Kotlin Coroutines Workshop

About me

Marcin Moskała
@marcinmoskala
marcinmoskala@gmail.com

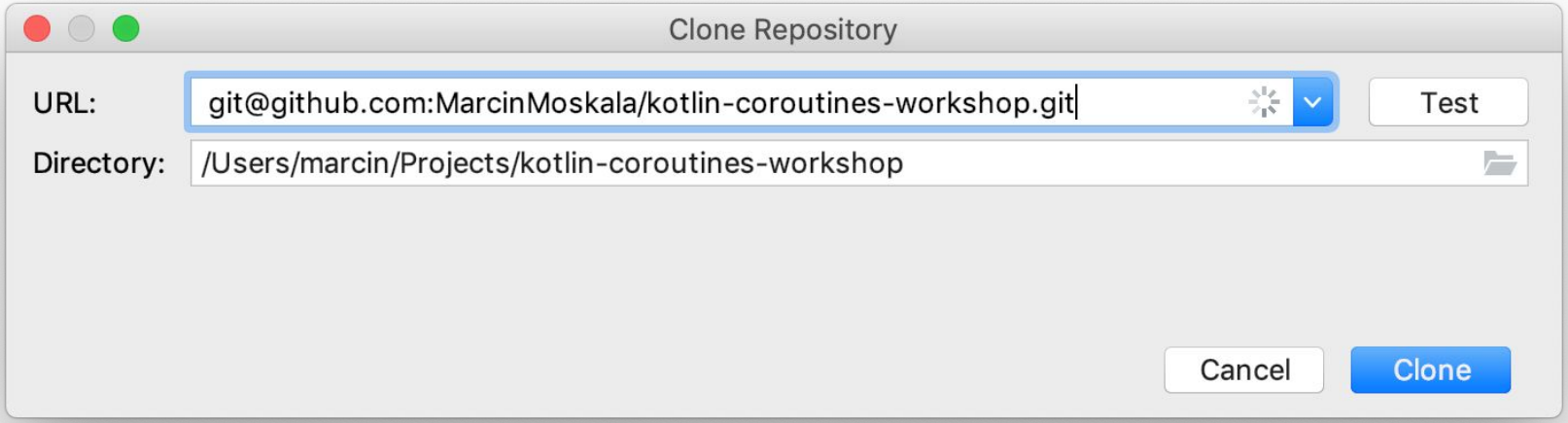
Trainer, Consultant,
Developer @ Allegro

The author of:

- Android Development in Kotlin
- Effective Kotlin

Founder of Kt. Academy

Clone project in IntelliJ



https://github.com/MarcinMoskala/kotlin-coroutines-workshop

Why?

Why?

```
fun onCreate() {
    val news = getNewsFromApi()
    val sortedNews = news.sortedBy { -it.publishedAt }
    view.showNews(sortedNews)
}
```

Why?

```
fun onCreate() {
    thread {
        val news = getNewsFromApi()
        val sortedNews = news.sortedBy { -it.publishedAt }
        view.showNews(sortedNews)
    }
}
```

Why?

```
fun onCreate() {
    thread {
        val news = getNewsFromApi()
        val sortedNews = news.sortedBy { -it.publishedAt }
        runOnUiThread {
            view.showNews(sortedNews)
        }
    }
}
```

Why?

```
fun onCreate() {
    getNewsFromApi { news ->
        val sortedNews = news.sortedBy { -it.publishedAt }
        view.showNews(sortedNews)
    }
}
```

Why?

Not simultaneous!

```
fun onCreate() {
    getConfigFromApi { config ->
        getNewsFromApi(config) { news ->
            getUserFromApi { user ->
                view.showNews(user, news)
            }
        }
    }
}
```

Callback hell!

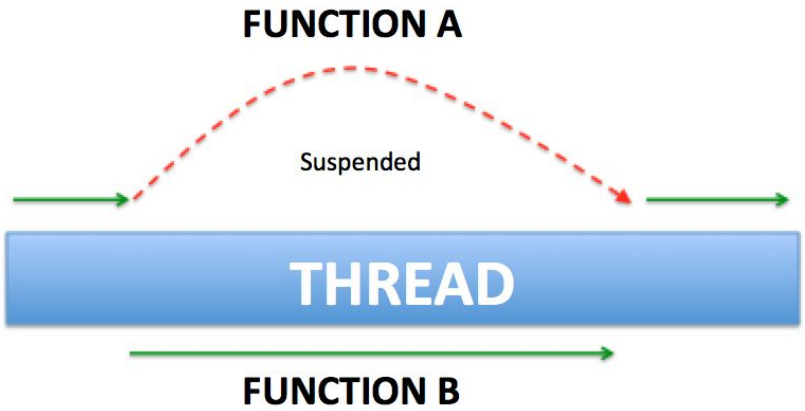
Why?

```
fun onCreate() {
    getNewsFromApi()
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe{ news ->
            val sortedNews = news
                .sortedBy { -it.publishedAt }
            view.showNews(sortedNews)
        })
}
```

Why?

```
fun onCreate() {
    Observable.zip(
        getConfigFromApi().flatMap { getNewsFromApi(it) },
        getUserFromApi(),
        Function2 { news: List<News>, config: Config ->
            Pair(news, config)
        })
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({ (news, config) ->
            view.showNews(news, config)
        })
}
```

```kotlin
suspend fun getNewsFromApi(): List<News> { ... }

fun onCreate() {
    launch(Dispatchers.Main) {
        val news = getNewsFromApi()
        val sortedNews = news.sortedBy { -it.publishedAt }
        view.addNews(sortedNews)
    }
}
```



Why?

```
suspend fun getConfigFromApi(): Config { ... }
suspend fun getNewsFromApi(config: Config): List<News> { ... }
suspend fun getUserDataFromApi(): UserData { ... }

fun onCreate() {
    launch(Dispatchers.Main) {
        val user = getUserDataFromApi()
        val config = getConfigFromApi()
        val news = getNewsFromApi(config)
        view.showNews(user, news)
    }
}
```

Why?

```
suspend fun getConfigFromApi(): Config { ... }
suspend fun getNewsFromApi(config: Config): List<News> { ... }
suspend fun getUserDataFromApi(): UserData { ... }

fun onCreate() {
    launch(Dispatchers.Main) {
        val user = async { getUserDataFromApi() }
        val config = async { getConfigFromApi() }
        val news = async { getNewsFromApi(config.await()) }
        view.showNews(user.await(), news.await())
    }
}
```

Why?

```
fun onCreate() {
    launch(Dispatchers.Main) {
        for (page in 0 until getNumberOfPages()) {
            val news = getNewsFromApi(page = page)
            view.addNews(news)
        }
    }
}
```

Why?

```
fun getNews() {
    val news = getNewsFromDatabase()
            .filter { it.published }
            .sortedByDescending { it.publishedAt }
    respond(news)
}
```

Why?

```
fun getNews() {
    thread {
        val news = getNewsFromDatabase()
                .filter { it.published }
                .sortedByDescending { it.publishedAt }
        respond(news)
    }
}
```

Why?

```
fun getNews() {
    launch(Dispatchers.Default) {
        val news = getNewsFromDatabase()
                .filter { it.published }
                .sortedByDescending { it.publishedAt }
        respond(news)
    }
}
```

