Extending Functions

We'll cover the following

How to extend

Functions are objects in Kotlin, and you can inject methods into functions, like you can inject methods into classes. In Java 8, the functional interface `Function<T, R>` has an `andThen()` method to combine two functions, and we can use this to compose operations—see Functional Programming in Java. Kotlin's `Function` doesn't have that method, but we can inject an `andThen()` method into Kotlin functions like this, for example:

```
// extendfunctions.kts
fun <T, R, U> ((T) -> R).andThen(next: (R) -> U): (T) -> U =
  { input: T -> next(this(input)) }
```

The extension function signature says that `andThen()` is added to a function that takes a parametrized type `T` and returns a result of type `R`. The parameter passed to `andThen()` has to be a function that takes as a parameter a variable of type `R`—the return type of the function on which `andThen()` is called—and it returns a result of parameterized type `U`. The resulting composed function created by `andThen()` takes a parameter of type `T` and returns a result of type `U`. In the body of `andThen()` we return a lambda expression. This lambda passes its parameter to the function on which `andThen()` is called and passes the result to the next function—that is, the parameter to `andThen()`. Thus, `andThen()` passes the result of one function as input to the next.

Extending functions

Let's write two standalone functions that we'll use to exercise the above function:

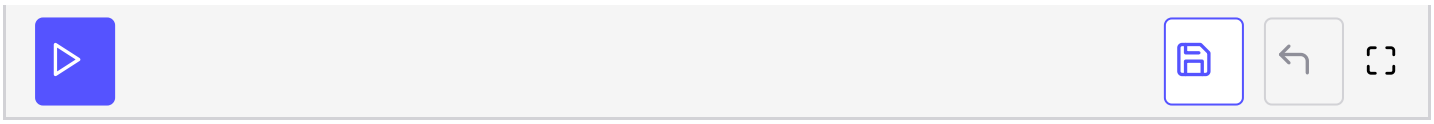
```
// extendfunctions.kts
fun increment(number: Int): Double = number + 1.toDouble()

fun double(number: Double) = number * 2
```

Now we can use `andThen()` to combine the result of `increment()` with a call to the `double()` function, like this:

```
val incrementAndDouble = ::increment.andThen(::double)

println(incrementAndDouble(5)) //12.0
```



extendfunctions.kts

On a reference to the `increment()` function, obtained using the `::` construct, we call the `andThen()` method and pass a reference to the `double()` method. The result is a function that will combine calls to these two functions, `increment()` and then `double()`.

We'll apply this technique in Memoization, the Groovy Way in Kotlin, to inject a `memoize()` method into functions.

In the next lesson, we'll see how to use `infix` for function fluency.