Flavors of Collections

We'll cover the following

- Types of collections in Kotlin
- Convenience methods added by Kotlin
- Views

In Java, we're used to different types of collections: `List`, `Set`, `Map`, and so on. We can use them in Kotlin as well. The mutable collection interfaces of Java are split into two interfaces in Kotlin: an immutable read-only interface and a mutable read-write interface. Kotlin also provides a number of convenience methods on collections in addition to those from the JDK.

Types of collections in Kotlin

When you're ready to iterate over the elements in any of these collections, Kotlin makes that task easier and much more fluent than in Java. At a high level, you may use the following collections in Kotlin:

- `Pair`—a tuple of two values.
- `Triple`—a tuple of three values.
- `Array`—indexed fixed-sized collection of objects and primitives.
- `List`—ordered collection of objects.
- `Set`—unordered collection of objects.
- `Map`—associative dictionary or map of keys and values.

Since Java already offers a wide range of collections in the JDK, you may wonder what role Kotlin plays when working with collections. The improvement comes in two forms: through extension functions and views. Let's discuss each of these separately.

Convenience methods added by Kotlin

Through the kotlin.collections package, of the standard library, Kotlin adds many useful convenience functions to various Java collections. You can use the Java collections from within Kotlin, in ways you're familiar with. For the same collections, when coding in Kotlin, you can also use the methods that Kotlin has added.

For example, in Java, we may iterate over the elements of a `List` of `String` stored in a reference `names` using the traditional `for`-loop like so:

```
//Java code
for(int i = 0; i < names.size(); i++) {
  System.out.println(names.get(i));
}
```

Alternatively, for greater fluency, we may use the for-each iterator:

```
//Java code
for(String name : names) {
  System.out.println(name);
}
```

The latter is less noisy than the former, but we get only the elements and not the index in this case. This is also true if you use the functional style `forEach` in Java instead of the imperative style for-each iterator. Kotlin makes it convenient to get both the index and then value, by adding a `withIndex()` method. Here's an example of using it:

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

println(names.javaClass)

for ((index, value) in names.withIndex()) {
  println("$index $value")
}
```

extension.kts

In this example, we call the `withIndex()` method on an instance of `ArrayList` obtained from the JDK, using Kotlin's `listOf()` method. This method returns a

special iterator over an `IndexedValue`, which is a data class. As we saw in

Destructuring, Kotlin makes it easy to extract values from data classes using destructuring. Using that facility, we obtained both the index and the value in the code. Here's the output:

```
class java.util.Arrays$ArrayList
0 Tom
1 Jerry
```

The output shows that the instance at hand is the JDK `ArrayList` class, but we're able to iterate more conveniently in Kotlin than in Java. The `withIndex()` is but one example of the vast number of convenience methods that Kotlin has added to the JDK classes. Take some time to familiarize yourself with the methods in the `kotlin.collections` package.

Views

Immutable collections are much safer to use when programming in the functional style, using concurrency, or programming asynchronous applications. Most collections in Java are mutable, but in recent years Java has introduced immutable versions. One downside, though, is that both mutable and immutable versions of the collections in Java implement the same interfaces. So any attempt to modify an immutable collection, like using the `add()` method on a `List`, for example, will result in an `UnsupportedOperationException` at runtime. Kotlin doesn't want you to wait that long to know that the operation is invalid. This is where the views come in.

Kotlin provides two different views for lists, sets, and maps: the read-only or immutable view, and the read-write or mutable view. Both views simply map to the same underlying collection in Java. There's no runtime overhead, and there are no conversions at compile time or runtime when you use these views instead of the original collection. The read-only view permits only read operations; any attempt to write using a read-only reference will fail at compile time.

For example, both `List` and `MutableList` are Kotlin views around `ArrayList`, but operations to add an element or set a value at an index using a `List` reference will fail at compile time.

A word of caution—don't assume that the use of read-only views provides thread

safety. The read-only reference is to a mutable collection and, even though you

can’t change the collection, it’s not guaranteed that the referenced collection isn’t changed in another thread. Likewise, if you use multiple views on a same instance, where some views are read-only and the others are read-write, you have to be extra careful to verify that no two threads use the read-write interface to modify the collection at the same time.

QUIZ

Which Kotlin function can be used to get both index and value from a collection?

Retake Quiz

In the next lesson, we'll first look at `Pair` and `Triple`, and then we'll move on to other more complex and powerful, collections.