examples/Massive.kt

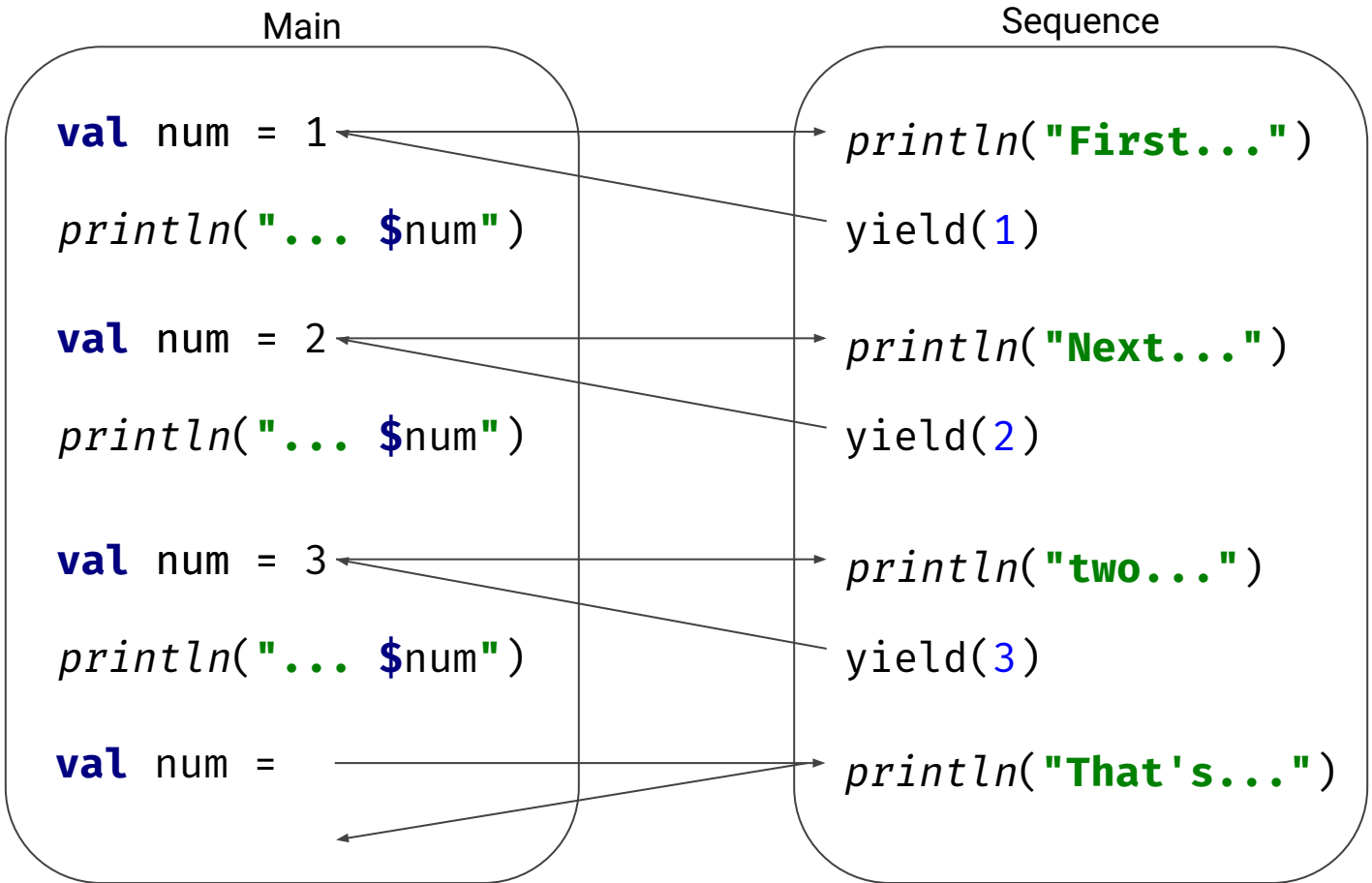
Sequence builders

Sequence builders

```
val childNumbers = sequence {
    println("First number is... one!")
    yield(1)
    println("Next is... eeeee two!")
    yield(2)
    println("twotwotwo... ummmm three!")
    yield(3)
    println("That's all I've learned")
}

fun main() {
    for (num in childNumbers) {
        println("Next number is $num")
    }
}
```

```
First number is... one!
Next number is 1
Next is... eeeee two!
Next number is 2
twotwotwo... ummmm three!
Next number is 3
That's all I've learned
```

Sequence builders

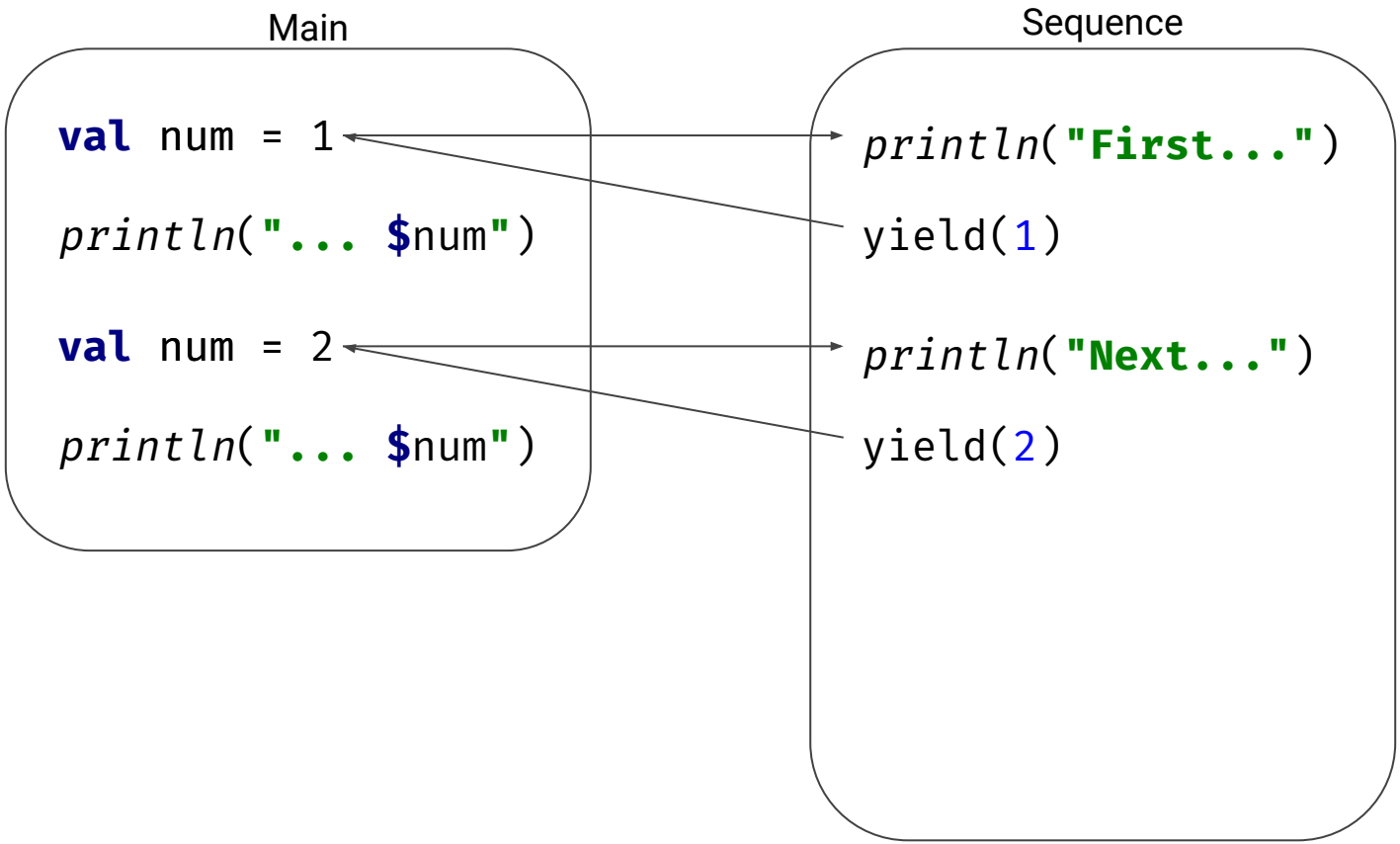


Sequence builders

```
val childNumbers = sequence {
    println("First number is... one!")
    yield(1)

    println("Next is... eeeee two!")
    yield(2)

    println("twotwotwo... ummmm three!")
    yield(3)

    println("That's all I've learned")
}

fun main() {
    val iterator = childNumbers.iterator()
    println("What is first?")
    println("Yes, it is ${iterator.next()}")
    println("What is next?")
    println("Good, it is ${iterator.next()}")
}
```

```
What is first?
First number is... one!
Yes, it is 1
What is next?
Next is... eeeee two!
Good, it is 2
```

Sequence builders



Task 1: Fibonacci and primes

Implement Fibonacci numbers, users sequence (and for ambitious Eratosthenes sieve)

```
val fibonacci = sequence {
    var prev = 1
    var prevprev = 1
    yieldAll(listOf(prev, prevprev))
    while (true) {
        val next = prev + prevprev
        yield(next)
        prevprev = prev
        prev = next
    }
}
```

Task solution

```
val primes = sequence {
    var rest = generateSequence(2) { it + 1 }
    while (true) {
        val prime = rest.first()
        yield(prime)
        rest = rest.drop(1).filter { it % prime != 0 }
    }
}
```

Understanding coroutines

How?

```
suspend fun getConfig(): String {
    println("A")
    delay(1000L)
    println("B")
    return "Some config"
}
```

How?

```
private val executor = Executors.newSingleThreadScheduledExecutor {
    Thread(it, "scheduler").apply { isDaemon = true }
}

suspend fun delay(time: Long): Unit = suspendCoroutine { cont ->
    executor.schedule({ cont.resume(Unit) }, time, TimeUnit.MILLISECONDS)
}
```

Continuation

```
interface Continuation<in T> {
    val context: CoroutineContext
    fun resume(value: T)
    fun resumeWithException(exception: Throwable)
}
```

```
interface Continuation<in T> {
  val context: CoroutineContext
  fun resumeWith(result: Result<T>)
}
```

