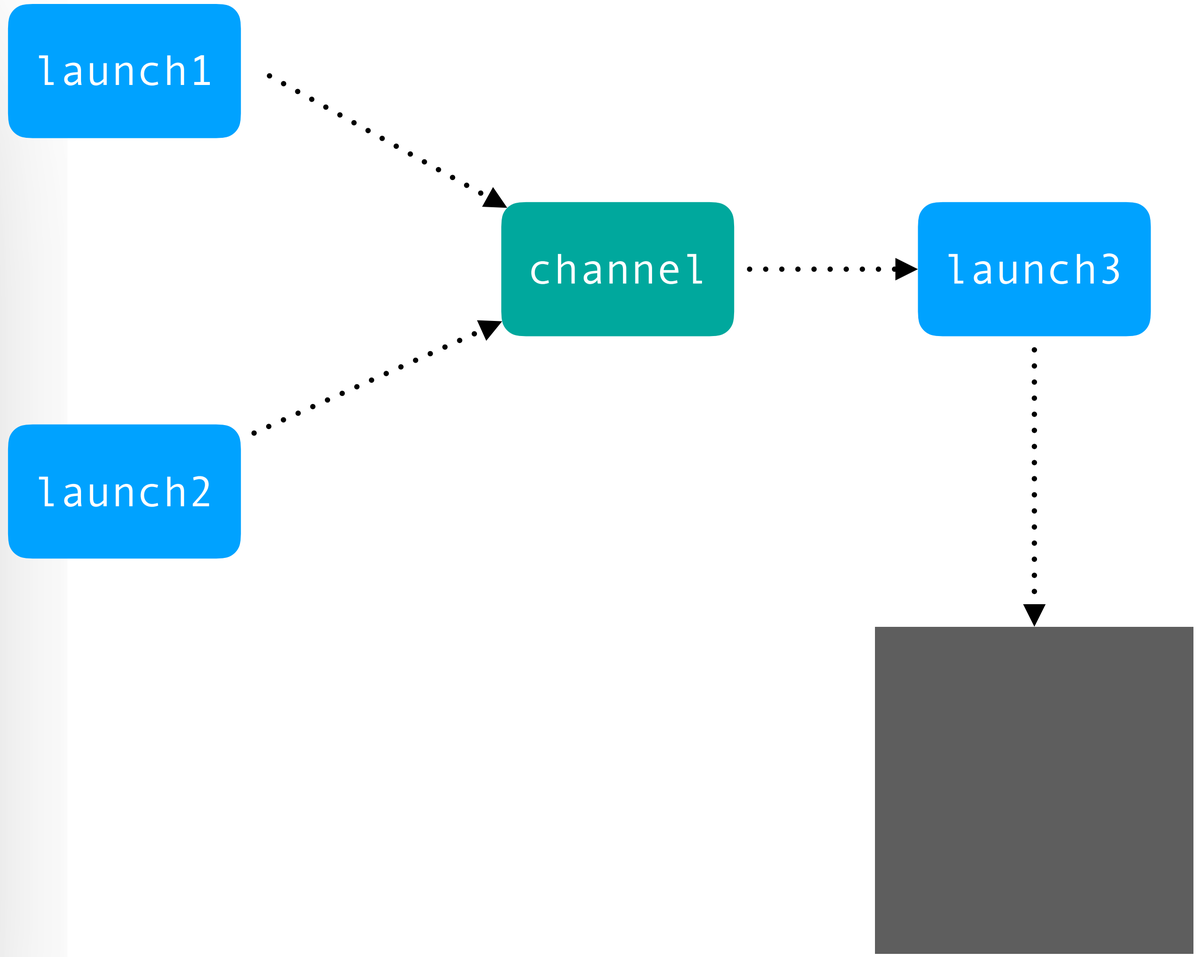
Qual a saída esperada?


```
// RENDEZVOUS (DEFAULT)
val channel = Channel<String>()
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



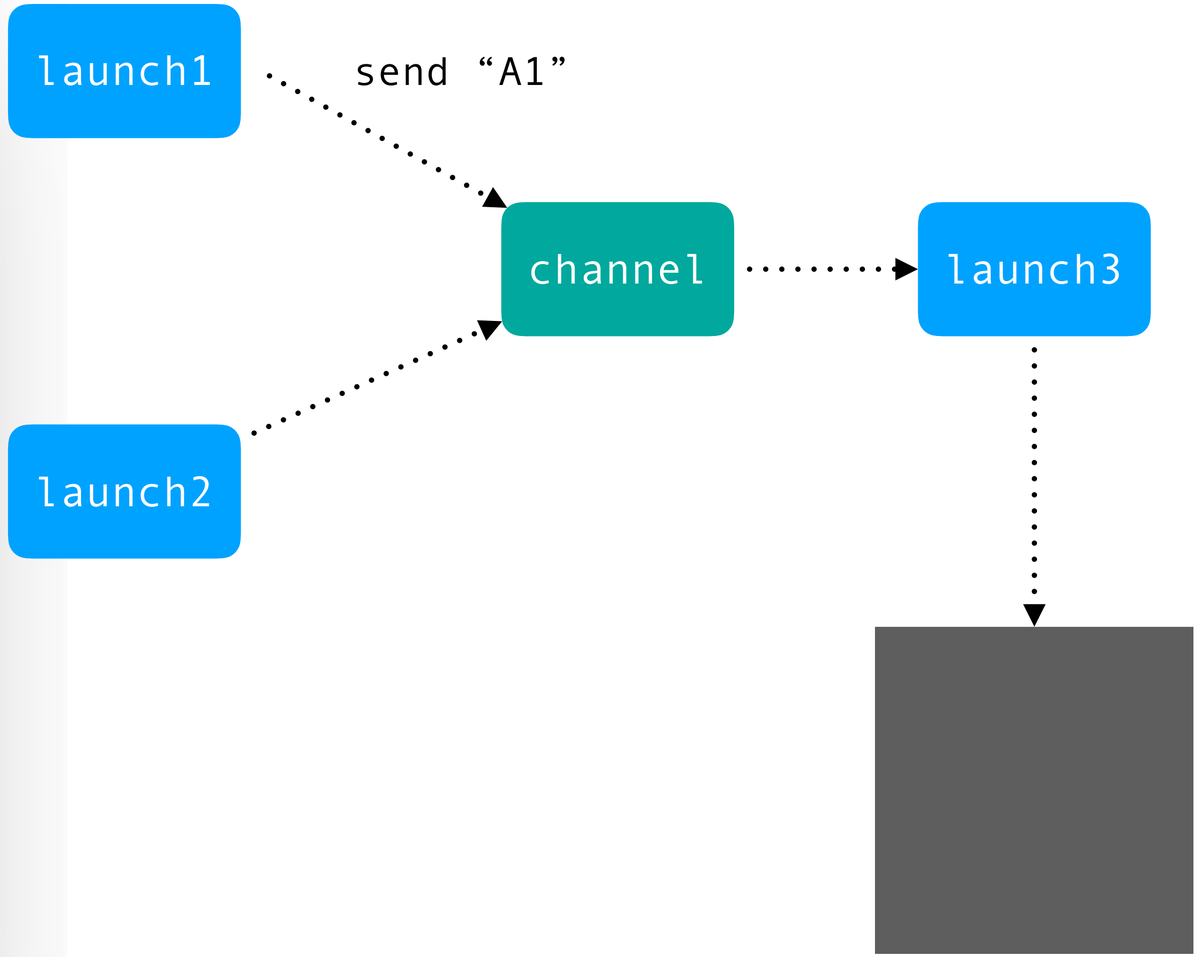
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



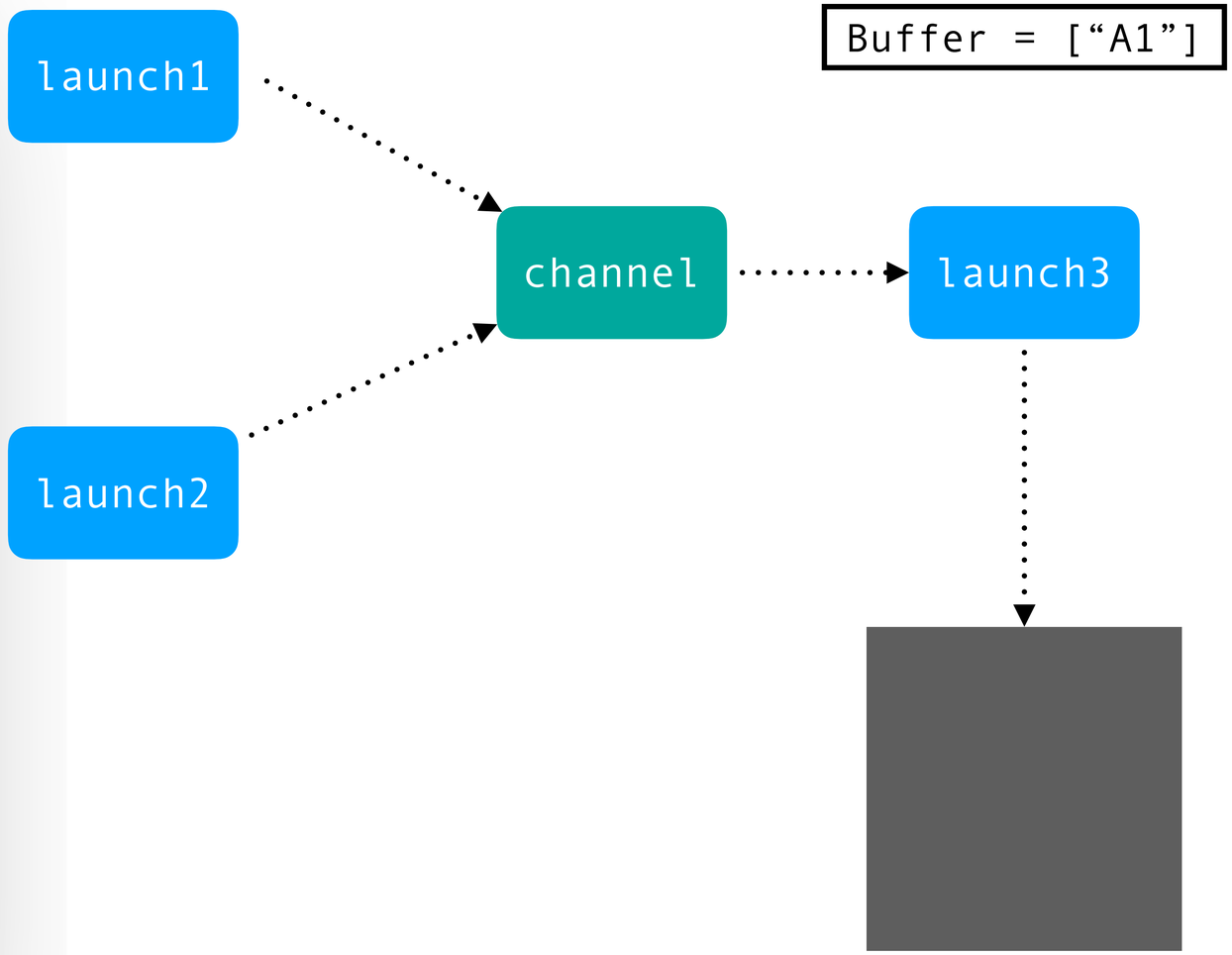
