

Creating asynchronous applications with Kotlin coroutines



Konrad Kamiński
[Allegro.pl](https://allegro.pl)

Agenda

- Introduction to coroutines.
- Threads and lifecycle of coroutines.
- Concurrent programming.

Every part will conclude with exercises.
They require IntelliJ IDEA Community/Ultimate
with Kotlin 1.3 support.

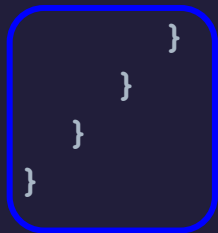
```
fun sendAuditMessage(userId: String) {  
    val accountId = getUserAccountId(userId)  
    val productCount = getProductCount(accountId)  
    sendMessage("User $userId has got $productCount products")  
}
```

```
suspend fun getUsername(userId: Int): String =  
    withContext(requestId.asContextElement()) {  
        loggingGetUser(userId).name  
    }
```

```
fun getUserAccountId(userId: String): String  
fun getProductCount(accountId: String): Int  
fun sendMessage(message: String): Unit
```



```
fun sendAuditMessage(userId: String, callback: () -> Unit) {  
    getUserAccountId(userId) { accountId ->  
        getProductCount(accountId) { productCount ->  
            sendMessage("User $userId has got $productCount products") {  
                callback.invoke()  
            }  
        }  
    }  
}
```



← Callback hell

```
fun getUserAccountId(userId: String, callback: (String) -> Unit): Unit  
fun getProductCount(accountId: String, callback: (Int) -> Unit): Unit  
fun sendMessage(message: String, callback: () -> Unit): Unit
```



```
fun sendAuditMessage(userId: String)=
    getUserAccountId(userId)
        .thenCompose { accountId ->
            getProductCount(accountId)
        }
        .thenAccept { productCount ->
            sendMessage("User $userId has got $productCount products")
        }
```

Combinators

```
fun getUserAccountId(userId: String): CompletableFuture<String>
fun getProductCount(accountId: String): CompletableFuture<Int>
fun sendMessage(message: String)
```



```
suspend fun sendAuditMessage(userId: String) {  
    val accountId = getUserAccountId(userId)  
    val productCount = getProductCount(accountId)  
    sendMessage("User $userId has got $productCount products")  
}
```

```
suspend fun getUserAccountId(userId: String): String  
suspend fun getProductCount(accountId: String): Int  
suspend fun sendMessage(message: String): Unit
```



```
suspend fun sendAuditMessage(userId: String): Unit

fun sendAuditMessage(userId: String, callback: Continuation<Unit>): Any?

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

val COROUTINE_SUSPENDED: Any = Any()
```



```
suspend fun myfun(param: Int): String =
    suspendCoroutine { callback: Continuation<String> ->
        // usually we'll invoke callback methods in a different thread
        // if we want to return a value
        callback.resumeWith(
            Result.success("Result of myfun with $param"))

        // if we want to throw an exception
        callback.resumeWith(
            Result.failure(Exception("Something went wrong")))
    }

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




```
launch {  
    //suspending functions can be invoked here  
}  
  
fun CoroutineScope.launch(block: suspend CoroutineScope.() -> Unit): Job  
  
interface CoroutineScope  
  
object GlobalScope: CoroutineScope  
  
interface Job {  
    suspend fun join()  
}  
  
suspend fun Collection<Job>.joinAll()
```



```
runBlocking {  
    //suspending functions can be invoked here  
}
```

```
fun <T> runBlocking(block: suspend CoroutineScope.() -> T): T
```



```
coroutineScope {  
    //all coroutines created here will have to be finished  
    //before we „exit“ this coroutineScope  
}
```

```
suspend fun <T> coroutineScope(block: suspend CoroutineScope.() -> T): T
```



Exercises

- Implement tasks 1-8. They are in `Task1.kt`, `Task2.kt`, etc. files.
- `Task5` is in `Task5.java` file.
- Tests for tasks are in `Test1`, `Test2`, etc. classes.
- Each test should be executed in separate VM.
`gradlew testPart1` should be enough.



```
interface CoroutineContext {  
    operator fun <E: Element> get(key: Key <E>): E?  
  
    operator fun plus(context: CoroutineContext): CoroutineContext  
}  
  
interface CoroutineScope {  
    val coroutineContext: CoroutineContext  
}  
  
runBlocking {  
    val job = coroutineContext[Job]!!  
    val dispatcher = coroutineContext[CoroutineInterceptor]!!  
    println(job)  
}
```



```
fun CoroutineScope.launch(  
    context: CoroutineContext = EmptyCoroutineContext,  
    block: suspend CoroutineScope.() -> Unit): Job
```

```
launch(someContext) { ... }
```

```
fun <T> runBlocking(  
    context: CoroutineContext = EmptyCoroutineContext,  
    block: suspend CoroutineScope.() -> T): T
```

