Structured concurrency with Kotlin coroutines

Alessandro Candolini

December 16, 2018

Agenda

1. Warm up
2. Without 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: String)
        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()
    }
  }
}
```

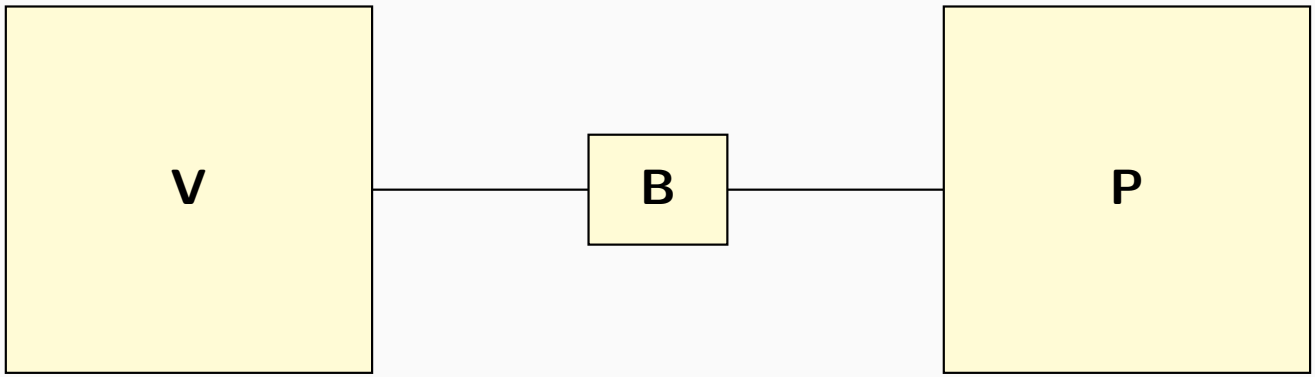
Can we move the binding somewhere else?

```
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

Without structured concurrency

Welcome structured concurrency

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?