Fluency with Any Object

We'll cover the following

- Behavior of the four methods
- From a verbose and noisy code
- Removing repetitive references with apply
- Getting results using run
- Passing an object as argument using let
- Chaining void functions using also

`infix` reduces some noise, but when working with just about any object, Kotlin makes code less verbose and more expressive. The language does this by adding a few convenience functions. Learning those methods can make your everyday coding a pleasant experience and make your code more fluent.

Coding in Kotlin involves invoking functions, methods, and passing lambda expressions, among other things. The language offers some support for minimizing noise with these day-to-day operations.

Specifically, Kotlin has four significant methods that can make code fluent: `also()`, `apply()`, `let()`, and `run()`. Each of these methods takes a lambda expression as a parameter and returns something back after invoking the given lambda. The format of a call to one of these methods, on a context object, looks like this:

```
result = context.oneOfTheseFourMethods { optionalParameter ->
        ...body...
        ...what's this (receiver) here?...
        optionalResult
}
```

Depending on the method you call, the `optionalParameter` and this, the receiver of the call, will be assigned differently. The `optionalResult` received from the call will differ as well. Once we learn a bit more about these functions, we'll see how we

can benefit from these functions to create concise and expressive code. Stay tuned.

Behavior of the four methods

Let's exercise these four methods and report back on the arguments received by the lambdas passed, the `this` receiver within each lambda expression, the result returned by them, and the result received on the calling side of these four methods.

```
val format = "%-10s%-10s%-10s%-10s"
val str = "context"
val result = "RESULT"

fun toString() = "lexical"

println(String.format("%-10s%-10s%-10s%-10s%-10s",
"Method", "Argument", "Receiver", "Return", "Result"))
println("================================================")

val result1 = str.let { arg ->
    print(String.format(format, "let", arg, this, result))
    result
}
println(String.format("%-10s", result1))

val result2 = str.also { arg ->
    print(String.format(format, "also", arg, this, result))
    result
}
println(String.format("%-10s", result2))

val result3 = str.run {
    print(String.format(format, "run", "N/A", this, result))
    result
}
println(String.format("%-10s", result3))

val result4 = str.apply {
    print(String.format(format, "apply", "N/A", this, result))
    result
}
println(String.format("%-10s", result4))
```

anymethods.kts

Each of the lambdas ends with an expression `result`, but some of them may get ignored, as we'll see. The first two lambdas receive a parameter arg and the last two don't. Study the output to learn about the behavior of the methods:

```
Method    Argument  Receiver  Return    Result
=============================================

let       context   lexical   RESULT    RESULT
also      context   lexical   RESULT    context
run       N/A       context   RESULT    RESULT
apply     N/A       context   RESULT    context
```

The `let()` method passes the context object, the object on which it's called, as an argument to the lambda. The lambda's `this`, or receiver, is lexically scoped and bound to the `this` in the defining scope of the lambda—we discussed lexical scoping in Closures and Lexical Scoping. The result of the lambda is passed as the result of the call to `let()`.

The `also()` method also passes the context object as the argument to its lambda, and the receiver is tied to `this` in lexical scoping. However, unlike `let()`, the `also()` method ignores the result of its lambda and returns the context object as the result. The return types of the lambda that `also()` receives is `Unit`, thus the `result` returned is ignored.

The `run()` method doesn't pass any argument to its lambda but binds the context object to the `this`, or receiver, of the lambda. The result of the lambda is returned as the result of `run()`.

The `apply()` method also doesn't pass any argument to its lambda and binds the context object to the `this`, or receiver, of the lambda. But unlike the `run()` method, the `apply()` method ignores the result of the lambda—`Unit` type used for return—and returns the context object to the caller.

Let's summarize these four methods' behaviors:

- All four methods execute the lambdas given to them.
- `let()` and `run()` execute the lambda and return the result of lambda to the caller.
- `also()` and `apply()` ignore the result of the lambda and, instead, return the context object to their callers.
- `run()` and `apply()` run the lambda in the execution context—`this`—of the context object on which they are called.

Let that sink in. Don't try to memorize these, it will fall in place with practice. The

key is to understand that there are differences in the receiver and what is returned

from these methods.