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



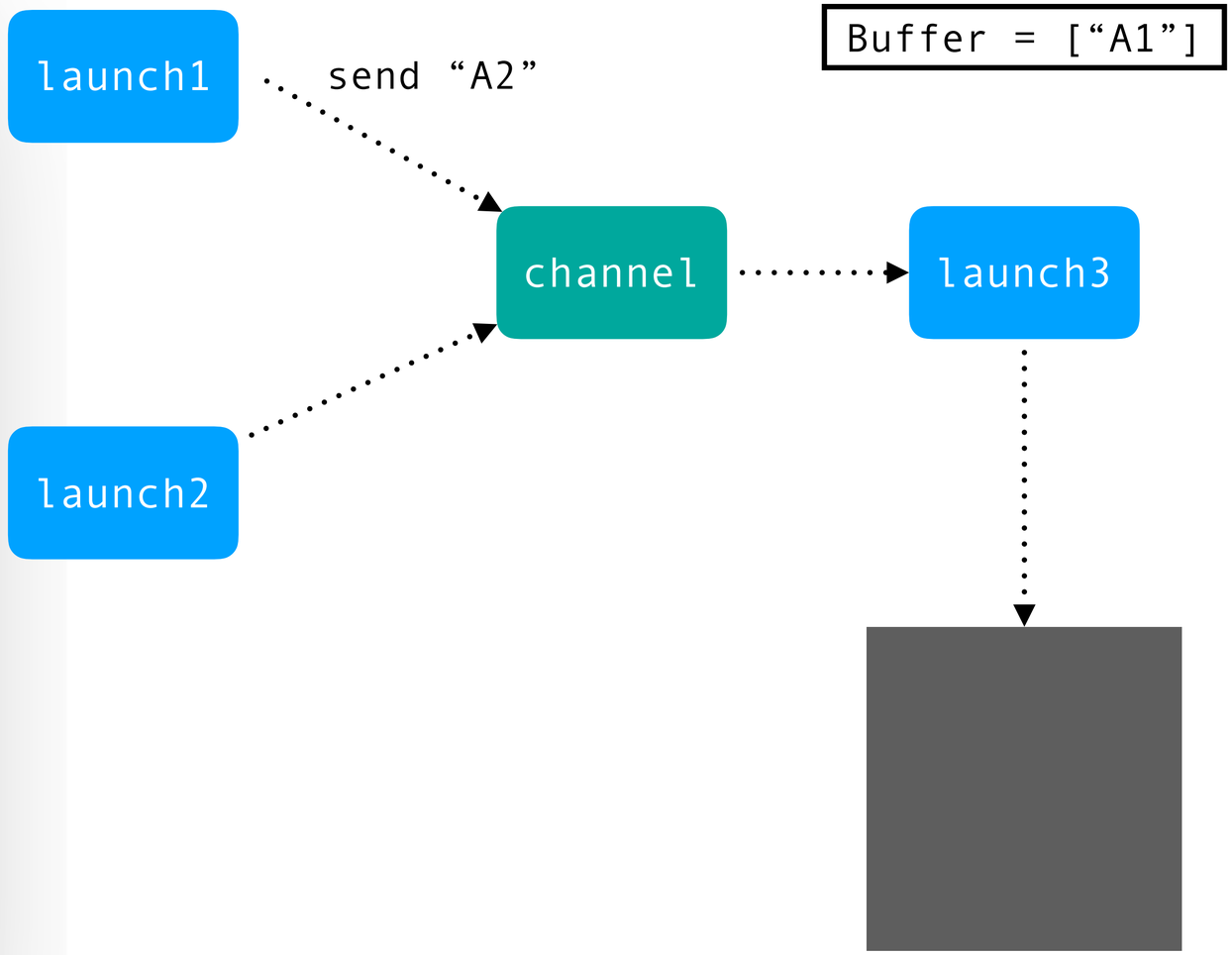
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



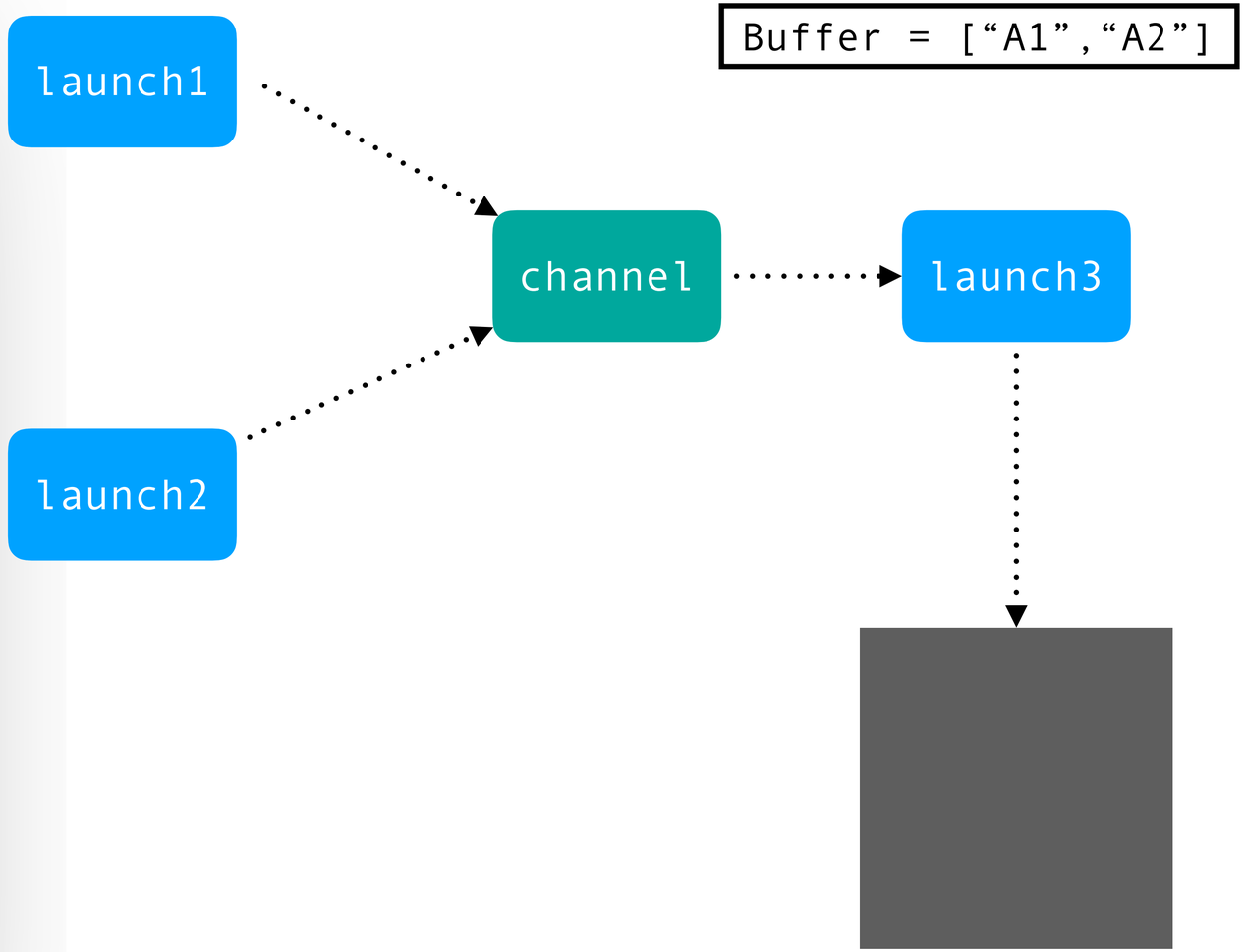
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



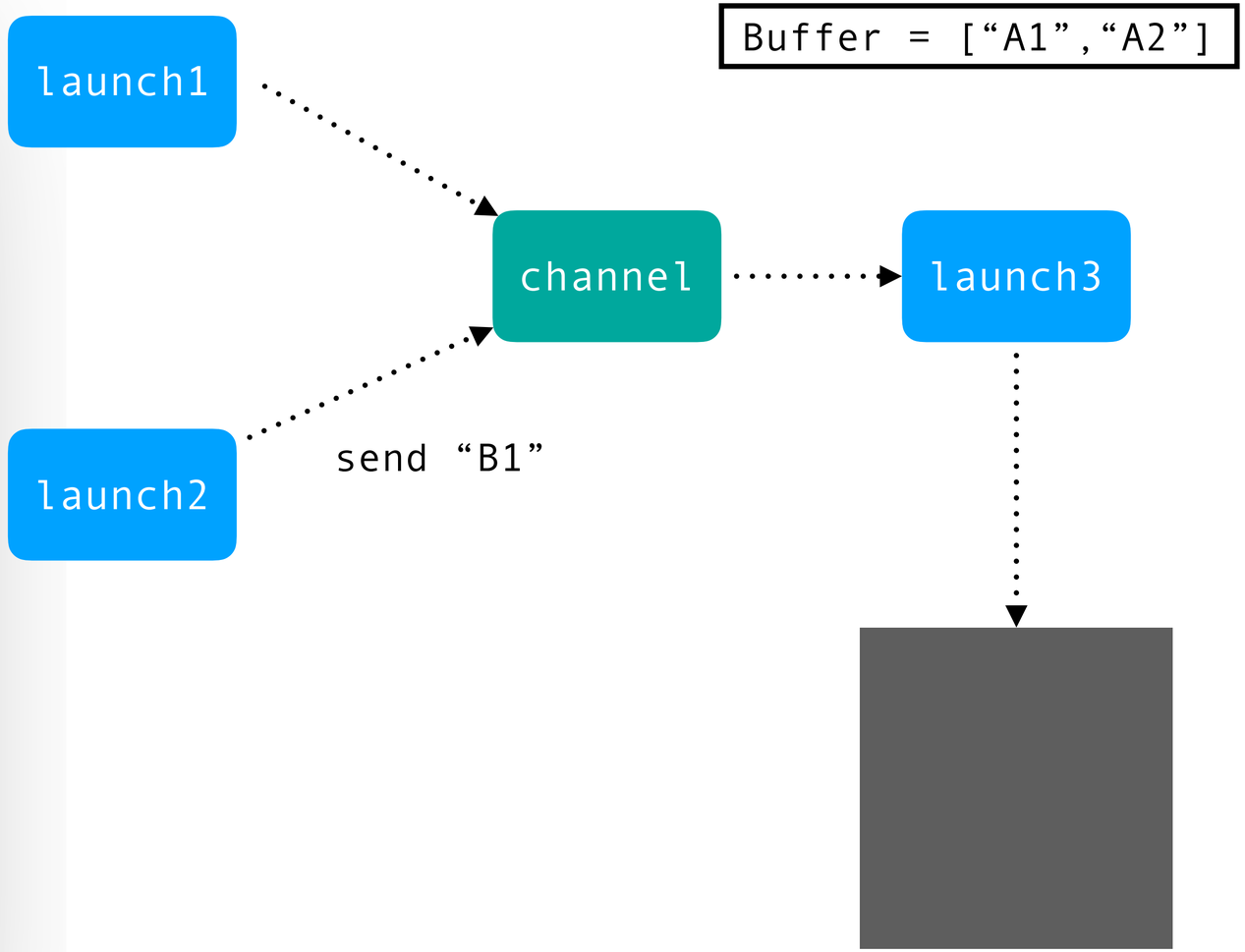