Coroutines are state machines

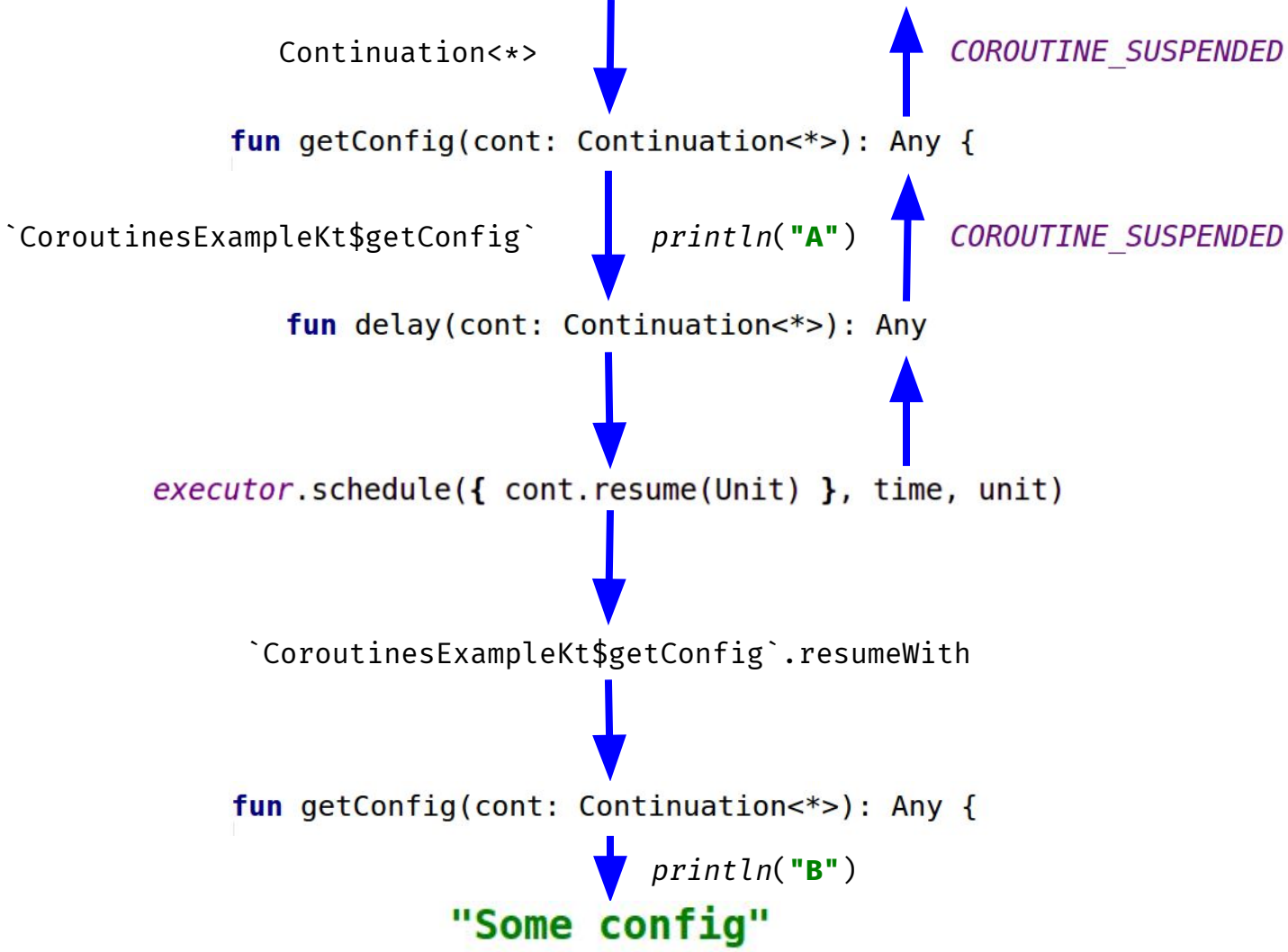
Index of the current state

Data for the current state

Ability to wait patiently

```kotlin
fun getConfig(continuation: Continuation<*>): Any {
    val cont = continuation as? `CoroutinesExampleKt$getConfig`
            ?: `CoroutinesExampleKt$getConfig`(continuation)

    if (cont.label == 0) {
        println("A")
        cont.label = 1
        if (delay(1000L, MILLISECONDS, cont) == COROUTINE_SUSPENDED) {
            return COROUTINE_SUSPENDED
        }
    }
    if (cont.label == 1) {
        println("B")
        return "Some config"
    }
    throw Error()
}
```

```kotlin
suspend fun getConfig(): String {
    println("A")
    delay(1000L)
    println("B")
    return "Some config"
}
```


Continuation<*>
COROUTINE_SUSPENDED
fun getConfig(cont: Continuation<*>): Any {
`CoroutinesExampleKt$getConfig`
println("A")
COROUTINE_SUSPENDED
fun delay(cont: Continuation<*>): Any
executor.schedule({ cont.resume(Unit) }, time, unit)
`CoroutinesExampleKt$getConfig`.resumeWith
fun getConfig(cont: Continuation<*>): Any {
println("B")
"Some config"

```kotlin
fun getConfig(continuation: Continuation<*>): Any {
    val cont = continuation as? `CoroutinesExampleKt$getConfig`
            ?: `CoroutinesExampleKt$getConfig`(continuation)

    if (cont.label == 0) {
        println("A")
        cont.label = 1
        if (delay(1000L, MILLISECONDS, cont) == COROUTINE_SUSPENDED) {
            return COROUTINE_SUSPENDED
        }
    }
    if (cont.label == 1) {
        println("B")
        return "Some config"
    }
    throw Error()
}
```

```kotlin
suspend fun getConfig(): String {
    println("A")
    delay(1000L)
    println("B")
    return "Some config"
}
```

How?

```
suspend fun presenterOnCreate() {
    val config = getConfig()
    val news = getNewsFromApi(config)
    val sortedNews = news.sortedBy { it.dateTime }
    showNews(sortedNews)
}
```

```kotlin
fun presenterOnCreate(continuation: Continuation<*>): Any {
    val cont = continuation as? `CoroutinesExampleKt$presenterOnCreate`
            ?: `CoroutinesExampleKt$presenterOnCreate`(continuation)

    var config: Any = cont.config
    var news: Any = cont.news
    if (cont.label == 0) {
        cont.label = 1
        config = getConfig(cont)
        if (config == COROUTINE_SUSPENDED) {
            return COROUTINE_SUSPENDED
        }
    }
    if (cont.label == 1) {
        cont.label = 2
        news = getNewsFromApi(config as String, cont)
        if (news == COROUTINE_SUSPENDED) {
            return COROUTINE_SUSPENDED
        }
    }
    if (cont.label == 2) {
        val sortedNews = (news as List<News>).sortedBy { it.dateTime }
        showNews(sortedNews, config as String)
        return Unit
    }
    throw Error()
}
```

Coroutines support vs Coroutines library

Built-in into Kotlin

Single modifier (suspend),
few primitives (like startCoroutine)
and interfaces (like Continuation)

Allows you to reproduce any
concurrence style

Distributed in separate library

Lot's of classes and functions (like
launch, runBlocking, CommonPool)

Gives you concrete
concurrence style

Task 2: Continuation

Can you steal continuation? Stop a function in the middle to resume it later using `suspendCoroutine` function.

Modify `continuationSteal` functions in the lines where you see a comment with “TODO”. Store continuation in `continuation` variable. After resume, print passed value using `console.println`.

Test in `continuation.ContinuationStealTests`.

```
val res = suspendCoroutine<T> { cont ->
    continuation = cont
}
console.println(res)
```

Kotlin coroutines library

Coroutine builders

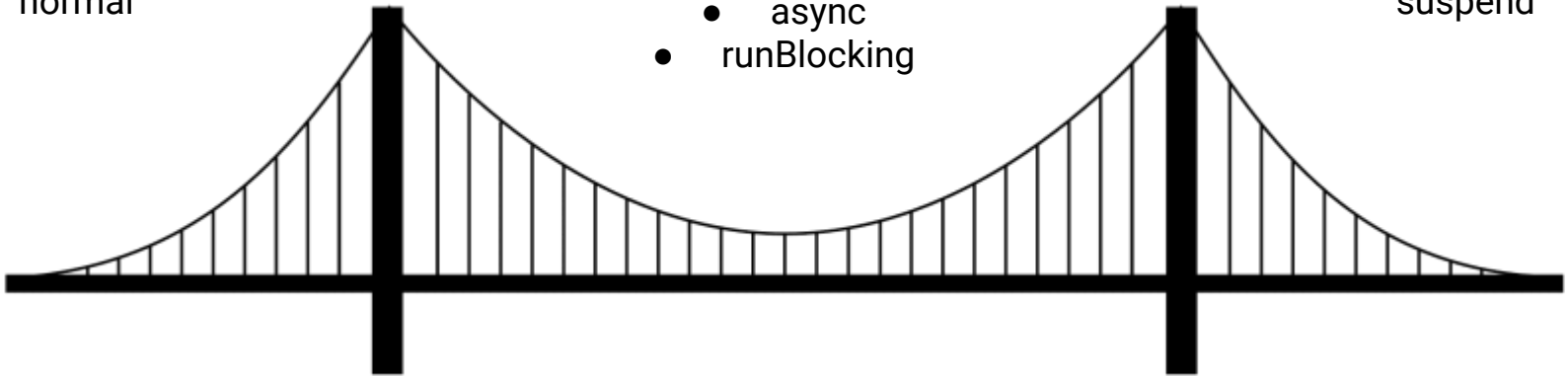
Coroutine builders

Coroutine builders:

- launch
- async
- runBlocking

normal

suspend



```
suspend fun suspendFun() {}

fun normalFun() {
    suspendFun()
}
```

launch

```
fun main() {
    GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
    Thread.sleep(2000L)
}
```

1.kt

launch

```
fun main() {
    GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
    Thread.sleep(2000L)
}
```

b1.kt

runBlocking

```kotlin
fun main() {
    runBlocking {
        delay(1000L)
        println("World!")
    }
    runBlocking {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
}
```

b2.kt

runBlocking

```
fun main() {
    GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
    runBlocking {
        delay(2000L)
    }
}
```

2.kt

runBlocking

```
fun main() = runBlocking {
    GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
    delay(2000L)
}
```

3.kt

runBlocking

```
class MyTest {
    @Test
    fun testMySuspendingFunction() = runBlocking {
        //...
    }
}
```

3.kt

async/await

```
fun main() = runBlocking {
    val value1 = GlobalScope.async {
        delay(1000L)
        1
    }
    val value2 = GlobalScope.async {
        delay(1000L)
        20
    }
    println("Calculating")
    print(value1.await() + value2.await())
}
```

b3.kt

Coroutine Context

Structured concurrency

```
fun main() = runBlocking {
    GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
    delay(2000L)
}
```

3.kt

Parental responsibilities

Parental responsibilities:

1. Wait for its childrens

CoroutineScope

Every coroutine builder is an extension on CoroutineScope and inherits its coroutineContext to automatically propagate both context elements and cancellation.

```
fun CoroutineScope.launch(
        context: CoroutineContext = EmptyCoroutineContext,
        start: CoroutineStart = CoroutineStart.DEFAULT,
        onCompletion: CompletionHandler? = null,
        block: suspend CoroutineScope.() -> Unit
): Job
```

```
GlobalScope.launch { /*...*/ }
```

CoroutineScope

```
interface CoroutineScope {
    val coroutineContext: CoroutineContext
}
```

CoroutineContext

It is an indexed set of Element instances. An indexed set is a mix between a set and a map. Every element in this set has a unique Key. Keys are compared by reference.

