KotlinConf

Deep dive into Coroutines on JVM

Roman Elizarov
elizarov at JetBrains

There is no magic

KotlinConf

Continuation Passing Style (CPS)

A toy problem

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

Direct style

Direct style

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

Direct style

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

$\left.\begin{array}{l} \\ \\ \end{array}\right\}$ Continuation

Continuation-Passing Style

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

$\left.\begin{array}{l}\texttt{val post = createPost(token, item)}\\ \texttt{processPost(post)}\end{array}\right\}$ Continuation

CPS == Callbacks

Continuation-Passing Style

```kotlin
fun postItem(item: Item) {
    requestToken { token ->
        createPost(token, item) { post ->
            processPost(post)
        }
    }
}
```

Coroutines Direct Style

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

How does it work?

Behind the scenes

Kotlin suspending functions

Kotlin

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

CPS Transformation

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

Java/JVM

```
Object createPost(Token token, Item item, Continuation<Post> cont) { … }
```

CPS Transformation

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

Java/JVM

callback

```
Object createPost(Token token, Item item, Continuation<Post> cont) { … }
```

Continuation

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

```
Object createPost(Token token, Item item, Continuation<Post> cont) { … }
```

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

Continuation

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

```
Object createPost(Token token, Item item, Continuation<Post> cont) { … }
```

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

Continuation

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

```
Object createPost(Token token, Item item, Continuation<Post> cont) { … }
```

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

Continuation

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

```
Object createPost(Token token, Item item, Continuation<Post> cont) { … }
```

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

Continuation

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

```
Object createPost(Token token, Item item, Continuation<Post> cont) { … }
```

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

Continuation is a generic callback interface

KotlinConf

Direct to CPS

Direct code

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

Continuations

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

Initial continuation

Continuations

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

$\left.\begin{array}{l}\texttt{val post = createPost(token, item)}\\ \texttt{processPost(post)}\\ \texttt{\}}\end{array}\right\}$ Continuation

Continuations

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

$\Big\}$ Continuation

Convert to CPS?

Callbacks?

```
fun postItem(item: Item) {

    requestToken { token ->

        createPost(token, item) { post ->

            processPost(post)

        }
    }
}
```

Labels

```kotlin
suspend fun postItem(item: Item) {
// LABEL 0
    val token = requestToken()
// LABEL 1
    val post = createPost(token, item)
// LABEL 2
    processPost(post)
}
```

Labels

```
suspend fun postItem(item: Item) {
    switch (label) {
      case 0:
          val token = requestToken()
      case 1:
          val post = createPost(token, item)
      case 2:
          processPost(post)
   }
}
```

State

```
suspend fun postItem(item: Item) {
    val sm = object : CoroutineImpl { … }
    switch (sm.label) {
      case 0:
          val token = requestToken()
      case 1:
          val post = createPost(token, item)
      case 2:
          processPost(post)
   }
}
```

CPS Transform

```
fun postItem(item: Item, cont: Continuation) {
    val sm = object : CoroutineImpl { … }
    switch (sm.label) {
      case 0:
          requestToken(sm)
      case 1:
          createPost(token, item, sm)
      case 2:
          processPost(post)
   }
}
```

Save state

```
fun postItem(item: Item, cont: Continuation) {
    val sm = …
    switch (sm.label) {
      case 0:
          sm.item = item
          sm.label = 1
          requestToken(sm)
      case 1:
          createPost(token, item, sm)
      case 2:
          processPost(post)
   }
}
```

Callback

```
fun postItem(item: Item, cont: Continuation) {
    val sm = object : CoroutineImpl { … }
    switch (sm.label) {
      case 0:
          sm.item = item
          sm.label = 1
          requestToken(sm)
      case 1:
          createPost(token, item, sm)
      case 2:
          processPost(post)
   }
}
```

State Machine as Continuation

Callback

```
fun postItem(item: Item, cont: Continuation) {
    val sm = object : CoroutineImpl {
        fun resume(…) {
            postItem(null, this)
        }
    }
    switch (sm.label) {
      case 0:
          sm.item = item
          sm.label = 1
          requestToken(sm)
      case 1:
          createPost(token, item, sm)
      …
}
```