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



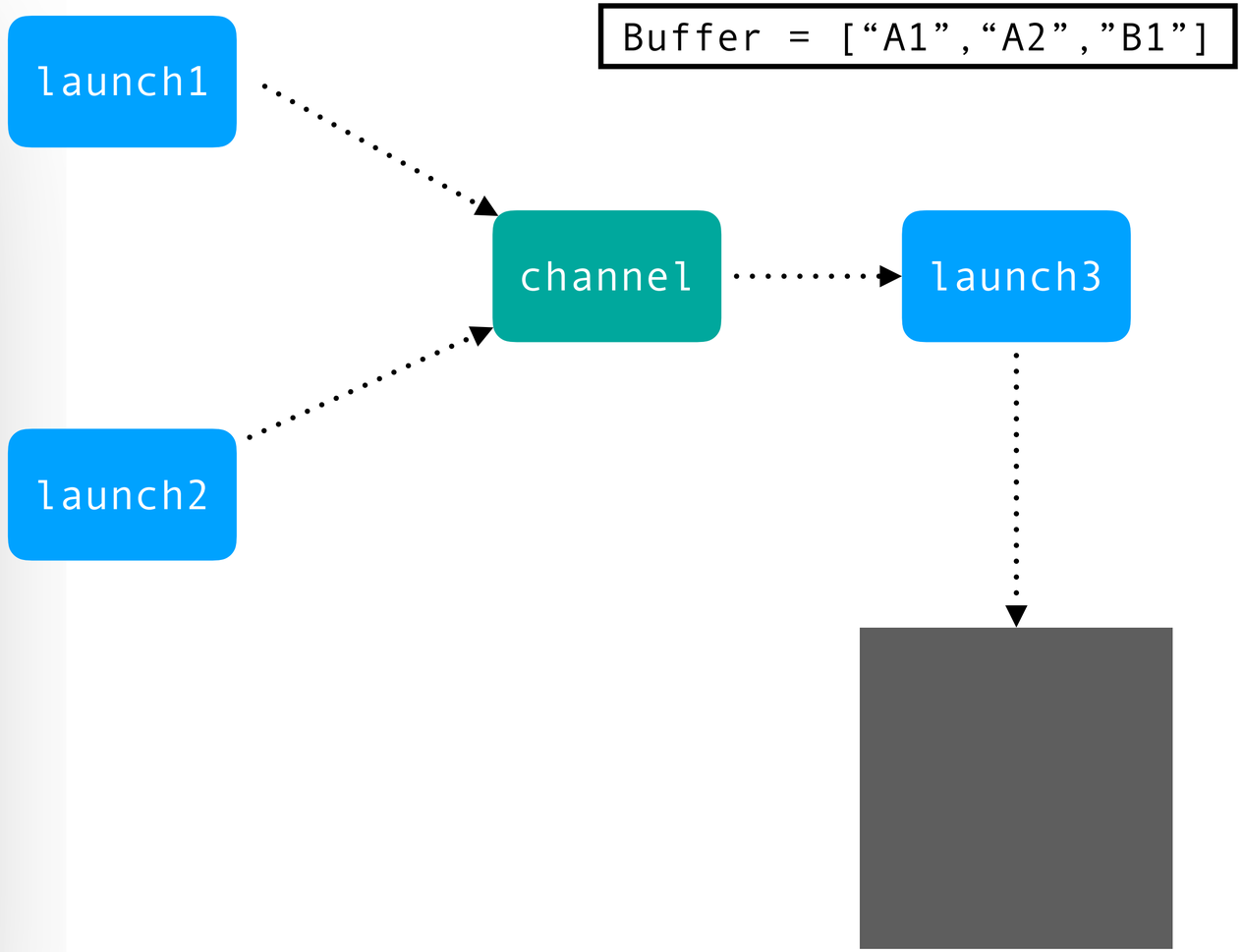
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



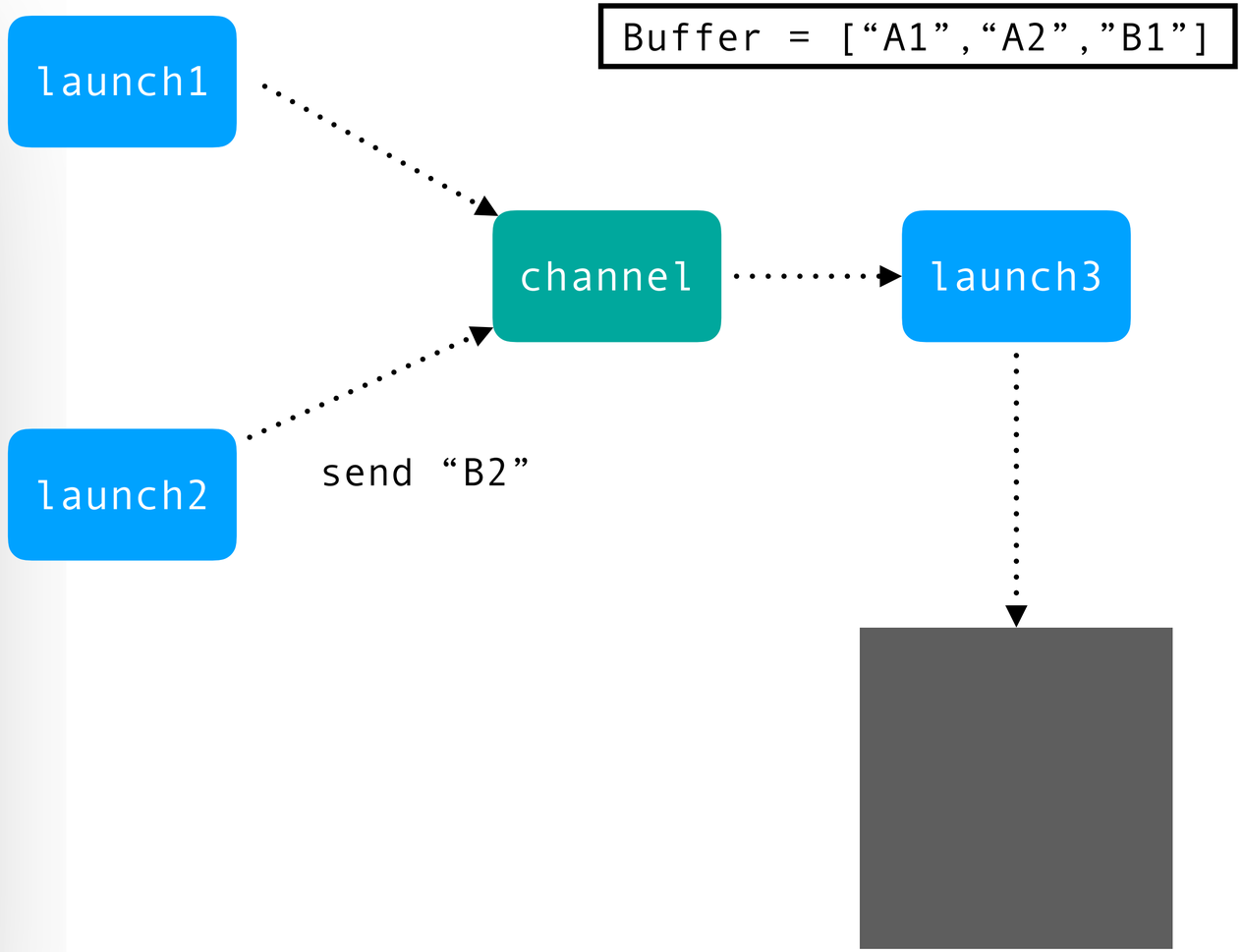
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



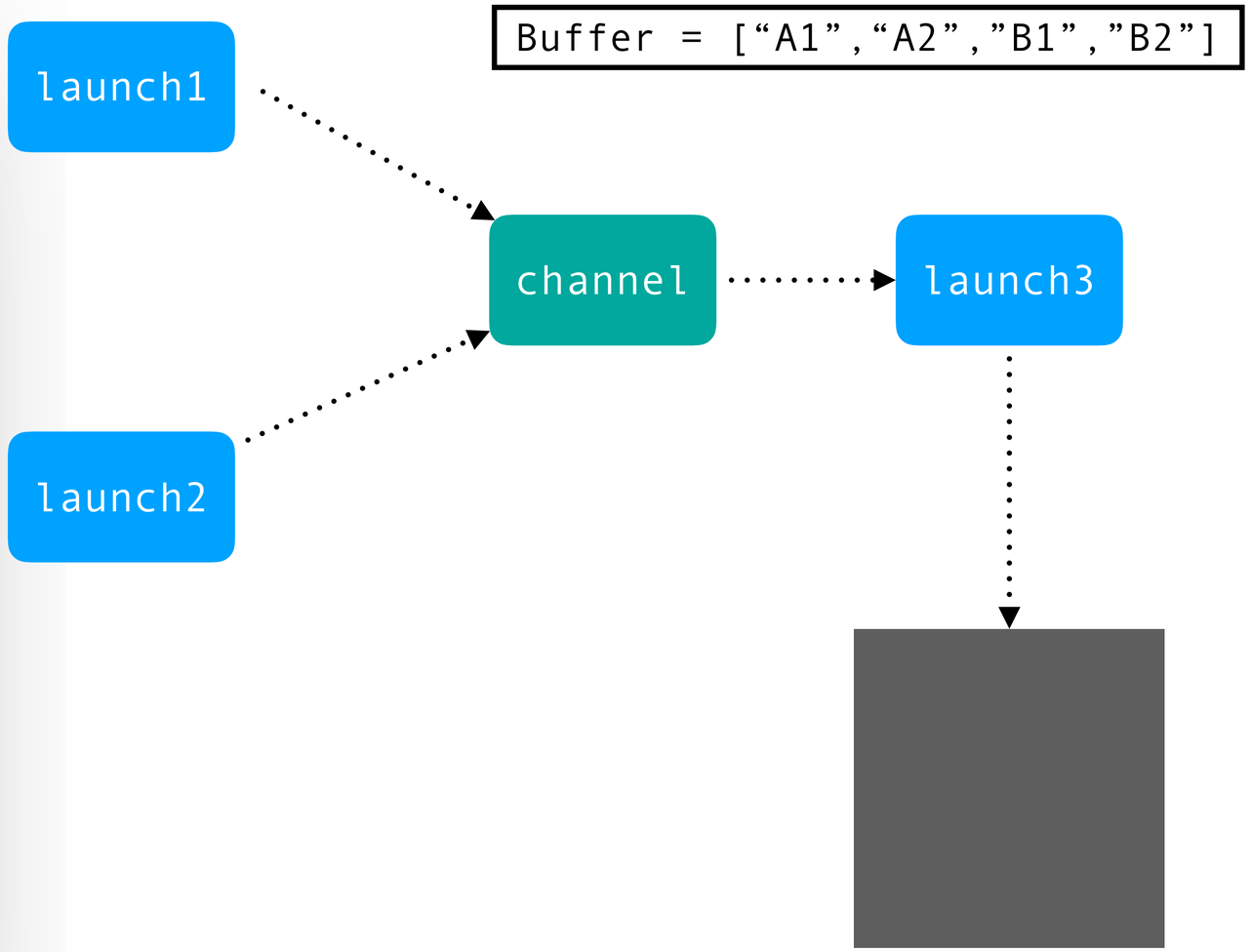