An element of the coroutine context is a singleton context by itself.

```
interface CoroutineContext {
    operator fun <E : Element> get(key: Key<E>): E?
    fun <R> fold(initial: R, operation: (R, Element) -> R): R
    operator fun plus(context: CoroutineContext): CoroutineContext
    fun minusKey(key: Key<*>): CoroutineContext

    interface Element : CoroutineContext {
        val key: Key<*>
    }

    interface Key<E : Element>
}
```

CoroutineContext key elements

- `CoroutineName` - For debugging purposes

Naming coroutines for debugging purposes

```
fun CoroutineScope.log(msg: String) =
    println("[${coroutineContext[CoroutineName]?.name}] $msg")

fun main() = runBlocking(CoroutineName("main")) {
   log("Started main coroutine")
   val v1 = async(CoroutineName("v1coroutine")) {
       delay(500)
       log("Computing v1")
       "KOKO"
   }
   val v2 = async(CoroutineName("v2coroutine")) {
       delay(1000)
       log("Computing v2")
       6
   }
   log("The answer for v1 = ${v1.await()}")
}
```

4.kt

Coroutine name implementation

```
public data class CoroutineName(
    val name: String
) : AbstractCoroutineContextElement(CoroutineName) {

    override fun toString(): String = "CoroutineName($name)"

    companion object Key : CoroutineContext.Key<CoroutineName>
}
```

Coroutine name implementation

```
class CoroutineName(
    val name: String
) : AbstractCoroutineContextElement(CoroutineName) {

    companion object Key : CoroutineContext.Key<CoroutineName>
}
```

Coroutine name implementation

```
class CoroutineName(val name: String) : CoroutineContext.Element {

    override val key: CoroutineContext.Key<*> = CoroutineName

    companion object : CoroutineContext.Key<CoroutineName>
}
```

Task 3: CounterContext

Implement a coroutine context that with a counter. We get next numbers using `next` function:

```
coroutineContext[CounterContext]?.next()
```

Coroutine Context: Job

CoroutineContext key elements

- `CoroutineName` - For debugging purposes
- `Job` - Lifecycle of coroutine

Job cancellation and awaiting

```
fun main() = runBlocking {
    val job = Job()
    launch(job) {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
    }
    delay(1300L) // delay a bit
    println("main: I'm tired of waiting!")
    job.cancel()
    job.join()
    println("main: Now I can quit.")
}
```

5.kt

Job cancellation and awaiting

```
fun main() = runBlocking {
    val job = launch {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
    }
    delay(1300L) // delay a bit
    println("main: I'm tired of waiting!")
    job.cancel()
    job.join()
    println("main: Now I can quit.")
}
```

6.kt

Closing resources with finally

```
fun main() = runBlocking {
    val job = launch {
        try {
            repeat(1000) { i ->
                println("I'm sleeping $i ...")
                delay(500L)
            }
        } finally {
            println("I'm running finally")
        }
    }
    delay(1300L)
    println("main: I'm tired of waiting!")
    job.cancelAndJoin()
    println("main: Now I can quit.")
}
```

Other useful extension functions

```
fun CoroutineScope.cancel(cause: CancellationException? = null) {
   val job = coroutineContext[Job] ?: error("Scope cannot be cancelled because it does not
have a job: $this")
   job.cancel(cause)
}

fun Job.cancelChildren(cause: CancellationException? = null) {
   children.forEach { it.cancel(cause) }
}

val CoroutineScope.isActive: Boolean
   get() = coroutineContext[Job]?.isActive ?: true

fun CoroutineContext.ensureActive() {
   val job = get(Job) ?: error("Context cannot be checked for liveness because it does not
have a job: $this")
   job.ensureActive()
}

fun Job.ensureActive() {
   if (!isActive) throw getCancellationException()
}
```

Exception handling

Exceptions in coroutines

```
fun main() = runBlocking<Unit> {
    launch {
        delay(1_000)
        throw Error()
    }
    launch {
        delay(2_000)
        println("Done")
    }
}
```

e1.kt

Parental responsibilities

Parental responsibilities:

1. Wait for its childrens
2. Fail when any children fails and close all other

Supervision job

```
fun main() = runBlocking {
    val supervisor = SupervisorJob()
    with(CoroutineScope(coroutineContext + supervisor)) {
        launch(CoroutineExceptionHandler { _, _ ->  }) {
            delay(1000)
            throw AssertionError("Cancelled")
        }
        launch {
            delay(2000)
            println("AAA")
        }
    }
    supervisor.join()
}
```

e2.kt

CoroutineContext key elements

- `CoroutineName` - For debugging purposes
- `Job` - Lifecycle of coroutine
- `CoroutineExceptionHandler` - Exception handling

CoroutineExceptionHandler

```
fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("Caught $exception")
    }
    val job = GlobalScope.launch(handler) {
        throw AssertionError()
    }
    job.join() // Caught java.lang.AssertionError
}
```

7.kt

Dispatchers

CoroutineContext key elements

- `CoroutineName` - For debugging purposes
- `Job` - Lifecycle of coroutine
- `CoroutineExceptionHandler` - Exception handling
- `ContinuationInterceptor` - Mainly dispatchers

ContinuationInterceptor

```
public interface ContinuationInterceptor : CoroutineContext.Element {

   public fun <T> interceptContinuation(
      continuation: Continuation<T>
   ): Continuation<T>

    public fun releaseInterceptedContinuation(
        continuation: Continuation<*>
    ) {  }

   companion object Key : CoroutineContext.Key<ContinuationInterceptor>
}
```

Coroutine dispatchers

```
launch(Dispatchers.Default) {
    // ...
}
```

CPU operations. It is backed by a shared pool of threads on JVM.

```
launch(Dispatchers.Main) {
    // ...
}
```

Main thread operations. Platform specific main thread if exists.

```
launch(Dispatchers.IO) {
    // ...
}
```

Blocking (Network, Disc) operations. Designed for offloading blocking IO tasks to a shared pool of threads.

```
launch(Dispatchers.Unconfined) {
    // ...
}
```

Does not change thread.

Single thread context

```
val dispatcher = Executors.newSingleThreadExecutor()
    .asCoroutineDispatcher()

val dispatcher = Executors.newFixedThreadPool(100)
    .asCoroutineDispatcher()
```

Different dispathers

```
fun main() = runBlocking<Unit> {
    fun getThreadName() = Thread.currentThread().name
    launch {
        println("main runBlocking      : I'm working in thread ${getThreadName()}")
    }
    launch(Dispatchers.Unconfined) {
        println("Unconfined            : I'm working in thread ${getThreadName()}")
    }
    launch(Dispatchers.Default) {
        println("Default               : I'm working in thread ${getThreadName()}")
    }
    launch(newSingleThreadContext("MyOwnThread")) {
        println("newSingleThreadContext: I'm working in thread ${getThreadName()}")
    }
}
```

8.kt

Making scope

```
val scope = CoroutineScope(Dispatchers.Main)

fun CoroutineScope(context: CoroutineContext): CoroutineScope =
        object : CoroutineScope {
            override val coroutineContext: CoroutineContext =
                    if (context[Job] != null) context else context + Job()
        }

val emptyScope = GlobalScope

object GlobalScope : CoroutineScope {
    override val coroutineContext: CoroutineContext
        get() = EmptyCoroutineContext
}

open class MainScope : CoroutineScope {
    override val coroutineContext: CoroutineContext = Dispatchers.Main
}
```

Task 4: BasePresenter

Update `BasePresenter` to:

- Cancel all jobs in `onDestroy`
- Run all jobs on `UI` thread
- Handle all errors by `onError`

MainScope

```
fun MainScope(): CoroutineScope =
     ContextScope(SupervisorJob() + Dispatchers.Main)
```

```
MainScope().launch {
   //...
}
```

withContext

```
val dispatcher = Executors.newFixedThreadPool(100)
        .asCoroutineDispatcher()


suspend fun someIntensiveOperation() = withContext(dispatcher) {
    //...
}
```

withTimeout

```
suspend fun timeRestricted(): Response = withTimeout(5_000) {
    //...
}

suspend fun timeRestricted(): Response? = withTimeoutOrNull(5_000) {
    //...
}
```

Task 4 part 2: UserRepository

Update `UserRepository` to never read data on the main thread, and to make it prepared for multiple concurrent calls.

Task 5: Coffee 1

You have a coffee shop. You defined the process of how to make a coffee in `coffee/Coffee.kt`.

a) How much does it take now? Can you make it faster?
   Assume that every thread can be treated as a separate barista.

b) What if instead of having a truly long operation, `longOperation` would use Thread.sleep(1000)?
   Can you process orders in a more efficient manner in such a case?

coroutineScope

Concurrent using async

```
suspend fun makeAsyncCalculations(): String {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    return "The answer is ${one.await() + two.await()}"
}
```

Concurrent using async

```
suspend fun makeAsyncCalculations(): String {
    val one = GlobalScope.async { doSomethingUsefulOne() }
    val two = GlobalScope.async { doSomethingUsefulTwo() }
    return "The answer is ${one.await() + two.await()}"
}
```

Error handling!

9.kt

Concurrent using async

```
suspend fun makeAsyncCalculations(): String {
    val one = GlobalScope.async { doSomethingUsefulOne() }
    val two = GlobalScope.async { doSomethingUsefulTwo() }
    GlobalScope.launch {
        doSomethingInBackground()
    }
    return "The answer is ${one.await() + two.await()}"
}
```

Completion awaiting!

