Default Values & Named Parameters

Discover how to use default parameter values to increase the flexibility of your functions and how to call functions using named parameters.

WE'LL COVER THE FOLLOWING

- Default Parameter Values
  - Named Parameters
  - In a World Without Default Values...
- Quiz
- Exercises
- Summary

In Kotlin, you can define default values for function parameters to make them optional.

Default Parameter Values

Let's say you have a function that joins a collection of strings into a single string:

```
fun join(strings: Collection<String>, delimiter: String) = strings.joinToString(delimiter)
```

For instance, `join(listOf("Kotlin", "Java"), " > ")` would return `"Kotlin > Java"`.

With this function definition, you always have to pass in a `delimiter`. But most commonly, you'll want to have a comma as delimiter, so it makes sense to set this as the default value:

```
fun join(strings: Collection<String>, delimiter: String = ", ") = strings.joinToString(delimi
```

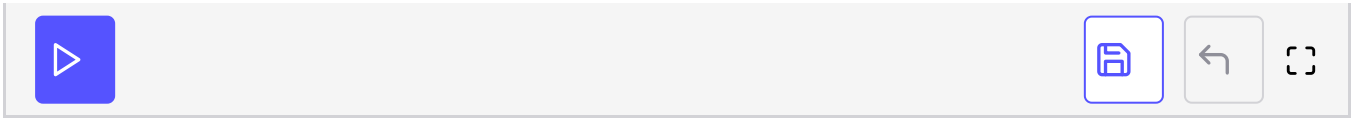
Default values are defined by simply adding an assignment to the parameter definition. This way, you now have different ways to call this function:

```
fun join(strings: Collection<String>, delimiter: String = ", ") = strings.joinToString(delimi

fun main() {
  val planets = listOf("Saturn", "Jupiter", "Earth", "Uranus")

  val joined1 = join(planets, " - ")
  val joined2 = join(planets)

  println(joined1)
  println(joined2)
}
```



You're still free to use a custom delimiter by passing in a value, but you can now omit the delimiter to simplify the function call. Think of these as **optional parameters** that can majorly increase the flexibility of your functions (i.e., of your API).

Kotlin's own `joinToString` function makes heavy use of default values because it's a prime example for their usability. You *can* pass in a delimiter, prefix, postfix, a transformation and more. But you can also call it with any subset of these or without any arguments.

> **Note:** Although the Kotlin compiler *could* infer the type of the `delimiter` parameter from the default value, you are required to define the type explicitly. This is because function signatures define an API that must be well-defined and that you should have put thought into (e.g., using an abstract type like `Collection` or `Iterable` instead of `List` to increase the function's flexibility).

Named Parameters

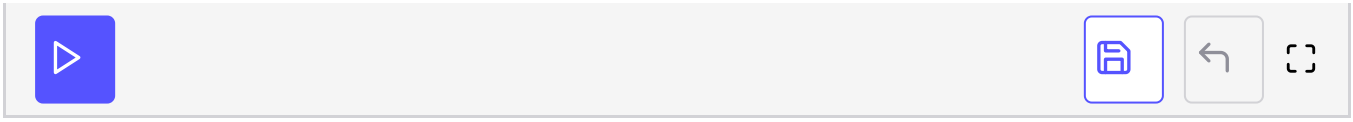
Using named parameters, there are even more ways you can call the `join` function:

```
fun join(strings: Collection<String>, delimiter: String = ", ") = strings.joinToString(delimi

fun main() {
  val planets = listOf("Saturn", "Jupiter", "Earth", "Uranus")

  val joined1 = join(planets, delimiter = " - ")
  val joined2 = join(strings = planets)
  val joined3 = join(strings = planets, delimiter = " - ")
  val joined4 = join(delimiter = ", ", strings = planets)

  println(joined1)
  println(joined2)
  println(joined3)
  println(joined4)
}
```



When you call a function in Kotlin, you can explicitly state the name of any input parameter. If you do this for a parameter, all the following ones have to be named as well.

Named parameters improve readability whenever the meaning of arguments is not clear from the function call itself, which is often the case when a function has many input parameters.

Note that you can change the order of your arguments when using named parameters.

In a World Without Default Values...

In a programming language without default values, you need to use *function overloading* to implement an equivalent `join` function. In Java for instance:

```
String join(Collection<String> strings, String delimiter) { ... }
String join(Collection<String> strings) { return join(strings, ", "
) }  // Overloaded function defines default value
```

Obviously, this requires more code, and it takes longer to identify the default values. More importantly, this so-called *telescoping (anti-)pattern* requires an exponential number of overloaded functions for more parameters with default values.

Quiz

Optional and Named Parameters

1

Which of the following function declarations uses default values correctly?

COMPLETED 0%

1 of 3

Exercises

Adjust your multiplication table function from the lesson “Basic Functions” by making both the `rows` and `columns` parameter optional with a default value. Then call your function:

1. by passing arguments for both parameters
2. by passing only a `rows` argument
3. by passing only a `columns` argument
4. by passing no argument

Also try passing your arguments in different orders using named parameters.

Problem | Solution

```
// Add your code here
```

Summary

You can make function parameters optional by defining a default value.

- Add the default values to the function signature, e.g. `fun makePizza(toppings: List<String> = emptyList())`.
  - You still have to define the parameter types explicitly.
  - There is no need to overload functions for this use case, which saves lots of boilerplate code.
- Use named parameters to pass any subset of optional parameters or to improve readability at call site.

The next lesson introduces Kotlin's concept of extension functions, a powerful feature to extend the API of existing and third-party entities.