Callback

```
fun postItem(item: Item, cont: Continuation) {
    val sm = cont as? ThisSM ?: object : ThisSM {
        fun resume(…) {
            postItem(null, this)
        }
    }
    switch (sm.label) {
      case 0:
          sm.item = item
          sm.label = 1
          requestToken(sm)
      case 1:
          createPost(token, item, sm)
      …
}
```

Restore state

```
fun postItem(item: Item, cont: Continuation) {
    val sm = …
    switch (sm.label) {
      case 0:
          sm.item = item
          sm.label = 1
          requestToken(sm)
      case 1:
          val item = sm.item
          val token = sm.result as Token
          sm.label = 2
          createPost(token, item, sm)
      …
}
```

Continue

```
fun postItem(item: Item, cont: Continuation) {
    val sm = …
    switch (sm.label) {
      case 0:
          sm.item = item
          sm.label = 1
          requestToken(sm)
      case 1:
          val item = sm.item
          val token = sm.result as Token
          sm.label = 2
          createPost(token, item, sm)
      …
}
```

State Machine vs Callbacks

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

```
fun postItem(item: Item) {
    requestToken { token ->
        createPost(token, item) { post ->
            processPost(post)
        }
    }
}
```

State Machine vs Callbacks

Reuse closure / state object

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

Create new closure

```
fun postItem(item: Item) {
    requestToken { token ->
        createPost(token, item) { post ->
            processPost(post)
        }
    }
}
```

State Machine vs Callbacks

```
suspend fun postItems(items: List<Item>) {
    for (item in items) {
        val token = requestToken()
        val post = createPost(token, item)
        processPost(post)
    }
}
```

Easy loops and higher-order functions

State Machine vs Callbacks

```
suspend fun postItems(items: List<Item>) {
    for (item in items) {
        val token = requestToken()
        val post = createPost(token, item)
        processPost(post)
    }
}
```

Easy loops and higher-order functions

A horrid callback mess

```
fun postItems(items: List<Item>) {
    …
}
```

KotlinConf

Integration

Zoo of futures on JVM

```kotlin
interface Service {
    fun createPost(token: Token, item: Item): Call<Post>
}
```

```kotlin
interface Service {
    fun createPost(token: Token, item: Item): Call<Post>
}

suspend fun createPost(token: Token, item: Item): Post =
    serviceInstance.createPost(token, item).await()
```

```
suspend fun <T> Call<T>.await(): T {
    …
}
```

Callbacks everywhere

```
suspend fun <T> Call<T>.await(): T {
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            // todo
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            // todo
        }
    })
}
```

```kotlin
suspend fun <T> Call<T>.await(): T = suspendCoroutine { cont ->
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            if (response.isSuccessful)
                cont.resume(response.body()!!)
            else
                cont.resumeWithException(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

```kotlin
suspend fun <T> suspendCoroutine(block: (Continuation<T>) -> Unit): T
suspend fun <T> suspendCoroutine(block: (Continuation<T>) -> Unit): T
```

Regular function

Inspired by **call/cc** from Scheme

Install callback

```
suspend fun <T> Call<T>.await(): T = suspendCoroutine { cont ->
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            if (response.isSuccessful)
                cont.resume(response.body()!!)
            else
                cont.resumeWithException(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

Install callback

```kotlin
suspend fun <T> Call<T>.await(): T = suspendCoroutine { cont ->
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            if (response.isSuccessful)
                cont.resume(response.body()!!)
            else
                cont.resumeWithException(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

Analyze response

```
suspend fun <T> Call<T>.await(): T = suspendCoroutine { cont ->
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            if (response.isSuccessful)
                cont.resume(response.body()!!)
            else
                cont.resumeWithException(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

