Challenges in Building for Fluency

We'll cover the following

Building fluency into code takes some effort. Explore if an extension function will be useful. Sometimes an `infix` method may help remove some clutter in code—to get rid of the dot and parenthesis. Maybe an implicit receiver will come to the rescue. You can choose from a number of techniques, and sometimes they may appear to conflict with one another. Let's look at an example of this situation.

To use the `infix` notation, so you can remove the dot and parenthesis, you need an object reference. For instance, a `person` reference is needed to use infix notation to make the following fluent: `person run "fast"`. But by using an implicit reference to the instance of this hypothetical `Person` class, we can get rid of the object reference in the call and write only `run("fast")`. Wait, it appears that we can either get rid of the dot along with parenthesis or we can dismiss the object reference, but not both. One work-around would be to use `this` instead of `person` since the implicit reference can be explicitly referenced using `this`. But writing `this run "fast"` doesn't appeal to my fluency- demanding pallet. What gives?

It's easy to get discouraged when we run into conflicts like this. But by step- ping back to the basics, we can think through ways to arrive at a working solution. Let's devise a solution: to create a fluent syntax without dots, parenthesis, and this.

Using extension functions

Suppose we're creating an application that will keep track of events and dates. Maybe we want to mention that some event happened `2 days ago` and another event will happen maybe `3 days from_now`. With some extension functions, we can make the words roll off our tongues and emerge as code—Kotlin code, that is.

We'll start with the code to extend the `Int` class with a `days()` method to facilitate

the fluent domain-specific method call:

```
// DateUtil.kt
package datedsl

import java.util.Calendar
import datedsl.DateUtil.Tense.*

infix fun Int.days(timing: DateUtil.Tense) = DateUtil(this, timing)
```

The newly injected `days()` method takes the yet-to-be-written `DateUtil.Tense` `enum` and returns an instance of the `DateUtil` class, also yet to be written. The `import` statement brings in the values of the `enum` into the scope for easy use with the `when` argument matching we'll use soon.

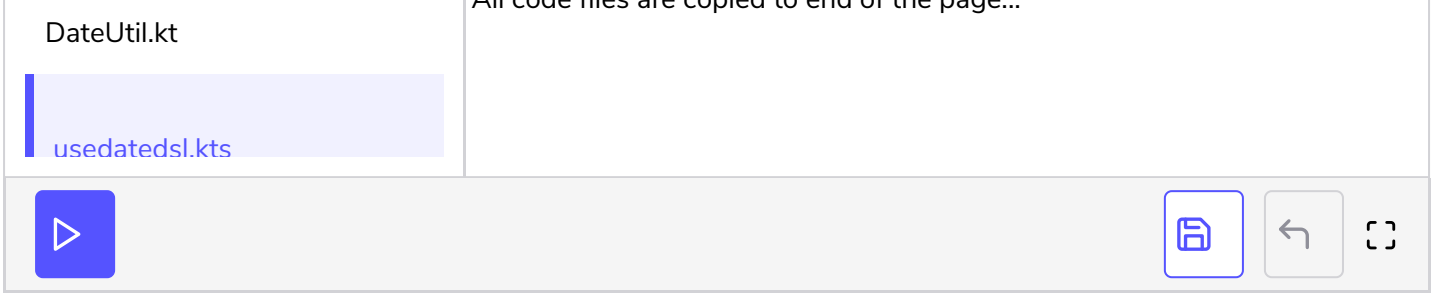
The `DateUtil` class will take the number, the `Int` instance on which `days()` was called, and the given `enum` and take the necessary steps to return the appropriate date for the hypothetical event. Let's take a look at the `DateUtil` class now.

```
class DateUtil(val number: Int, val tense: Tense) {
  enum class Tense {
    ago, from_now
  }

  override fun toString(): String {
    val today = Calendar.getInstance()

    when (tense) {
      ago -> today.add(Calendar.DAY_OF_MONTH, -number)
      from_now -> today.add(Calendar.DAY_OF_MONTH, number)
    }

    return today.getTime().toString()
  }
}
```

DateUtil.kt

The `DateUtil` class is nesting the `enum` `Tense` and storing the constructor parameters as immutable properties. The `toString()` method takes different actions for different values of the tense variable and returns the appropriate instance of time.

The easy part is exercising this code to use the domain-specific method we added to `Int`. Let's take a look:



To run this script locally, you first have to compile the DateUtil class with following commands.

```
kotlinc-jvm DateUtil.kt -d datedsl.jar
kotlinc-jvm -classpath datedsl.jar -script usedatedsl.kts
```

Depending on when you run it, you'll see an output similar to the following:

```
Sun Aug 11 05:11:38 MDT 2019
Fri Aug 16 05:11:38 MDT 2019
```

Using receiver and `infix`

In the previous example, the extension function did the trick, but things aren't that simple most of the time. Let's take a look at a DSL that's going to demand a lot more.