From a verbose and noisy code

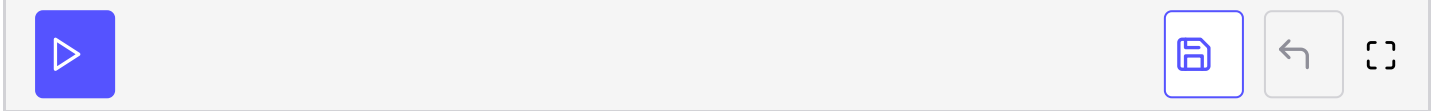
We'll now use these four methods to make a piece of code fluent. For this purpose, we start by defining a `Mailer` class:

```
class Mailer {
    val details = StringBuilder()
    fun from(addr: String) = details.append("from $addr...\n")
    fun to(addr: String) = details.append("to $addr...\n")
    fun subject(line: String) = details.append("subject $line...\n")
    fun body(message: String) = details.append("body $message...\n")
    fun send() = "...sending...\n$details"
}
```

The fluency of code to interact with the methods of the class is the focus here, thus the class doesn't do anything useful other than registering the calls into a `StringBuilder`.

Here's a rather verbose example of using this `Mailer` class.

```
val mailer = Mailer()
mailer.from("builder@agiledeveloper.com")
mailer.to("venkats@agiledeveloper.com")
mailer.subject("Your code sucks")
mailer.body("...details...")
val result = mailer.send()
println(result)
```



Before we discuss the quality of the code, let's make sure it works:

```
...sending...
from builder@agiledeveloper.com...
to venkats@agiledeveloper.com...
subject Your code sucks...
body ...details......
```

The code worked, but the calls to the methods on `Mailer` were noisy, repetitive, and not very pleasant. This is where the four convenience methods we discussed come into play.

Removing repetitive references with `apply`

Refer back to the output table from the example in Behavior of the Four Methods; the `apply()` method executes the lambda in the context of the object on which it's called and returns the context object back to the caller. The `apply()` method can form a chain of method calls, like so:

```
obj.apply{...}.apply{...}...
```

Let's rewrite the previous code that uses Mailer to use `apply()`:

```
val mailer =
    Mailer()
        .apply { from("builder@agiledeveloper.com") }
        .apply { to("venkats@agiledeveloper.com") }
        .apply { subject("Your code sucks") }
        .apply { body("details") }

val result = mailer.send()

println(result)
```

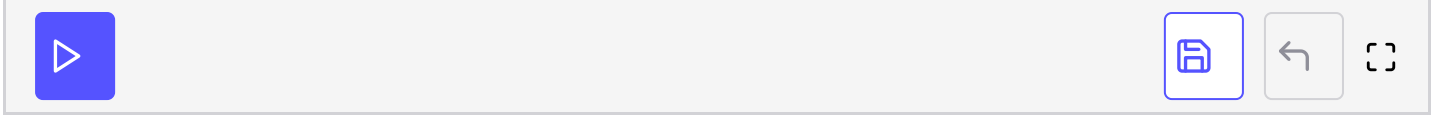
None of the methods of `Mailer` return back the `Mailer` instance on which they were called. That means we can't combine multiple calls to `Mailer` methods. The `apply()` solves that issue nicely. With each call to `apply()`, we start with a `Mailer` and end with the same instance. This allows us to chain—that is, continue making multiple calls on a `Mailer` instance—without repeating the reference name. The good news is less repetition of the reference, but the code is now noisy with all these `apply {...` calls.

The `apply()` method executes the lambda in the context of its target object. As a result we may place multiple calls to `Mailer` within the lambda, like so:

```
val mailer = Mailer().apply {
    from("builder@agiledeveloper.com")
    to("venkats@agiledeveloper.com")
    subject("Your code sucks")
    body("details")
}

val result = mailer.send()

println(result)
```



Without changing the `Mailer` class, the user of the `Mailer` is able to make multiple calls on an instance without repeatedly using the reference dot notation. That reduces the noise significantly in code. Also, a builder-pattern like expressions can be used for any class, even those that were not designed for chaining of calls to setters.

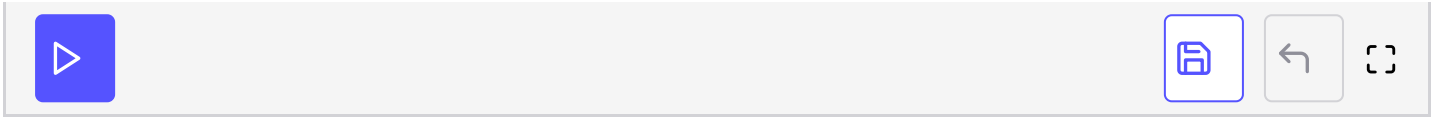
Getting results using `run`

In the previous example, we made multiple calls to methods of `Mailer`, but eventually we wanted to get the result of the `send()` method call. `apply()` is a great choice if we want to keep the `Mailer` reference at the end of the calls and do more work with the instance. But if we're after the result of a sequence of calls on an object and don't care for the instance, we may use `run()`. The `run()` method returns the result of the lambda, unlike `apply()`; but just like `apply()`, it runs the lambda in the context of the target object.