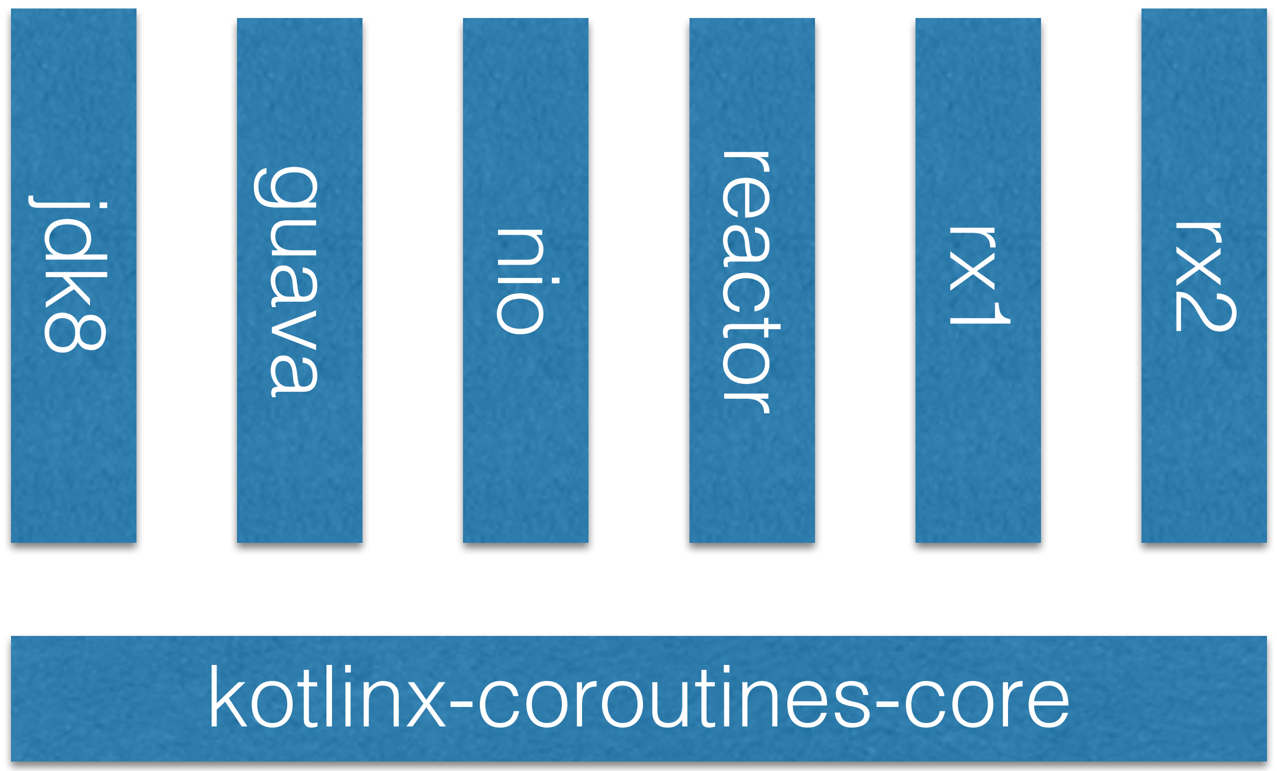
Analyze response

```
suspend fun <T> Call<T>.await(): T = suspendCoroutine { cont ->
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            if (response.isSuccessful)
                cont.resume(response.body()!!)
            else
                cont.resumeWithException(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

That's all

Out-of-the box integrations



Contributions are welcome

KotlinConf

Coroutine context

What thread it resumes on?

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

$\left.\begin{matrix} \\ \\ \end{matrix}\right\}$ Continuation

It depends!

What thread it resumes on?

```
fun postItem(item: Item) {
    launch(UI) {
        val token = requestToken()

        val post = createPost(token, item)

        processPost(post)
    }
}
```

$\left.\begin{matrix}\\ \\ \end{matrix}\right\}$ Continuation

Continuation Interceptor

```
interface ContinuationInterceptor : CoroutineContext.Element {

    companion object Key : CoroutineContext.Key<ContinuationInterceptor>

    fun <T> interceptContinuation(continuation: Continuation<T>):
                                                           Continuation<T>
}
```

Continuation Interceptor

```
interface ContinuationInterceptor : CoroutineContext.Element {

    companion object Key : CoroutineContext.Key<ContinuationInterceptor>

    fun <T> interceptContinuation(continuation: Continuation<T>):
                                                         Continuation<T>
}
```

Continuation Interceptor

```
interface ContinuationInterceptor : CoroutineContext.Element {

    companion object Key : CoroutineContext.Key<ContinuationInterceptor>

    fun <T> interceptContinuation(continuation: Continuation<T>):
                                                            Continuation<T>
}
```

Dispatched continuation