Our daily lives are filled with meetings and there seems to be no escape. Might as well make scheduling the next one a tad easier, right? Let's design a fluent syntax to make that happen:

```
// meetingdsl.kts
"Release Planning" meeting {
  start at 14.30
  end by 15.20
}
```

Let's take small steps to achieve this goal. Ignore what's inside the lambda for now and focus only on the `"Release Planning" meeting {}` part. What do we need to make this work? Two things.

First, we need to inject a `meeting()` method into the `String` class—an extension function. Second, we need to make `meeting()` an `infix` method so we can drop the dot. The parenthesis is not an issue since the parameter is the last lambda, the only lambda.

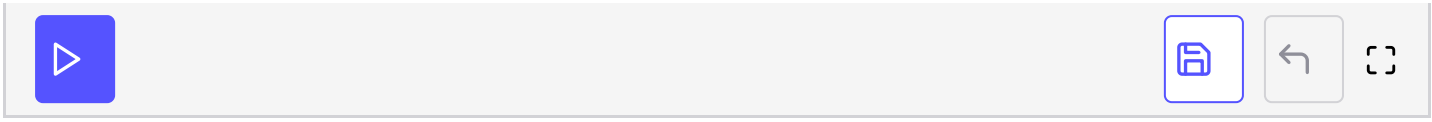
Let's get this small part working first:

```
infix fun String.meeting(block: () -> Unit) {
  println("step 1 accomplished")
}
"Release Planning" meeting {}
```

The `meeting()` extension function will be called when the DSL is executed. It'll print that message we placed, and Kotlin will warn that we're not using the `block` variable—all good so far.

Within the lambda, we want to update the state—the details about the meeting timings. For this we may use a `Meeting` class that'll be the holder of the details, the state. Since the lambda will populate the state for an instance of `Meeting`, we might as well run it in the context of the instance. To accomplish this, we should change the lambda's signature. Let's take that next small step.

```
class Meeting
infix fun String.meeting(block: Meeting.() -> Unit) {
    val meeting = Meeting()
    meeting.block()
    println(meeting)
}
"Release Planning" meeting {
    println("With in lambda: $this")
}
```



We have a `Meeting` class, but it's not much yet. The `block` parameter of `String.meeting()` now expects a receiver of type `Meeting`. Within the `String.meeting()` method we create an instance of `Meeting`, run the lambda in the context of that instance, and print the instance.

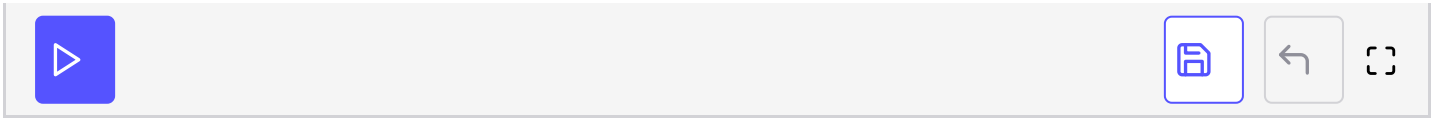
```
With in lambda: Meetingdsl2$Meeting@1a2e563e
Meetingdsl2$Meeting@1a2e563e
```

The output shows that the same instance created within `String.meeting()` is also seen as `this`, the receiver, inside the lambda expression.

Next step, let's create `at` and `by` methods in the `Meeting` class and run them both from within the lambda. That's going to grow out `Meeting` class from its infancy.

While at it, let's also add a constructor parameter to the `Meeting` class to store the meeting title:

```
class Meeting(val title: String) {
    var startTime: String = ""
    var endTime: String = ""
    private fun convertToString(time: Double) = String.format("%.02f", time)
    fun at(time: Double) { startTime = convertToString(time) }
    fun by(time: Double) { endTime = convertToString(time) }
    override fun toString() = "$title Meeting starts $startTime ends $endTime"
}

infix fun String.meeting(block: Meeting.() -> Unit) {
    val meeting = Meeting(this)
    meeting.block()
    println(meeting)
}
"Release Planning" meeting {
    at(14.30)
    by(15.20)
}
```



The `at()` method stores the given `Double` value into a property `startTime` after converting to a `String`. Likewise, the `by()` method stores the `endTime`. The `toString()` method reports the state of the `Meeting` object. The `String.meeting()` method is using the constructor of `Meeting`.

