Creating Internal DSLs in Kotlin

1. Domain Specific Languages

Domain Specific Languages are languages that we create that are very specific to a particular domain or a particular application.

- Allows us to create a very targeted API for the users of our application.
  - Depending on who the users are.

2. Types: External vs Internal

External DSLs are where you get to define your own language all by yourself. But the burden is that you'll have to parse it yourself.

- You have the flexibility of the syntax.
  Internal DSLs are DSLs that write on an existing language. The compiler/interpreter becomes a tool that you can leverage.
- You're able to bend the language.
  - If you want to be able to express a certain syntax you need to make the language work that way.

3. What makes Kotlin special for internal DSLs?

Kotlin is already pretty fluent, and Kotlin is able to do a bunch of things that are typically difficult in statically typed languages.

Optional semicolon

Semicolons break the flow and minimizes our fluency. It makes the code less ceremonious.

drop ()'s and .'s using infix

Let's take a look at some code to explore the idea of ceremony.

```
class Car {
        fun drive(dist: Int) {
                println("driving...")
        }
}
```

```
val car = Car()
car.drive(10)
```

If we look at the code above it appears quiet "codey" (for lack of a better term!). How about we mute those noises that we have from the `.`'s and `()`'s.

We can use `infix` notation with the `infix` keyword. We can create `infix` methods that we add to our projects. This way we can drop the `.`'s and `()`'s.

```
class Car {
        infix fun drive(dist: Int) {
                println("driving...")
        }
}

val car = Car()

car drive 10
```

Extension methods give you the power of fluency.

```
val greet = "hello"

println(greet.shout())
```

When I run this code, it obviously doesn't work as there is no `shout` method that the `String` class has. So to work around this we can write that method ourselves.

```
fun String.shout() = toUpperCase()

println(greet.shout())
```

Now we can start bending our classes by injecting methods into these classes.

No () for passing last lambda

"Good code invites the reader, bad code pushes the reader away" - Venkat

- The more parentheses and semicolons create noise in the code.

Look at the excessive noise in the code below:

```
fun process(func: (Int) -> Unit, n: Int) {
        func(n * 2)
}

process({e -> println(e) }, 2)
```

Let's try to reduce that...

```
fun process(n: Int, func: (Int) -> Unit) {
        func(n * 2)
}

//process(2, { e -> println(e) })
process(2) { e -> println(e) }
```

The lambda is the last parameter. In this case lambdas get special treatment. Which allows us to save a parentheses and make our code less noisy. Thus clearing the clutter.

Implicit Receivers

Let's take a quick detour and visit to JavaScript

```
function greet(name) {
        console.log(`$name`);
}

greet('Jane');
```

One of the cool features of JavaScript is that you can take arbitrary functions and turn them into methods of your class.

```
function greet(name) {
        console.log(`${this.toUpperCase()} $name`);
}

greet('hello', 'Jane');
```

This is something you can do in Kotlin through lambda expressions.

```
fun call(greet: (String) -> Unit) {
        greet('Jane')
}
```

```
call { name ->
        println("$name")
}
```

Here we added a context object associated with our lambda. When it executes it is run in the context of this object: `(String)`.

```
fun call(greet: (String).(String) -> Unit) {
        //greet('Jane')
        "Hello".greet("Jane")
}

call { name ->
        println("${this.toUpperCase()} $name")
}
```

`this` and `it` can come in handy

`this` and `it` can become parameters in your context object that adds to fluency here.

```
fun call(greet: (String).(String) -> Unit) {
        //greet('Jane')
        "Hello".greet("Jane")
}

call { name ->
        println("${this.toUpperCase()} $it")
}
```

Let's go ahead and create some DSLs!

```
val ago = "ago"

infix fun Int.days(tense: String) {
        println("called")
}

//2.days(ago)

2 days ago
```

This is how you can write an infix function and inject the days into an integer and give fluency into your code to perform that kind of behavior.

```
val ago = "ago"
val from_now = "from now"

infix fun Int.days(tense: String) {
        when(tense) {
                ago -> println(LocalDataTime.now.minusDays(this.toLong()))
                from_now -> println(LocalDataTime.now.plusDays(this.toLong()))
                else -> println("?")
        }
}

2 days ago
2 days from_now
```

Planning a Meeting DSL

```
class Meeting(name: String) {
        val start = this
        infix fun at(time: IntRange){
                println("$name meeting starts at $time")
        }
}

infix fun String.meeting(block: Meeting.() -> Unit) {
        Meeting(this).block() //fire the block of code in the context of the meeting (context object)
}

"planning" meeting {
        start at 3:15
}
```