Let's modify the code to use `run()` instead of `apply()`:

```
val result = Mailer().run {
    from("builder@agiledeveloper.com")
    to("venkats@agiledeveloper.com")
    subject("Your code sucks")
    body("details")
    send()
}

println(result)
```



Each of the method calls within the lambda executed on the `Mailer` instance that was used as a target to `run()`. The result of the `send()` method, the `String`, is returned by the `run()` method to the caller. The `Mailer` instance on which `run()` was called isn't available anymore.

To keep the target object at the end of a sequence of calls, use `apply()`; to keep the result of the last expression within the lambda instead, use `run()`. In either case, use these methods only if you want to run the lambda in the context of the target.

Passing an object as argument using `let`

Suppose you receive an instance from a function but want to pass that instance to another method as argument. That sequence of operation will break the flow of code, the fluency in general. The `let()` method will help restore the fluency in that case.

Let's look at an example to illustrate this point. Here are two functions—one that returns a `Mailer` object and one that takes that in as a parameter:

```
fun createMailer() = Mailer()

fun prepareAndSend(mailer: Mailer) = mailer.run {
    from("builder@agiledeveloper.com")
    to("venkats@agiledeveloper.com")
    subject("Your code suks")
    body("details")
    send()
  }
```

The internals of the `createMailer()` and `prepareAndSend()` aren't important for this discussion. Let's focus on the code that uses these two functions:

```
val mailer = createMailer()
val result = prepareAndSend(mailer)
println(result)
```

We first stored the result of `createMailer()` into a variable, then passed it to `prepareAndSend()` and stored that result into a variable named `result` for final printing. That's boring. Where's the flow? you protest.

Ignoring the `println()`, we could change the calls to the two functions like this:

```
val result = prepareAndSend(createMailer())
```

That'll work, but fluency is definitely missing there; it feels heavy with multiple parenthesis, like the regular code we write. We want to be able to take the result of one operation and perform the next step on it, to nicely compose one call to the next. Let the `let()` method lead us there.

```
val result = createMailer().let { mailer ->
  prepareAndSend(mailer)
}
```

On the output from `createMailer()` we call `let()` and pass a lambda expression to it. Within the lambda, the `Mailer` instance—that is, the target of `let()`, the result of `createMailer()`—is available as a parameter. We are then passing it to the `prepareAndSend()` function. Hmm, really, this is an improvement? That's a reasonable question, but this code is like a delicious meal being prepared—it's not fully baked yet.

We can make one teeny-tiny change, to get rid of that parameter name:

```
val result = createMailer().let {
  prepareAndSend(it)
}
```

A notch better, but... The lambda isn't doing much; it's taking a parameter and passing it to the `prepareAndSend()` method. Instead of using a lambda here, we may use a method reference—we saw this in Using Function References.

```
val result = createMailer().let(::prepareAndSend)
```

Now that's a lot better. The result of `createMailer()` is passed to `let()`, which then passes that to the `prepareAndSend()` method, and whatever that function returns, `let()` hands it back to us.

If you want to use the result of the lambda we passed to `let()` as argument, then the method `let()` is a good choice. But to continue doing some work with the target on which `let()` was called, `also()` is the method you're looking for.

Chaining void functions using `also`

The `also()` method is useful to chain a series of `void` functions that otherwise don't fall into a call chain.

Suppose we have a bunch of `void` functions—returning `Unit` in Kotlin—like these:

```
fun prepareMailer(mailer: Mailer):Unit {
  mailer.run {
    from("builder@agiledeveloper.com")
    to("venkats@agiledeveloper.com")
    subject("Your code suks")
    body("details")
  }
```

```
}

fun sendMail(mailer: Mailer): Unit {
  mailer.send()
  println("Mail sent")
}
```

Using the `createMailer()` function we saw earlier, we can create a `Mailer` instance and then pass it to a series of functions that don't return anything. But that won't have a good flow:

```
val mailer = createMailer()
prepareMailer(mailer)
sendMail(mailer)
```

We can restore the chain of function calls using `also()` since `also()` passes the target to the lambda as a parameter, ignores the return from the lambda, and returns back the target of the call. Here's the fluent code to use these `void` functions:

```
createMailer()
  .also(::prepareMailer)
  .also(::sendMail)
```

By using these four functions, we can make the code we write each day a tad more fluent, not just pleasing to our eyes, but reducing the stress of reading the code as well.

Getting used to writing such fluent code takes some effort and practice.

The next lesson explores the concept of implicit receivers.