The output shows that the code is doing what's expected—the `at()` and `by()` methods are invoked in the context of a `Meeting` instance:

```
Release Planning Meeting starts 14.30 ends 15.20
```

We're left with good news and bad news. The good news is it worked. The bad news—the DSL is far from where we want it to be. What does at and by mean? That's not readable at all. Also, why that parenthesis? Let's first get rid of that parenthesis. For that, as you know, we'll use `infix`.

```
infix fun at(time: Double) { startTime = convertToString(time) }
infix fun by(time: Double) { endTime = convertToString(time) }
```

The only change we made to the `Meeting` class is placing the `infix` keyword twice, once for `at()` and once for `by()`. Now can we drop the parenthesis from the DSL? Not so fast. While `infix` is great, it comes with some severe limitations. To use it, you need an instance on which the method will be called, followed by a space, then

the name of the method, another space, and then finally a single argument. Sadly, we can't write `at 14.30` and expect that to work; an instance reference is needed before `at`. Let's compromise: we'll use `this` for the object reference, but only for a few moments.

```
"Release Planning" meeting {
  this at 14.30
  this by 15.20
}
```

We're almost there—all we have to do is replace `this` with `start` on one line and with `end` on the other line. For the change to the first line, we can define `start` as a variable that is bound to `this`. That'll do the trick.

Within the lambda the receiver is implicit. We know that `at(14.30)` was actually `this.at(14.30)`. If we write `start at 14.30`, will Kotlin see it as `start.at(14.30)` and then as `this.start.at(14.30)` and be happy to compile and produce the desired result? Let's find out right away—we're running short on nails.

```
class Meeting(val title: String) {
  var startTime: String = ""
  var endTime: String = ""
  val start = this
  val end = this

  private fun convertToString(time: Double) = String.format("%.02f", time)
  infix fun at(time: Double) { startTime = convertToString(time) }
  infix fun by(time: Double) { endTime = convertToString(time) }

  override fun toString() = "$title Meeting starts $startTime ends $endTime"
}

infix fun String.meeting(block: Meeting.() -> Unit) {
  val meeting = Meeting(this)

  meeting.block()

  println(meeting)
}

"Release Planning" meeting  {
  start at 14.30
  end by 15.20
}
```

meetingdsl.kts

That's the complete code to make the DSL work. The only changes we did in this last step are: (1) we added two properties `start` and `end` to the `Meeting` class, and (2) we replaced `this` within the DSL with `start` and `end` on the last two lines, respectively.

We achieved fluency, but there's a catch in this implementation—it doesn't prevent the users of our DSL from calling `start by` instead of `start at`, and `end at` instead of `end by`. As well, after typing `start` or `end`, the autocompletion feature of IDEs will show both `at` and `by`, thus misleading the user. This is because both `start` and `end` are properties that return the same instance and both `at` and `by` are methods on `Meeting`. We can prevent this potential error by moving the `at` and `by` methods to separate classes, like so:

```
open class MeetingTime(var time: String = "") {
  protected fun convertToString(time: Double) = String.format("%.02f", time)
}

class StartTime : MeetingTime() {
  infix fun at(theTime: Double) { time = convertToString(theTime) }
}

class EndTime : MeetingTime() {
  infix fun by(theTime: Double) { time = convertToString(theTime) }
}

class Meeting(val title: String) {
  val start = StartTime()
  val end = EndTime()

  override fun toString() =
    "$title Meeting starts ${start.time} ends ${end.time}"
}

infix fun String.meeting(block: Meeting.() -> Unit) {
  val meeting = Meeting(this)

  meeting.block()

  println(meeting)
}

"Release Planning" meeting  {
  start at 14.30
  end by 15.20
}
```

meetingdslevolved.kts

The `MeetingTime` class serves as a base class, holds a `time` property of type `String`,

and contains the `convertToString()` function that'll be used by its derived classes.

The `StartTime` class extends `MeetingTime` and contains the `at()` method. Likewise, the `EndTime` class is similar to `StartTime`, except it contains the `by()` method. Since the start time and end time are now stored in the classes `StartTime` and `EndTime`, respectively, the `Meeting` class doesn't need these two fields. The `start` property of `Meeting` now returns an instance of `StartTime` instead of this. Likewise, the `end` property returns an instance of `EndTime`.

That took some effort and trickery to get the fluent syntax working. Let's shoot for something a bit more intense next and, along the way, you can learn about the type safety Kotlin provides for internal DSLs in the next lesson.

Code Files Content !!!

⤋ ⤋

```
--------------------------------------------------------------------------
|  DateUtil.kt [1]
--------------------------------------------------------------------------


package datedsl

import java.util.Calendar
import datedsl.DateUtil.Tense.*

infix fun Int.days(timing: DateUtil.Tense) = DateUtil(this, timing)

class DateUtil(val number: Int, val tense: Tense) {
  enum class Tense {
    ago, from_now
  }

  override fun toString(): String {
    val today = Calendar.getInstance()

    when (tense) {
      ago -> today.add(Calendar.DAY_OF_MONTH, -number)
      from_now -> today.add(Calendar.DAY_OF_MONTH, number)
    }

    return today.getTime().toString()
  }
```

```
}


--------------------------------------------------------------------------
|  usedatedsl.kts [1]
--------------------------------------------------------------------------


import datedsl.*
import datedsl.DateUtil.Tense.*

println(2 days ago)
println(3 days from_now)




********************************************************************************
```