Chapter 2. The Kotlin Collections Framework

In the preceding chapter we offered an overview of the syntax of the Kotlin language. As with any language, syntax is a foundation but, really, no more than that. When it comes to getting actual work done, syntax alone won't carry the water. To do that you need expressions and idioms that are easy to assemble into useful code, and that are as easy for other developers to understand and modify.

One important aspect of nearly every modern language is its *collections framework*: ways of grouping objects, and libraries of functions that manipulate them.

At the time it was introduced, Java's collection framework was state of the art. Today, more than 20 years later, the basic data structures provided by newer languages have not changed much. All of the containers that we're familiar with from the Java framework (or even the earliest versions of the C++ `stdlib`) are still there: `Iterable`, `Collection`, `List`, `Set`, and `Map` (to use their Java names). In response to broad acceptance of functional styles of programming, however, collections frameworks for modern languages like Swift and Scala usually provide a set of common, higher-order functions that operate on the collections: `filter`, `map`, `flatmap`, `zip`, and more. You will, indeed, find these functions in the collections framework from the Kotlin Standard Library.

In this chapter, we will first visit the collections themselves and a few interesting extensions that the Kotlin language empowers. After that, we will dig into some of the powerful higher-order functions that operate on the collections.

Collection Basics

Kotlin's collections framework embeds the data structures from the Java Collections Framework as a subset. It wraps the basic Java classes with

some new features and adds functional transformations that operate on them.

Let's start this deep dive into the collections library with a quick look at some of the extensions to the data structures themselves.

Java Interoperability

Because seamless interoperablity with Java is a central goal of the Kotlin language, Kotlin collection data types are based on their Java counter-parts. Figure 2-1 illustrates the relationship.

Figure 2-1. The Kotlin collection type hierarchy and its relation to Java.

By making Kotlin collection types subtypes of their Java analogs, Kotlin preserves all of functionality of the Java Collections Framework. For the most part, Kotlin extends, but does not alter the Java framework. It just adds the new, functional methods.

There is one significant exception: mutability.

Mutability

It is, perhaps, only logical that a language that embeds mutability in its syntax would also embed mutability in its collection system.

Kotlin defines two distinct type hierarchies in its collections framework, one for collections that are mutable and one for collections that are not. This can be seen in Example 2-1.

Example 2-1. Mutable and Immutable Lists

```
val mutableList = mutableListOf(1, 2, 4, 5)
val immutableList = listOf(1, 2, 4, 5)
mutableList.add(4)    // compiles

// doesn't compile: ImmutableList has no `add` method.
immutableList.add(2)
```

NOTE

Mutable is the opposite of *immutable*. A mutable object can be changed and an immutable one cannot. The distinction is critical when trying to optimize code. Since they cannot change, immutable objects can be shared safely among multiple threads. A mutable object, however, must be made explicitly thread-safe if it is to be shared. Thread safety requires locking or copying, which may be expensive.

Unfortunately, Kotlin cannot guarantee the immutablity of its immutable collections. Immutable collections simply do not have mutator functions (`add`, `remove`, `put`, etc.). Especially when a Kotlin collection is passed to Java code—where Kotlin's immutability constraints are not enforced by the type system—there can be no assurance that the contents of the collection will not change.

Note that the mutability of a collection is not related to the mutability of the object that the collection contains. As a very simple example, consider the following code:

```
val deeplist = listOf(mutableListOf(1, 2), mutableListOf(3, 4))

// Does not compile: "Unresolved reference: add"
deeplist.add(listOf(3))

deeplist[1][1] = 5      // works
deeplist[1].add(6)      // works
```

The variable `deeplist` is a `List<MutableList<Int>>`. It is and always will be a list of two lists. The contents of the lists that `deeplist` contains, however, can grow, shrink, and change.

The creators of Kotlin are actively investigating all things immutable. The prototype `kotlinx.collections.immutable` library is intended to be a set of truly immutable collections. To use them in your own Android/Kotlin project, add the following dependency to your `build.gradle` file:

```
implementation \
'org.jetbrains.kotlinx:kotlinx-collections-immutable:$IC
```

While the *Kotlinx Immutable Collections Library* uses state-of-the-art algorithms and optimizes them so that they are very fast compared to other JVM implementations of immutable collections, these true immutable collections are still an order of magnitude slower than their mutable analogs. Currently, there's nothing to be done about it. However, many modern developers are willing to sacrifice some performance for the safety that immutability brings, especially in the context of concurrency.[1]

Overloaded Operators

Kotlin supports a disciplined ability to overload the meanings of certain infix operators, in particular, `+` and `-`. Kotlin's collections framework makes good use of this capability. To demonstrate, let's look at a naive implementation of a function to convert a `List<Int>` to a `List<Double>`:

```
fun naiveConversion(intList: List<Int>): List<Double> {
    var ints = intList
    var doubles = listOf<Double>()
    while (!ints.isEmpty()) {
        val item = ints[0]
        ints = ints - item
        doubles = doubles + item.toDouble()
    }
    return doubles
}
```

Don't do this. The only thing that this example does efficiently is demonstrate the use of the two infix operators `+` and `-`. The former adds an element to a list and the latter removes an element from it.

