Infix Functions

Discover what infix functions are and how you can implement your own to write more readable Kotlin code.

WE'LL COVER THE FOLLOWING

Infix functions are a great tool to improve readability of function calls in certain cases.

What are Infix Functions?

In programming, function calls are typically written in prefix notation, meaning that the function name is in front, followed by its arguments in parentheses:

```
gcd(54, 24)
```

With *infix functions*, however, the function name stands *between* the arguments. You've already seen this with the predefined `to` function, which creates `Pair`s (to put into a map):

```
"Dwayne" to 3.14159
```

Here, the function name `to` is surrounded by its two arguments, which brings us to the first limitation of infix functions: **an infix function must have exactly two parameters.** In the function call, the first argument is on the left

of the function name, and the second is on the right. Infix function calls don't need parentheses, which helps remove clutter from your code.

> Other infix functions you have already seen include `downTo` and `step`.

You can still call an infix function like a regular function as well. The following is equivalent to the example above:

```
to("Dwayne", 3.14159)
```

However, you can see that this significantly weakens code readability. Infix functions can improve readability for functions where it leads to a natural way of reading the function call.

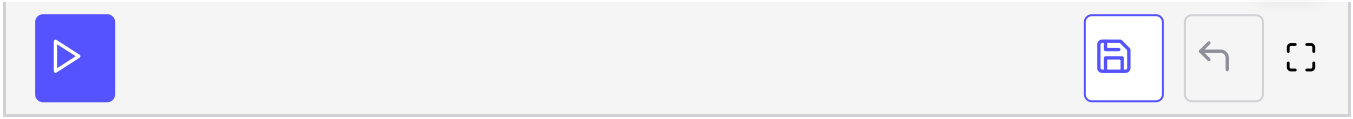
Implementing an Infix Function

Kotlin has a function to repeat a string a given number of times:

```
val str = "Ho "
str.repeat(3)
```

Let's say you want to be able to call this using infix notation as `3 times "Ho "`. To do so, you can define an infix function called `times`:

```
infix fun Int.times(str: String) = str.repeat(this)
```

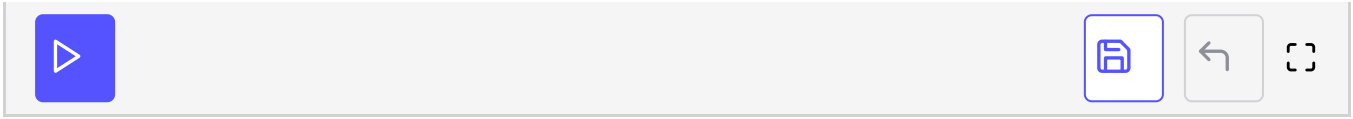


The `infix` modifier signifies that you're defining an infix function. Recall that an infix function must have two parameters, but the function signature in the example only has one. This is because an infix function must be either a class member or an extension function. The extended class becomes the first parameter type. Here, this is an extension function on the `Int` class, making the first parameter of type `Int`, and the second parameter is of type `String`. Inside the function body, you refer to the first parameter's value using `this`.

With this function definition, you can now use the syntax above:

```
infix fun Int.times(str: String) = str.repeat(this)


val greeting = 3 times "Ho "
println(greeting)
```



As you can tell, infix functions can sometimes help make the code read more naturally. This particular example can be further improved using *operator functions*, which will be covered in the next lesson.

Restrictions for Infix Functions

Infix functions must...

- be either *member functions* or *extension functions*.
- have exactly one additional parameter in the function signature (in addition to the extended class, which acts as the first parameter's type).
- *not* have varargs or default values.

Quiz

Infix Functions

1 Which of the following function calls uses infix notation?

Exercises

Implement an infix function `withoutLast` that removes the last appearance of a given substring in a given string.

Call your infix function on several examples, including:

- `"What's up Kotlin?" withoutLast "up"` should return `"What's Kotlin?"`
- `"What's up up Kotlin?" withoutLast "up"` should return `"What's up Kotlin?"`
- `"What's up upKotlin?" withoutLast "up"` should return `"What's up Kotlin?"`

Hint: You can use Kotlin's `String.substringBeforeLast` and `String.substringAfterLast` in your function body.

Summary

Infix functions allow function calls where the function name stands between its two arguments.

- They can improve code readability (e.g., Kotlin's `to` and `step` function).
- They must fulfill some (weak) restrictions, which are easy to fulfill as long as the function logically has two parameters.
- You can define your own infix functions using the `infix` modifier in your function declaration.

The following lesson will take this concept a step further with so-called *operator functions*.