Concurrent using async

```
suspend fun CoroutineScope.makeAsyncCalculations(): String {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    launch {
        doSomethingInBackground()
    }
    return "The answer is ${one.await() + two.await()}"
}
```

Dangerous

Tweet.kt

Coroutine scope

```
fun main() = runBlocking {
    launch {
        delay(200L)
        println("Task from runBlocking")
    }

    coroutineScope {
        launch {
            delay(500L)
            println("Task from nested launch")
        }

        delay(100L)
        println("Task from coroutine scope")
    }

    println("Coroutine scope is over")
}
```

```
Task from coroutine scope
Task from runBlocking
Task from nested launch
Coroutine scope is over
```

10.kt

Coroutine scope

```
suspend fun main() = coroutineScope<Unit> {
    delay(1000)
    print("Hello, World")
}
```

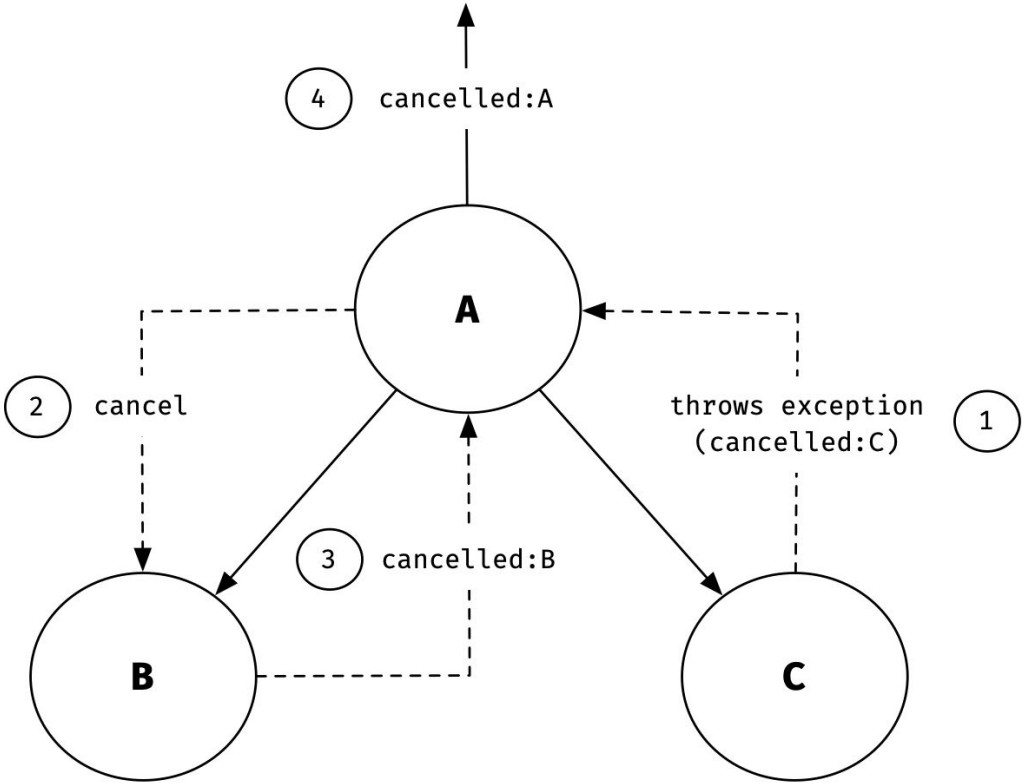
Main.kt

Structured concurrency with async

```
suspend fun makeAsyncCalculations(): String = coroutineScope {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    "The answer is ${one.await() + two.await()}"
}
```

11.kt

Structured concurrency



Job Cancellation (Abnormal)

Exception handling

```
suspend fun failedConcurrentSum(): Int =
coroutineScope {
    val one = async {
        try {
            delay(Long.MAX_VALUE)
            42
        } finally {
            println("First child was cancelled")
        }
    }
    val two = async<Int> {
        println("2nd child throws an exception")
        throw ArithmeticException()
    }
    one.await() + two.await()
}
```

```
fun main() = runBlocking<Unit> {
    try {
        failedConcurrentSum()
    } catch (e:
ArithmeticException) {
        println("Computation failed
with ArithmeticException")
    }
}
```

12.kt

Exception propagation

```kotlin
fun main() = runBlocking {
    val deferred = GlobalScope.async {
        println("Throwing exception from async")
        throw ArithmeticException()
    }
    try {
        deferred.await()
        println("Unreached")
    } catch (e: ArithmeticException) {
        println("Caught ArithmeticException")
    }
}
```

13.kt

Task 6: Request

Implement `getBestStudent` from `request/Request.kt` that should properly get the best student in the semester.

Tests in `ScopeTest`

Supervision scope

```kotlin
fun main() = runBlocking<Unit> {
    supervisorScope {
        launch(CoroutineExceptionHandler { _, _ ->  }) {
            delay(1000)
            throw AssertionError("Cancelled")
        }
        launch {
            delay(2000)
            println("AAA")
        }
    }
}
```

15.kt

Different worlds

```
fun getData(): Data { /*...*/ }
```

Fast, local

Coroutine Builders

```
suspend fun getData(): Data { /*...*/ }
```

Coroutine Scope

Long, Blocking

coroutineScope

```
fun CoroutineScope.getData(): Data { /*...*/ }
```

Side effects

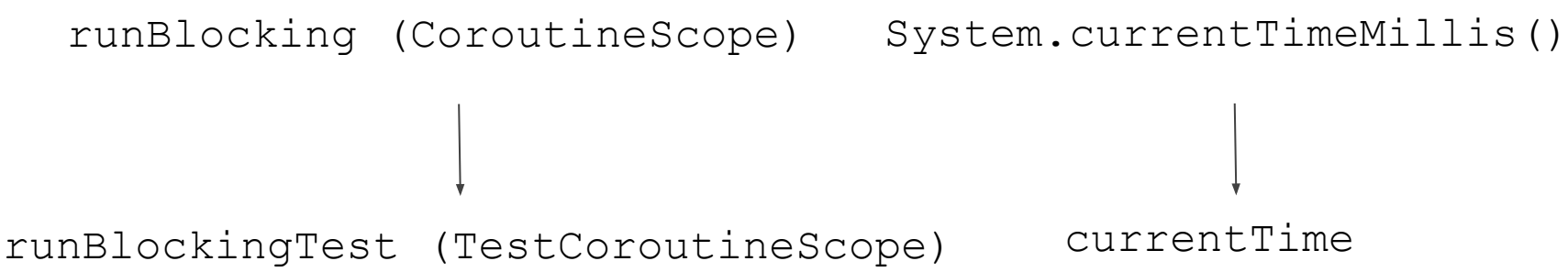
~~`fun asyncGetData(scope: CoroutineScope): Data = async {...}`~~

```
fun startProcess(scope: CoroutineScope) { /*...*/ }
```

~~`suspend fun CoroutineScope.getData(): Data { /*...*/ }`~~

Testing coroutines

Testing coroutines




```
@UseExperimental(ExperimentalCoroutinesApi::class)
@Test
fun testChecker() = runBlockingTest {
    val startTime = currentTime
    delay(1000)
    val endTime = currentTime
    assertEquals(startTime + 1000, endTime)
}
```

Experimental API

```
@UseExperimental(ExperimentalCoroutinesApi::class)
```

or

```
//build.gradle
compileTestKotlin {
    kotlinOptions {
        freeCompilerArgs +=
"-Xuse-experimental=kotlin.Experimental"
    }
}
```

Task 7: RequestTest

Refactor tests from `RequestTest` to make them faster and more reliable.

Coroutines and Android

Retrofit

```
interface GitHubServiceApiDef {
   @GET("orgs/jetbrains/repos?per_page=100")
   suspend fun getOrgReposCall(): List<Repo>

   @GET("repos/jetbrains/{repo}/contributors?per_page=100")
   suspend fun getRepoContributorsCall(@Path("repo") repo: String): List<User>
}
```

Since version 2.6.0

Room

```
@Dao
interface UsersDao {

    @Query("SELECT * FROM users")
    suspend fun getUsers(): List<User>

    @Query("UPDATE users SET age = age + 1 WHERE userId = :userId")
    suspend fun incrementUserAge(userId: String)

    @Insert
    suspend fun insertUser(user: User)

    @Update
    suspend fun updateUser(user: User)

    @Delete
    suspend fun deleteUser(user: User)
}
```

Since version 2.1

Work Manager

```
class UploadWorkWorker(...) : CoroutineWorker(...) {

    suspend fun doWork(): Result {
        val newNotes = db.queryNewNotes()
        noteService.upload(newNotes)
        db.markAsSynced(newNotes)
        return Result.success()
    }
}
```

work-runtime-ktx:2.0.0

ViewModel scope

```
viewModelScope


lifecycleScope.launch {
    showHint()
    delat(5_000)
    showSecondHint()
}
```

State problem

State problem

```
class UserDownloader(val api: NetworkService) {
    private val users = mutableListOf<User>()

    fun all(): List<User> = users

    suspend fun downloadNext(num: Int) = coroutineScope {
        repeat(num) {
            val newUser = api.getUser()
            users.add(newUser)
        }
    }
}
```

Downloader.kt

State problem

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 22
     at java.util.ArrayList.add(ArrayList.java:463)
     at UserDownloader$downloadNext$2.invokeSuspend(Main.kt:21)
     at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:32)
     at kotlinx.coroutines.DispatchedTask.run(Dispatched.kt:236)
     at kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(CoroutineScheduler.kt:594)
     at kotlinx.coroutines.scheduling.CoroutineScheduler.access$runSafely
(CoroutineScheduler.kt:60)
     at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(CoroutineScheduler.kt:742)
     Suppressed: java.lang.ArrayIndexOutOfBoundsException: 244
          ... 7 more
```

995294

Downloader.kt

State problem

```
var counter = 0

fun main() = runBlocking {
    GlobalScope.massiveRun {
        counter++
    }
    println("Counter = $counter")
}

suspend fun CoroutineScope.massiveRun(action: suspend () -> Unit) {
    val jobs = List(1000) {
        launch {
            repeat(1000) { action() }
        }
    }
    jobs.forEach { it.join() }
}
```

s1.kt

AtomicInteger

```kotlin
private var counter = AtomicInteger()

fun main() = runBlocking {
    GlobalScope.massiveRun {
        counter.incrementAndGet()
    }
    println("Counter = ${counter.get()}")
}
```

s2.kt

Thread confinement fine-grained

```kotlin
private var counter = 0

fun main() = runBlocking {
    val counterContext =
        newSingleThreadContext("CounterContext")

    GlobalScope.massiveRun {
        withContext(counterContext) {
            counter++
        }
    }
    println("Counter = $counter")
}
```

s3.kt

Mutex