```
runBlocking(someContext) { ... }
```

```
suspend fun <T> withContext (  
    context: CoroutineContext,  
    block: suspend CoroutineScope.() -> T): T
```

```
withContext(someContext) { ... }
```



```
abstract class CoroutineDispatcher: ContinuationInterceptor { ... }
```

```
Dispatchers.Default      // ThreadPool, numbers of threads = #CPU
```

```
Dispatchers.IO           // for IO, shares threads with Default
```

```
Dispatchers.Main         // for UI (JavaFX, Android, Swing)
```

```
Dispatchers.Unconfined // not confined to any specific thread
```

```
fun Executor.asCoroutineDispatcher(): CoroutineDispatcher
```

```
val context = Executors.newFixedThreadPool(5).asCoroutineDispatcher()
```



```
CoroutineName(name: String) // useful for debugging

CoroutineId(id: Long) // -ea

CoroutineExceptionHandler(handler: (CoroutineContext, Throwable) -> Unit)

fun <T> ThreadLocal<T>.asContextElement(
    value: T = get()): ThreadContextElement<T>

val currentId: ThreadLocal<String>

withContext(CoroutineName("ReadingUser") + currentId.asContextElement()) {
    //...
}
```




```
interface Job {  
    fun cancel()  
    val isActive: Boolean  
}
```

```
fun Job.cancelChildren()
```

```
suspend fun <T> withTimeout(  
    timeMillis: Long,  
    block: suspend CoroutineScope.() -> T): T
```



Exercises

- Implement tasks 9-17. They are in `Task9.kt`, `Task10.kt`, etc. files.
- Tests for tasks are in `Test9`, `Test10`, etc. classes.
- Each test should be executed in separate VM.
`gradlew testPart2` should be enough.



```
val result: Deferred<T> = async {  
    //suspending functions can be invoked here  
}  
  
fun <T> CoroutineScope.async(  
    context: CoroutineContext = EmptyCoroutineContext,  
    block: suspend CoroutineScope.() -> T): Deferred<T>  
  
interface Deferred<out T>: Job {  
    suspend fun await(): T  
}
```



```
interface SendChannel<in E> {  
    suspend fun send (element: E)  
    fun close (cause: Throwable? = null): Boolean  
}
```

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

```
suspend fun <E> ReceiveChannel<E>.consumeEach(action: (E) -> Unit)
```

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



```
Channel() // creates a RendezvousChannel
Channel(Channel.RENDEZVOUS) // creates a RendezvousChannel
Channel(20) // creates an ArrayChannel (buffered)
Channel(Channel.UNLIMITED) // creates unlimited buffer channel
Channel(Channel.CONFLATED) // creates a conflated channel
```



```
val result: ReceiveChannel<T> = produce {  
    send(a)  
    send(b)  
}
```

```
fun <T> CoroutineScope.produce(  
    context: CoroutineContext = EmptyCoroutineContext,  
    capacity: Int = 0, // RENDEZVOUS  
    block: suspend ProducerScope.() -> T): ReceiveChannel<T>
```

```
interface ProducerScope<in E>: CoroutineScope, SendChannel<E> {  
    val channel: SendChannel<E>  
}
```



```
val result: SendChannel<T> = actor {  
    var state = 0  
  
    consumeEach { msg ->  
        // do sth with msg and alter state  
    }  
}  
  
fun <T> CoroutineScope.actor(  
    context: CoroutineContext = EmptyCoroutineContext,  
    capacity: Int = 0, // RENDEZVOUS  
    block: suspend ActorScope.() -> T): SendChannel<T>  
  
interface ActorScope<in E>: CoroutineScope, ReceiveChannel<E> {  
    val channel: ReceiveChannel<E>  
}
```



```
interface Mutex {  
    suspend fun lock()  
    suspend fun unlock()  
}  
  
fun Mutex(locked: Boolean = false): Mutex  
  
suspend fun Mutex.withLock(action: () -> T): T  
  
mutex.withLock {  
    // critical section here  
}
```




```
val result: CompletableFuture<T> = future {  
    // you can use suspending functions here  
}
```

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

```
suspend fun <T> CompletionStage<T>.await(): T
```

```
class CompletableFuture<T>: CompletionStage<T>
```



Exercises

- Implement tasks 18-26. They are in `Task18.kt`, `Task19.kt`, etc. files.
- Tests for tasks are in `Test18`, `Test19`, etc. classes.
- Each test should be executed in separate VM.
`gradlew testPart3` should be enough.



Where to find more information

`github.com/Kotlin/kotlinx.coroutines`

`kotlinlang.org/docs/reference/coroutines.html`

Roman Elizarov talks

The logo for Devoxx, featuring the word "Devoxx" in a stylized font. The "D" and "E" are white on a black background, while the "V" and "O" are orange. The "x" is white with an orange outline. A small "TM" trademark symbol is at the top right.

Thank you



Konrad Kamiński
[Allegro.pl](https://allegro.pl)

github.com/konrad-kaminski/spring-kotlin-coroutine