Iterating over Arrays and Lists

We'll cover the following

- Iterate with values
- Iterate with indexes

You can seamlessly use any of the JDK collection classes and interfaces in Kotlin. Thus, you can use Java's array and `java.util.List` in Kotlin as well. Creating instances of these in Kotlin is simpler than in Java, and you can iterate over the values in these collections with greater ease in Kotlin.

Iterate with values

Let's first create an array of numbers and examine its type:

```
// iterate.kts
val array = arrayOf(1, 2, 3)

println(array.javaClass) //class [Ljava.lang.Integer;
```

To create an array of values, use the `arrayOf()` function that belongs to the kotlin package. The functions that belong to the kotlin package may be called without the kotlin prefix—`kotlin.arrayOf()`, for example—or without any explicit imports.

Since all the values given are of type `Int`, the array created in this example is an array of `Integer` values. To create a primitive `int` array, instead of an array of `Integer` objects, use the `intArrayOf()` function. Irrespective of which function we pick, we can iterate over the array of values using the `for(x in ...)` syntax like before.

```
// iterate.kts
for (e in array) { print("$e, ") } //1, 2, 3,
```

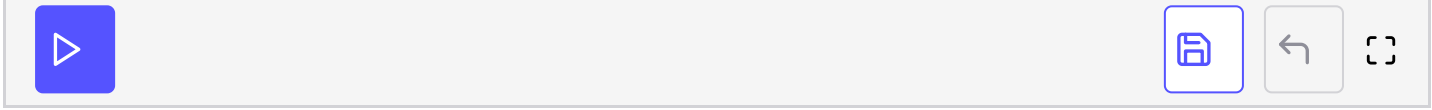
Likewise, you can create an instance of `List<T>` using the `listOf()` function and then iterate over its values using `for`:

then iterate over its values using `for`:

```
val list = listOf(1, 2, 3)

println(list.javaClass) //class java.util.Arrays$ArrayList

for (e in list) { print("$e, ") } //1, 2, 3,
```



iterate.kts

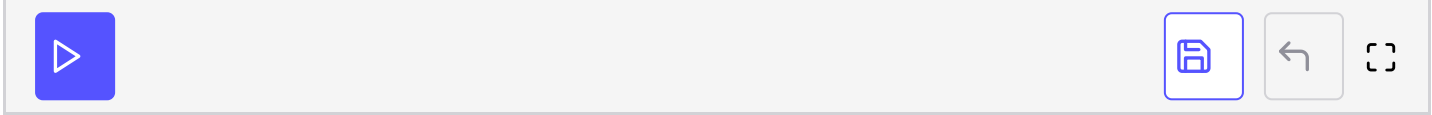
Just like in the iterations we saw earlier, the variable `e`—for element of the collection—is immutable, leading to safe iteration.

Iterate with indexes

The preceding iteration using `for` provided the values in the collection. But sometimes we need the index in addition to the value. The traditional `for` loop in C-like languages gives us the index, but not the value so easily. In this iteration we got the values easily, but getting the index, well, that shouldn't be hard either. The indices property provides a range of index values. Let's use that to iterate over a list of String values.

```
val names = listOf("Tom", "Jerry", "Spike")

for (index in names.indices) {
  println("Position of ${names.get(index)} is $index")
}
```



index.kts

Once we get the index value from the indices property we can obtain the value from the list at that position.

```
Position of Tom is 0
Position of Jerry is 1
Position of Spike is 2
```

Alternatively, we can get both the index and the position in one shot, using the `withIndex()` function along with an application of destructuring, which we saw in

Destructuring, like so:

```
// withIndex.kts
for ((index, name) in names.withIndex()) {
   println("Position of $name is $index")
}
```

You've seen how you can use the Java collections in Kotlin and also iterate over the values with greater ease. Later in the chapter we'll see how to achieve greater fluency using internal iterators, when we discuss the functional style of programming.

When iterating over a collection of values, we often want to process the values, sometimes differently, depending on the value or its type. The handy argument-matching syntax will remove so much boilerplate code, as we'll see in the next lesson.