```
private val mutex = Mutex()
private var counter = 0

fun main() = runBlocking {
    GlobalScope.massiveRun {
        mutex.withLock {
            counter++
        }
    }
    println("Counter = $counter")
}
```

s4.kt

Task 8

Fix Downloader.kt: it should always print the correct answer that is 1 000 000.

Task 9: Make factory

- You need to use machines (`Machine`) to produce codes. Create them using `FactoryControl::makeMachine`.
- You need 20 codes. Store them using `FactoryControl::storeCode`.
- There mustn't be more than 5 non-broken machines at the same time.
- Cannot use machine to produce a code more often than every 1000 ms
- Cannot produce a machine more often than every 800 ms

Task 9: Make factory

- We have a worker who makes machines every 800ms as long as there is less than 5 of them.
  - He won't produce more than 1000 machines. Please, use `repeat(1000)` instead of `while(true)`
- Every machine produces a code using `structured.produce` function every second. It saves this code to shared space.
  - In case of an error, it stops working.
  - Machine won't produce more than 1000 codes. Please, use `repeat(1000)` instead of `while(true)`
- We have a single manager that takes codes one after another and stores them using `control.storeCode`. Note that is it time consuming operation. He is the only one who can do that.
  - In case of no codes, he sleeps for 100ms
  - He ends everything when there are 20 codes stored.
  - He won't do it more than 1000 times. Please, use `repeat(1000)` instead of `while(true)`

Task: Make factory

```
suspend fun makeMachine(control: FactoryControl) = coroutineScope {
    val machine = control.makeMachine()
    repeat(1000) {
        delay(1000)
        codes += machine.produce()
    }
}
```

Task: Make factory, Machine

```
suspend fun makeWorker(control: FactoryControl) = coroutineScope {
    var activeMachines = AtomicInteger()
    supervisorScope {
        repeat(1000) {
            delay(800)
            if (activeMachines.get() < 5) {
                activeMachines.incrementAndGet()
                launch {
                    try {
                        makeMachine( control)
                    } catch (e: ProductionError) {
                        activeMachines.decrementAndGet()
                    }
                }
            }
        }
    }
}
```

Task: Make factory, Manager

```
suspend fun makeManager(scope: CoroutineScope, control: FactoryControl) = coroutineScope {
    launch {
        var storedCodes = 0
        repeat(1000) {
            if (codes.isNotEmpty()) {
                val code = mutex.withLock { codes.removeAt(0) }
                control.storeCode(code)
                storedCodes++
                if (storedCodes >= 20) {
                    scope.cancel()
                    return@launch
                }
            } else {
                delay(100)
            }
        }
    }
}
```

Task 10: API

Open backend/Api.kt and implement the missing functions. Tests in backend/ApiTests.kt.

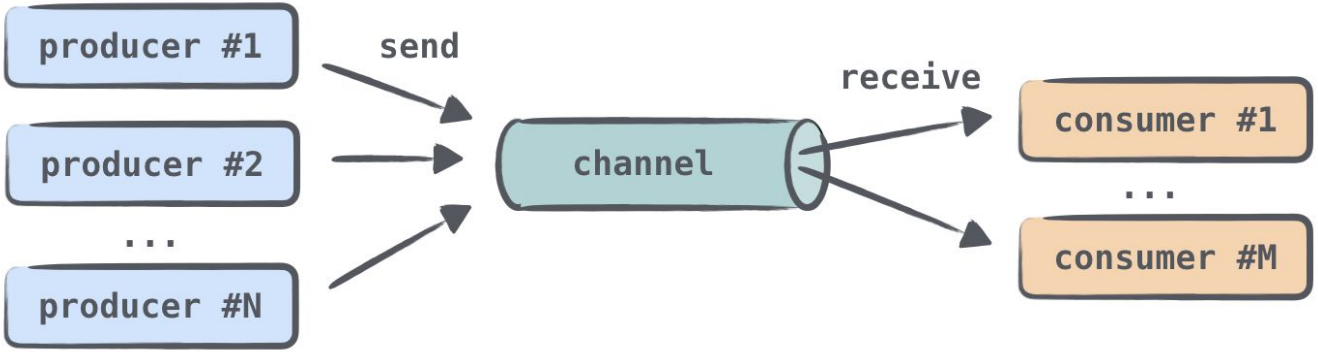
Task 11. Github 1

Fetch GitHub API for contributors to the JetBrains repositories, and aggregate them to have a name to the number of all contributions. Implement `getAggregatedContributions` in github/aggregated.kt.

Channels & Actors

Channel


coroutine #1
send
channel
receive
coroutine #2


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

Channel

```
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>
```

Channel

```
fun main() = runBlocking {
    val channel = Channel<Int>()
    launch {
        repeat(5) { index ->
            channel.send(index * 2)
            delay(1000)
        }
    }

    repeat(5) {
        val received = channel.receive()
        print(received)
    }
}
```

c1.kt

Channel

```
fun main() = runBlocking {
    val channel = Channel<Int>()
    launch {
        repeat(5) { index ->
            channel.send(index * 2)
            delay(1000)
        }
        channel.close()
    }

    for (i in channel) {
        print(i)
    }
}
```

c2.kt

Produce

```
fun CoroutineScope.produceNumbers() = produce {
    var x = 1
    while (true) send(x++)
}
```

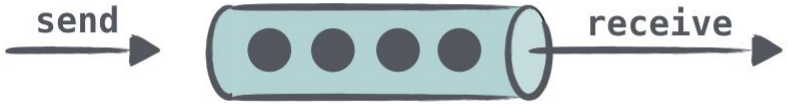
Channel types

Unlimited channel

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

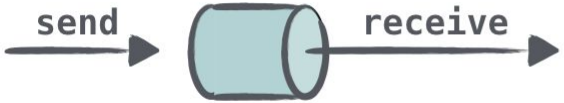
Buffered channel

```
val channel = Channel<Int>(20)
val channel = Channel<Int>(Channel.BUFFERED)
```



“Rendezvous” channel

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



Conflated channel

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

“Rendezvous” channel

```
fun main() = runBlocking<Unit> {
    val channel = Channel<String>() // Same as Channel<String>(Channel.RENDEZVOUS)

    launch {
        var i = 1
        repeat(5) {
            channel.send("Ping ${i++}")
            println("Message sent")
        }
        channel.close()
    }

    launch {
        for(text in channel) {
            println(text)
            delay(1000)
        }
    }
}
```

```
Ping 1
Message sent
Ping 2
Message sent
Ping 3
Message sent
Ping 4
Message sent
Ping 5
Message sent
```

c3.kt

Unlimited channel

```
fun main() = runBlocking<Unit> {
    val channel = Channel<String>(Channel.UNLIMITED)

    launch {
        var i = 1
        repeat(5) {
            channel.send("Ping ${i++}")
            println("Message sent")
        }
        channel.close()
    }

    launch {
        for(text in channel) {
            println(text)
            delay(1000)
        }
    }
}
```

```
Message sent
Message sent
Message sent
Message sent
Message sent
Ping 1
Ping 2
Ping 3
Ping 4
Ping 5
```

c4.kt

Buffered channel

```
fun main() = runBlocking<Unit> {
    val channel = Channel<String>(3)

    launch {
        var i = 1
        repeat(5) {
            channel.send("Ping ${i++}")
            println("Message sent")
        }
        channel.close()
    }

    launch {
        for(text in channel) {
            println(text)
            delay(1000)
        }
    }
}
```

```
Message sent
Message sent
Message sent
Ping 1
Message sent
Ping 2
Message sent
Ping 3
Ping 4
Ping 5
```

c5.kt

Conflated channel

```
fun main() = runBlocking<Unit> {
    val channel = Channel<String>(Channel.CONFLATED)

    launch {
        var i = 1
        repeat(5) {
            channel.send("Ping ${i++}")
            println("Message sent")
        }
    }

    launch {
        for(text in channel) {
            println(text)
            delay(1000)
        }
    }
}
```

```
Message sent
Message sent
Message sent
Message sent
Message sent
Ping 5
```

c6.kt

Task 12. Github 2

In github/Channels.kt, fetch GitHub API for contributors to the JetBrains repositories and:

- In `getContributions` send each list of contributors to `usersChannel` straight after receiving it.
- In `getAggregatedContributionsChannel` after each fetched list of contributions, update your aggregated list and send it to `usersChannel`.

Pipelines

```
fun CoroutineScope.produceNumbers() = produce<Int> {
   var x = 1
   while (true) send(x++)
}

fun CoroutineScope.square(
   numbers: ReceiveChannel<Int>
): ReceiveChannel<Int> = produce {
   for (x in numbers) send(x * x)
}

fun main() = runBlocking {
   val numbers = produceNumbers()
   val squares = square(numbers)
   for (i in 1..5) println(squares.receive())
   println("Done!")
   coroutineContext.cancelChildren()
}
```

c7.kt

Fan-out

```
fun CoroutineScope.produceNumbers() = produce {
    var x = 1 // start from 1
    while (true) {
        send(x++)
        delay(100)
    }
}

fun CoroutineScope.launchProcessor(id: Int, channel: ReceiveChannel<Int>) =
launch {
    for (msg in channel) {
        println("Processor #$id received $msg")
    }
}

fun main() = runBlocking {
    val producer = produceNumbers()
    repeat(5) { launchProcessor(it, producer) }
    delay(950)
    producer.cancel()
}
```

c8.kt

Fan-in

```
suspend fun sendString(channel: SendChannel<String>, s: String, time: Long) {
    while (true) {
        delay(time)
        channel.send(s)
    }
}

fun main() = runBlocking {
    val channel = Channel<String>()
    launch { sendString(channel, "foo", 200L) }
    launch { sendString(channel, "BAR!", 500L) }
    repeat(6) {
        println(channel.receive())
    }
    coroutineContext.cancelChildren()
}
```

c9.kt

Task 13. Coffee 2

You cannot assume that every thread is a different barista. You need a barista to be a coroutine (`launch`). Also replace `longOperation()` with `delay(1000)`.

a) Implement it in the way that orders are optimally handled by baristas.
b) You have only one coffee machine. Instead of `makeEspresso` use `EspressoMachine` that is shared between baristas.
c) Improve the coffee machine to not block thread but suspend it.

Ticker

```
fun main() = runBlocking {
   val tickerChannel = ticker(delayMillis = 100, initialDelayMillis = 0)
   var nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
   println("Initial element is available immediately: $nextElement")

   nextElement = withTimeoutOrNull(50) { tickerChannel.receive() }
   println("Next element is not ready in 50 ms: $nextElement")

   nextElement = withTimeoutOrNull(60) { tickerChannel.receive() }
   println("Next element is ready in 100 ms: $nextElement")

   println("Consumer pauses for 150ms")
   delay(150)

   nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
   println("Next element is available immediately after large consumer delay:
$nextElement")
   nextElement = withTimeoutOrNull(60) { tickerChannel.receive() }
   println("Next element is ready in 50ms after consumer pause in 150ms: $nextElement")
   tickerChannel.cancel()
}
```

c10.kt

Channels as actors