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



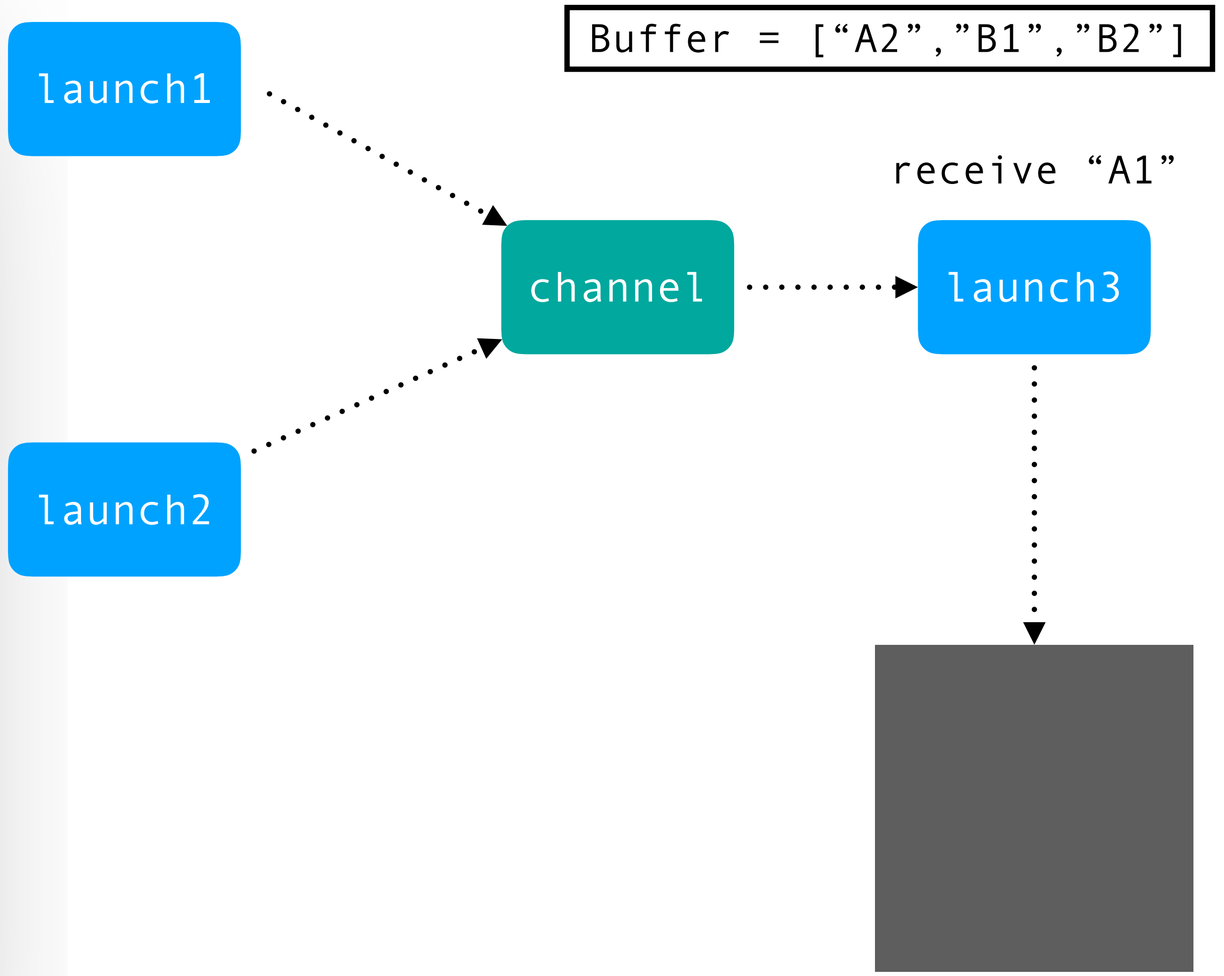
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



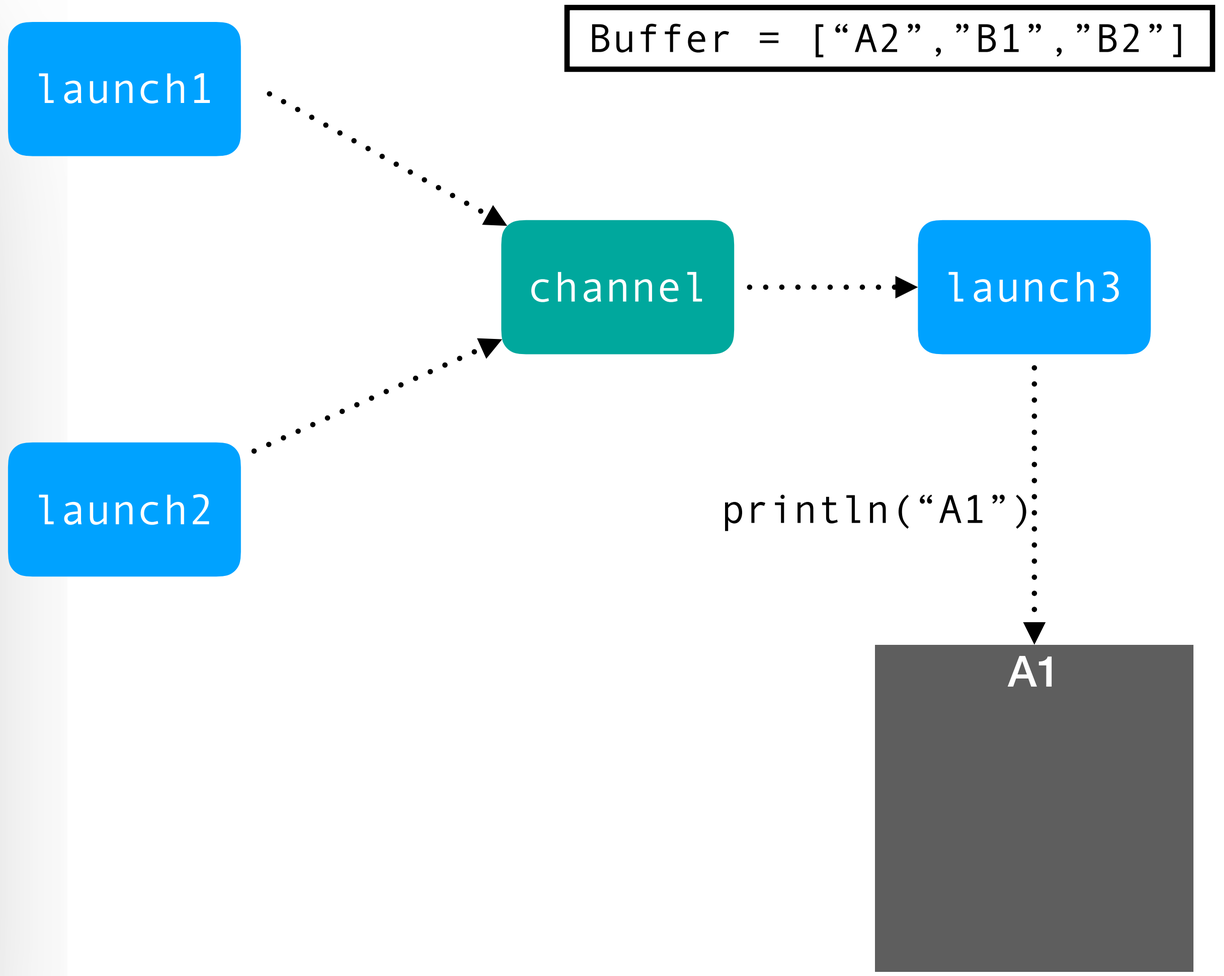
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



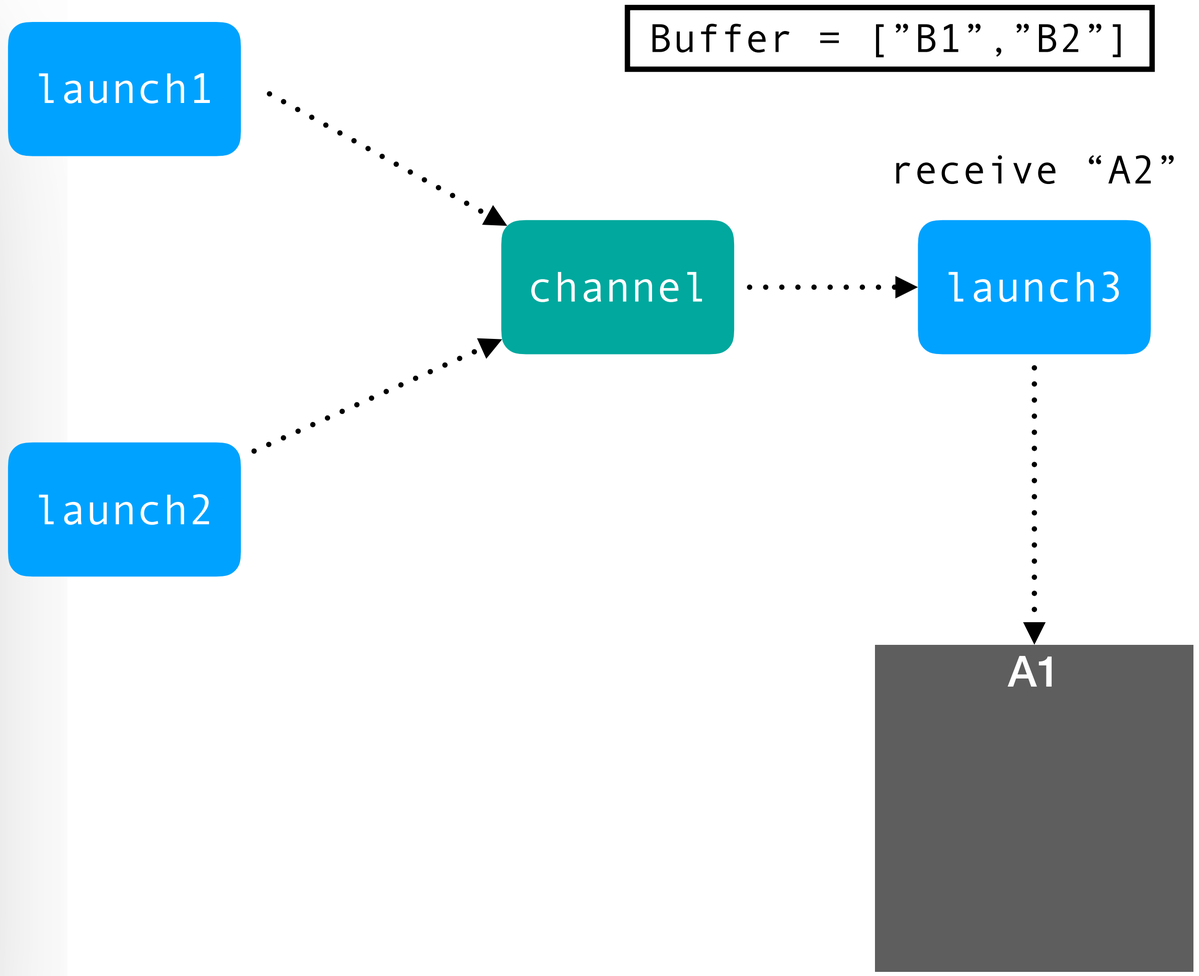
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