```
class DispatchedContinuation<in T>(
    val dispatcher: CoroutineDispatcher,
    val continuation: Continuation<T>
): Continuation<T> by continuation {

    override fun resume(value: T) {
       dispatcher.dispatch(context, DispatchTask(…))
    }

    …
}
```

Dispatches execution to another thread

KotlinConf

Starting coroutines

Coroutine builder

```
fun <T> future(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> T
): CompletableFuture<T>
```

A regular function

```
fun <T> future(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> T
): CompletableFuture<T>
```

```
fun <T> future(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> T
): CompletableFuture<T>
```

```
fun <T> future(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> T
): CompletableFuture<T>
```

suspending lambda

```kotlin
fun <T> future(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> T
): CompletableFuture<T> {
    val future = CompletableFuture<T>()
    block.startCoroutine(…)
    return future
}
```

```
fun <T> future(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> T
): CompletableFuture<T> {
    val future = CompletableFuture<T>()
    block.startCoroutine(…)
    return future
}
```

```
fun <T> future(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> T
): CompletableFuture<T> {
    val future = CompletableFuture<T>()
    block.startCoroutine(completion = object : Continuation<T> {
        …
    })
    return future
}
```

```kotlin
fun <T> future(…): CompletableFuture<T> {
    val future = CompletableFuture<T>()
    block.startCoroutine(completion = object : Continuation<T> {
        override val context: CoroutineContext get() = context

        override fun resume(value: T) {
            future.complete(value)
        }

        override fun resumeWithException(exception: Throwable) {
            future.completeExceptionally(exception)
        }
    })
    return future
}
```

```
fun <T> future(…): CompletableFuture<T> {
    val future = CompletableFuture<T>()
    block.startCoroutine(completion = object : Continuation<T> {
        override val context: CoroutineContext get() = context

        override fun resume(value: T) {
            future.complete(value)
        }

        override fun resumeWithException(exception: Throwable) {
            future.completeExceptionally(exception)
        }
    })
    return future
}
```

That’s all, folks!

KotlinConf

Job cancellation

Launch coroutine builder

```
fun launch(
    context: CoroutineContext = DefaultDispatcher,
    block: suspend () -> Unit
): Job { … }
```

Launching coroutine

```
val job = launch {
    …
}
```

```
val job = launch {
    …
}

job.join()
```

```
val job = launch {
    …
}

job.join()
job.cancel()
```

Job

```
interface Job : CoroutineContext.Element {

    companion object Key : CoroutineContext.Key<Job>

    …
}
```

Using coroutine context

```
launch {
    val job = coroutineContext[Job]!!
    …
}
```

Using coroutine context

```
launch {
    val job = coroutineContext[Job]!!
    val interceptor = coroutineContext[CoroutineInterceptor]!!
    …
}
```

Timeouts

```
launch {
    withTimeout(10, TimeUnit.SECONDS) {
        …
    }
}
```

Cooperative cancellation

Cooperative cancellation

```
launch {
    while (true) {
        …
    }
}
```

Cooperative cancellation

```
launch {
    while (isActive) {
        …
    }
}
```

Cooperative cancellation

```
launch {
    while (true) {
        delay(…)
        …
    }
}
```

Cancellable suspension

```
suspend fun <T> Call<T>.await(): T =
    suspendCancellableCoroutine { cont ->
        enqueue(…)
    }
```

Cancellable continuation

```
suspend fun <T> Call<T>.await(): T =
    suspendCancellableCoroutine { cont: CancellableContinuation<T> ->
        enqueue(…)
}
```

Completion handler

```
suspend fun <T> Call<T>.await(): T =
    suspendCancellableCoroutine { cont: CancellableContinuation<T> ->
        enqueue(…)
        cont.invokeOnCompletion {
            this@await.cancel()
        }
    }
```

Completion handler

```
suspend fun <T> Call<T>.await(): T =
    suspendCancellableCoroutine { cont: CancellableContinuation<T> ->
        enqueue(…)
        cont.invokeOnCompletion {
            this@await.cancel()
        }
    }
```

KotlinConf

Communicating Sequential Processes (CSP)

Shared Mutable State

The choice

Shared Mutable State ⟷ Share by Communicating

Example

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val chan = Channel<Int>()
    launch(coroutineContext) {
        repeat(10) { i ->
            delay(100)
            chan.send(i)
        }
        chan.close()
    }
    launch(coroutineContext) {
        for (i in chan) {
            println(i)
        }
    }
}
```

