Structured concurrency with Kotlin coroutines

Alessandro Candolini

December 16, 2018

Agenda

1. Warm up
2. Before structured concurrency
3. Welcome structured concurrency
4. Exceptions

Warm up

```
interface Contract { // mvp contract

    interface View {

    }

    interface Presenter {

    }
}
```

```
interface Contract { // higher level contract

    interface View {
        fun render(state : ViewState)
    }

    interface Presenter {
        fun perform(action : ViewAction)
    }

    sealed class ViewAction
    sealed class ViewState
}
```

```
interface Contract { // more granular contract

    interface View {
        fun showLoading()
        fun hideLoading()
        fun showError(error: CharSequence)
        fun showResults(items: List<Item>)
    }

    interface Presenter {
        fun onRefresh()
    }
}
```

```
// usually

class SomeActivity : Activity(), Contract.View

class SomeFragment: Fragment(), Contract.View

class SomeCustomView : ViewGroup(), Contract.View
```

```
class Presenter : Contract.Presenter {

    override fun onRefresh() = TODO()

}
```

- Two-way bindings
- Passive view

- View instance holds a reference to its presenter
- Presenter instance holds a reference to the associated view instance[1]

[1] Circular dependencies; see `https://www.martinfowler.com/eaaDev/uiArchs.html`

```
class Presenter : Contract.Presenter {

    override fun onRefresh() {
      view.showLoading()  // <--  view?
    }
}
```

```
class Presenter(
        private val view: Contract.View
) : Contract.Presenter {

    override fun onRefresh() {
      view.showLoading()
    }
}
```

```
interface Contract {

    interface View /* ... */

    interface Presenter {
        fun bind(view : View) // <---
        fun unbind()                // <---
        fun onRefresh()
    }
}
```

```
class Presenter : Contract.Presenter {

    private var view : Contract.View? = null

    override fun bind(view: Contract.View) {
        this.view = view
    }
    override fun unbind() {
        this.view = null
    }
    override fun onRefresh() {
        view?.showLoading()
    }
}
```

java.lang.IllegalStateException(s)

```
override fun onSaveInstanceState(outState: Bundle)
```

```
class SomeFragment : Fragment(), Contract.View {
  @Inject
  lateinit var presenter: Contract.Presenter

  override fun onViewCreated(view: View,
          savedInstanceState: Bundle?) {
     // ...
     presenter.bind(this)
     presenter.onRefresh()
  }

  override fun onDestroyView() {
     presenter.unbind()
     super.onDestroyView()
  }
}
```

```
// why not this?
class SomeFragment : Fragment(), Contract.View {
  // ...

  override fun onSaveInstanceState(outState: Bundle) {
      presenter.unbind()
      super.onSaveInstanceState(outState)
  }
}
```

```
// why not this?
class Presenter(view : Contract.View) :
        Contract.Presenter {

  private var isAttached : Boolean = false

  override fun bind() {isAttached = true}
  override fun unbind() {isAttached = false}

  override fun onRefresh() {
    if(isAttached) {
      view.showLoading()
    }
  }
}
```

```
// what about memory leaks ?
class Presenter(view : Contract.View) :
        Contract.Presenter {
  private val executor : Executor = /*...*/

  override fun onRefresh() {
    view.showLoading()
    executor.execute {
        // long-running operation
        updateUI()
    }
  }

  private fun updateUI() {
   view.hideLoading()
  }
}
```

```
class Presenter(view : Contract.View) :
        Contract.Presenter {
  private val executor : Executor = /*...*/

  override fun onRefresh() {
    view.showLoading()
    executor.execute {
        // long-running operation
        updateUI()
    }
  }

  private fun updateUI() {
    // do nothing!
  }
}
```