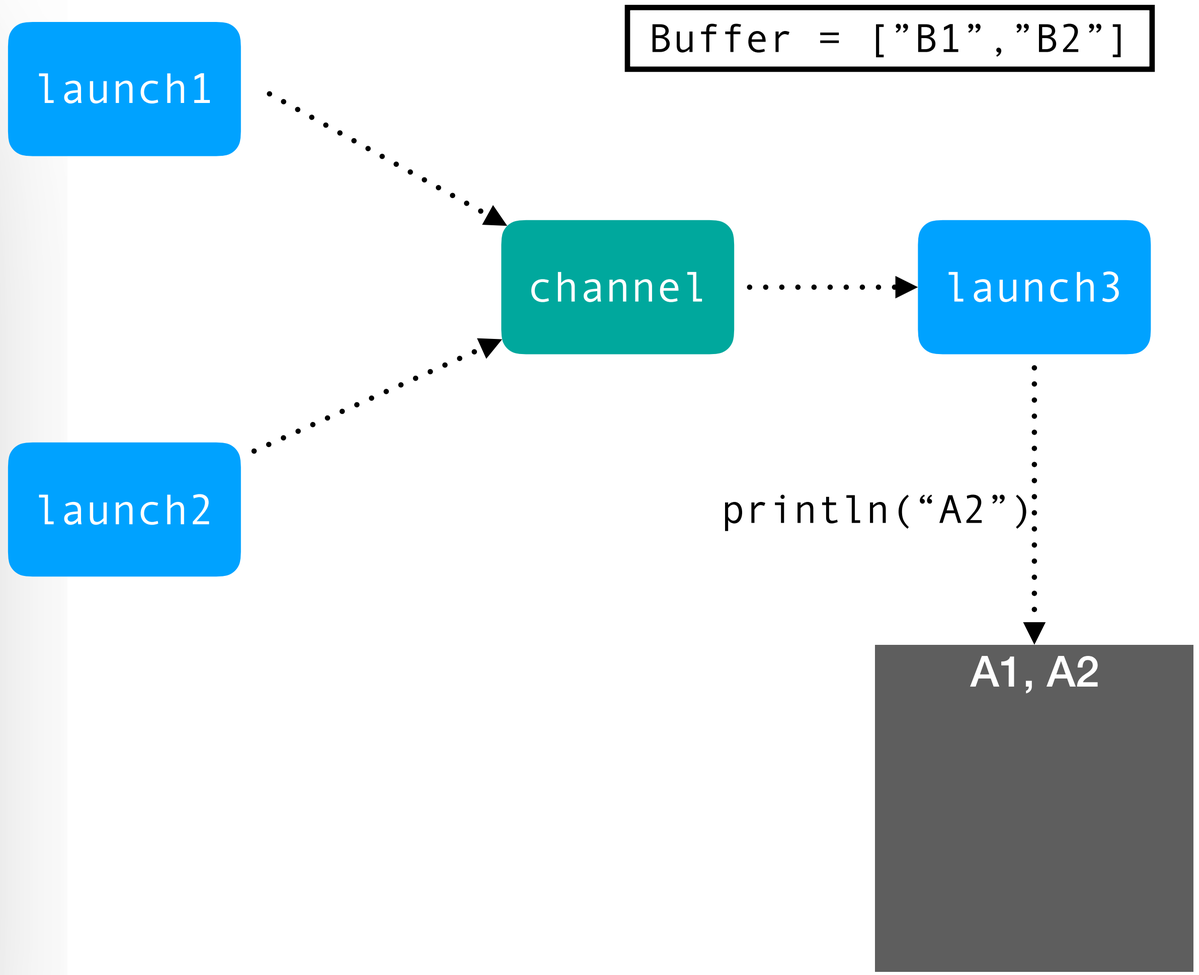
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



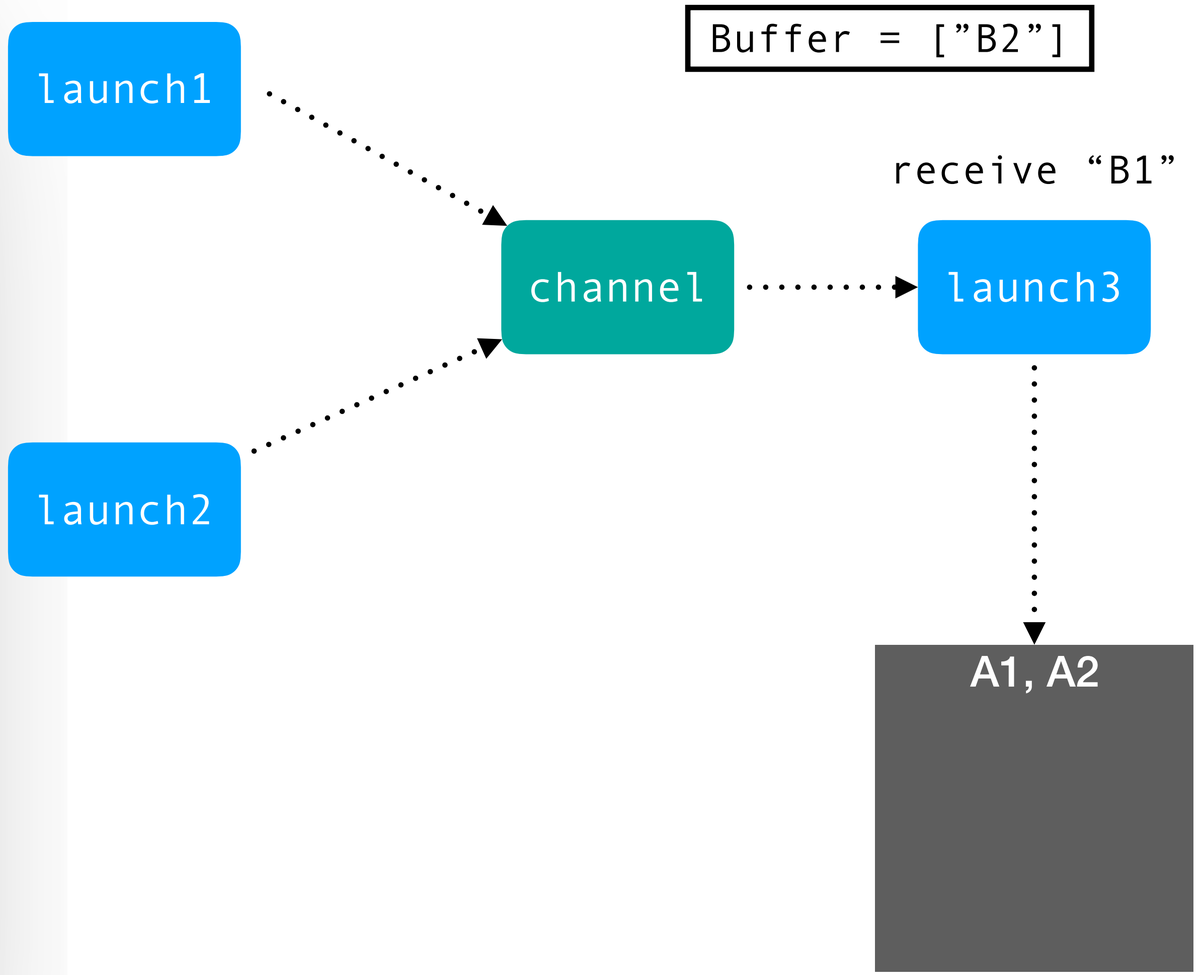
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



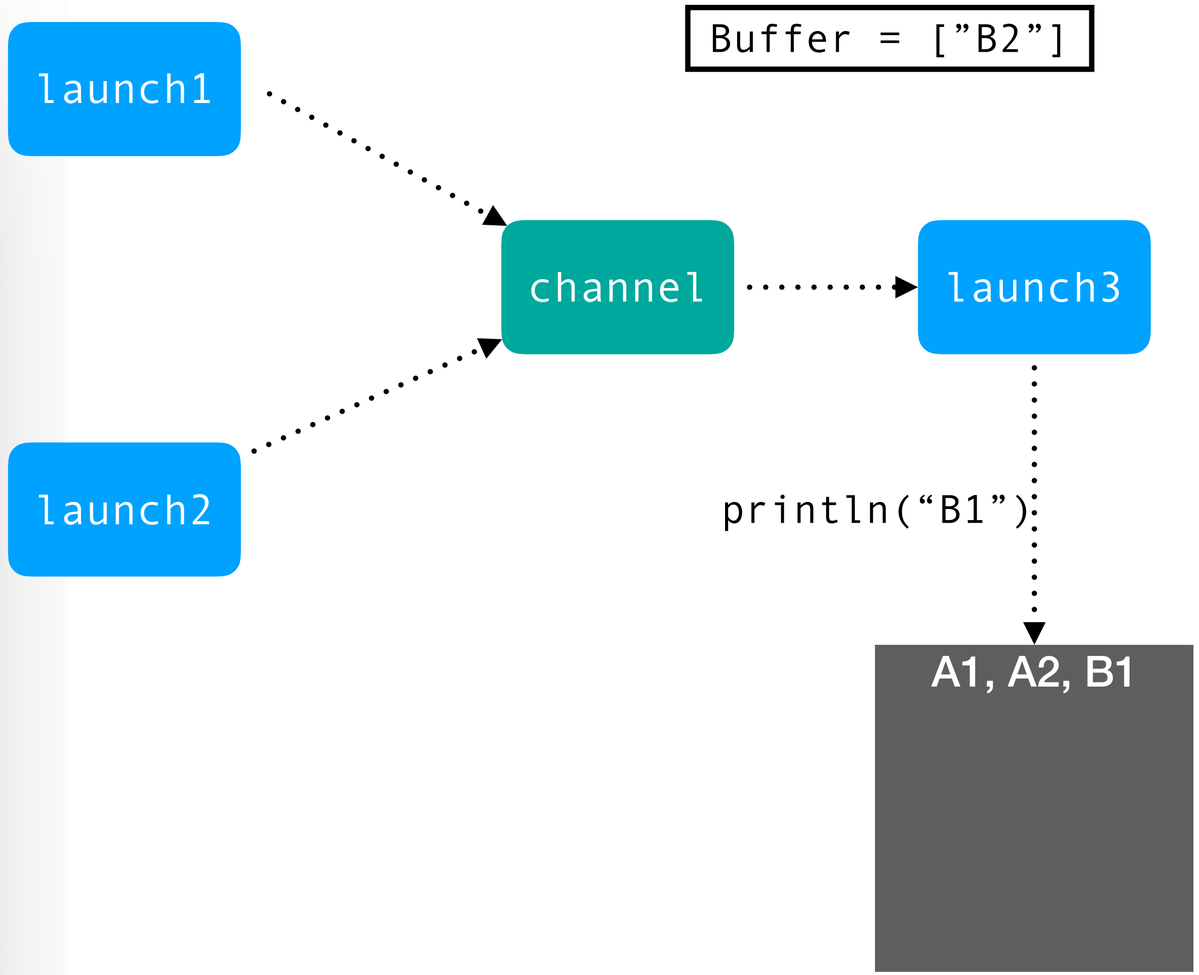
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