```
fun CoroutineScope.counterActor(): Channel<CounterMsg> {
    val channel = Channel<CounterMsg>()
    launch {
        var counter = 0
        for (msg in channel) {
            when (msg) {
                is IncCounter -> counter++
                is GetCounter -> msg.response.complete(counter)
            }
        }
    }
    return channel
}

fun main() = runBlocking<Unit> {
    val channel = counterActor()
    GlobalScope.massiveRun { channel.send(IncCounter) }
    val response = CompletableDeferred<Int>()
    channel.send(GetCounter(response))
    println("Counter = ${response.await()}")
    channel.close()
}
```

c11.kt

Actors

```
sealed class CounterMsg
object IncCounter : CounterMsg()
class GetCounter(val response: CompletableDeferred<Int>) : CounterMsg()

fun CoroutineScope.counterActor() = actor<CounterMsg> {
   var counter = 0
   for (msg in channel) {
       when (msg) {
           is IncCounter -> counter++
           is GetCounter -> msg.response.complete(counter)
       }
   }
}

fun main(args: Array<String>) = runBlocking<Unit> {
   val counter = counterActor()
   GlobalScope.massiveRun { counter.send(IncCounter) }
   val response = CompletableDeferred<Int>()
   counter.send(GetCounter(response))
   println("Counter = ${response.await()}")
   counter.close()
}
```

c12.kt

Task 14: Make factory with actors

- Worker is informed every 800. If there are less than 5 machines it produces a new one.
- Every machine produces a code using `structured.produce` function every second or breaks (random).
- Manager collects all the codes, and stores them using `storeCode`, and if there are more than 20 stored is ends everything.

Task: Make factory with actors

```
fun CoroutineScope.setupFactory(control: FactoryControl) {
    val managerChannel = managerActor(control)
    val workerChannel = workerActor(control, managerChannel)
    launch {
        repeat(1000) {
            delay(800)
            workerChannel.send(Thick())
        }
        managerChannel.close()
        workerChannel.close()
    }
}
```

Task: Make factory with actors

```
fun CoroutineScope.managerActor(control: FactoryControl) = actor<ManagerMessages> {
    var counter = 0
    for (msg in channel) {
        when (msg) {
            is NewCode -> {
                control.storeCode(msg.code)
                counter++
                if (counter >= 20) {
                    cancel()
                }
            }
        }
    }
}

class NewCode(val code: String) : ManagerMessages
```

Task: Make factory with actors

```
fun CoroutineScope.workerActor(control: FactoryControl, managerChannel:
SendChannel<NewCode>) = actor<WorkerMessages> {
    var machinesAlive = 0
    for (msg in channel) {
        when (msg) {
            is Thick -> {
                if (machinesAlive < 5) {
                    startMachine(control, channel, managerChannel)
                    machinesAlive++
                }
            }
            is MachineDestroyed -> {
                machinesAlive--
            }
        }
    }
}

class Thick : WorkerMessages
class MachineDestroyed : WorkerMessages
```

Task: Make factory with actors

```
fun CoroutineScope.startMachine(control: FactoryControl, workerChannel:
SendChannel<WorkerMessages>, managerChannel: SendChannel<NewCode>) {
    val machine = control.makeMachine()
    launch {
        try {
            repeat(1000) {
                delay(1000)
                val code = machine.produce()
                managerChannel.send(NewCode(code))

            }
        } catch (error: ProductionError) {
            workerChannel.send(MachineDestroyed())
        }
    }
}
```

Flow

Cold flows, hot channels

List

Sequence

Channel

Flow

Cold flows, hot channels

```
val channel = produce {
    while (true) {
        val x = computeNextValue()
        send(x)
    }
}
```

Cold flows, hot channels

```
val flow = flow {
    while (true) {
        val x = computeNextValue()
        emit(x)
    }
}
```

Cold flows, hot channels

```
fun main() = runBlocking {
    println("Started producing")
    val channel = produce {
        println("Channel started")
        for (i in 1..3) {
            delay(100)
            send(i)
        }
    }

    delay(100)
    println("Calling channel...")
    channel.consumeEach { value -> println(value) }
    println("Consuming again...")
    channel.consumeEach { value -> println(value) }
}
```

```
Started producing
Channel started
Calling channel...
1
2
3
Consuming again...
```

c13.kt

Cold flows, hot channels

```
fun main() = runBlocking {
    println("Started producing")
    val channel = flow {
        println("Flow started")
        for (i in 1..3) {
            delay(100)
            emit(i)
        }
    }

    delay(100)
    println("Calling flow...")
    channel.collect { value -> println(value) }
    println("Consuming again...")
    channel.collect { value -> println(value) }
}
```

```
Started producing
Calling flow...
Flow started
1
2
3
Consuming again...
Flow started
1
2
3
```

f1.kt

Flow builders

`flow { emit(value) }` - Typical flow builder emitting next values

`flowOf(value1, value2, ...)` - Flow with all the values

`list.asFlow()` - Flow with all the values on Iterable, Array or Sequence

`function.asFlow()` - Flow with value returned from a function

`channelFlow { send(value) }` - Flow similar to a channel

`callbackFlow<Int> { send(value) }` - For making flow from callbacks

onEach

```
flowOf(1,2,3)
        .onEach { print(it) } // 123
        .map { it * 10 }
        .collect {}
```

```
fun <T> Flow<T>.onEach(
    action: suspend (T) -> Unit
): Flow<T> = transform { value ->
    action(value)
    emit(value)
}
```

f2.kt

map

```
flowOf(1,2,3)
        .onEach { print(it) } // 123
        .map { it * 10 }
        .collect { print(it) } // 102030
```

```
fun <T, R> Flow<T>.map(
    crossinline transform: suspend (value: T) -> R
): Flow<R> = transform { value ->
    emit(transform(value))
}
```

f3.kt

filter

```
flowOf(1,2,3)
        .filter { it % 2 == 1 }
        .collect { print(it) } // 13
```

```
fun <T> Flow<T>.filter(
    crossinline predicate: suspend (T) -> Boolean
): Flow<T> = transform { value ->
    if (predicate(value)) emit(value)
}
```

f4.kt

Scan

```
(1..10).asFlow()
        .scan(0) { acc, v -> acc + v }
        .collect { println(it) }
```

```
0
1
3
6
10
15
21
28
36
45
55
```

```
fun <T, R> Flow<T>.scan(
    initial: R,
    operation: suspend (accumulator: R, value: T) -> R
): Flow<R> = flow {
    var accumulator: R = initial
    emit(accumulator)
    collect { value ->
        accumulator = operation(accumulator, value)
        emit(accumulator)
    }
}
```

Similar to fold, but emitting intermediate values

f5.kt

flatMapConcat

```kotlin
suspend fun main() {
    measureTimeMillis {
        ('A'..'C').asFlow()
                .flatMapConcat { flowFrom(it) }
                .collect { print(it) }
                // A_0 A_1 A_2 B_0 B_1 B_2 C_0 C_1 C_2
    }.let(::print) // 9060
}


fun flowFrom(elem: Any) = flowOf(0, 1, 2)
        .onEach { delay(1000) }
        .map { "${it}_${elem} " }
```

f6.kt

flatMapMerge

```
suspend fun main() {
    measureTimeMillis {
        ('A'..'C').asFlow()
                .flatMapMerge { flowFrom(it) }
                .collect { print(it) }
                // A_0 B_0 A_1 C_0 B_1 A_2 B_2 C_1 C_2
    }.let(::print) // 3079
}


fun flowFrom(elem: Any) = flowOf(0, 1, 2)
        .onEach { delay(1000) }
        .map { "${it}_${elem} " }
```

f7.kt

flatMapMerge

```
suspend fun main() {
    measureTimeMillis {
        ('A'..'C').asFlow()
                .flatMapMerge(concurrency = 2) { flowFrom(it) }
                .collect { print(it) }
                // 0_A 0_B 1_B 1_A 2_B 2_A 0_C 1_C 2_C
    }.let(::print) // 6129
}


fun flowFrom(elem: Any) = flowOf(0, 1, 2)
        .onEach { delay(1000) }
        .map { "${it}_${elem} " }


flatMapMerge(concurrency = 1) == flatMapConcat
```

f8.kt

flatMapLatest

```kotlin
suspend fun main() {
    measureTimeMillis {
        ('A'..'C').asFlow()
                .onEach { delay(1500) }
                .flatMapLatest { flowFrom(it) }
                .collect { print(it) } // 0_A 0_B 0_C 1_C 2_C
    }.let(::print) // 7620
}
```

f9.kt

Buffer

```
suspend fun main() = coroutineScope {
    flowOf("A", "B", "C")
        .onEach  { println("onEach $it") }
        .collect { println("collect $it") }
}
```

```
onEach A
collect A
onEach B
collect B
onEach C
collect C
```

```
suspend fun main() = coroutineScope {
    flowOf("A", "B", "C")
        .onEach  { println("onEach $it") }
        .buffer(100)
        .collect { println("collect $it") }
}
```

```
onEach A
onEach B
onEach C
collect A
collect B
collect C
```

f10.kt

Conflate

```
suspend fun main() = coroutineScope {
    val flow = flow {
        for (i in 1..30) {
            delay(100)
            emit(i)
        }
    }

    print(flow.onEach { delay(1000) }.toList())
    // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]

    print(flow.conflate().onEach { delay(1000) }.toList())
    // [1, 10, 20, 30]
}
```

f11.kt

Zip and combine