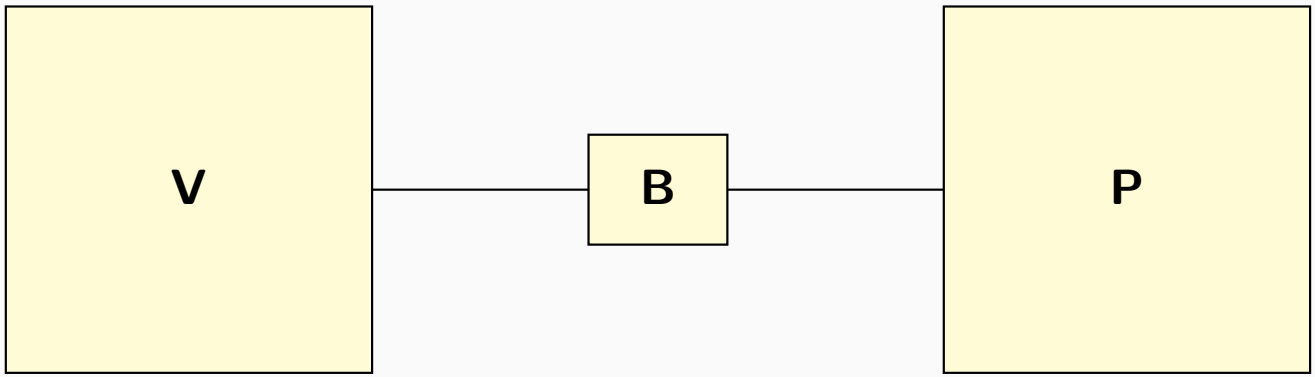
Can we handle this wiring somewhere else?

```
class SomeFragment : Fragment(), Contract.View {
  // ...

  override fun showLoading() {
    if (isAdded() ) {
       // update the android view
    }
  }
}
```

```
class SomeFragment : Fragment(), Contract.View {
  // ...

  override fun showA() = if(isAdded()){/* ... */}
  override fun hideA() = if(isAdded()){/* ... */}
  override fun showB() = if(isAdded()){/* ... */}
  override fun hideB() = if(isAdded()){/* ... */}
  override fun showC() = if(isAdded()){/* ... */}
  override fun hideC() = if(isAdded()){/* ... */}
}
```


V
B
P

Options are:

- Presenter
- View
- “Binder“ between view and presenter

Pros:

- Testability

Cons:[2]

- Verbosity
- Noise in the contracts

[2] Base class/type class can help though

Moving things out of the UI thread

Who is responsible?

- View?
- Binder (if any)?
- Presenter?
- Interactors/usecases?
- Repository/service layer?

```
sealed class Model

interface UseCase {
    suspend fun fetch(): Model
}
```

```
class Presenter(private val useCase: UseCase) :
        Contract.Presenter {

 // ...

   override fun onRefresh() { // deprecated way
     launch(Dispatchers.IO) {
       val model =  useCase.fetch()
       launch(Dispatchers.Main) {
         view?.showResults(model)
       }
     }
   }

}
```

```
class Presenter(private val useCase: UseCase) :
        Contract.Presenter {

// ...

  override fun onRefresh() { // more on this later!
    GlobalScope.launch(Dispatchers.IO) {
      val model =  useCase.fetch()
      GlobalScope.launch(Dispatchers.Main) {
        view?.showResults(model)
      }
    }
  }

}
```

```
class Presenter(
  private val useCase: UseCase
  private val ui: CoroutineDispatcher,
  private val io: CoroutineDispatcher
) : Contract.Presenter {
  // ...

  override fun onRefresh() {
    launch(io) {
      val model =  useCase.fetch()
      launch(ui) {
        view?.showResults(model)
      }
    }
  }
}
```

What's the difference between

```
launch(ui) {
  /*...*/
  launch(io) {
    /*...*/
  }
}
```

and

```
launch(io) {
  /*...*/
  launch(ui) {
    /*...*/
  }
}
```

```
class Presenter(
  private val useCase: UseCase
  private val ui: CoroutineDispatcher,
  private val io: CoroutineDispatcher
) : Contract.Presenter {
  // ...

  override fun onRefresh() {
    launch(ui) {
      val model = withContext(io){
          useCase.fetch()
      }
      view?.showResults(model)
    }
  }
}
```

```
class Presenter(
  private val useCase: UseCase
  private val ui: CoroutineDispatcher,
  private val io: CoroutineDispatcher
) : Contract.Presenter {
  // ...

  override fun onRefresh() {
    launch(ui) {
      try {
        val model = /*...*/
        view?.showResults(model)
      } catch(e : Exception) { // more on this later!
        view?.showError(/*...*/)
      }
    }
  }
}
```