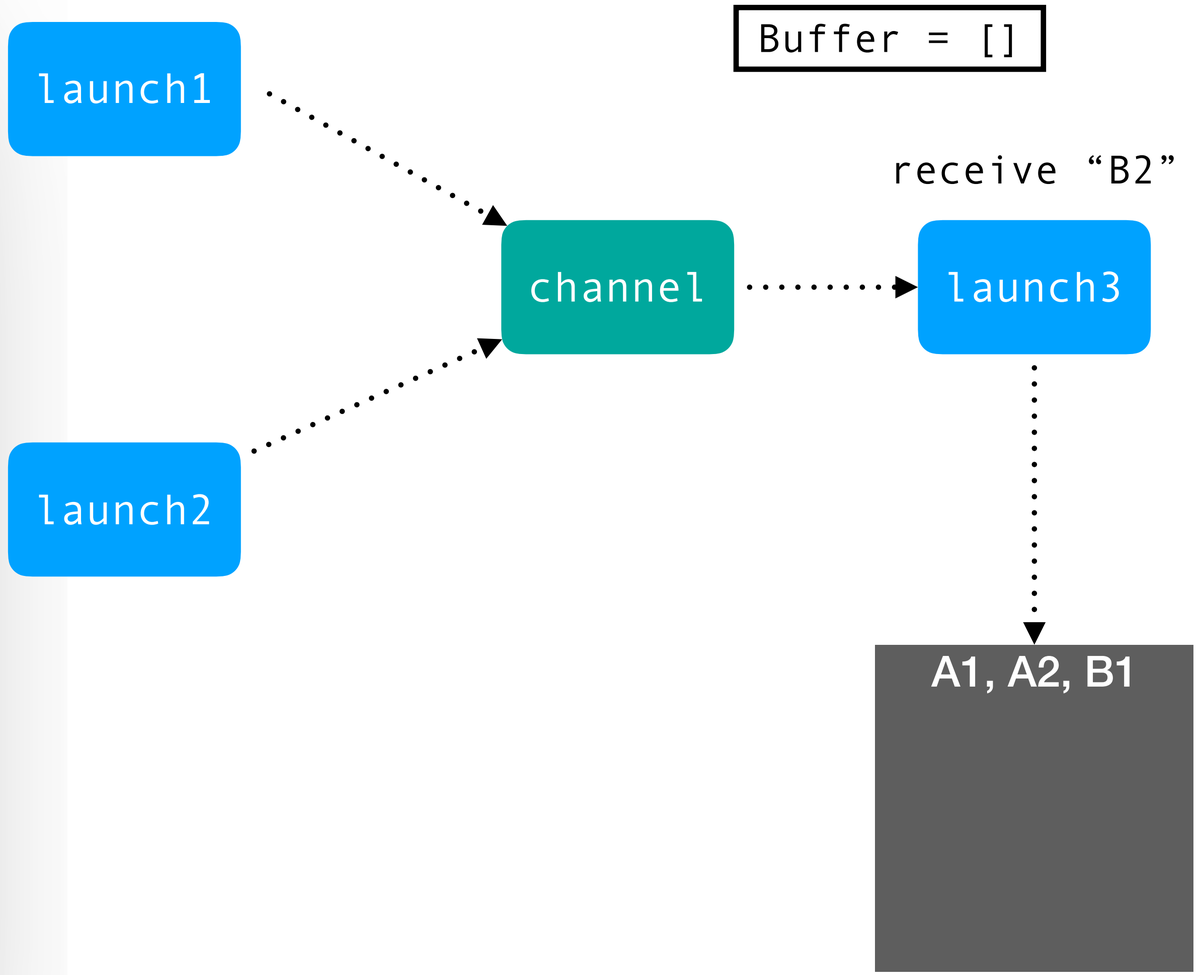
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



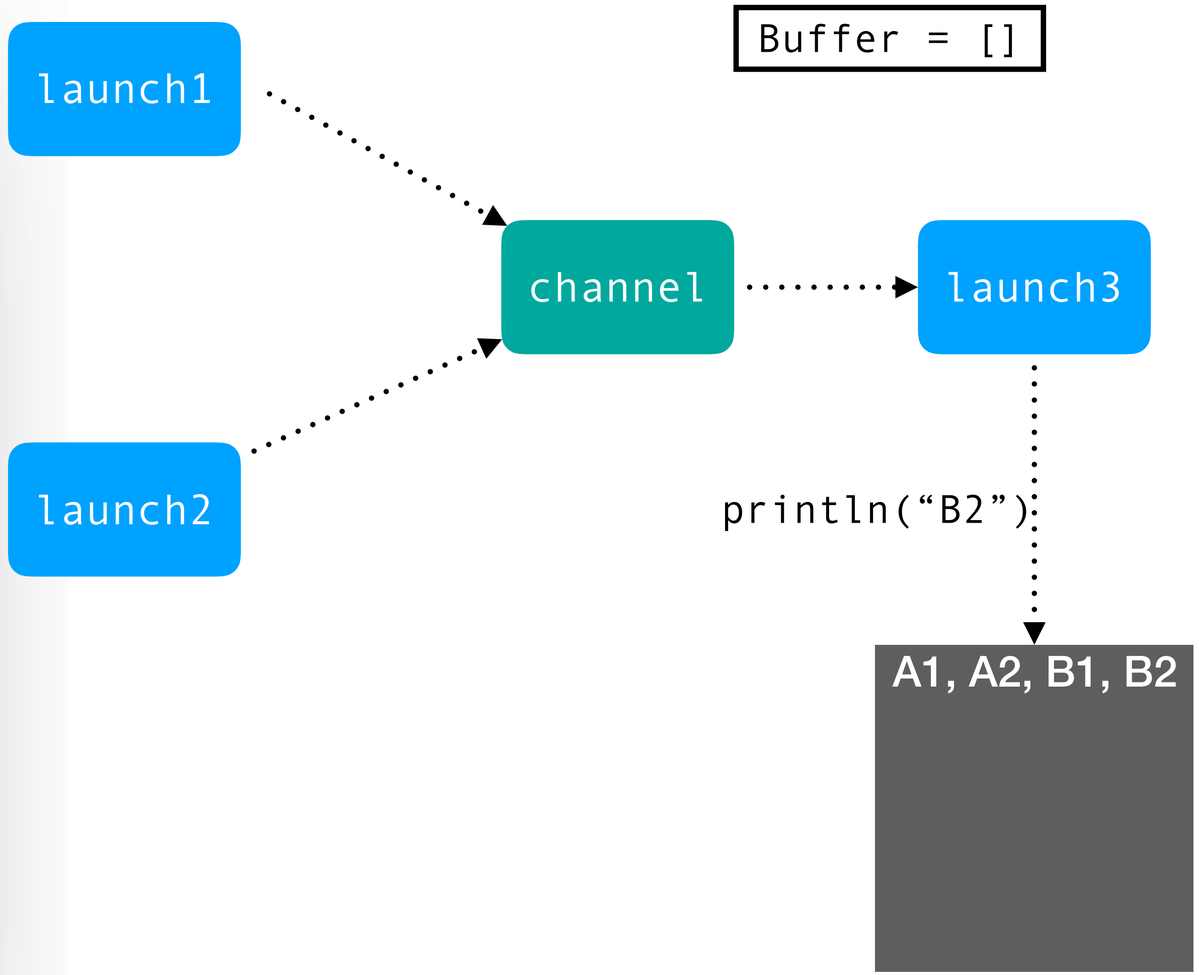
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



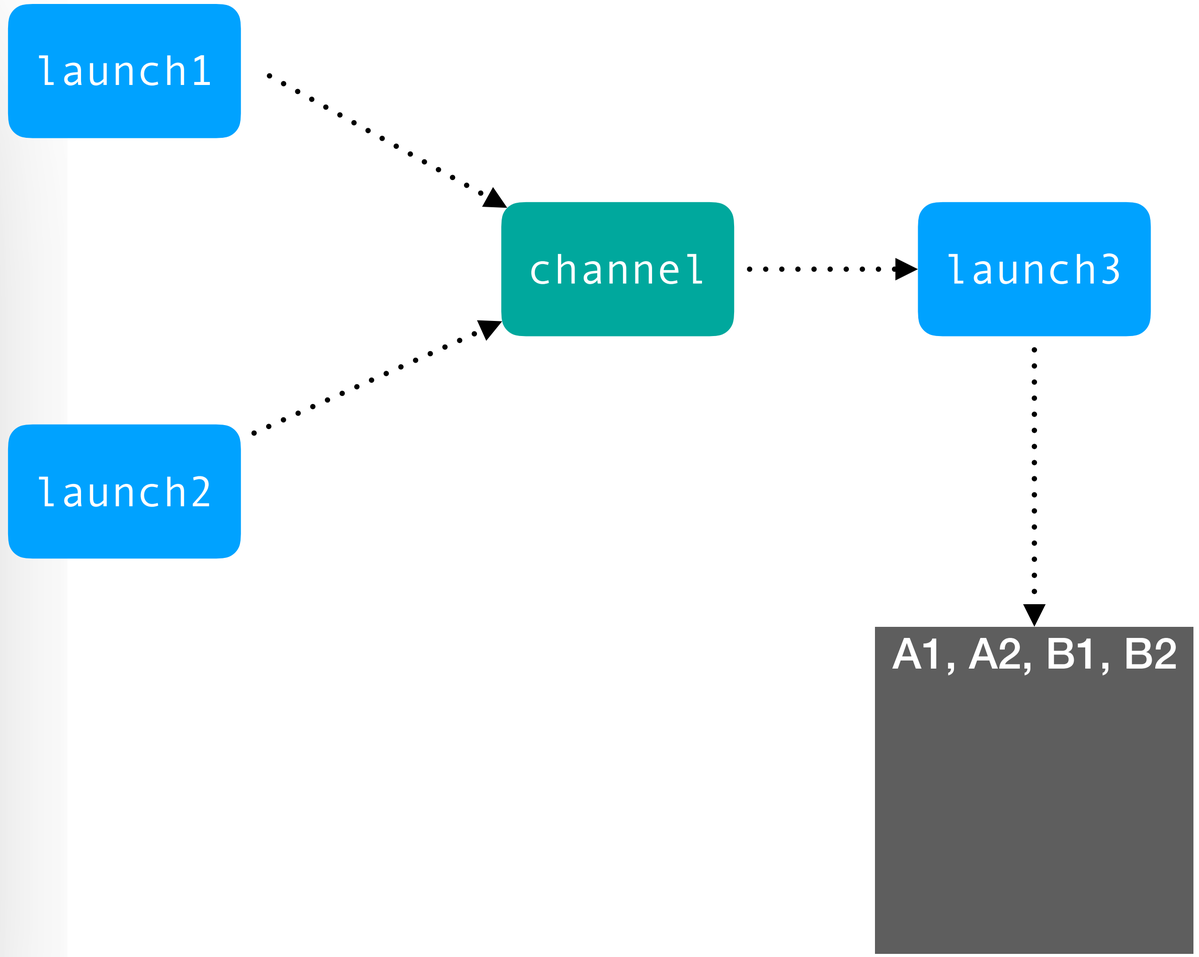
Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



Qual a saída esperada?


