Creating Functions

We'll cover the following

Creating functions, and methods, in Kotlin is refreshingly different from creating methods in Java. The greater flexiblity that Kotlin offers removes unnecessary ceremony when creating functions. Let's start with short functions, explore type inference, and then look at defining parameters and multiline functions.

KISS functions

Kotlin follows the mantra "Keep it simple, stupid"—the KISS principle—to function definitions. Small functions should be simple to write, no noise, no fluff. Here's one of the shortest functions you can write in Kotlin, followed by a call to it.

```
fun greet() = "Hello"

println(greet())
```

kiss.kts

Function declarations start with the keyword `fun`—Kotlin wants you to remember to have fun every time you look at a function or a method. The function name is followed by a parameter list, which may be empty. If the function is a *single-expression function*, which is very short, then separate the body from the declaration using the `=` operator instead of using the `{}` block syntax. For short functions, the return type can be inferred. Also, the `return` keyword isn't allowed for single-expression functions, which are functions without a block body.

Run this script to see Kotlin greet you:

```
Hello
```

Let’s examine what the above function actually returns.

Return type and type inference

The `greet()` function returns a `String`, but we didn’t explicitly specify that. That’s because Kotlin can infer the return type of functions with a non-block body—that is, functions without `{}`. The return type inference happens at compile time. Let’s verify this by making a mistake in code:

```
fun greet() = "Hello"

val message: Int = greet() //ERROR
//type mismatch: inferred type is String but Int was expected
```

Based on the context, Kotlin determines that `greet()` is returning a `String`. We’re assigning that to a variable `message` of type `Int`, which is a *no-no*, and the code will fail to compile.

Type inference is safe to use and the type checking happens at compile time; use it for internal APIs and where the functions are a single expression separated by `=`. But if the function is for external use or the function is more complex, specify the return type explicitly. This will help both you and the users of your functions clearly see the return type. It will also prevent surprises that could arise if the return type is inferred and the implementation is changed to return a different type.

Kotlin will infer the return type of functions only when the function body is a single expression and not a block.

Let’s modify the `greet()` function to explicitly specify the return type.

```
fun greet(): String = "Hello"
```

The return type is prefixed with a `:` and goes right after the parameter list. The `return` keyword is still not permitted since the body of the function is a single expression and not a block.

Leave out the return type if it's obvious, specify it otherwise.

What if the function isn't returning anything? Let's visit those pesky `void` functions next.

All functions are expressions

As you saw in More Expressions, Fewer Statements, Kotlin favors expressions over statements. Based on that principle, functions must be treated as expressions instead of statements, and we can nicely compose a sequence of calls to invoke a method on the return value of each one.

Kotlin uses a special type called `Unit` that corresponds to the `void` type of Java. The name `Unit` comes from type theory and represents a singleton that contains no information. You can consistently use `Unit` to specify that you're not returning anything useful. Also, Kotlin will infer the type as `Unit` if the function isn't returning anything. Let's take a look at this inference first.

```
fun sayHello() = println("Well, hello")

val message: String = sayHello() //ERROR
//type mismatch: inferred type is Unit but String was expected
```

inferunit.kts

The `sayHello()` function prints out a message on the standard output using `println()`, which we know is a `void` function in Java, but it returns `Unit` in Kotlin. We use type inference for the return type of `sayHello()`, but be assured that it's inferred as `Unit`. We verify the inferred type by assigning the result of `sayHello()` to a variable of type `String`, and that will fail to compile due to type mismatch.

Instead of using type inference, we may explicitly specify `Unit` as the return type as well. Let's change `sayHello()` to specify the return type and then assign the result to a variable of type `Unit`:

```
fun sayHello(): Unit = println("Well, hello")

val message: Unit = sayHello()
```

```
println("The result of sayHello is $message")
```

specifyunit.kts

Since even void functions return `Unit` in Kotlin, all functions can be treated as expressions and we may invoke methods on the results from any function. For its part, the `Unit` type has `toString()`, `equals()`, and `hashCode()` methods. Though not highly useful, you may invoke any of those methods. For example, in the previous code we pass the `message` variable, which is of type `Unit`, to `println()`, and that internally calls the `toString()` method of Unit. Let's take a look at the output:

```
Well, hello
The result of sayHello is kotlin.Unit
```

The `toString()` method of Unit merely returns a String with value `kotlin.Unit`, the full name of the class.