What's the difference between

```
fun onRefresh() {
  launch(dispatcher) {
      /*...*/
  }
}
```

and

```
suspend fun onRefresh() {
      /*...*/
}
```

and

```
suspend fun onRefresh() {
  launch(dispatcher) {
      /*...*/
  }
}
```

```
suspend fun f(a : A) : B
suspend fun g(b : B) : C

suspend fun h(a : A) : C = g(f(a)) // no concurrency
```

```
suspend fun f(a : A, callback : Callback<A>)
suspend fun g(b : B, callback : Callback<C>)

interface Callback<T> {
  fun onSuccess(t : T)
  fun onError(throwable : Throwable)
}
```

```
fun h(a:A, callback: Callback<C>) {
  f(a, object : Callback<B> {
        override fun onSuccess(t: B) {
          g(t,object : Callback<C> {
              override fun onSuccess(t: C) {
                 callback.onSuccess(t)
              }
              override fun onError(throwable: Throwable)
                 callback.onError(throwable)
              }
          })
        }
        override fun onError(throwable: Throwable) {
          callback.onError(throwable)
        }
  })
}
```

```
class Presenter(dispatcher : CoroutineDispatcher) {

    fun doA() {
        launch(dispatcher) { /*...*/ }
    }
    fun doB() {
        launch(dispatcher) { /*...*/ }
    }
    fun doC() {
        launch(dispatcher) { /*...*/ }
    }
}
```

Concurrency

Two operations are concurrent if they are not ordered by *happens before* relation.[3].

[3] Leslie Lamport's paper
https://www.microsoft.com/en-us/research/uploads/prod/2016/12/Time-Clocks-and-the-Ordering-of-Events-in-a-Distributed-System.pdf

"Concurrency is the composition of independently executing processes, typically functions, but they don't have to be."

"Parallelism is the simultaneous execution of multiple things, possibly related, possibly not."

Rob Pike

Rob Pike - 'Concurrency Is Not Parallelism'

Figure 1: `https://www.youtube.com/watch?v=cN_DpYBzKso&t=1061s`

Programming Techniques | S. L. Graham, R. L. Rivest Editors

Communicating Sequential Processes

C.A.R. Hoare
The Queen's University
Belfast, Northern Ireland

This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises.

Key Words and Phrases: programming, programming languages, programming primitives, program structures, parallel programming, concurrency, input, output, guarded commands, nondeterminacy, coroutines, procedures, multiple entries, multiple exits, classes, data representations, recursion, conditional critical regions, monitors, iterative arrays
CR Categories: 4.20, 4.22, 4.32

1. Introduction

Among the primitive concepts of computer programming, and of the high level languages in which programs are expressed, the action of assignment is familiar and well understood. In fact, any change of the internal state of a machine executing a program can be modeled as an assignment of a new value to some variable part of that machine. However, the operations of input and output, which affect the external environment of a machine, are not nearly so well understood. They are often added to a programming language only as an afterthought.

Among the structuring methods for computer programs, three basic constructs have received widespread recognition and use: A repetitive construct (e.g. the **while** loop), an alternative construct (e.g. the conditional **if..then..else**), and normal sequential program composition (often denoted by a semicolon). Less agreement has been reached about the design of other important program structures, and many suggestions have been made: Subroutines (Fortran), procedures (Algol 60 [15]), entries (PL/I), coroutines (UNIX [17]), classes (SIMULA 67 [5]), processes and monitors (Concurrent Pascal [2]), clusters (CLU [13]), forms (ALPHARD [19]), actors (Hewitt [1]).

The traditional stored program digital computer has been designed primarily for deterministic execution of a single sequential program. Where the desire for greater speed has led to the introduction of parallelism, every attempt has been made to disguise this fact from the programmer, either by hardware itself (as in the multiple function units of the CDC 6600) or by the software (as in an I/O control package, or a multiprogrammed operating system). However, developments of processor technology suggest that a multiprocessor machine, constructed from a number of similar self-contained processors (each with its own store), may become more powerful, capacious, reliable, and economical than a machine which is disguised as a monoprocessor.