```
// BUFFERED (buffer = 4)
val channel = Channel<String>(4)
launch {
    channel.send("A1")
    channel.send("A2")
}
launch {
    channel.send("B1")
    channel.send("B2")
}
launch {
    repeat(4) {
        println(channel.receive())
    }
}
```



Channels podem ser fechados para indicar que não há mais elementos chegando.

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

launch {
    for (value in 1..5) channel.send(value)
    channel.close() // fim do envio
}

// Recebemos os itens até que o
// channel seja fechado
for (value in channel) println(value)
```

```
val channel = Channel<Int>(Channel.CONFLATED)
launch {
    for (value in 1..5) channel.send(value)
}
launch {
    println(channel.receive())
    println(channel.receive())
}
```

Channel Conflated **emite o item mais recente**

```
println(5)
```

ClosedReceiveChannelException: **Channel was closed**

Channel & Coroutines

Coroutine builder + channel + extension functions

Producers

```
/* CoroutineScope.produce */

val hwChannel = produce<String> {
    send("Hello")
    send("World!")
}

//ou

// especificando o contexto
val hwChannel = produce<String>(Dispatchers.IO) {
    send("Hello")
    send("World!")
}
```

Consumers

```
/* ReceiveChannel.consumeEach */

hwChannel.consumeEach { println(it) }

// ao invés de

for (value in hwChannel) println(value)
```

Algumas considerações...

1. ***Channels*** são ótimos para modelar **fontes de dados quentes**

2. Deve-se usar um ***channel*** quando precisar **enviar dados** de **uma *coroutine* para outra**

a) ***coroutines*** são concorrentes

b) precisamos de **sincronização** para trabalhar com qualquer dado na presença de **concorrência**.

Mas e se **não** precisarmos de concorrência ou sincronização, mas apenas de **fluxos de dados sem bloqueio**?

Seja bem-vindo ao Kotlin **Flow**!

