Destructuring

We'll cover the following

What is destructuring

Structuring, or construction, is creating an object from values in different variables. Destructuring is the opposite—to extract values into variables from within an existing object. This facility is useful to remove noisy, repetitive code. Kotlin has the destructuring capability much like in languages such as JavaScript. But unlike JavaScript, the destructuring in Kotlin is based on the position of properties instead of the names of the properties.

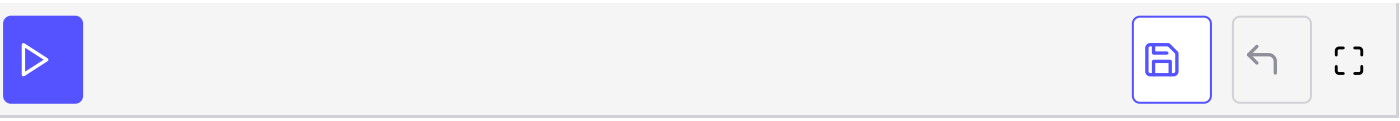
Let's start with a piece of code that is verbose and then refactor that code, using destructuring, to make it concise. `Triple` is a class in the Kotlin standard library that represents a tuple. We'll look at it further in Using Pair and Triple. For now, we'll use `Triple` to return a group of three values:

```
// destructuring.kts
fun getFullName() = Triple("John", "Quincy", "Adams")
```

Here's a traditional, boring, call to the above function, to receive the result and assign to three different variables.

```
val result = getFullName()
val first = result.first
val middle = result.second
val last = result.third

println("$first $middle $last") //John Quincy Adams
```

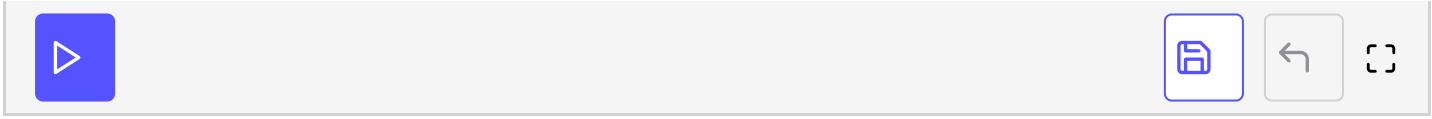


destructuring.kts

That took multiple lines of code and a few dot operations. But when the return type of a function is a `Pair`, `Triple`, or any data class, we can use destructuring to extract the values into variables, elegantly and concisely. Let's rewrite the code, this time to use destructuring.

```
val (first, middle, last) = getFullName()

println("$first $middle $last") //John Quincy Adams
```



destructuring.kts

Four lines reduced to a concise single line of code. It appears like the `getFullName()` function suddenly returned multiple values—a nice illusion. The three immutable variables `first`, `middle`, and `last` are defined in that line and are immediately assigned the three properties of the result `Triple`, in the order `first`, `second`, and `third`, respectively. In reality, this works because the `Triple` class has specialized methods to assist with destructuring; you'll learn about this later in the course. The order in which the properties are destructured is the same as the order in which the properties are initialized in the source object's constructor.

How to skip values

Suppose we don't care about one of the properties of the object being returned. For example, if we don't want the middle name, we can use an underscore (`_`) to skip it.

```
// destructuring.kts
val (first, _, last) = getFullName()

println("$first $last") //John Adams
```

Similarly, you may skip more than one property by using multiple _s, like so:

```
// destructuring.kts
val (_, _, last) = getFullName()
```

```
println(last) //Adams
```

You may place `_` at any position that you want to ignore. If you want to stop at a particular position and ignore the rest, you don't need to fill all remaining spaces with _s. For example, to get only the middle name we can write:

```
// destructuring.kts
val (_, middle) = getFullName()

println(middle) //Quincy
```

In addition to using destructuring when the return type is a data class, you can also use destructuring to extract `key` and `value` from `Map` entries—see Using Map.

If you're curious how destructuring works under the hood and what those special methods that I alluded to are, stay tuned; we'll revisit destructing and explore further in Data Classes.

QUIZ

1 What is the ouput of the following code snippet?

```
fun getFullName() = Triple("a", "b", "c")

val (last) = getFullName()

println("$last")
```

2 Which operator is used to skip values during destructuring?

Retake Quiz

The next lesson concludes the discussion for this chapter.