In order to use such a machine effectively on a single task, the component processors must be able to communicate and to synchronize with each other. Many methods of achieving this have been proposed. A widely adopted method of communication is by inspection and updating of a common store (as in Algol 68 [18], PL/I, and many machine codes). However, this can create severe problems in the construction of correct programs and it may lead to expense (e.g. crossbar switches) and unreliability (e.g. glitches) in some technologies of hardware implementation. A greater variety of methods has been proposed for synchronization: semaphores [6], events (PL/I), conditional critical regions [10], monitors and queues (Concurrent Pascal [2]), and path expressions [3]. Most of these are demonstrably adequate for their purpose, but there is no widely recognized criterion for choosing between them.

This paper makes an ambitious attempt to find a single simple solution to all these problems. The essential proposals are:

(1) Dijkstra's guarded commands [8] are adopted (with a slight change of notation) as sequential control structures, and as the sole means of introducing and controlling nondeterminism.

(2) A parallel command, based on Dijkstra's *parbegin* [6], specifies concurrent execution of its constituent sequential commands (processes). All the processes start simultaneously, and the parallel command ends only when they are all finished. They may not communicate with each other by updating global variables.

(3) Simple forms of input and output command are introduced. They are used for communication between concurrent processes.

This research was supported by a Senior Fellowship of the Science Research Council.

Author's present address: Programming Research Group, 45, Banbury Road, Oxford, England.

Figure 2: Tony Hoare's seminal paper

"The most obvious application of the new ideas is to the specification, design, and implementation of computer systems which continuously act and interact with their environment. The basic idea is that these systems can be readily decomposed into subsystems which operate concurrently and interact with each other as well as with their common environment. The parallel composition of subsystems is as simple as the sequential composition of lines or statements in a conventional programming language."

Tony Hoare (CSP book, 2015)

```
class PresenterWithDebouncer(
        private val base: Contract.Presenter
) : Contract.Presenter by base {

    override fun onRefresh() {
        base.onRefresh() // HOW ??
    }
}
```

```
class Test {

    @Mock
    lateinit var useCase : UseCase

    lateinit var presenter : Contract.Presenter

    @Before
    fun setup() {
        presenter = Presenter(useCase,
                Dispatchers.Unconfined,
                Dispatchers.Unconfined
            )
    }
}
```

```
interface Contract {

    interface Presenter {
        suspend fun onRefresh()

        fun bind(view: View)
        fun unbind()
    }

}
```

```
class SomeFragment : Fragment(), Contract.View {
  // ...

  override fun showA() = launch(ui){/* ... */}
  override fun hideA() = launch(ui){/* ... */}
  override fun showB() = launch(ui){/* ... */}
  override fun hideB() = launch(ui){/* ... */}
  override fun showC() = launch(ui){/* ... */}
  override fun hideC() = launch(ui){/* ... */}
}
```

```
class SomeFragment : Fragment(), Contract.View {
  // ...

  override fun onViewCreated(view: View,
              savedInstanceState: Bundle?) {
    swipeRefresh.setOnRefreshListener {
       launch(io){presenter.refresh()}
    }
  }
}
```

Before structured concurrency

```
class Presenter( /*...*/ ) : Contract.Presenter {

  private var view: Contract.View? = null
  private val jobs = mutableListOf<Job>()

  override fun onRefresh() {
    jobs += launch(ui) {/*...*/}
  }
  override fun unbind() {
    view = null
    jobs.forEach { it.cancel() }
    jobs.clear()
  }
}
```

```
class Presenter(
  private val useCase: UseCase,
  private val ui: Scheduler,
  private val io: Scheduler
) : Contract.Presenter {

  private val bag = CompositeDisposable()

  override fun onRefresh() {
    bag.add(
      useCase
        .observeOn( /*...*/ )
        .subscribeOn( /*...*/ )
        .subscribe { /* ... */ }
    )
  }
}
```

```
class Presenter(
  private val useCase: UseCase,
  private val ui: Scheduler,
  private val io: Scheduler
) : Contract.Presenter {

  private val bag = CompositeDisposable()

  override fun unbind() {
    view = null
    bag.clear() // not dispose!
  }
}
```

Question

Why disposing?

```
class Presenter( /*...*/ ) : Contract.Presenter {

  private var view: Contract.View? = null
  private val jobs = mutableListOf<Job>()

  override fun onRefresh() {
    jobs += launch(ui) {/*...*/}
  }
  override fun unbind() {
    view = null
    jobs.forEach { it.cancel() } // release resource
    jobs.clear() // avoid memory leaks
  }
}
```