PicPay

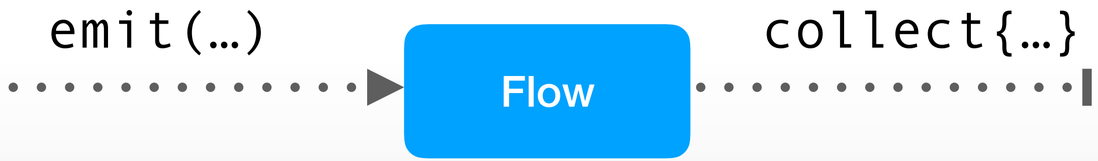
Um `flow` é um *stream* "frio" de valores

```
val myFlow: Flow<String> = flow {
    emit("Hello Streams")
}
```

`myFlow` é apenas a referência da instância do *Flow*

O código dentro do *flow{...}* não está ativo e ainda não há recursos vinculados a ele. 👍🏻

Flow possui duas operações principais




```kotlin
val myFlow: Flow<String> = flow { emit("Hello Streams") }

myFlow.collect { println(it) } // "Hello Streams"
```

Algumas maneiras de criar um Flow

```
flow{…}
flowOf(…)
.asFlow()
```

```kotlin
val flow1: Flow<Int> = (1..3).asFlow()

val flow2: Flow<Int> = flowOf(1, 2, 3)

val flow3: Flow<Int> = flow { for (i in 1..3) emit(i) }
```

Flows podem ser transformados usando alguns operadores

PicPay

Alguns listeners de "eventos"

```
// Executado antes do primeiro `emit`
.onStart {...}

// Executado junto com cada emissão
.onEach {...}

// Executado depois de todas as emissões
.onCompletion {...}

// Executado caso ocorra alguma exception
.catch {...}
```

PicPay

Coroutine context & Flow

Use `flowOn` para especificar o contexto de execução.

```
val myFlow = flow {
    emit("1.2")
}.flowOn(Dispatchers.IO)
.map {
    it.toFloat()
}.flowOn(Dispatchers.Default)
.map {
    "Value: $it"
}
```

PicPay

Coroutine context & Flow

Use `flowOn` para especificar o contexto de execução.

```
val myFlow = flow {
    emit("1.2")
}.flowOn(Dispatchers.IO)
.map {
    it.toFloat()
}.flowOn(Dispatchers.Default)
.map {
    "Value: $it"
}
```

Trecho executado com `Dispatchers.IO`

PicPay

Coroutine context & Flow

Use `flowOn` para especificar o contexto de execução.

```
val myFlow = flow {
    emit("1.2")
}.flowOn(Dispatchers.IO)
.map {
    it.toFloat()
}.flowOn(Dispatchers.Default)
.map {
    "Value: $it"
}
```

Trecho executado com `Dispatchers.Default`

PicPay

Coroutine context & Flow

Use `flowOn` para especificar o contexto de execução.

```
val myFlow = flow {
    emit("1.2")
}.flowOn(Dispatchers.IO)
.map {
    it.toFloat()
}.flowOn(Dispatchers.Default)
.map {
    "Value: $it"
}
```

Trecho executado no contexto de quem chamar o *collect*

PicPay

Exemplos

```
class MovieRepositoryImpl(...) : MovieRepository {

  override suspend fun getMovie(): Flow<Movie> {
    // do something ....
    return movieDataSource.getMovieData()
              .map { mapToMovie(it) }
              .filter { it.released }
              .onEach { saveMovie(it) }
              .flowOn(Dispatchers.IO)
  }
}
```

Exemplos

```
class MovieRepositoryImpl(...) : MovieRepository {

  override suspend fun getMovie(): Flow<Movie> {
    // do something ....
    return movieDataSource.getMovieData()
              .map { mapToMovie(it) }
              .filter { it.released }
              .onEach { saveMovie(it) }
              .flowOn(Dispatchers.IO)
  }
}
```

Exemplos

```
interface MovieUseCase {

  suspend fun getMovie(...): Flow<Movie>

}

...


movieUseCase.getMovie(...)
    .onStart { notifyMovieStarted() }
    .onCompletion { notifyMovieFinished() }
    .catch { notifyMovieError(it) }
    .collect { showMovie(it) }
```

Exemplos

```
interface MovieUseCase {

  suspend fun getMovie(...): Flow<Movie>

}

...
```

```
movieUseCase.getMovie(...)
    .onStart { notifyMovieStarted() }
    .onCompletion { notifyMovieFinished() }
    .catch { notifyMovieError(it) }
    .collect { showMovie(it) }
```

Wrapping callbacks com `callbackFlow`

```
fun View.clicks(): Flow<Unit> = callbackFlow {
    //
}
```

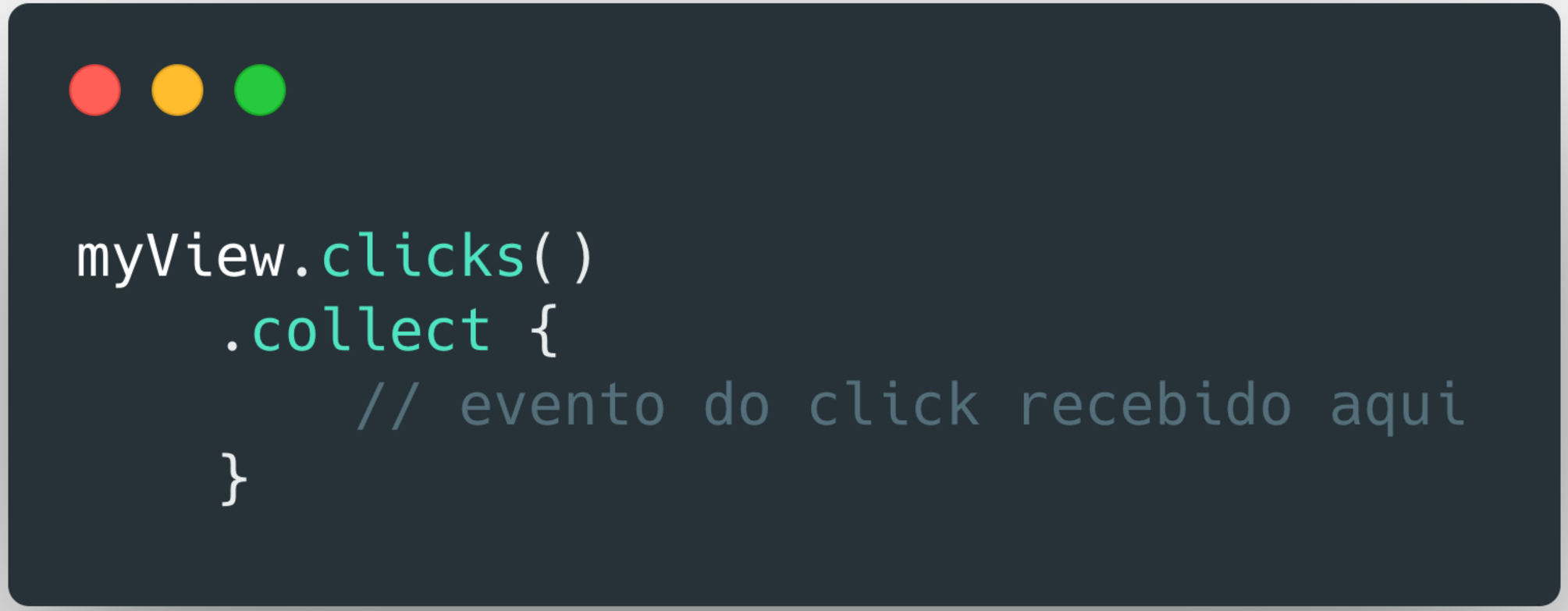
Wrapping callbacks com `callbackFlow`

```
fun View.clicks(): Flow<Unit> = callbackFlow {

    this@clicks.setOnClickListener { this.offer(Unit) }

}
```

Wrapping callbacks com `callbackFlow`

```
fun View.clicks(): Flow<Unit> = callbackFlow {
    this@clicks.setOnClickListener { this.offer(Unit) }
    awaitClose { this@clicks.setOnClickListener(null) }
}
```

Wrapping callbacks com `callbackFlow`



```
myView.clicks()
      .collect {
            // evento do click recebido aqui
      }
```

Wrapping callbacks com `callbackFlow`

```
myView.clicks()
      .debounce(300L)
      .collect {
          // evento do click recebido aqui
      }
```

Wrapping callbacks com `callbackFlow`

```
fun saveDocument(
    collectionId: String,
    docs: HashMap<String, String>
): Flow<DocumentReference> = callbackFlow {
    firestore.collection(collectionId)
        .add(docs)
        .addOnSuccessListener {
            offer(it)  // emite o item recebido em caso de sucesso
        }
        .addOnCanceledListener {
            cancel() // cancela sem enviar uma exception
        }
        .addOnFailureListener {
            // cancela e envia uma exception
            cancel(CancellationException("onFailureListener", it))
        }

    awaitClose { /* executar algo após a execução (dispose por exemplo) */ }
}
```

Considerações Finais ✍🏻

- Channel & *Flow* podem andar juntos
- Migração tranquila de *RxJava/RxKotlin/RxAndroid* para Channel & Flow
- Nova versão do Room e WorkManager já usam *Flow*
- API ainda Experimental

PicPay

Dúvidas? 🤔

PicPay

Obrigado!

Jeziel Lago

Android Developer | PicPay

/jeziellago

@jeziellago

PicPay

Links

http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html

https://medium.com/@elizarov/reactive-streams-and-kotlin-flows-bfd12772cda4

https://medium.com/@elizarov/simple-design-of-kotlin-flow-4725e7398c4c

https://medium.com/@elizarov/execution-context-of-kotlin-flows-b8c151c9309b

https://medium.com/@elizarov/cold-flows-hot-channels-d74769805f9#9f07

https://proandroiddev.com/what-is-concurrent-access-to-mutable-state-f386e5cb8292

https://play.kotlinlang.org/hands-on/Introduction%20to%20Coroutines%20and%20Channels/01_Introduction

https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow/