Asynchronous data streams with kotlin flow

What is a flow in Kotlin?

1. It's a kotlin language feature that serves as a reactive programming framework
2. It's all about being notified about changes in your code and sending them through a pipeline that potentially modifies them
3. A flow is a coroutine that can emit multiple values over a period of time

Callback hell before

```
fun requestTokenAsync(): Promise<Token> { … }
fun createPostAsync(token: Token, item: Item): Promise<Post> …
fun processPost(post: Post) { … }
```

```
fun postItem(item: Item) {
    requestTokenAsync()
        .thenCompose { token -> createPostAsync(token, item) }
        .thenAccept { post -> processPost(post) }
}
```

Direct style with Kotlin Coroutines

```
suspend fun requestToken(): Token { … }
suspend fun createPost(token: Token, item: Item): Post { … }
fun processPost(post: Post) { … }
```

```
suspend fun postItem(item: Item) {
    val token = requestToken()
    val post = createPost(token, item)
    processPost(post)
}
```

Like *regular* code

Direct style with Kotlin Coroutines

```
suspend fun requestToken(): Token { … }
suspend fun createPost(token: Token, item: Item): Post { … }
fun processPost(post: Post) { … }
```

```
suspend fun postItem(item: Item) {
    val token = requestToken()
    val post = createPost(token, item)
    processPost(post)
}
```

Asynchronous yet sequential

```
suspend fun requestToken(): Token { … }
suspend fun createPost(token: Token, item: Item): Post { … }
fun processPost(post: Post) { … }
```

```
suspend fun postItem(item: Item) {
    val token = requestToken()
    val post = createPost(token, item)
    processPost(post)
}
```

Asynchronous yet sequential

```
suspend fun requestToken(): Token { … }
suspend fun createPost(token: Token, item: Item): Post { … }
fun processPost(post: Post) { … }
```

```
suspend fun postItem(item: Item) {
    val token = requestToken()
    val post = createPost(token, item)
    processPost(post)
}
```

Asynchronous yet sequential

```
suspend fun requestToken(): Token { … }
suspend fun createPost(token: Token, item: Item): Post { … }
fun processPost(post: Post) { … }
```

```
suspend fun postItem(item: Item) {
    val token = requestToken()
    val post = createPost(token, item)
    processPost(post)
}
```

```
suspend fun foo(): Response
```

One response

```
suspend fun foo(): List<Response>
```

Many responses

```
suspend fun foo(): List<Response>
```

```
suspend fun foo(): List<Response> = buildList {
    …
}
```

```
suspend fun foo(): List<Response> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```

```
suspend fun foo(): List<Response> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```

```
fun main() = runBlocking {
    val list = foo()
    for (x in list) println(x)
}
```

```
suspend fun foo(): List<Response> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```

```
fun main() = runBlocking {
    val list = foo()
    for (x in list) println(x)
}
```

```
suspend fun foo(): List<Response> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```

```
fun main() = runBlocking {
    val list = foo()
    for (x in list) println(x)
}
```

```
suspend fun foo(): List<Response> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```

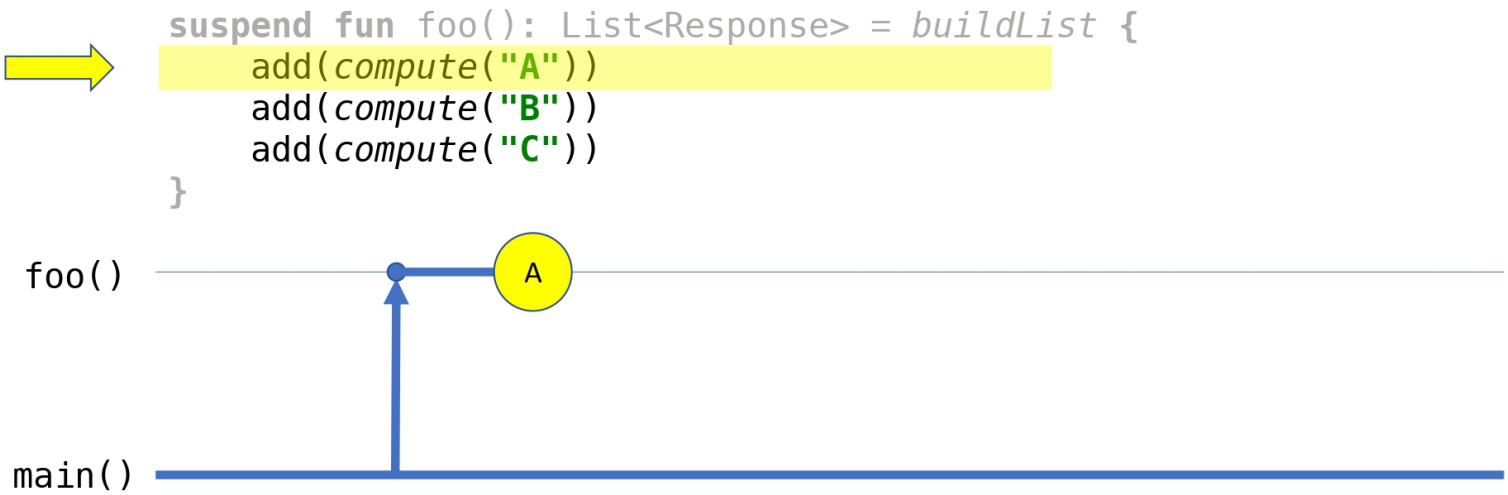
foo()

main()

```
fun main() = runBlocking {
    val list = foo()
    for (x in list) println(x)
}
```

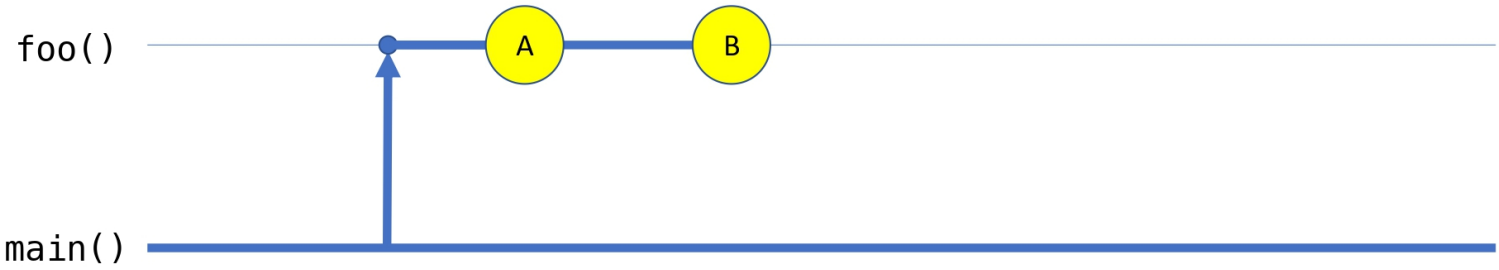
```
suspend fun foo(): List<Response> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```


```
fun main() = runBlocking {
    val list = foo()
    for (x in list) println(x)
}
```



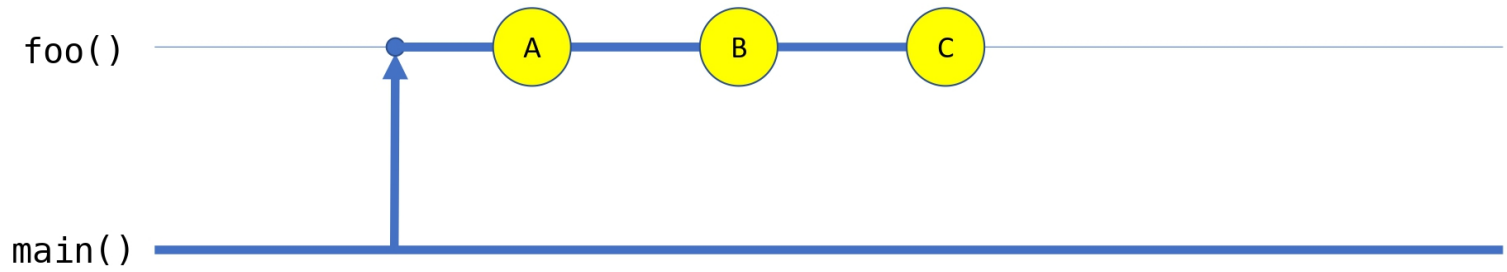

```
fun main() = runBlocking {
    val list = foo()
    for (x in list) println(x)
}
```

```
suspend fun foo(): List<Response> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```



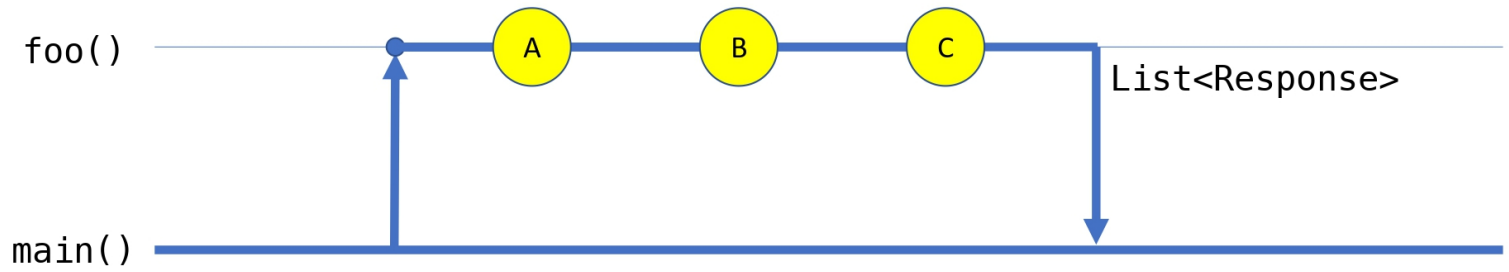

```
fun main() = runBlocking {
    val list = foo()
    for (x in list) println(x)
}
```

```
suspend fun foo(): List<Response> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```



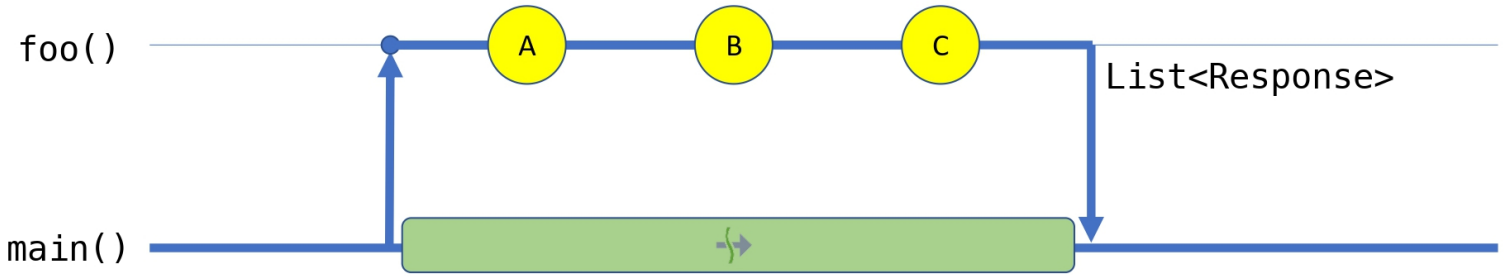

```
fun main() = runBlocking {
    val list = foo()
    for (x in list) println(x)
}
```

```
suspend fun foo(): List<Response> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```



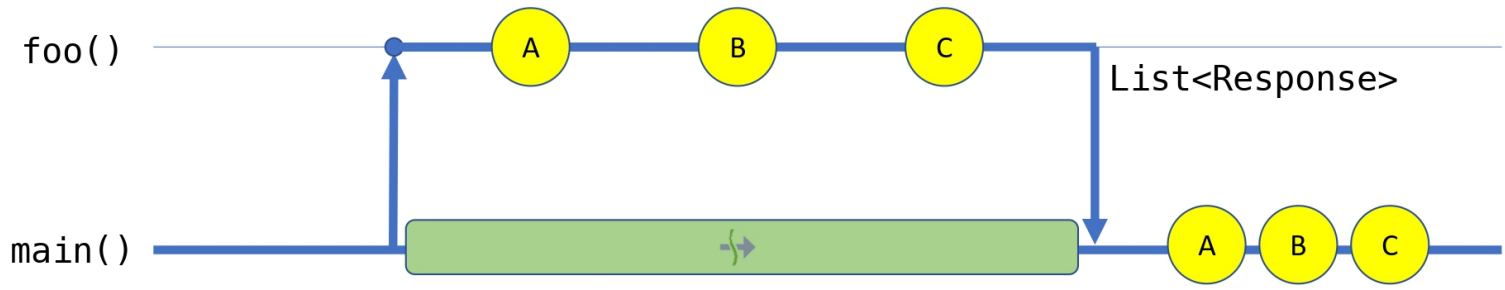

```
fun main() = runBlocking {
    val list = foo()
    for (x in list) println(x)
}
```

```
suspend fun foo(): List<Response> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```



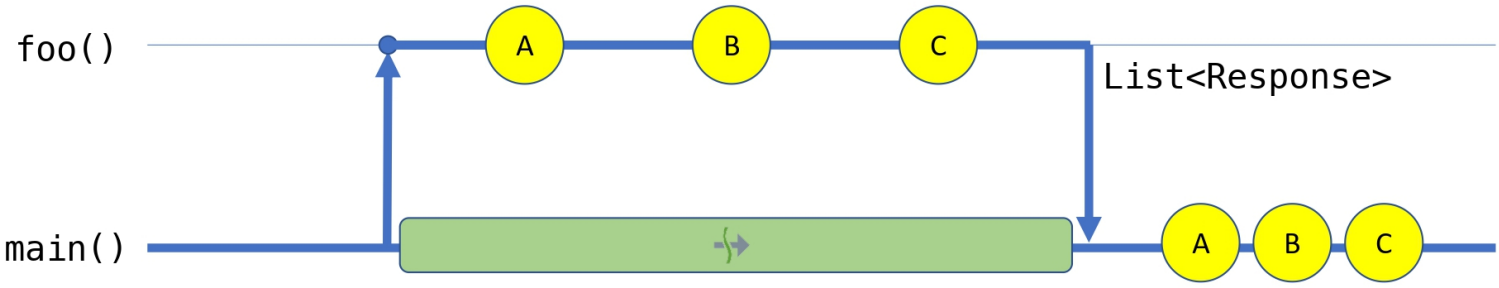

```
fun main() = runBlocking {
    val list = foo()
    for (x in list) println(x)
}
```

```
suspend fun foo(): List<Response> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```




```
fun main() = runBlocking {
    val list = foo()
    for (x in list) println(x)
}
```

```
suspend fun foo(): List<Response> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```




```
fun main() = runBlocking {
    val list = foo()
    for (x in list) println(x)
}
```

Channel

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    …
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```

```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```

```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```

```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

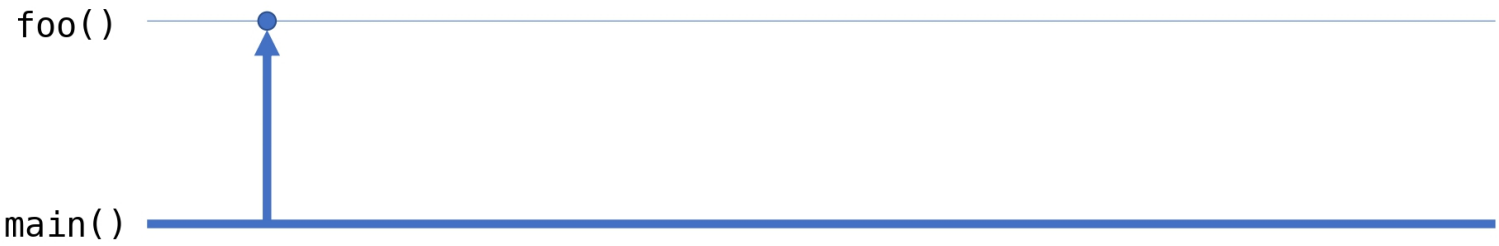
```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```

foo()

main()

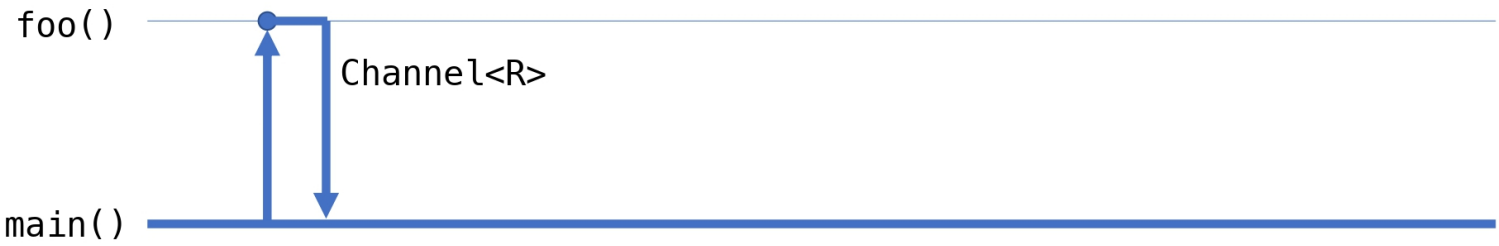
```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```




```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```




```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```


```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

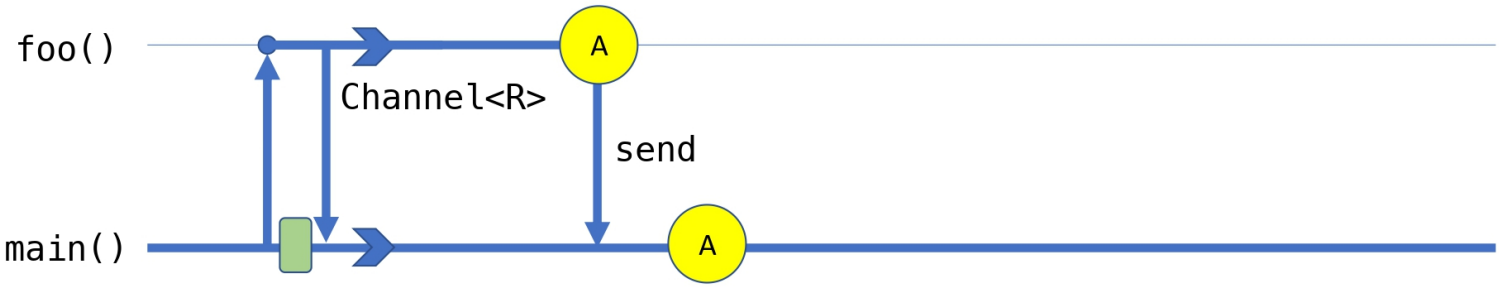
```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```


```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```

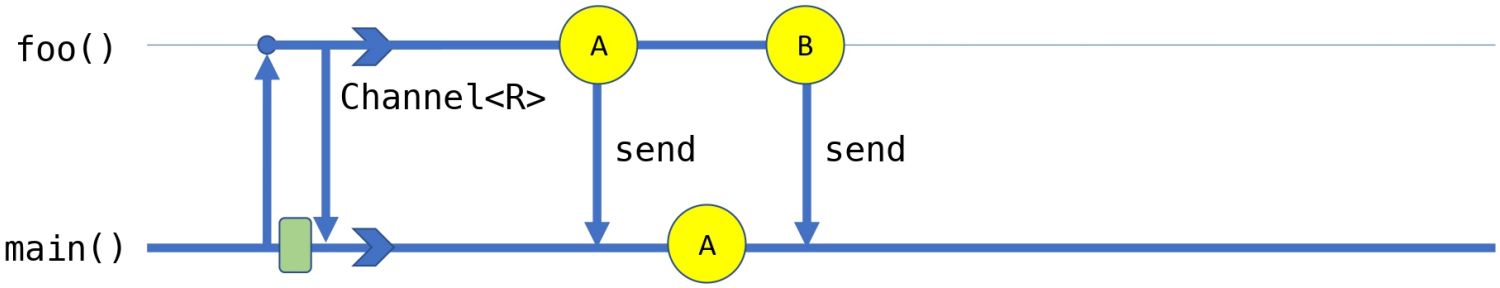

```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```



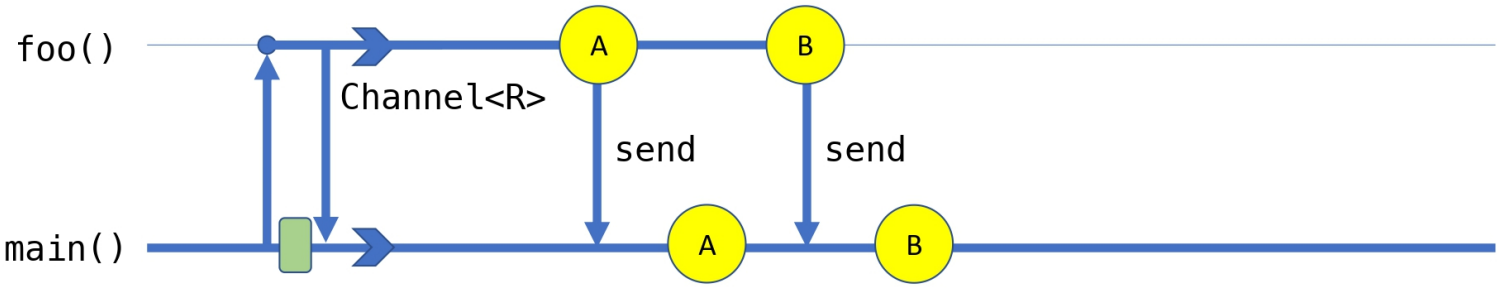

```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```



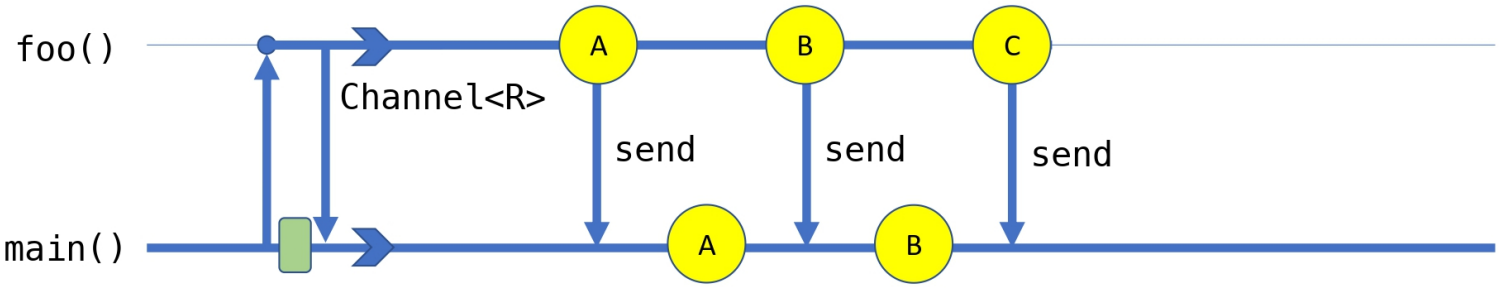

```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```



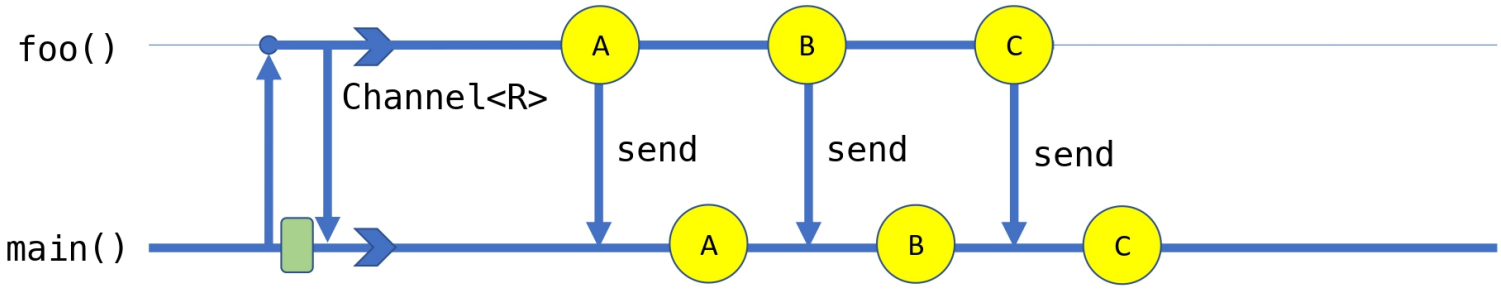

```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```



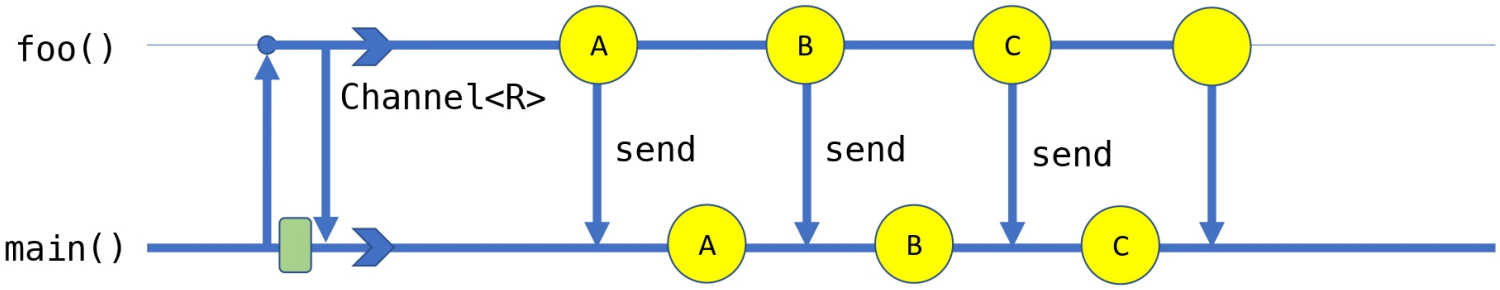

```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```



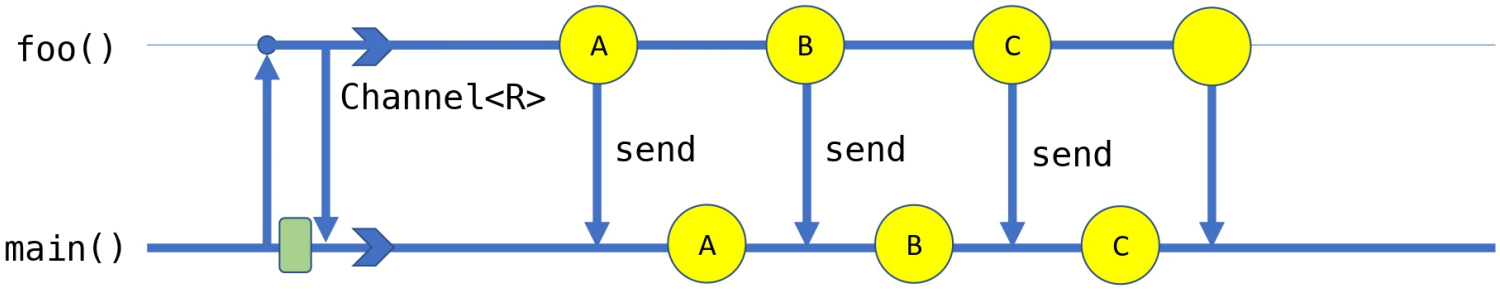

```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```



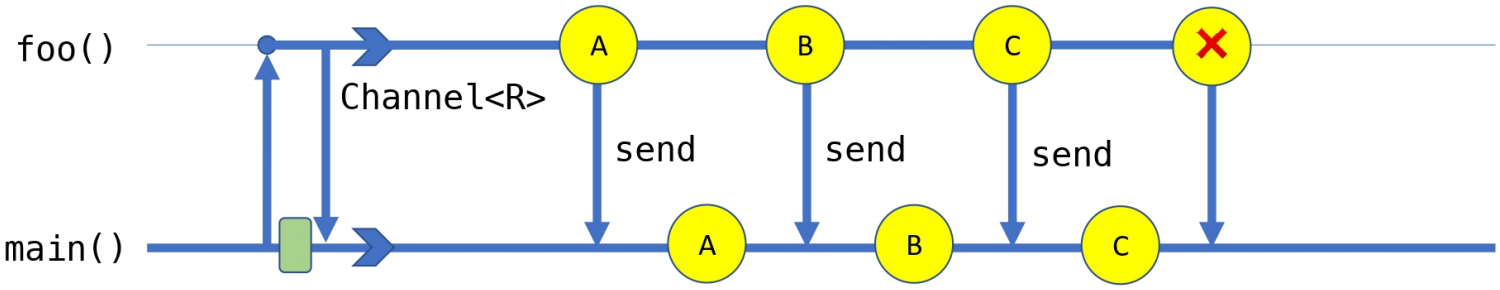

```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```




```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```




```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

Channel is hot 🔥

Channel is hot 🔥

```
fun main() = runBlocking {
    val channel = foo()
    for (x in channel) println(x)
}
```

Channel is hot 🔥

```
fun main() = runBlocking {
    val channel = foo()
    // for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```


```
fun main() = runBlocking {
    val channel = foo()
    // for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response> = produce {
    send(compute("A"))
    send(compute("B"))
    send(compute("C"))
}
```


```
fun main() = runBlocking {
    val channel = foo()
    // for (x in channel) println(x)
}
```

```
fun CoroutineScope.foo(): ReceiveChannel<Response>
```

Kotlin Flow

```
fun foo(): Flow<Response> = flow {
    ...
}
```

```
fun foo(): Flow<Response> = flow {
    ...
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

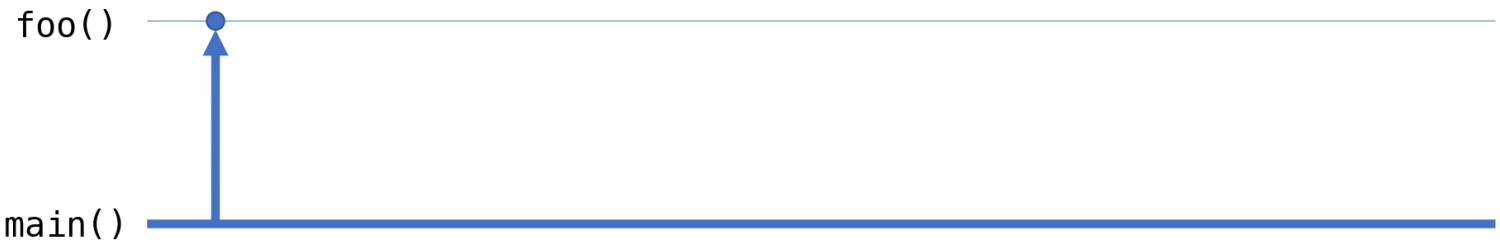
```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```

foo()

main()

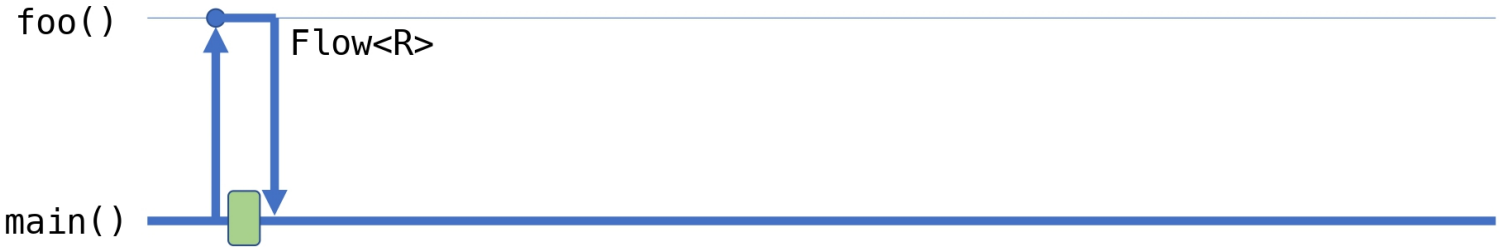
```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```



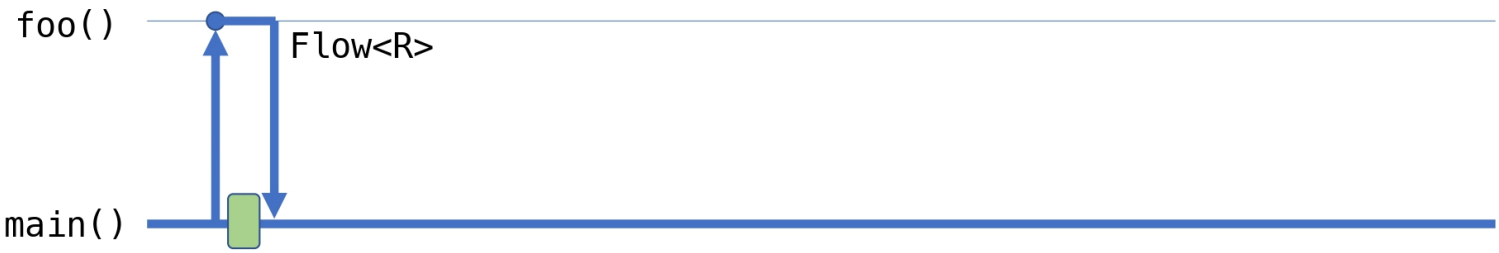

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```



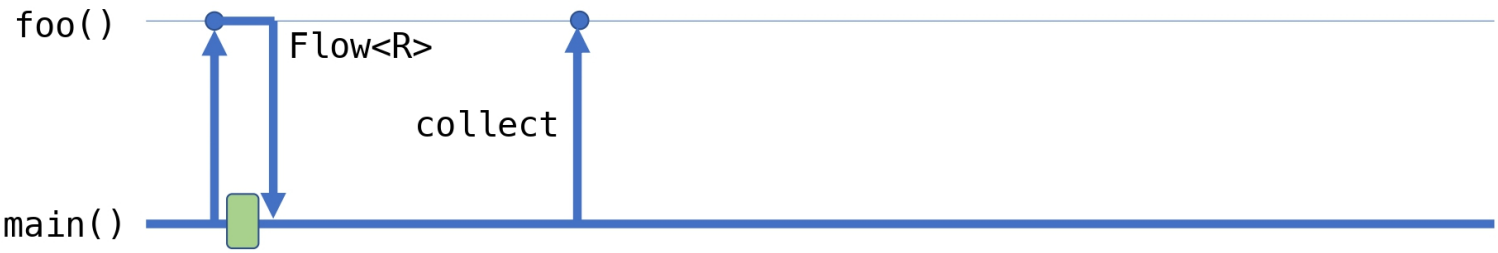

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```

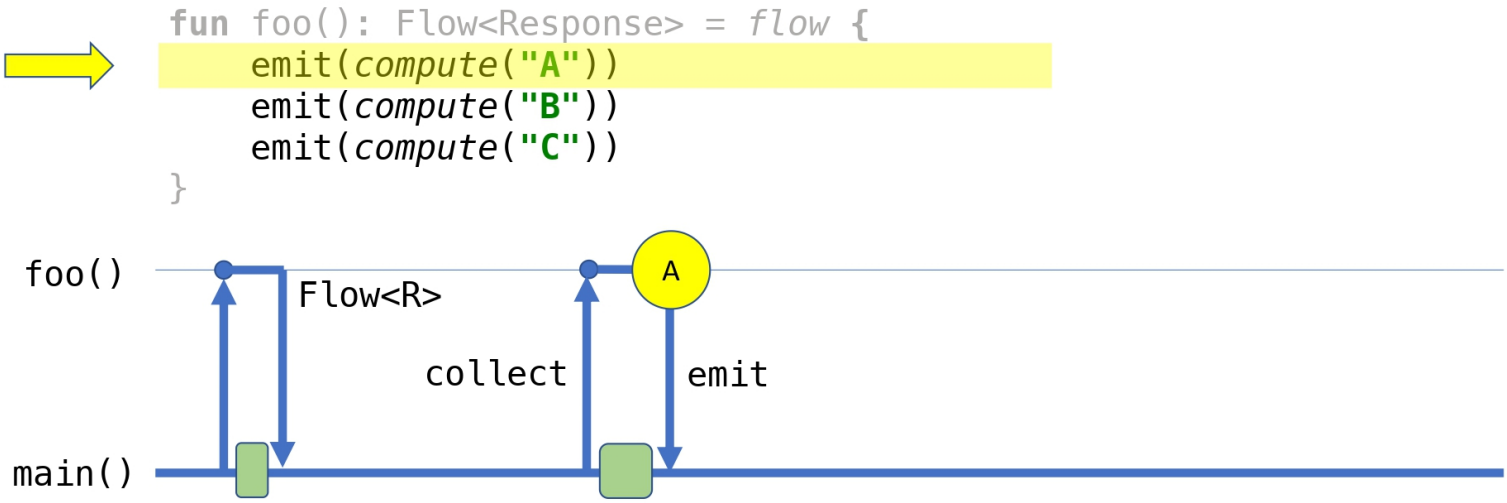



```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```

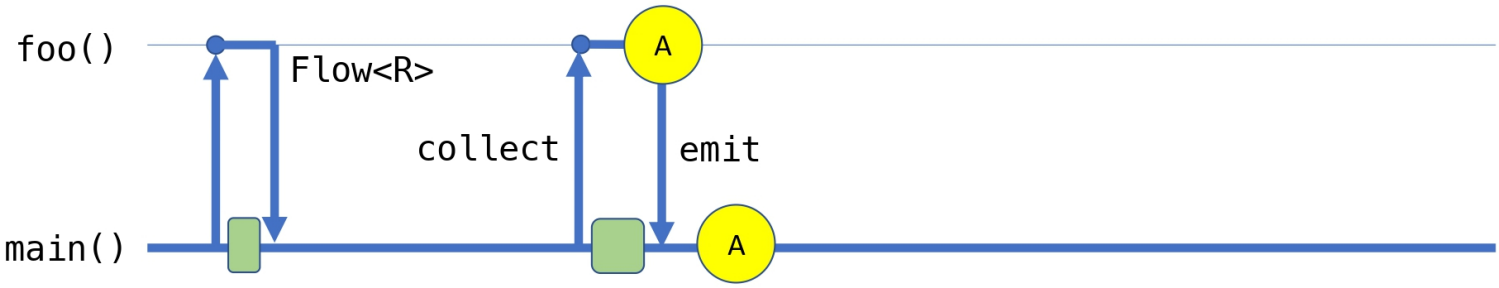



```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```



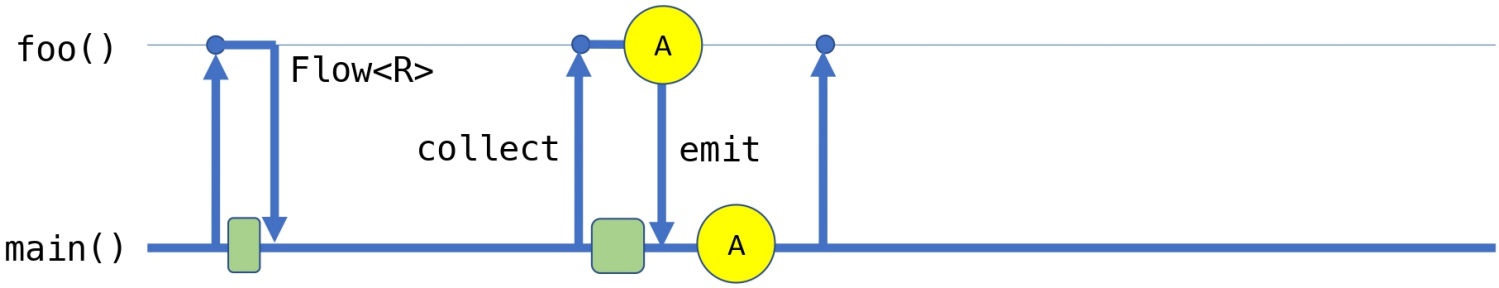

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```



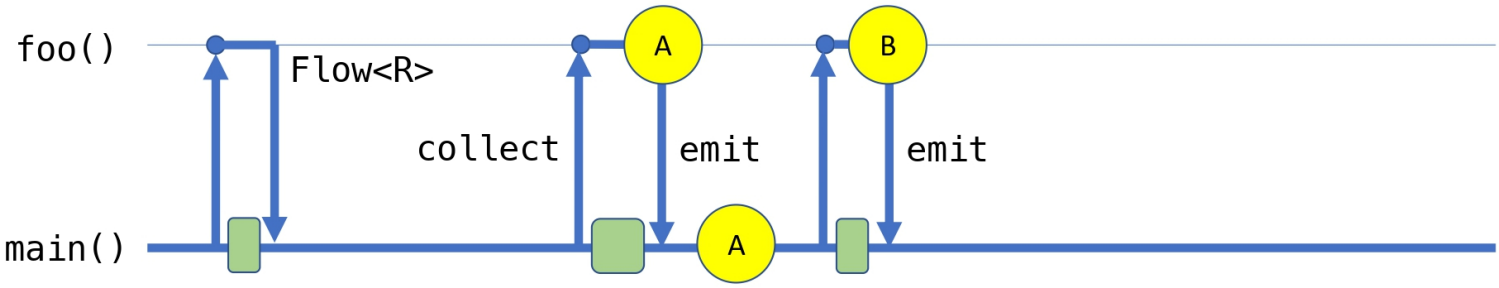

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```



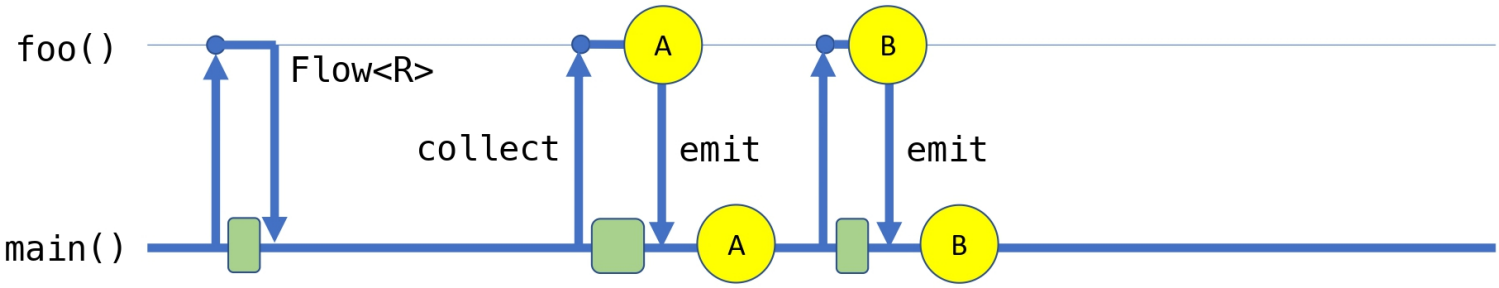

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```



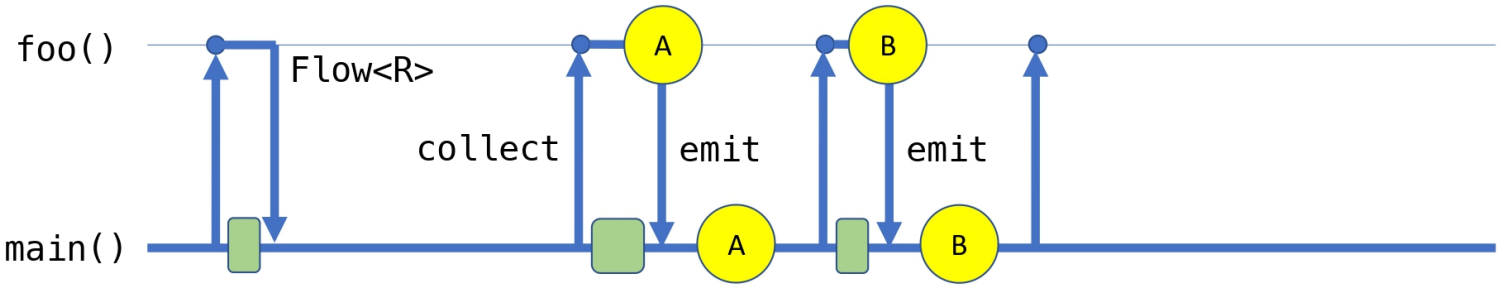

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```



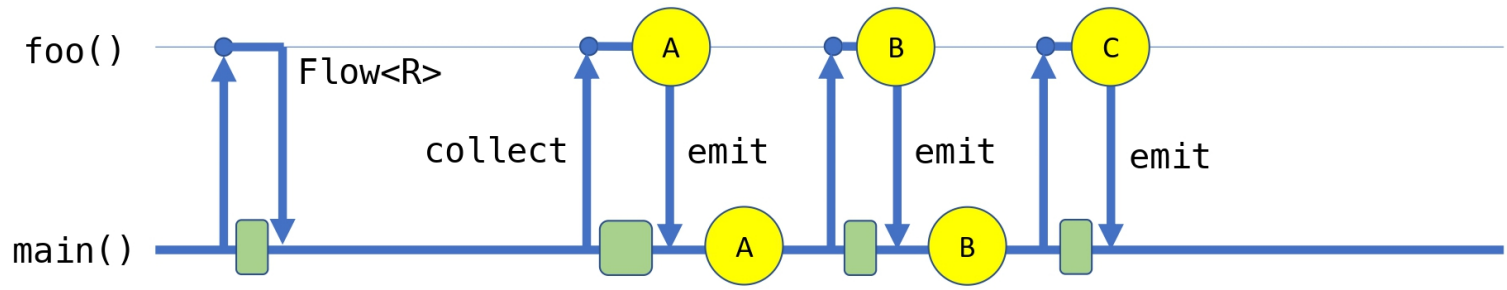

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```



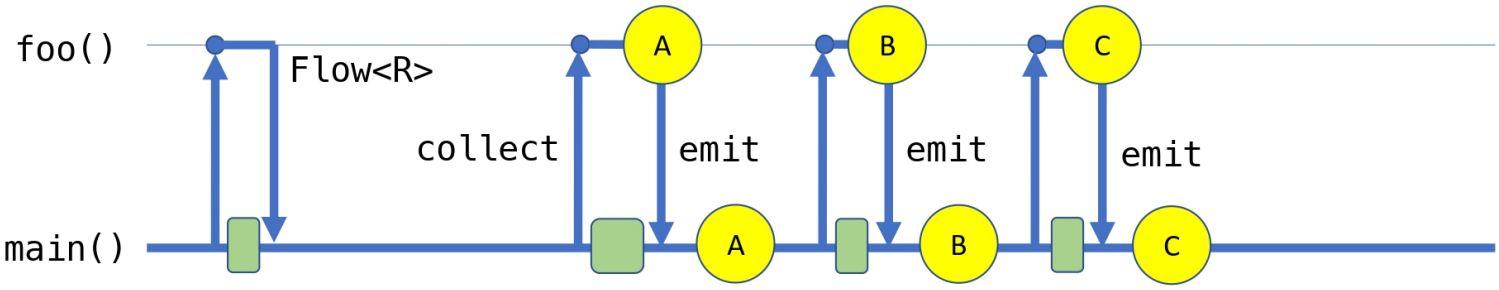

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```



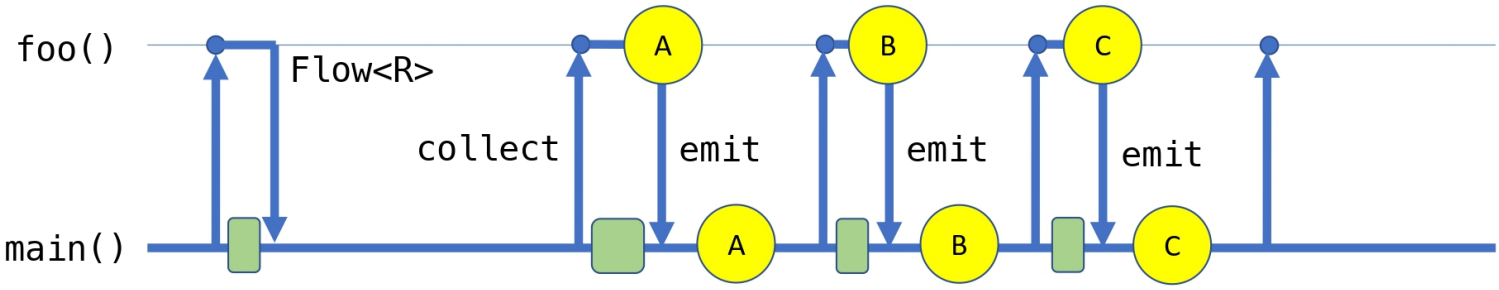

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```



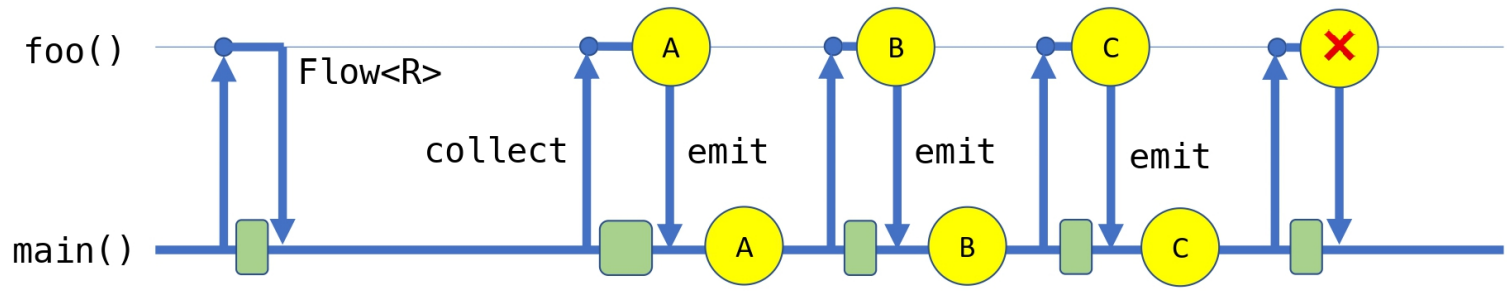

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```



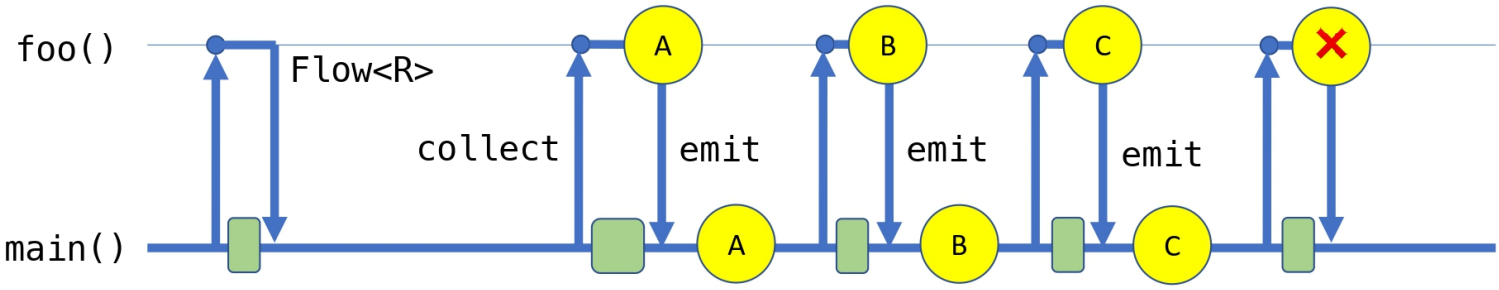

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```




```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```




```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

Flow is cold ❄️

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

Flow is cold ❄️

```
fun main() = runBlocking {
    val flow = foo()
    // flow.collect { x -> println(x) }  😀
}
```

Flow is declarative

Flow is declarative

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```

Declaration

```
fun strings(): Flow<String> = flow {
    emit("A")
    emit("B")
    emit(”C")
}
```

```
fun strings(): Flow<String> = flow {
    …
}

fun foo(): Flow<Response> =
    strings().map { name ->
        compute(name)
    }
```

```
fun strings(): Flow<String> = flow {
    …
}

fun foo(): Flow<Response> =
    strings().map { name ->
        compute(name)
    }
```

```
fun strings(): Flow<String> = flow {
    …
}

fun foo(): Flow<Response> =
    strings().map { name ->
        compute(name)
    }
```

```
fun strings(): Flow<String> = flow {
    …
}

fun foo(): Flow<Response> =
    strings().map { name ->
        compute(name)
    }
```

Operators

```
fun foo(): Flow<Response> =
    flowOf("A", "B", "C").map { name ->
        compute(name)
    }
```

Operators

Flow vs List

```
suspend fun foo(): List<Response> =
    listOf("A", "B", "C").map { name ->
        compute(name)
    }
```

```
fun foo(): Flow<Response> =
    flowOf("A", "B", "C").map { name ->
        compute(name)
    }
```

Flow vs List

```
suspend fun foo(): List<Response> =
    listOf("A", "B", "C").map { name ->
        compute(name)
    }
```

Runs – Imperative

```
fun foo(): Flow<Response> =
    flowOf("A", "B", "C").map { name ->
        compute(name)
    }
```

Defined – Declarative

```
suspend fun <T> Flow<T>.collect(…)
```

Runs the flow

Flow vs List

```
suspend fun foo(): List<Response> =
    listOf("A", "B", "C").map { name ->
        compute(name)
    }
```

Runs – Imperative

```
fun foo(): Flow<Response> =
    flowOf("A", "B", "C").map { name ->
        compute(name)
    }
```

Defined – Declarative

```
suspend fun <T> Flow<T>.toList(): List<T>
```

Runs the flow

Flow vs List

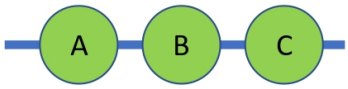
Execution order

```
suspend fun foo(): List<Response> =
    listOf("A", "B", "C").map { name ->
        compute(name)
    }
```

```
fun foo(): Flow<Response> =
    flowOf("A", "B", "C").map { name ->
        compute(name)
    }
```

Flow vs List

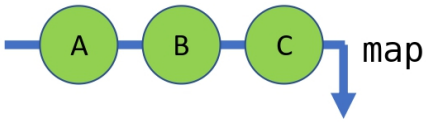
```
suspend fun foo(): List<Response> =
    listOf("A", "B", "C").map { name ->
        compute(name)
    }
```



```
fun foo(): Flow<Response> =
    flowOf("A", "B", "C").map { name ->
        compute(name)
    }
```

Flow vs List

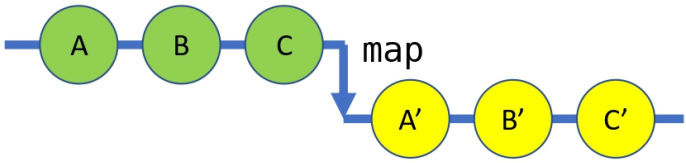
```
suspend fun foo(): List<Response> =
    listOf("A", "B", "C").map { name ->
        compute(name)
    }
```



```
fun foo(): Flow<Response> =
    flowOf("A", "B", "C").map { name ->
        compute(name)
    }
```

Flow vs List

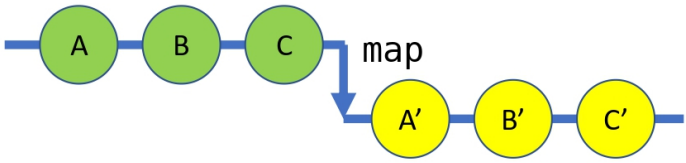
```
suspend fun foo(): List<Response> =
    listOf("A", "B", "C").map { name ->
        compute(name)
    }
```



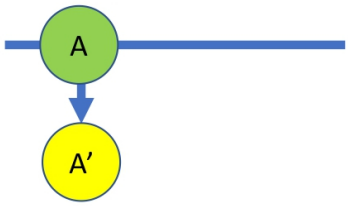

```
fun foo(): Flow<Response> =
    flowOf("A", "B", "C").map { name ->
        compute(name)
    }
```

Flow vs List

```
suspend fun foo(): List<Response> =
    listOf("A", "B", "C").map { name ->
        compute(name)
    }
```

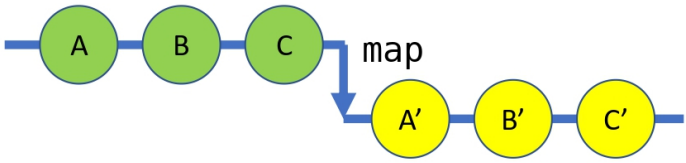



```
fun foo(): Flow<Response> =
    flowOf("A", "B", "C").map { name ->
        compute(name)
    }
```

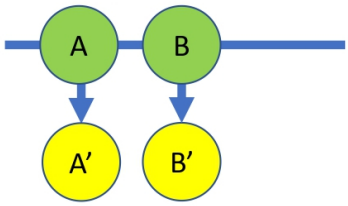


Flow vs List

```
suspend fun foo(): List<Response> =
    listOf("A", "B", "C").map { name ->
        compute(name)
    }
```

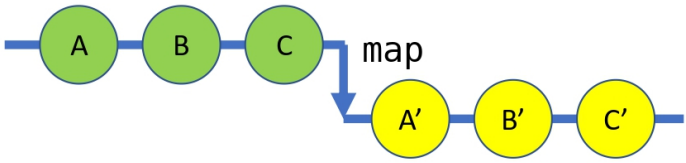



```
fun foo(): Flow<Response> =
    flowOf("A", "B", "C").map { name ->
        compute(name)
    }
```



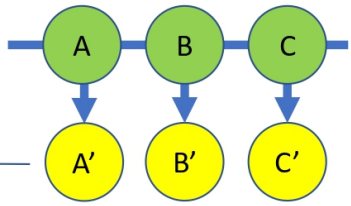
Flow vs List

```
suspend fun foo(): List<Response> =
    listOf("A", "B", "C").map { name ->
        compute(name)
    }
```

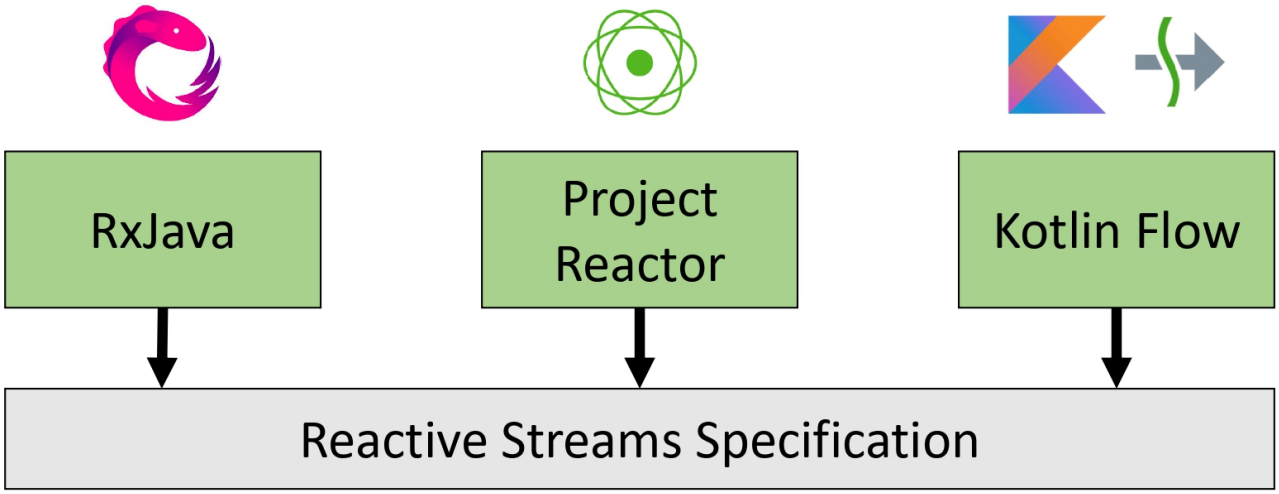



```
fun foo(): Flow<Response> =
    flowOf("A", "B", "C").map { name ->
        compute(name)
    }
```

React on emitted values



Flow is reactive 🚀



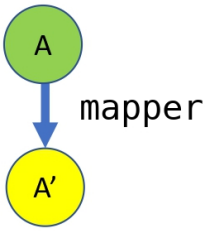
Publisher<T>

```
fun <T : Any> Publisher<T>.asFlow(): Flow<T>
```


```
fun <T : Any> Flow<T>.asPublisher(): Publisher<T>
```

Why Flow?

Flowable<T>



Synchronous

```
fun map(mapper: (T) -> R): Flowable<R>

fun flatMapSingle(mapper: (T) -> SingleSource<R>): Flowable<R>
```

Asynchronous

Flowable<T>

Synchronous

A

mapper

A'

```
fun map(mapper: (T) -> R): Flowable<R>
fun flatMapSingle(mapper: (T) -> SingleSource<R>): Flowable<R>
```

Asynchronous

Synchronous

A

predicate

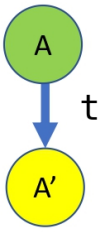
A

```
fun filter(predicate: (T) -> Boolean): Flowable<T>
```

🤯

Asynchronous

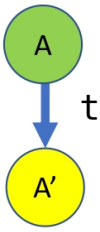
Flow<T>



transform

```
fun map(transform: suspend (T) -> R): Flow<R>
```

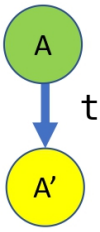
Flow<T>



transform

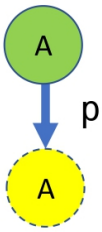
```
fun map(transform: suspend (T) -> R): Flow<R>
```

Flow<T>



transform

```
fun map(transform: suspend (T) -> R): Flow<R>
```



predicate

```
fun filter(predicate: suspend (T) -> Boolean): Flow<T>
```

Operator avoidance

~~`startWith(value)`~~	→	`onStart { emit(value) }`
~~`startWith(flow)`~~	→	`onStart { emitAll(flow) }`
~~`delaySubscription(time)`~~	→	`onStart { delay(time) }`
~~`delayEach(time)`~~	→	`onEach { delay(time) }`
~~`onErrorReturn(value)`~~	→	`catch { emit(value) }`
~~`onErrorResume(flow)`~~	→	`catch { emitAll(flow) }`
~~`generate(…)`~~	→	`flow { … }`

Composable

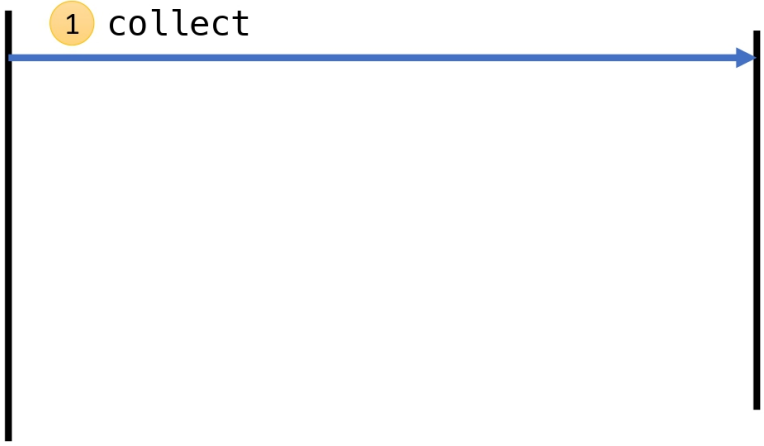
Reactive +   = 

Flow under the hood

```kotlin
interface Flow<out T> {
    suspend fun collect(collector: FlowCollector<T>)
}
```

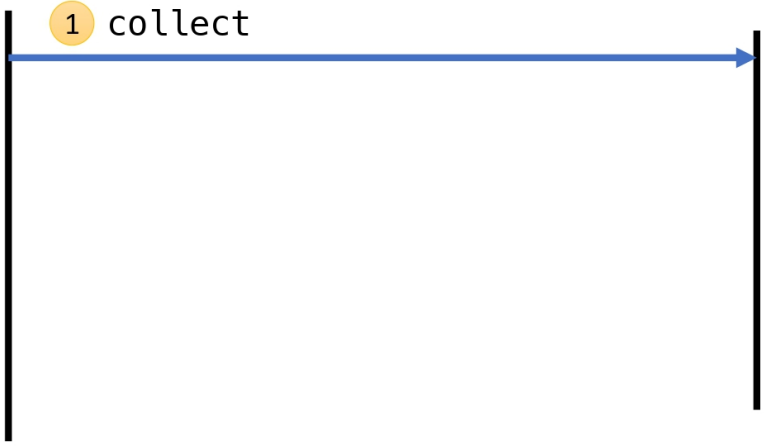
```kotlin
interface FlowCollector<in T> {
    suspend fun emit(value: T)
}
```

```
flow.collect { value ->
    println(value)
}
```



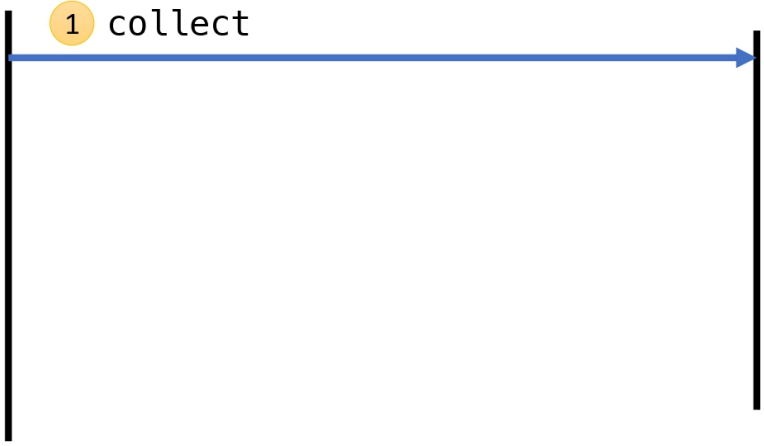
```
flow.collect { value ->
    println(value)
}
```

```
val flow = flow {
    emit("A")
}
```



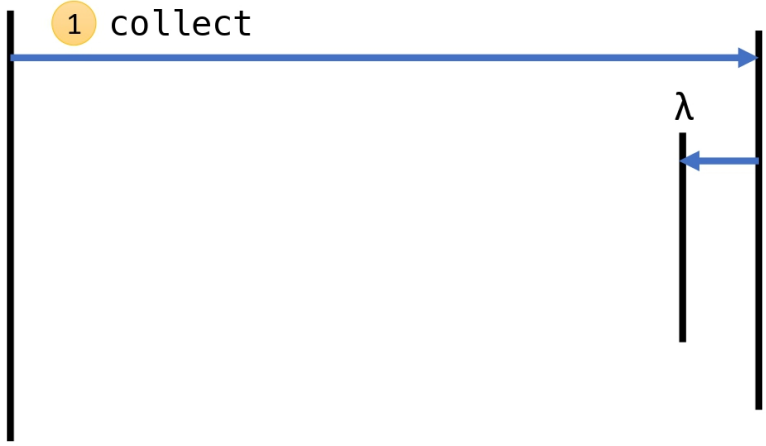
```
flow.collect { value ->
    println(value)
}
```

```
val flow = flow {
    emit("A")
}
```



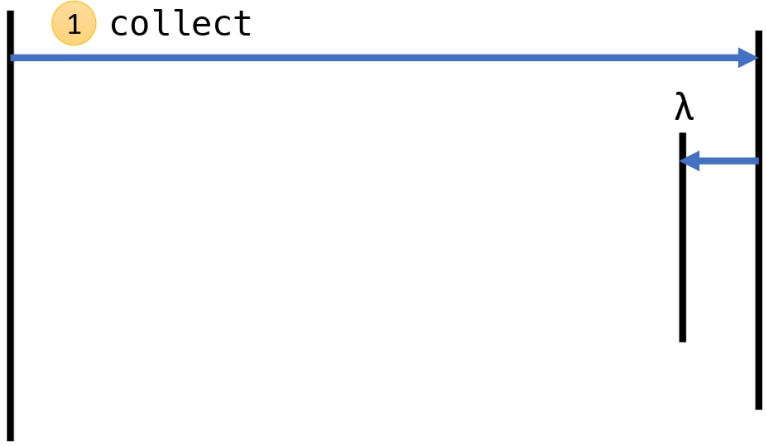
```
flow.collect { value ->
    println(value)
}
```

```
val flow = flow {
    emit("A")
}
```



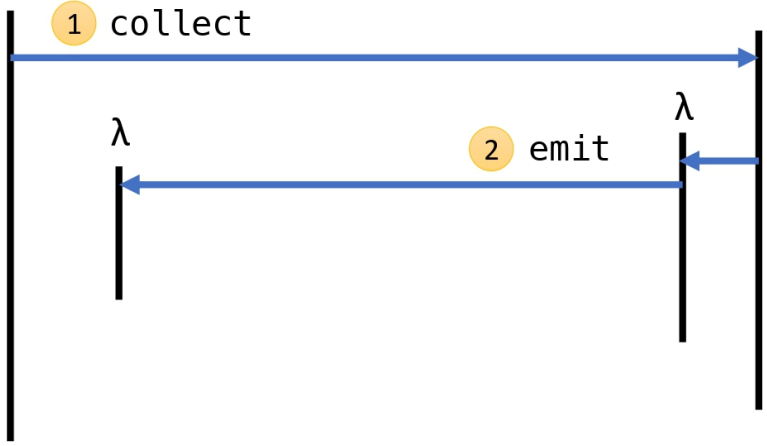
```
flow.collect { value ->
    println(value)
}
```

```
val flow = flow {
    emit("A")
}
```



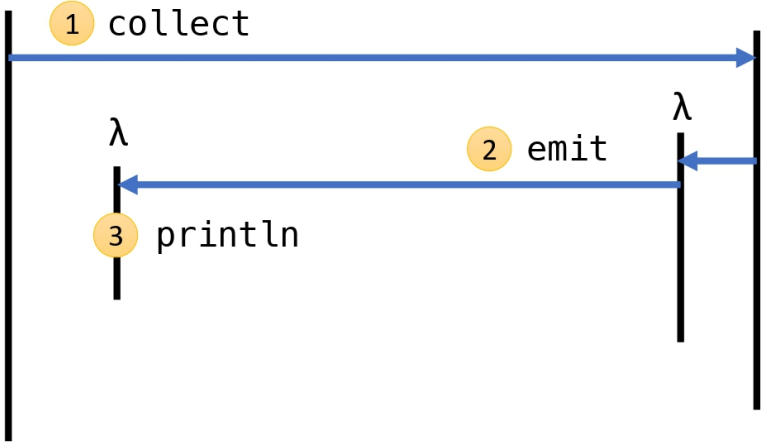
```
flow.collect { value ->
    println(value)
}
```

```
val flow = flow {
    emit("A")
}
```



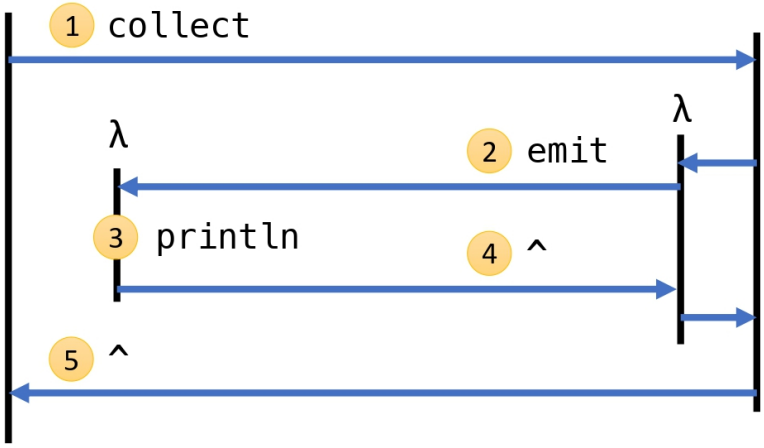
```
flow.collect { value ->
    println(value)
}
```

```
val flow = flow {
    emit("A")
}
```



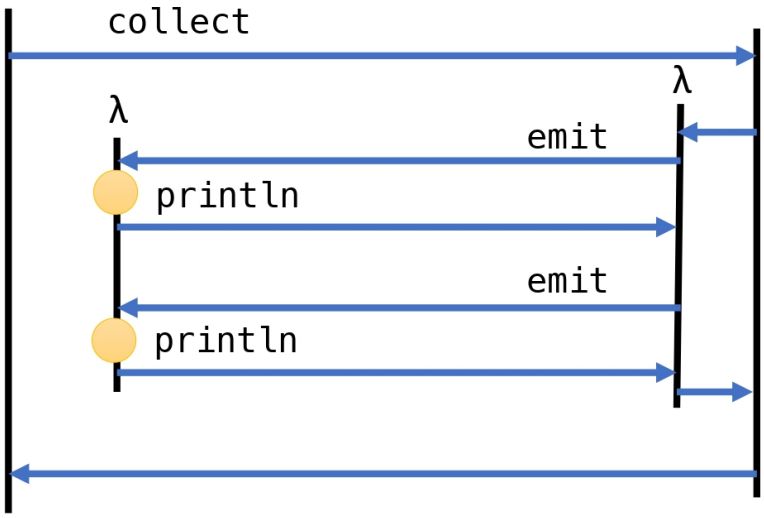
```
flow.collect { value ->
    println(value)
}
```

```
val flow = flow {
    emit("A")
}
```



```
flow.collect { value ->
    println(value)
}
```

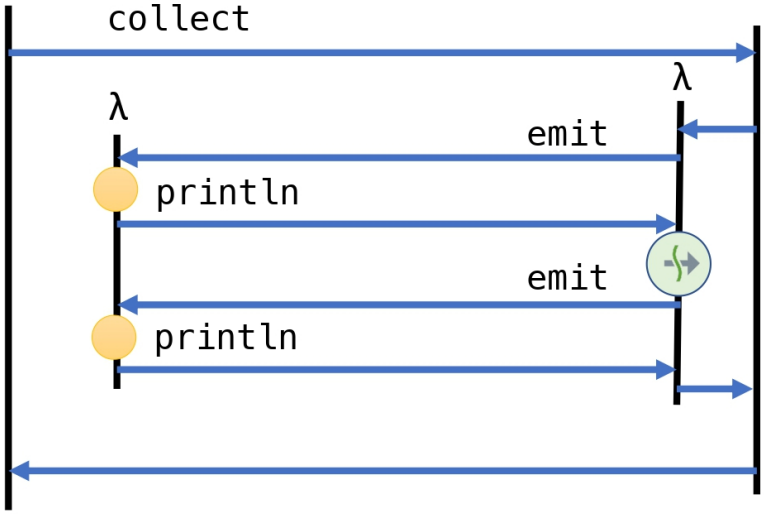
```
val flow = flow {
    emit("A")
    emit("B")
}
```



Asynchronous emitter

```
val flow = flow {
    emit("A")
    delay(100)
    emit("B")
}
```

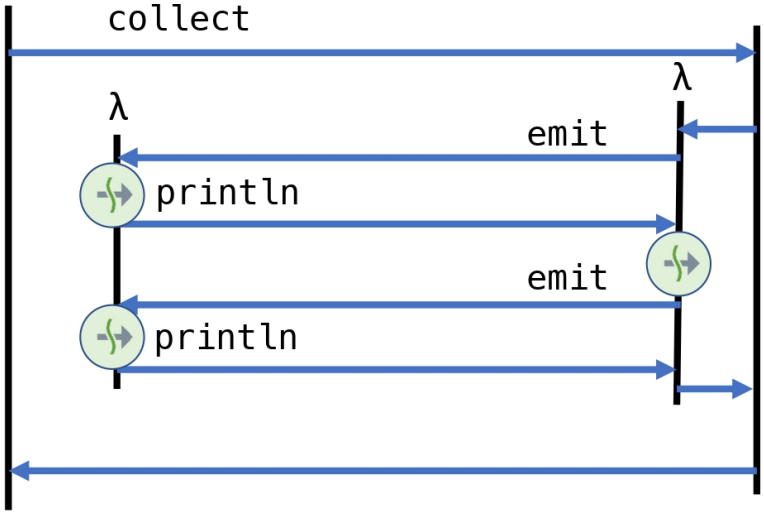
```
flow.collect { value ->
    println(value)
}
```



Backpressure

```
flow.collect { value ->
    delay(100)
    println(value)
}
```

```
val flow = flow {
    emit("A")
    delay(100)
    emit("B")
}
```



Simple design

Simple design ⇒ performance

Kotlin Flow Plays Scrabble

- Benchmark originally developed by José Paumard
- Implemented for RxJava by David Karnok

Kotlin Flow Plays Scrabble

- Benchmark originally developed by José Paumard
- Implemented for RxJava by David Karnok

```
SequencePlaysScrabble          9.824 ± 0.190  ms/op
```

https://github.com/Kotlin/kotlinx.coroutines/tree/develop/benchmarks/src/jmh/kotlin/benchmarks/flow/scrabble/README.md

Kotlin Flow Plays Scrabble

- Benchmark originally developed by José Paumard
- Implemented for RxJava by David Karnok

```
SequencePlaysScrabble          9.824 ± 0.190  ms/op
RxJava2PlaysScrabbleOpt       23.653 ± 0.379  ms/op
```

https://github.com/Kotlin/kotlinx.coroutines/tree/develop/benchmarks/src/jmh/kotlin/benchmarks/flow/scrabble/README.md

Kotlin Flow Plays Scrabble

- Benchmark originally developed by José Paumard
- Implemented for RxJava by David Karnok

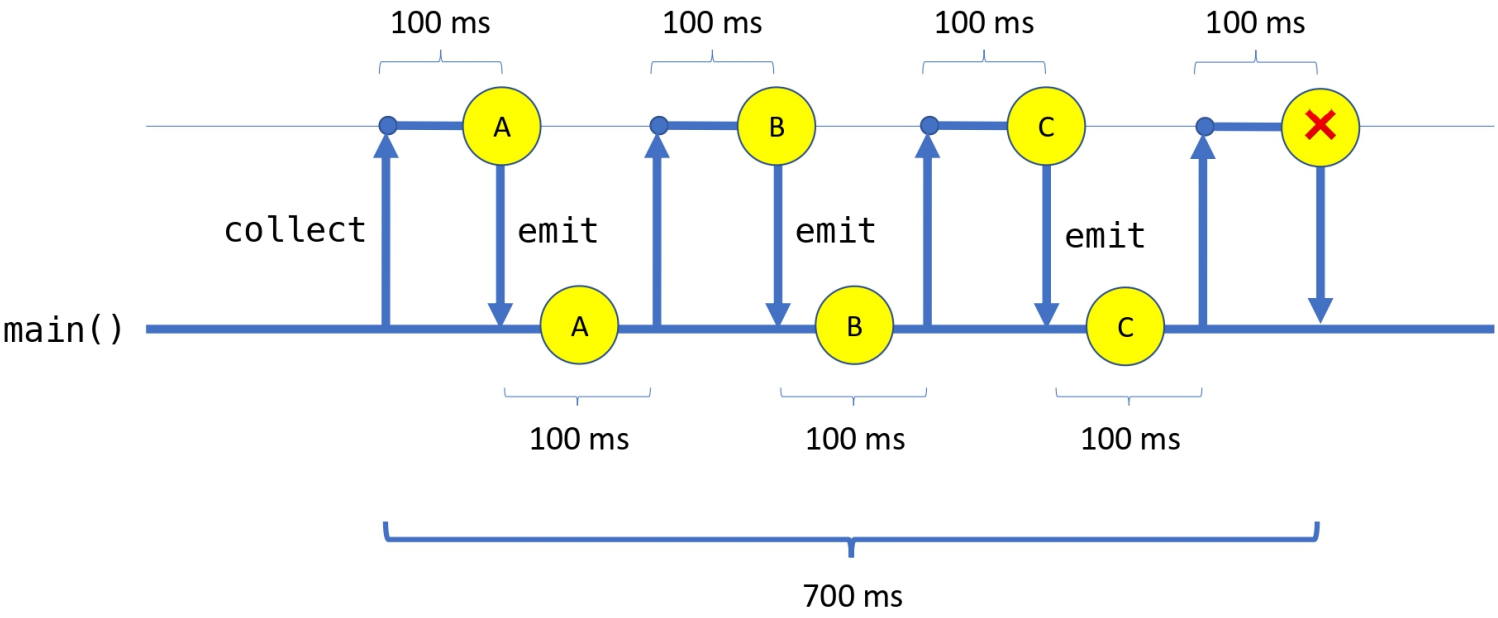
```
SequencePlaysScrabble          9.824 ± 0.190  ms/op
RxJava2PlaysScrabbleOpt       23.653 ± 0.379  ms/op
FlowPlaysScrabbleOpt          13.958 ± 0.278  ms/op
```

https://github.com/Kotlin/kotlinx.coroutines/tree/develop/benchmarks/src/jmh/kotlin/benchmarks/flow/scrabble/README.md

Flow is asynchronous

Flow is asynchronous yet sequential

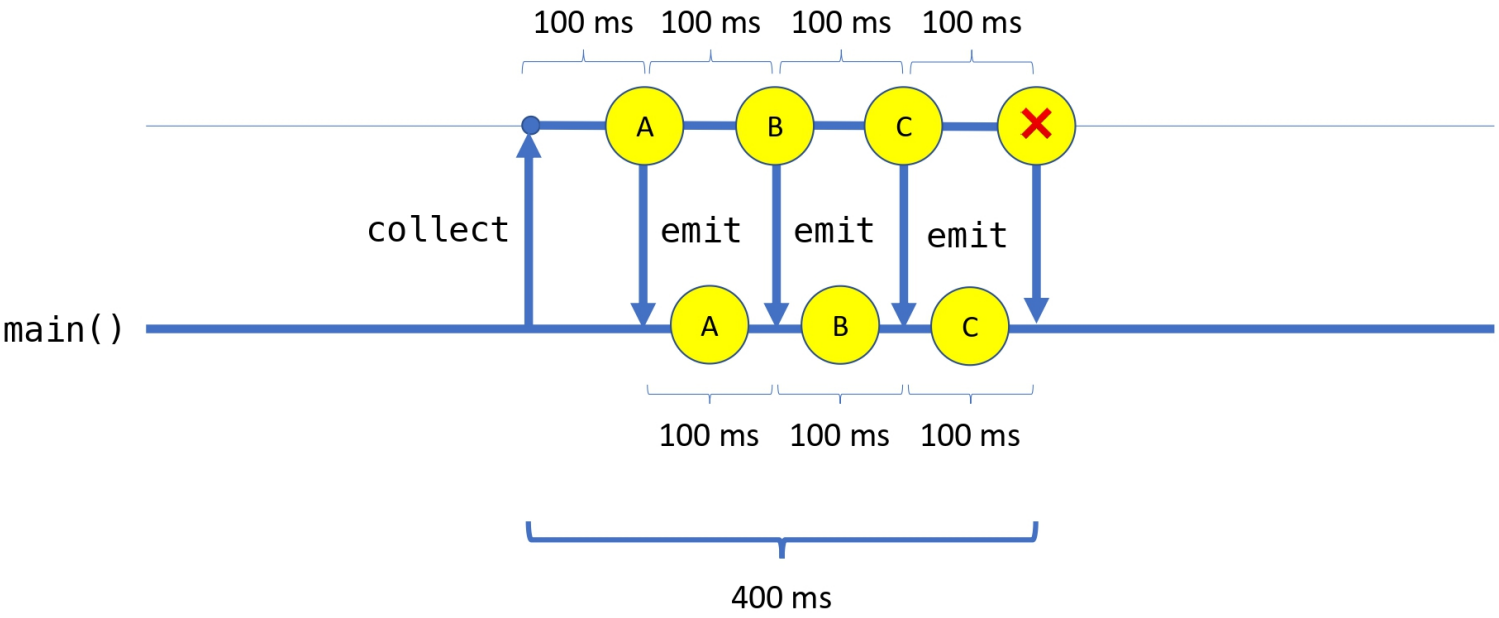
Single coroutine



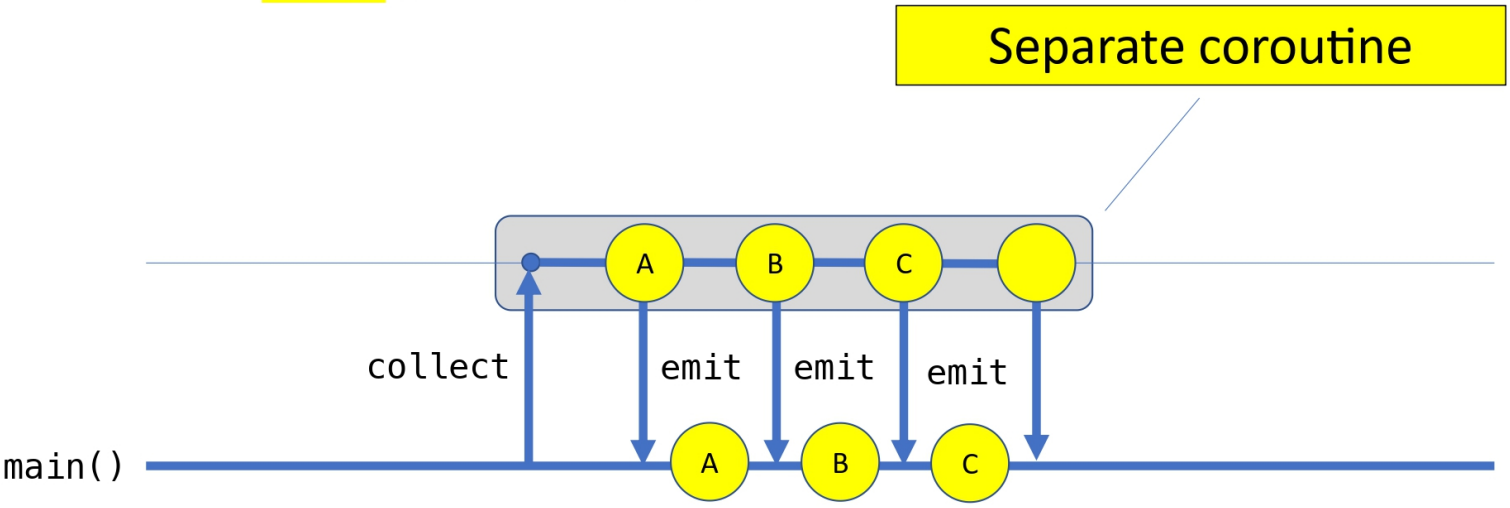
Going concurrent with a flow

```
flow.buffer().collect { … }
```

```
flow.buffer().collect { … }
```

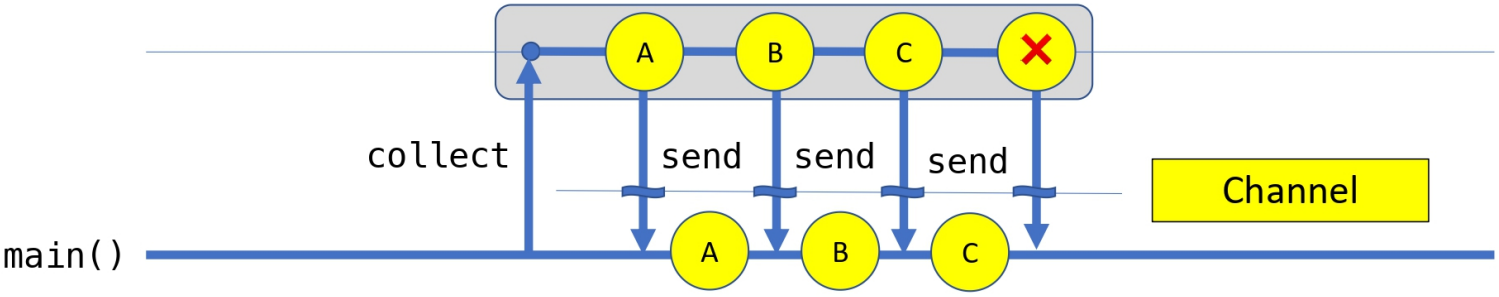


```
flow.buffer().collect { … }
```



```
flow.buffer().collect { … }
```

Declarative & safe



Flow execution context

```
fun foo(): Flow<Response> =
    flowOf("A", "B", "C").map { name ->
        compute(name)
    }
```

```
fun foo(): Flow<Response> =
    flowOf("A", "B", "C").map { name ->
        compute(name)
    }
```

Where does it execute?

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> =
    flowOf("A", "B", "C").map { name ->
        compute(name)
    }
    .flowOn(Dispatchers.Default)
```

Executes in background

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

```
fun foo(): Flow<Response> =
    flowOf("A", "B", "C").map { name ->
        compute(name)
    }
    .flowOn(Dispatchers.Default)
```

Context preservation

```
fun main() = runBlocking {
    val flow = foo()
    flow.collect { x -> println(x) }
}
```

Executes in collector's context

Flow in reactive UI

```
fun events(): Flow<Event>
```

```
fun events(): Flow<Event>

scope.launch {
    …
}
```

```
fun events(): Flow<Event>


scope.launch {
    events().collect { event ->
        …
    }
}
```

```
fun events(): Flow<Event>


scope.launch {
    events().collect { event ->
        updateUI(event)
    }
}
```

```
fun events(): Flow<Event>
```

“Subscribe” to events

```
scope.launch {
    events().collect { event ->
        updateUI(event)
    }
}
```

```
fun events(): Flow<Event>
```

“Subscribe” to events

```
events()
    .onEach { event -> updateUI(event) }
    .launchIn(scope)
```

```
fun events(): Flow<Event>
```

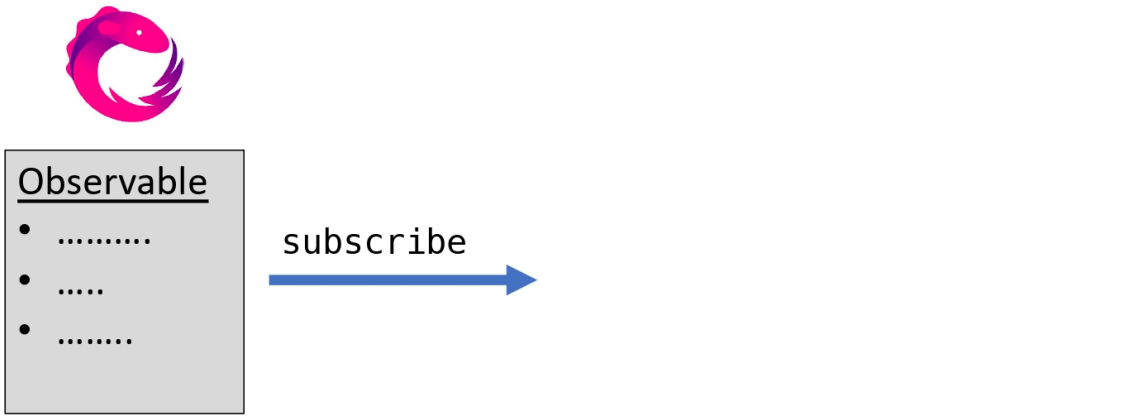
“Subscribe” to events

```
events()
    .onEach { event -> updateUI(event) }
    .launchIn(scope)
```