Question

Why a list and not only one reference?

```
class Presenter( /*...*/ ) : Contract.Presenter {

  private var view: Contract.View? = null
  private val jobs = mutableListOf<Job>()

  override fun onA(){ jobs += launch(ui){/*...*/}}
  override fun onB(){ jobs += launch(ui){/*...*/}}
  override fun onC(){ jobs += launch(ui){/*...*/}}

  override fun unbind() {
    view = null
    jobs.forEach { it.cancel() }
    jobs.clear()
  }
}
```

Assumption

all processes are independent and can run concurrently (otherwise, need to handle concurrency/coordination via channels, actors, etc)

Question

Why is adding to the list manually error prone?

```
class Presenter( /*...*/ ) : Contract.Presenter {

  private var view: Contract.View? = null
  private val jobs = mutableListOf<Job>()

  override fun onA(){ jobs += launch(ui){/*...*/}}
  override fun onB(){ launch(ui){/*leaking*/}}
  override fun onC(){ jobs += launch(ui){/*...*/}}

  override fun unbind() {
    view = null
    jobs.forEach { it.cancel() }
    jobs.clear()
  }
}
```

Parallel decomposition

```
suspend fun fetch() : List<Model> {

 val deferredA = async{ /* source A */ }
 val deferredB = async{ /* source B */ }
 val deferredC = async{ /* source C */ }

 val modelA = deferredA.await()
 val modelB = deferredB.await()
 val modelC = deferredC.await()
 return modelA + modelB + modelC
}
```

```
suspend fun fetch() : List<Model> {

 val deferredA = async{/*long running task*/}
 val deferredB = async{/*quickly throws exception*/}
 val deferredC = async{/*...*/ }

 val modelA = deferredA.await()  // <-- wait A
 val modelB = deferredB.await()
 val modelC = deferredC.await()
 return modelA + modelB + modelC
}
```

Welcome structured concurrency

```
val scope : CoroutineScope = /*...*/

fun onRefh() {
  scope.launch {/*...*/}
}
```

```
val scope : CoroutineScope = /*...*/
val dispatcher : CoroutineDispatcher = /*...*/

fun onRefresh() {
  scope.launch(dispatcher){/*...*/}
}
```

```
val scope : CoroutineScope = /*...*/
val dispatcher : CoroutineDispatcher = /*...*/

fun CoroutineScope.onRefresh() { // convention
  launch(dispatcher){/*...*/}
}
```

```
val scope : CoroutineScope = /*...*/
val dispatcher : CoroutineDispatcher = /*...*/

fun onRefresh() {
  scope.launch{ // parent
     launch(dispatcher){ // child
       /*...*/
     }
  }
}
```

```
class Presenter( /*...*/ ) :
        Contract.Presenter, CoroutineScope {

  override fun onRefresh() {
    launch {/*...*/}
  }
}
```

```
class Presenter( scope : CoroutineScope) :
      Contract.Presenter, CoroutineScope by scope {

  override fun onRefresh() {
    launch {/*...*/}
  }
}
```

```
class Presenter( /*...*/) :
          Contract.Presenter, CoroutineScope {

  private lateinit var job : Job
  override val coroutineContext: CoroutineContext
    get() = ui + job

  override fun bind(view : View) {
    job = Job()
  }

  override fun onRefresh() {
    launch(ui) {/*...*/}
  }
  override fun unbind() {
    job.cancel()
  }
}
```

```
class Presenter( /*...*/ ) : Contract.Presenter,
    CorotuineScope {

  override fun onA(){ launch{/*...*/}}
  override fun onB(){ launch{/*...*/}}
  override fun onC(){ launch{/*...*/}}

  override fun unbind() {
    job.cancel()
  }
}
```

Show code:

- Testability
- Parallel decomposition example

Exceptions

```
interface UseCase {

    fun fetch(): List<Item>

}
```

```
class Presenter(
        private val view: Contract.View,
        private val useCase: UseCase
) : Contract.Presenter {

    override fun onRefresh() {
        val items = useCase.fetch()
        view.showItems(items)
    }

    override fun onSubmit() = TODO()

}
```

Questions?