Main coroutine

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val chan = Channel<Int>()
    launch(coroutineContext) {
        repeat(10) { i ->
            delay(100)
            chan.send(i)
        }
        chan.close()
    }
    launch(coroutineContext) {
        for (i in chan) {
            println(i)
        }
    }
}
```

Channel

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val chan = Channel<Int>()
    launch(coroutineContext) {
        repeat(10) { i ->
            delay(100)
            chan.send(i)
        }
        chan.close()
    }
    launch(coroutineContext) {
        for (i in chan) {
            println(i)
        }
    }
}
```

Launch

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val chan = Channel<Int>()
    launch(coroutineContext) {
        repeat(10) { i ->
            delay(100)
            chan.send(i)
        }
        chan.close()
    }
    launch(coroutineContext) {
        for (i in chan) {
            println(i)
        }
    }
}
```

Child coroutine

Coroutine body

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val chan = Channel<Int>()
    launch(coroutineContext) {
        repeat(10) { i ->
            delay(100)
            chan.send(i)
        }
        chan.close()
    }
    launch(coroutineContext) {
        for (i in chan) {
            println(i)
        }
    }
}
```

Sequential code!

Send

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val chan = Channel<Int>()
    launch(coroutineContext) {
        repeat(10) { i ->
            delay(100)
            chan.send(i)
        }
        chan.close()
    }
    launch(coroutineContext) {
        for (i in chan) {
            println(i)
        }
    }
}
```

Close

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val chan = Channel<Int>()
    launch(coroutineContext) {
        repeat(10) { i ->
            delay(100)
            chan.send(i)
        }
        chan.close()
    }
    launch(coroutineContext) {
        for (i in chan) {
            println(i)
        }
    }
}
```

Receive for loop

```kotlin
fun main(args: Array<String>) = runBlocking<Unit> {
    val chan = Channel<Int>()
    launch(coroutineContext) {
        repeat(10) { i ->
            delay(100)
            chan.send(i)
        }
        chan.close()
    }
    launch(coroutineContext) {
        for (i in chan) {
            println(i)
        }
    }
}
```

Demo

Actors

The other way to look at CSP

The choice

Named channels ↔ Named coroutines

Actor == named coroutine & inbox channel

Example

```kotlin
fun main(args: Array<String>) = runBlocking<Unit> {
    val printer = actor<Int>(coroutineContext) {
        for (i in channel) {
            println(i)
        }
    }
    launch(coroutineContext) {
        repeat(10) { i ->
            delay(100)
            printer.send(i)
        }
        printer.close()
    }
}
```

Actor coroutine builder

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val printer = actor<Int>(coroutineContext) {
        for (i in channel) {
            println(i)
        }
    }
    launch(coroutineContext) {
        repeat(10) { i ->
            delay(100)
            printer.send(i)
        }
        printer.close()
    }
}
```

Actor body

```kotlin
fun main(args: Array<String>) = runBlocking<Unit> {
    val printer = actor<Int>(coroutineContext) {
        for (i in channel) {
            println(i)
        }
    }
    launch(coroutineContext) {
        repeat(10) { i ->
            delay(100)
            printer.send(i)
        }
        printer.close()
    }
}
```

Sequential!

Interacting with an actor

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val printer = actor<Int>(coroutineContext) {
        for (i in channel) {
            println(i)
        }
    }
    launch(coroutineContext) {
        repeat(10) { i ->
            delay(100)
            printer.send(i)
        }
        printer.close()
    }
}
```

KotlinConf

References

Guide to **kotlinx.coroutines** by example

- Basics
- Cancellation and Timeouts
- Composition
- Coroutine contexts
- Channels
- Shared Mutable State and Concurrency
- Select expressions

https://github.com/Kotlin/kotlinx.coroutines/blob/master/coroutines-guide.md

Thank you

Roman Elizarov
elizarov at JetBrains
relizarov

Any questions?