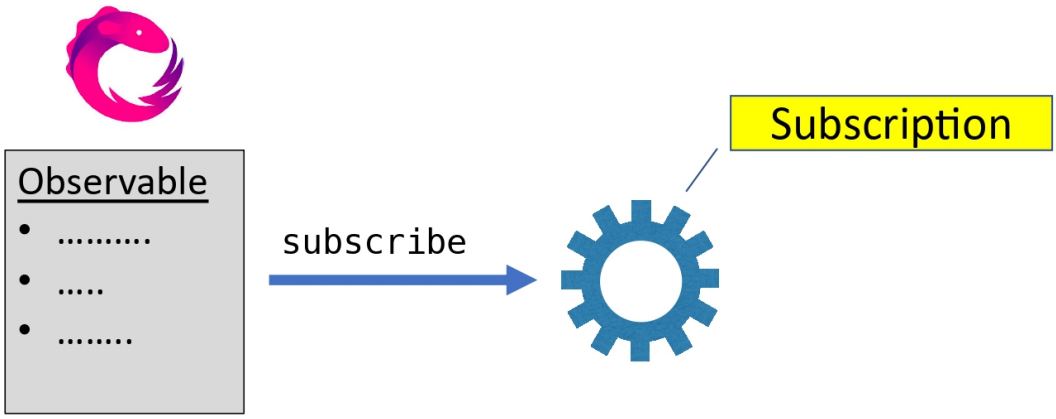
Managing Lifetime

Observable

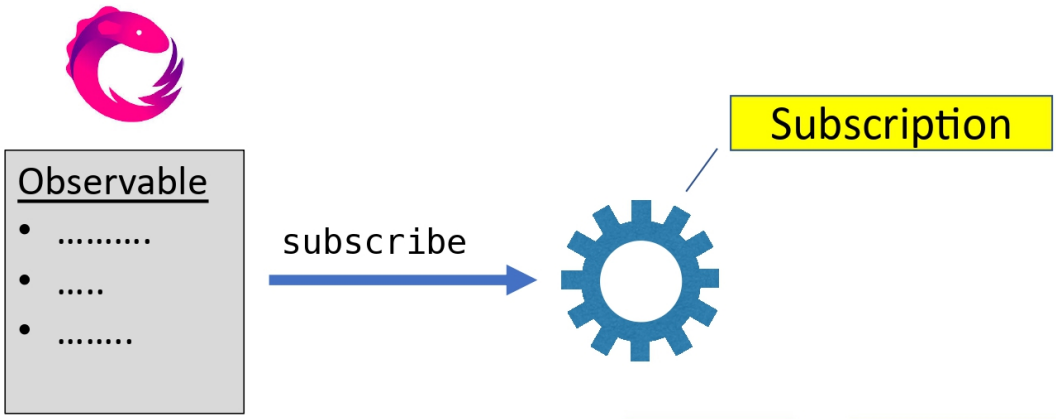
- ……
- …..
- ……..



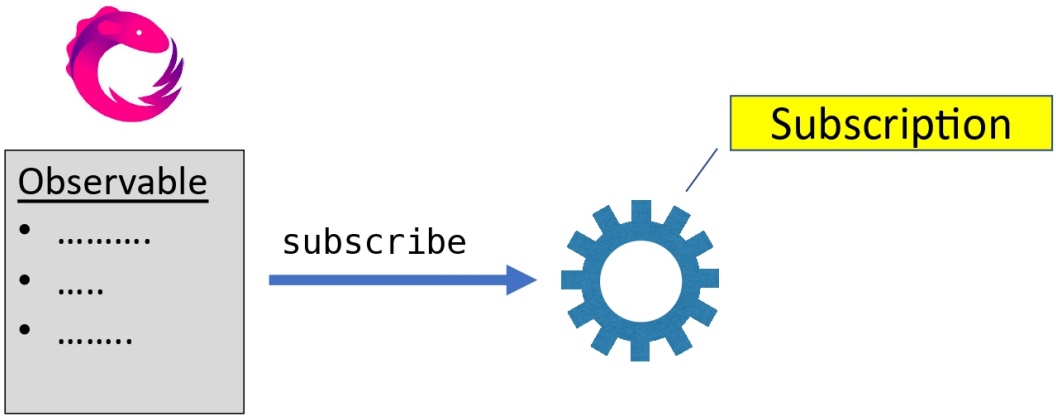

```
observable.subscribe { event ->
    updateUI(event)
}
```



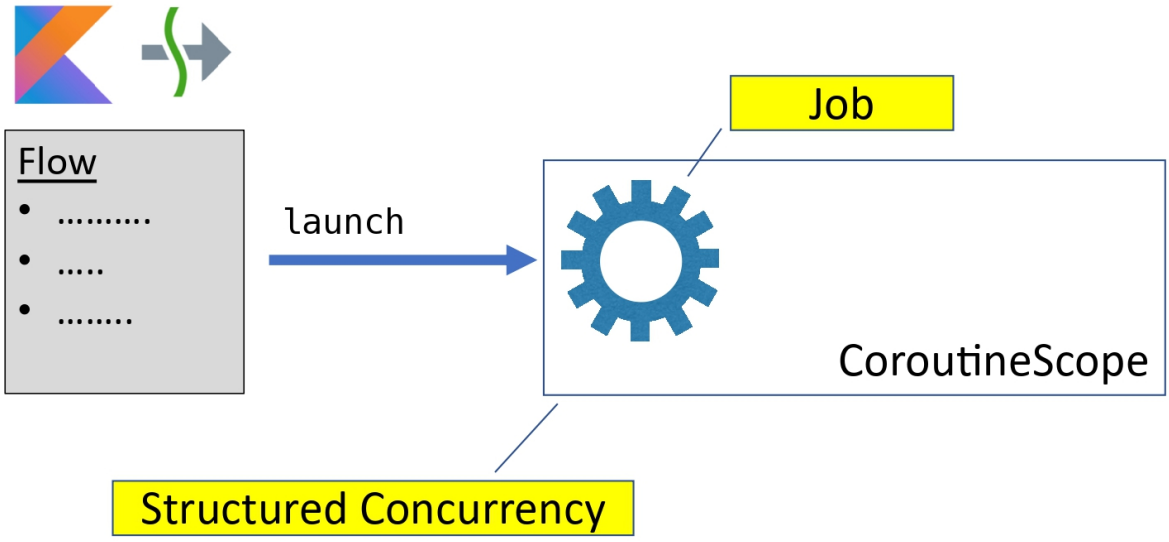

```
observable.subscribe { event ->
    updateUI(event)
}
```

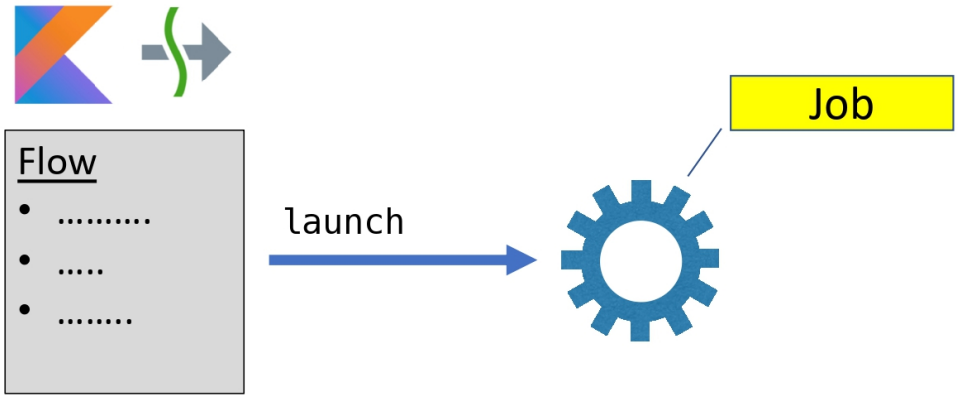



```
val composite = CompositeDisposable()

composite.add(observable.subscribe { event ->
    updateUI(event)
})
```

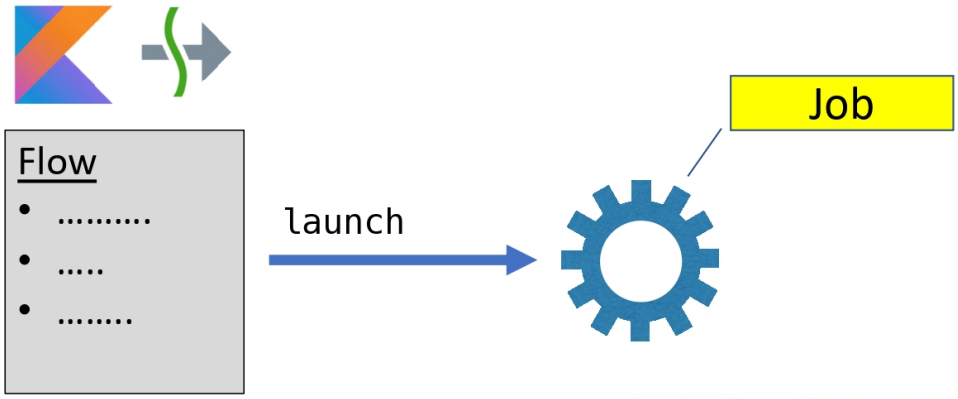



```
val composite = CompositeDisposable()

composite.add(observable.subscribe { event ->
    updateUI(event)
})

composite.clear()
```

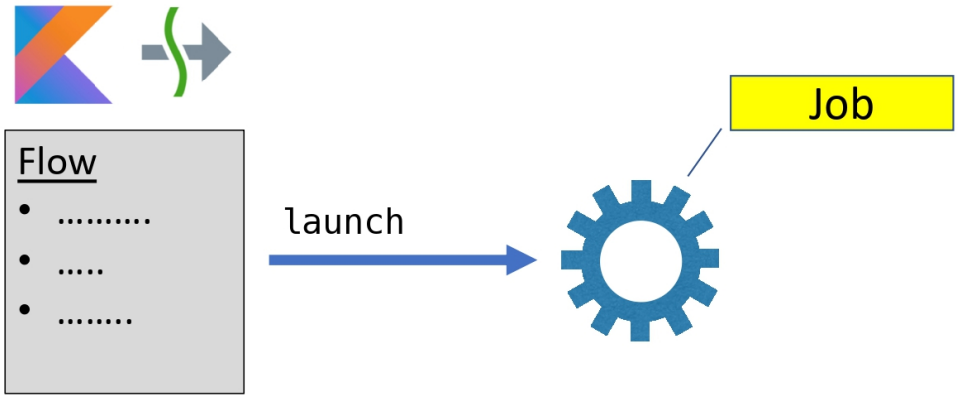

Flow
launch
Job
CoroutineScope
Structured Concurrency



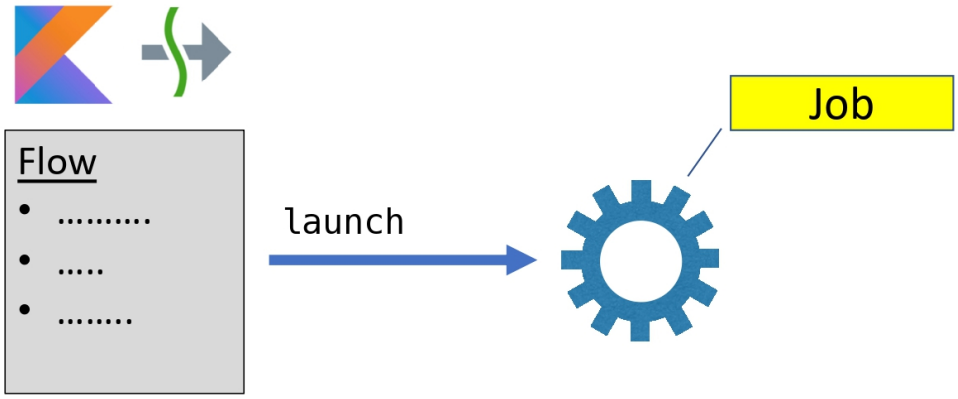

```
events()
    .onEach { event -> updateUI(event) }
    .launchIn(scope)
```



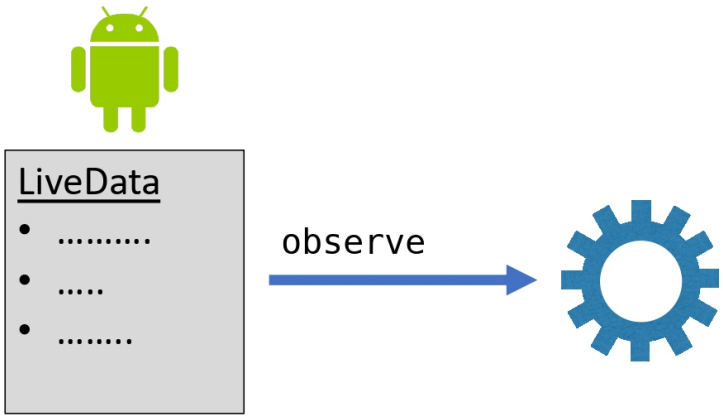

```
val scope = MainScope()

events()
    .onEach { event -> updateUI(event) }
    .launchIn(scope)
```




```
val scope = MainScope()

events()
    .onEach { event -> updateUI(event) }
    .launchIn(scope)

scope.cancel()
```




```
val scope = MainScope()

events()
    .onEach { event -> updateUI(event) }
    .launchIn(scope)

scope.cancel()
```




```
data.observe(livecycleOwner) { event ->
     updateUI(event)
}
```

Status & Roadmap

Status & Roadmap

🚀 Flow is stable in kotlinx.coroutines version 1.3.0

🔮 Future improvements

- ➢Out-of-the box support for UI models (StateFlow / EventFlow)
- ➢Sharing / caching flows
- ➢Concurrency / parallelism operators
- ➢Chunking / windowing operators

🦄 Want more? Give us your feedback
https://github.com/Kotlin/kotlinx.coroutines/issues

Learn more

- Kotlin Flow by example guide
  https://kotlinlang.org/docs/reference/coroutines/flow.html

- API docs
  https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/

- Stories in my blog
  https://medium.com/@elizarov