```kotlin
suspend fun main() {
    val f1 = flowOf(1, 2, 3).onEach { delay(1000) }
    val f2 = flowOf("A", "B", "C").onEach { delay(800) }

    f1.zip(f2) { t1, t2 -> t2 + t1}
            .collect { print("$it ") } // A1 B2 C3

    f1.combine(f2) { t1, t2 -> t2 + t1}
            .collect { print("$it ") } // A1 B1 B2 C2 C3

    flowOf(f1, f2).flattenConcat()
            .collect { print("$it ") } // 1 2 3 A B C

    flowOf(f1, f2).flattenMerge()
            .collect { print("$it ") } // A 1 B 2 C 3
}
```

CombiningFlows.kt

Zip and combine

```
suspend fun main() {
    val nums = (1..3).asFlow().onEach { delay(300) }
    val strs = flowOf("one", "two", "three").onEach { delay(400) }

    println("Zip:")
    val startTime = ct
    nums.zip(strs) { a, b -> "$a -> $b" }
            .collect { value ->
                println("$value at ${ct - startTime} ms")
            }

    println("Combine:")
    val startTime2 = ct
    nums.combine(strs) { a, b -> "$a -> $b" }
            .collect { value ->
                println("$value at ${ct - startTime2} ms")
            }
}

val ct: Long get() = System.currentTimeMillis()
```

```
Zip:
1 -> one at 468 ms
2 -> two at 873 ms
3 -> three at 1277 ms
Combine:
1 -> one at 415 ms
2 -> one at 619 ms
2 -> two at 818 ms
3 -> two at 924 ms
3 -> three at 1222 ms
```

f12.kt

Task 15. Flow Kata

Open flow/Kata.kt and implement the missing functions.

ChannelFlow

```
fun <T> Flow<T>.merge(other: Flow<T>): Flow<T> = channelFlow {
    launch {
        collect { send(it) }
    }
    other.collect { send(it) }
}

fun <T> contextualFlow(): Flow<T> = channelFlow {
    launch(Dispatchers.IO) {
        send(computeIoValue())
    }
    launch(Dispatchers.Default) {
        send(computeCpuValue())
    }
}
```

callbackFlow

```
fun flowFrom(api: CallbackBasedApi): Flow<T> = callbackFlow {
    val callback = object : Callback {
        override fun onNextValue(value: T) {
            try {
                sendBlocking(value)
            } catch (e: Exception) {
                // Handle exception from the channel:
                // failure in flow or premature closing
            }
        }
        override fun onApiError(cause: Throwable) {
            cancel(CancellationException("API Error", cause))
        }
        override fun onCompleted() = channel.close()
    }
    api.register(callback)
    awaitClose { api.unregister(callback) }
}
```

Exceptions

```
fun main() = runBlocking {
    try {
        launch {
            throwing()
        }
    } catch (e: IllegalStateException) {
        print("Caught")
    }
    print("Done")
}
```

Exception in thread "main" java.lang.IllegalStateException

Exceptions.kt

Exceptions

```
fun main() = runBlocking {
    try {
        val async = async {
            throwing()
        }
        async.await()
    } catch (e: IllegalStateException) {
        print("Caught")
    }
    print("Done")
}
```

Exception in thread "main" java.lang.IllegalStateException

Exceptions.kt

Exceptions

```
fun main() = runBlocking {
    try {
        coroutineScope {
            throwing()
        }
    } catch (e: IllegalStateException) {
        print("Caught")
    }
}
```

Caught

Exceptions.kt

Exceptions

```
fun main() = runBlocking {
    try {
        val async = async {
            throwing()
        }
        async.await()
    } catch (e: IllegalStateException) {
        print("Caught")
    }
    print("Done")
}
```

Exception in thread "main" java.lang.IllegalStateException

Exceptions.kt

Exceptions

```
fun main() = runBlocking {
    val channel = produce {
        send(1)
        send(2)
        throwing()
    }
    try {
        for (e in channel) {
            println("Got it")
        }
    } catch (e: IllegalStateException) {
        println("Caught")
    }
}
```

```
Got it
Got it
Caught
Exception in thread "main"
java.lang.IllegalStateException
```

Exceptions.kt

Exceptions

```
fun main() = runBlocking {
    val channel = produce(capacity = UNLIMITED) {
        send(1)
        send(2)
        throwing()
    }
    delay(100)
    try {
        for (e in channel) {
            println("Got it")
        }
    } catch (e: IllegalStateException) {
        println("Caught")
    }
}
```

Exception in thread "main"
java.lang.IllegalStateException

Exceptions.kt

Exceptions

```
fun main() = runBlocking<Unit> {
    val flow = flow {
        emit(1)
        emit(2)
        throwing()
    }
    try {
        flow.collect { println("Got it") }
    } catch (e: IllegalStateException) {
        println("Caught")
    }
}
```

```
Got it
Got it
Caught
```

Exceptions.kt

flowOn

```
withContext(Dispatchers.Main) {
   val singleValue = flow { ... } // will be executed on IO if context
wasn't specified before
       .map { ... } // Will be executed in IO
       .flowOn(Dispatchers.IO)
       .filter { ... } // Will be executed in Default
       .flowOn(Dispatchers.Default)
       .single() // Will be executed in the Main
}
```

f13.kt

Flow lifecycle

```
fun main() = runBlocking<Unit> {
    flow {
        (1..5).forEach {
            delay(1000)
            emit(it)
            if (it == 2) throw RuntimeException("Error on $it")
        }
    }   .onEach { println("On each $it") }
        .onStart { println("Starting flow") }
        .onCompletion { println("Flow completed") }
        .catch { ex -> println("Exception message: ${ex.message}") }
        .toList()
}
```

```
Starting flow
On each 1
On each 2
Flow completed
Exception message: Error on 2
```

f14.kt

Collect is suspending

```
fun main() = runBlocking<Unit> {
    measureTimeMillis {
        (1..5).asFlow()
                .onEach { event -> delay(100) }
                .collect() // We wait 500ms

        (1..5).asFlow()
                .onEach { event -> delay(100) }
                .collect() // We wait 500ms

    }.let(::print) // 1049
}
```

f15.kt

Async coroutines

```
fun main() = measureTimeMillis {
    runBlocking {
        measureTimeMillis {
            (1..5).asFlow()
                    .onEach { event -> delay(100) }
                    .launchIn(this)

            (1..5).asFlow()
                    .onEach { event -> delay(100) }
                    .launchIn(this)

        }.let(::print) // 15
    }
}.let(::print) // 591
```

f16.kt

Async coroutines

```
fun main() = runBlocking {
    flowOf(1,2,3)
            .map { it / 0 }
            .catch { emit(-1) }
            .collect { it / 0 }
}
```

ArithmeticException: / by zero

f17.kt

Async coroutines

```
fun main() = runBlocking {
    try {
        flowOf(1, 2, 3)
                .map { it / 0 }
                .catch { emit(-1) }
                .collect { it / 0 }
    } catch (e: ArithmeticException) {
        print("Got it")
    }
}
```

Got it

f18.kt

Async coroutines

```
flowOf(1, 2, 3)
        .map { it / 0 }
        .catch { emit(-1) }
        .onEach { it / 0 }
        .catch { print("Got it") }
        .collect()
```

Got it

f19.kt

Task 16. Factory based on flow

Implement a factory using a flow. In this case, machines do not break. You should start by creating every 800 ms a machine, and this machine should produce codes every second. You should produce 20 codes in total. Each code should be consumed using `control.storeCode`.

StateFlow

StateFlow

```
suspend fun main() = coroutineScope {
    val state = MutableStateFlow(1)
    println(state.value) // 1
    delay(1000)
    launch {
        state.collect { println("Value changed to $it") } // Value changed to 1
    }
    delay(1000)
    state.value = 2 // Value changed to 2
    delay(1000)
    launch {
        state.collect { println("and now it is $it") } // and now it is 2
    }
    delay(1000)
    state.value = 3 // Value changed to 3 and now it is 3
}
```

sf1.kt

Best practices

```kotlin
val job: Job = Job()
val scope = CoroutineScope(Dispatchers.Default + job)
```

```kotlin
val job = SupervisorJob()
val scope = CoroutineScope(Dispatchers.Default + job)
```

Prefer the Main dispatcher for root coroutine

```
val scope = CoroutineScope(Dispatchers.Default)

fun login() = scope.launch {
    withContext(Dispatcher.Main) { view.showLoading() }
    networkClient.login(...)
    withContext(Dispatcher.Main) { view.hideLoading() }
}
```

```
val scope = CoroutineScope(Dispatchers.Main)

fun login() = scope.launch {
    view.showLoading()
    withContext(Dispatcher.IO) { networkClient.login(...) }
    view.hideLoading()
}
```

Avoid usage of unnecessary async/await

```
launch {
    val data = async { /* code */ }.await()
    val data2 = async(Dispatchers.Default) { /* code */ }.await()
}
```

```
launch {
    val data = /* code */
    val data2 = withContext(Dispatchers.Default) { /* code */ }
}
```

Avoid cancelling scope job

```
class WorkManager {
    val job = SupervisorJob()
    val scope = CoroutineScope(Dispatchers.Default + job)

    fun doWork1() {
        scope.launch { /* do work */ }
    }

    fun doWork2() {
        scope.launch { /* do work */ }
    }

    fun cancelAllWork() {
        job.cancel()
    }
}
```

Avoid cancelling scope job

```
class WorkManager {
    val job = SupervisorJob()
    val scope = CoroutineScope(Dispatchers.Default + job)

    fun doWork1() {
        scope.launch { /* do work */ }
    }

    fun doWork2() {
        scope.launch { /* do work */ }
    }

    fun cancelAllWork() {
        scope.coroutineContext.cancelChildren()
    }
}
```

When needed concrete dispatcher, set it using withContext

```
suspend fun login(): Result = withContext(Dispatcher.Main) {
    view.showLoading()

    val result = withContext(Dispatcher.IO) {
        someBlockingCall()
    }

    view.hideLoading()
    return result
}
```

Avoid usage of global scope

```
GlobalScope.launch { /*...*/ }
```

```
scope.launch { /*...*/ }
viewModelScope.launch { /*...*/ }
lifecycleScope.launch { /*...*/ }
```

That’s it! Thank you