Default and Named Arguments

We'll cover the following

- Evolving functions with default arguments
- Improving readability with named arguments

Overloading of functions is common in Java and is the way to create functions that can take different number and type of arguments. That's an option in Kotlin as well, but the feature of default arguments is a simpler and better way to evolve functions, though it requires recompilation as it breaks binary compatibility. Also, named arguments is a great way to create readable code. We'll focus on those two features now.

Evolving functions with default arguments

Quickly glance at the `greet()` function we wrote earlier, repeated here for convenience.

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

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

The `greet()` function has a hardcoded string "Hello", but what if we want to provide the flexibility to the caller of the function to provide a pleasantry of its choice?

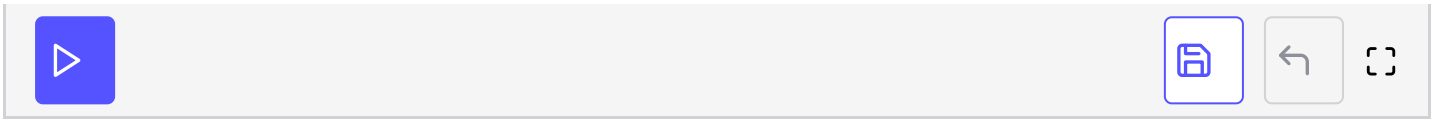
If we add a new parameter to the function, then any existing code that calls the function will break because those will be shy of the much needed, albeit new, parameter. In languages like Java we use overloading for this purpose, but that may lead to code duplication. Kotlin makes this task easy with default arguments.

A default argument is a parameter that takes a default value right after the declaration. If the caller doesn't pass a value for that parameter, then the default value is used. Specify the name of the parameter, colon, type, followed by the

assignment to the default value, using `=`. Let's change the `greet` method to take on an additional parameter, but with a default value.

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

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



defaultarguments.kts

The existing code that calls `greet()` with only one argument for name continues to work. Any new call to `greet()` may pass either one argument, for name, or two arguments, for `name` and `msg`. In the first call, since `msg` isn't provided a value, the default argument `Hello` is used. In the second call the given argument `Howdy` is used and the default argument is ignored.

In `greet()`, the parameter with default argument is placed after the regular parameter—that is, the one with no default argument. You may wonder if it makes sense to swap their order of appearance. Yes, but with consequences:

- Since a value will be required for the regular parameter, the caller will be forced to provide a value for the parameter with default argument as well—that defeats the purpose of having the default argument.
- The caller may skip the default parameter if they use named arguments—we'll see this in Improve Readability with Named Arguments.
- A parameter with default argument may precede a last parameter that stands for a lambda expression—we'll see the benefit of this in Use Lambda as the Last Parameter.

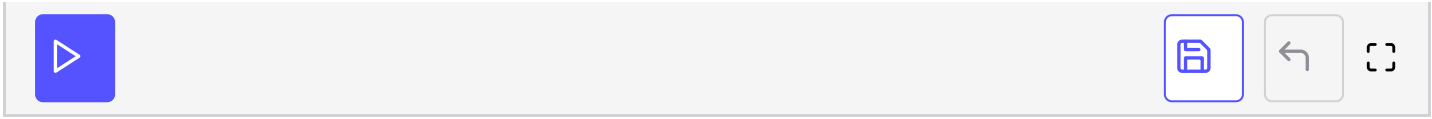
In short, to make default arguments effective, use them on trailing parameters and follow them, optionally, only with parameters for lambda expressions.

The default argument doesn't have to be a literal; it may be an expression. Also, you may compute the default arguments for a parameter using the parameters to its left.

The default argument used in the `greet()` function has a hardcoded value. Let's change that to use the parameter to its left:

```
fun greet(name: String, msg: String = "Hi ${name.length}") = "$msg $name"


println(greet("Scott", "Howdy")) //Howdy Scott
println(greet("Scott")) //Hi 5 Scott
```



defaultcompute.kts

Once again, when two arguments are passed to `greet()`, the default argument is ignored; it's not computed. On the other hand, when only the value for the name parameter is passed, then the value for the `msg` parameter is computed from the default argument expression. In the second call to `greet()`, we pass only the name and the result; `msg` is computed based on the value of the `name` parameter—a geeky high five to good friend Scott.

In this example, swapping the position of `name` and `msg` will result in a compilation error that name is uninitialized in the default argument expression—another reason for placing parameters with default arguments in the trailing positions.

Improving readability with named arguments

Code is written once but read and updated many times. Anything we can do to reduce the burden on the readers is a welcome step. In a call like `greet("Scott", "Howdy")` it's not hard to surmise what the arguments stand for. But you might come across a call like this:

```
// namedarguments.kts
createPerson("Jake", 12, 152, 43)
```

You may wonder what those numbers mean, and that's certainly no fun when dealing with impending deadlines. You'll have to switch context and look at the documentation for the function or its declaration to find the meaning of these magic numbers.

```
// namedarguments.kts
fun createPerson(name: String, age: Int = 1, height: Int, weight: Int) {
   println("$name $age $height $weight")
}
```

Poorly written code can turn the politest human into a terrible cusser—readability matters. Kotlin is a language of fluency, and that principle shines in method calls as well.

Without changing anything in how a function is defined, you can make calls to methods more readable, with little effort. This is where named arguments come in.

Let's make the call to `createPerson()` readable:

```
// namedarguments.kts
createPerson(name = "Jake", age = 12, weight = 43, height = 152)
```

That's a lot better—no guessing and no fussing over what those parameters mean. You can assign a value to a parameter's name right there in the function call. Even though the function takes `weight` as the last parameter, it isn't required to be the last in the call; named arguments may be placed in any order.

The confusion in this code is among the different integer values; the value for `name` in `createPerson` is intuitive. We can place named arguments after positional arguments, as shown in the following two examples:

```
// namedarguments.kts
createPerson("Jake", age = 12, weight = 43, height = 152)
```

```
// namedarguments.kts
createPerson("Jake", 12, weight = 43, height = 152)
```

While the last one is valid Kotlin syntax, from the readability point of view it may be better to use named argument for `age` as well.

Since `age` has a default argument, we may leave it out if we used either named values for all other parameters, or positional arguments for all parameters to its left and named arguments for all other parameters.

For example, the following two calls that leave out the argument for `age` are valid:

```
// namedarguments.kts
createPerson(weight = 43, height = 152, name = "Jake")

createPerson("Jake", weight = 43, height = 152)
```

Named arguments make method calls readable and also eliminate potential errors when adding new parameters to existing functions. Use them when the parameters passed in aren't obvious.

QUIZ

Q

What will be the output of the following code snippet?

```
fun test(num:Int= 1, letter: Char ='A'): String = "number is $num and letter is $letter"

println(test(2, 'B'))
```

Retake Quiz

Let’s now turn our attention to two other features of Kotlin in the next lesson that can reduce the noise in method calls.