Since all functions return something useful, or a `Unit at the very least, they all serve as expressions and the result may be assigned to a variable or used for further processing.

The functions we've used so far in this chapter didn't take any parameters, but in reality functions typically accept parameters. Let's now focus on defining parameters and passing arguments to functions.

Defining parameters

Some languages like Haskell and F# can dive into functions and infer the types of the parameters. Personally, I'm not a fan of that; changing the implementation of a function may result in change to the types of the parameters. That's unsettling for me. Kotlin insists on specifying the types for parameters to functions and methods. Provide the type of the parameter right after the parameter's name, separated by `:`.

Let's change the `greet()` function to take a parameter of String type.

```
fun greet(name: String): String = "Hello $name"
```

```
println(greet("Eve")) //Hello Eve
```

passingarguments.kts

The type specification in Kotlin has the consistent form `candidate` : Type, where the candidate can be one of the following: variable declaration using `val` or `var`, function declaration to specify the return type, function parameters, and an argument passed to the catch block.

While the type of the parameter `name` is required, we may omit the return type if we want Kotlin to infer the return type. If you have more than one parameter, list them comma separated within the parenthesis.

You saw in Prefer val over var, that we should prefer immutability over mutability and that Kotlin forces you to choose between `val` and `var` when defining local variables. But when defining the `greet()` function, we didn't specify `val` or `var` for the parameter. That has a good reason.

Effective Java, Third Edition advises programmers to use `final` and prefer immutability as much as possible. Kotlin doesn't want us to make a choice here for function and method parameters; it decided that modifying parameters passed to functions is a bad idea. You can't say `val` or `var` for parameters—they're implicitly `val`, and any effort to change the parameters' values within functions or methods will result in compilation errors.

We've seen really short functions so far. Let's look at writing more complex functions.

Functions with block body

When a function is small, a single expression, we can separate the body of the function from the declaration using the `=` operator. If the function is more complex than that, then place the body in a block `{}` and don't use `=`. You have to specify the return type for any function with a block body; otherwise, the return type is inferred as `Unit`.

Let's write a function that returns the maximum among numbers in a given array.

```
fun max(numbers: IntArray): Int {
```

```
fun max(numbers: IntArray): Int {
  var large = Int.MIN_VALUE

  for (number in numbers) {
    large = if (number > large) number else large
  }

  return large
}

println(max(intArrayOf(1, 5, 2, 12, 7, 3))) //12
```

multilinefunction.kts

The `max()` function takes an array as a parameter, specifies `Int` as the return type, and has a body wrapped in a block `{}`. In this case the return type isn't optional—you must specify it since the body of the function is a block. Also, the `return` keyword is required.

A word of caution—don't use `=` followed by a block `{}` body. If you explicitly specify the return type, and follow it with the `=`, and then a block body, the compiler will raise an error.

What if we omit the return type, but used `=` and then a block body instead of a single expression? For example, what if we write something like the following?

```
fun notReally() = { 2 }
```

Kotlin won't infer the return type by stepping into the code block. But, it will assume the entire block to be a lambda expression or an anonymous function—a topic we'll cover later in the course. Kotlin thinks that `notReally()` is a function that returns a lambda expression—oops.

For the sake of fun, admittedly a skewed definition of fun, let's explore the effect of using `=` with a block, but without a return type.

```
fun f1() = 2
fun f2() = { 2 }
fun f3(factor: Int) = { n: Int -> n * factor }

println(f1()) //2
println(f2()) //() -> kotlin.Int
println(f2()()) //2
println(f3(2)) //(kotlin.Int) -> kotlin.Int
println(f3(2)(3)) //6
```

caveat.kts

The function `f1()` is inferred to return an `Int`. But Kotlin inferred that the function `f2()` is returning a lambda expression that takes no parameters and returns an `Int`. Likewise, it inferred that the function `f3()` is returning a lambda that takes `Int` and returns `Int`.

It's highly unlikely that someone who writes such code will have many friends—no point writing code like that and spending the rest of your life alone. If you want to create functions that return lambdas, we'll see better ways to do that later. In short, don't mix `=` with block body.

QUIZ

1

Which error will the following code snippet generate?

```
fun func() = 2

val message: String = func()
```

2

Which is a valid function signature in Kotlin?

3

Which keyword in kotlin is equivilant to `void` in java?

Retake Quiz

Now that you know how to create functions; let’s see how easy it is to evolve existing functions in Kotlin.