Reusable Modules

It's as easy as ABI

Hi everyone, I'm Jordan an Android Developer working here on Marks & Spencer's mobile platform team.

Where among many things, we care about the health of the builds within the business.

This has led me onto a deep dive of something I think is important; reusable modules.

This might sound silly, after all, modules are normally reusable things.

Over the next ten minutes or so, I want to have a look at two things that impact the re-usability of a module of code. Then we'll look at how we can visualise this. Finally, I'll come up with some ground rules for you all to take away.

I want to preface this talk by saying that I specialise in Android and the JVM. I haven't branched out and tested this in Kotlin JS or Multiplatform. I assume the same rules roughly apply, but don't quote me on that.

This is a Kotlin conference after all, so I think we should make a start with some Kotlin.

Visibility Modifiers

`public`, `internal` and `private`

The key part of this talk that directly relates to Kotlin is visibility modifiers.

This is such a key foundational part of the language that I am sure you know that they are.

But, as I've been delving into this topic of Reusable modules I've found that the impact of them is often overlooked.

We have what I consider to be the three core visibility modifiers in Kotlin: `public` `internal` `private`

`public`

```
public class LoginViewModel()
    : ViewModel() {


}
```

- class is accessible out of compilation module
- `public` is optional

First is `public`, here we've create a new ViewModel and we've made it public. We've done this by writing `public` before the `class` keyword.

Generally, this also works without the `public` keyword.

This means that anything consuming a module containing this code can access and use this class.

`internal`

```
public class LoginViewModel internal constructor()
    : ViewModel() {


}
```

- `internal constructor`
- `LoginViewModel` cannot be made constructed of the compilation module

Next, we have `internal`. In this example we have placed an `internal` keyword before the `constructor`.

This means that the constructor is not public. It can only be accessed by the same module that this `ViewModel` is compiled in.

This is a great way of scoping some code to just the module we are in.

`private`

```
public class LoginViewModel internal constructor()
    : ViewModel() {
    private val _state = MutableStateFlow<UiState>(UiState.Loading)
    val state: StateFlow<UiState> = _state.asStateFlow()
}
```

- `private` `MutableStateFlow`
- `public` `StateFlow`

Finally, `private`.

Here, we've added two new properties to our `LoginViewModel`. One of which is private, this field is use a `MutableStateFlow` from the coroutines library.

We've also made a non-mutable field public.

The `private` field means this property is `private` to the scope it is defined in. By that, if you've defined something top level in a file it is accessible to an entire file. If you define it like we have here, it is accessible to the class it is defined in.

Okay this is ask well and good. But not super interesting. Let's look at some Gradle.

Dependencies

`api` and `implementation`

Gradle is the "go to" build tool for Kotlin, others exist like Maven etc etc. But I want to focus on this.

We use Gradle to turn our code from text to a an executable program.

Now for Gradle. You probably know this, but Gradle is the go to tool for building our code. That is taking it from letters on our screen to a program that executes.

The thing we care about here is dependencies and how we declare them! The two ways to do this require us to this are:

`api` and `implementation`.

`implementation`

```
dependencies {

    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.8.6'

}
```

– Dependency is **consumed** by this **compilation** module
–
Available at compilation and runtime
When we declare a dependency as the `implementation` keyword we say that I want this dependency to be consumed by this module.

This means that the module can independently compile against and execute against the classes defined in a library.

`api`

```
dependencies {

    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.8.6'
    api 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.9.0'

}
```

– Dependency is **consumed** and **produced** by this compilation module
–
Available at compilation and runtime
When we declare a dependency as an `api` we say that I want this dependency to be consumed by this module, but also be produced by it.

This means that the module can independently compile against and execute against the classes defined in a library. Not only that, but it will produce that library for consumers of this module.

Application Binary Interface

Our usage of modifiers and dependencies comes together to form something called an application binary interface.

What is an application binary interface?

The ABI itself is the enumeration of all classes/functions and properties that are public from a module. It's how other modules see and think about other modules.

It's also used for binary compatibility; by comparing one ABI (say from a prior release) to the ABI in a pull request you can confirm you don't break the public API of a library across library versions.

Okay, that sounds good. But does one of these ABIs look like?

```
@Lkotlin/Metadata;
public final class uk/co/jordanterry/LoginViewModel : androidx/life-
cycle/ViewModel {



    public final fun getState ()Lkotlinx/coroutines/flow/StateFlow;



}
```

Here is the ABI of the `LoginViewModel` I put together earlier in this talk.

```
@Lkotlin/Metadata;
public final class uk/co/jordanterry/LoginViewModel : androidx/life-
cycle/ViewModel {
```

– We see our `public final class`
– Fully qualified `LoginViewModel`
– Fully qualified Androidx `ViewModel`

```
    public final fun getState ()Lkotlinx/coroutines/flow/StateFlow;
```

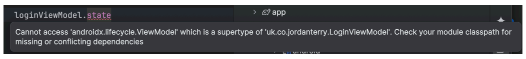
– public final function called `getState`
– Fully qualified reference to KotlinX `StateFlow`

```
    public final class uk/co/jordanterry/LoginViewModel : androidx/life-
cycle/ViewModel {
```

oh oh! Something doesn't add up.

```
dependencies {
    api 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.9.0'
    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.8.6'


}
```



```
loginViewModel.state
```

Cannot access 'androidx.lifecycle.ViewModel' which is a supertype of 'uk.co.jordanterry.LoginViewModel'. Check your module classpath for missing or conflicting dependencies

```
dependencies {


    api 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.9.0'
    api 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.8.6'


}
```

Recap

– The usage of visibility modifiers impacts your ABI
– Your ABI is a proxy for what consumers of other modules see
– You should produce dependencies that have classes on your ABI

Why should you care about this?

– Developer Experience
– Compile Time
– Correctness

Want to visualise this?

– Kotlin Binary Compatibility Validator
– Metalava
– Dependency Analysis Gradle Plugin

Questions?