What cannot be achieved in Swift (yet?)

I'm Vincent 👋 🇫🇷

Disclaimer

It's going to be a frustating talk!

We're going to see some cool ideas...

...that we can't actually use 😭

So what's the motivation for this talk?

Last year, WWDC brought us Property Wrappers and Function Builders

So let's have a look some features from "the other guys"...

...that could someday (next week?) be added to Swift!

(And we might even get a better understanding of Swift along the way!)

#1 Smart Casting

```kotlin
// Kotlin

val x: String? = "Hi"

if (x != null) {
  x.length // no need to force unwrap!
}
```

```kotlin
// Kotlin

val obj: Any = "This is a String"

if(obj is String) {
    obj.length // no need to explicitly cast
}
```

→ Smart casting is pretty similar to `if-let` in Swift

→ With one little advantage: it doesn't require the introduction of a new identifier

#2 Extending Function Types

```swift
// Swift

extension (Double) -> Double {
    // ...
}

non-nominal type '(Double) -> Double' cannot be extended
```

```kotlin
// Kotlin

typealias RealFunction = (Double) -> Double

val RealFunction.derivative: RealFunction
    get() = { x ->
        val h = 1e-3

        round((this(x + h) - this(x - h)) / (2 * h) / h) * h
    }

fun main() {
    val squared: (Double) -> Double = { it * it }

    print(squared.derivative(4.0)) // 8.0
}
```

→ Definetely not a game changer...

→ ...but a nice addition for apps that deal heavily with functions!

#3 Annotations

Let's take a look at how network calls can be implemented on Android

```kotlin
// Kotlin

data class MyServiceResponse(/* ... */)

interface MyService {
    @FormUrlEncoded
    @POST("/myservice/endpoint")
    fun call(@Header("Authorization") authorizationHeader: String,
             @Field("first_argument") firstArgument: String,
             @Field("second_argument") secondArgument: Int
            ): Observable<MyServiceResponse>
}
```

→ That was the entire "implementation"!

→ No need to write the actual implemention by hand, as it can be generated at runtime

→ All the information needed to generate the code is being provided through the annotations

→ In Swift, we're not quite there yet!

→ We do have Property Wrappers…

→ …but they cannot be applied to methods or arguments

Maybe one day we'll have similar annotations in Swift?

#4 Annotations

```
// Dependency Injection in Java with Dagger

class CoffeeMaker {
  @Inject Heater heater;
  @Inject Pump pump;
}

@Module
class DripCoffeeModule {
  @Provides static Heater provideHeater() {
    return new ElectricHeater();
  }

  @Provides static Pump providePump(Thermosiphon pump) {
    return pump;
  }
}
```

Looks similar to Swinject, right?

But there's one big difference!

→ Swinject resolves dependencies at **runtime**

→ Dagger resolves dependencies at **compile time**

→ Which gives opportunities to perform **optimizations** and **catch errors** before an app ships

→ This works because Java supports **Annotation Processors**

→ Which are classes called at compile time, that **read annotations** and **generate code** accordingly

Let's take a closer look!

```swift
// Swift

@propertyWrapper
struct MyWrapper<Value> {
    var value: Value

    init(wrappedValue: Value) {
        self.value = wrappedValue
    }

    var wrappedValue: Value {
        get { /* some custom wrapping logic */ }
        set { /* some custom wrapping logic */ }
    }
}
```

What happens at compile time?

```swift
// Swift

struct Person {
  @MyWrapper let name: String
}

let john = Person(name: "John")
```

→ Whenever we write `john.name`, the compiler actually **replaces** it by `john.name.wrappedValue`

→ This processing of the annotation is **hardcoded** into the compiler

→ In Kotlin, **Annotation Processors** let developers define custom ways to process annotations 🚀

→ In Swift we don't have an equivalent for now...

→ ...but tools like Sourcery can get pretty close

#5 Coroutines

Asynchronous code is always a challenge!

In Swift we have a choice of libraries to tackle the issue

RxSwift, Combine, PromiseKit, etc.

But how about something built into the language itself?

```kotlin
// Kotlin

class LoginRepository(...) {

    // Notice the keyword suspend
    suspend fun makeLoginRequest(jsonBody: String): Result<LoginResponse> {
        // Blocking network request code
    }
}
```

```kotlin
// Kotlin

class LoginViewModel(
    private val loginRepository: LoginRepository
): ViewModel() {

    fun login(username: String, token: String) {

        // We use the function launch to start a coroutine
        viewModelScope.launch {
            val jsonBody = "{ username: \"$username\", token: \"$token\"}"

            // makeLoginRequest is an async call
            val result = loginRepository.makeLoginRequest(jsonBody)

            // However, we can assign and manipulate its result
            // just like with a sync function!
            when (result) {
                is Result.Success<LoginResponse> -> // Happy path
                else -> // Show error in UI
            }
        }
    }
}
```

Feels like the DispatchQueue API, but on steroids

(It's a bit similar to the async/await pattern)

That's it for today!

Now wait & see what happens next week 🧐

What cannot be achieved in Swift (yet?)

Vincent Pradeilles @v_pradeilles – Worldline 🇫🇷