The operand to the left of a `+` or `-` operator can define the behavior of that operator. Containers, when they appear to the left of a `+` or `-`, define two implementations for each of those two operators: one when the right-hand operand is another container and the other when it is not.

Adding a noncontainer object to a container creates a new container that has all of the elements from the left-hand operand (the container) with the new element (the right-hand operand) added. Adding two containers together creates a new container that has all of the elements from both.

Similarly, subtracting an object from a container creates a new container with all but the first occurrence of the left-hand operand. Subtracting one container from another produces a new container that has the elements of the left-hand operand, with *all* occurrences of *all* the elements in the right-hand operand removed.

NOTE

The `+` and `-` operators preserve order when the underlying container is ordered. For instance:

```
(listOf(1, 2) + 3)
    .equals(listOf(1, 2, 3))    // true
(listOf(1, 2) + listOf(3, 4))
    .equals(listOf(1, 2, 3, 4)) // true
```

Creating Containers

Kotlin does not have a way to express container literals. There is no syntactic way, for instance, of making a `List` of the numbers 8, 9, and 54. Nor is there a way of making a `Set` of the strings "Dudley" and "Mather." Instead, there are handy methods for creating containers that are nearly as elegant. The code in Example 2-1 showed two simple examples of creating lists. There are also `...Of` methods for creating mutable and immutable lists, sets, and maps.

Creating literal maps requires knowing a clever trick. The `mapOf` function takes a list of `Pairs` as its argument. Each of the pairs provides a key (the pair's first value) and a value (the pair's second value). Recall that Kotlin supports an extended set of infix operators. Among these operators is `to`, which creates a new `Pair` with its left operand as the first element and its right operand as the second element. Combine these two features and you can, conveniently, build a `Map` like this:

```
val map = mapOf(1 to 2, 4 to 5)
```

The type of the content of a container is expressed using a generic syntax very similar to Java's. The type of the variable map in the preceding code,

for instance, is `Map<Int, Int>`, a container that maps `Int` keys to their `Int` values.

The Kotlin compiler is quite clever about inferring the types of the contents of containers created with their factory methods. Obviously in this example:

```
val map = mutableMapOf("Earth" to 3, "Venus" to 4)
```

the type of `map` is `MutableMap<String, Int>`. But what about this?

```
val list = listOf(1L, 3.14)
```

Kotlin will choose the nearest type in the type hierarchy tree that is an ancestor of all of the elements of the container (this type is called the *upper bound type*). In this case it will choose `Number`, the nearest ancestor of both `Long` and `Double`. The variable `list` has the inferred type `List<Number>`.

We can add a `String`, though, as in the following:

```
val list = mutablelistOf(1L, 3.14, "e")
```

The only type that is an ancestor to all of the elements, a `Long`, a `Double`, and a `String`, is the root of the Kotlin type hierarchy, `Any`. The type of the variable `list` is `MutableList<Any>`.

Once again, though, recall from Chapter 1 that the type `Any` is not the same as the type `Any?`. The following will not compile (assuming the definition from the preceding example):

```
list.add(null)  // Error: Null cannot be a value of a non-null type Any
```

In order to allow the list to contain `null`, we'd have to specify its type explicitly:

```
val list: MutableList<Any?> = mutablelistOf(1L, 3.14, "e")
```

We can create collections now. So, what do we do with them?

Functional Programming

We operate on them! Nearly all of the operations that we will discuss here are based on the paradigm of functional programming. In order to understand their context and motivation, let's review the paradigm.

Object-oriented programming (OOP) and *functional programming* (FP) are both paradigms for software design. Software architects understood the promise of functional programming soon after its invention in the late 1950s. Early functional programs tended to be slow, though, and it's only recently that the functional style has been able to challenge a more pragmatic imperative model for performance. As programs get more complex and difficult to understand, as concurrency becomes inevitable, and as compiler optimization improves, functional programming is changing from a cute academic toy into a useful tool that every developer should be able to wield.

Functional programming encourages *immutability*. Unlike the functions in code, mathematical functions don't change things. They don't "return" anything. They simply have a value. Just as "4" and "2 + 2" are names for the same number, a given function evaluated with given parameters is simply a name (perhaps a verbose name!) for its value. Because mathematical functions do not change, they are not affected by time. This is immensely useful when working in a concurrent environment.

Though different, FP and OOP paradigms can coexist. Java was, certainly, designed as an OO language, and Kotlin, fully interoperable, can duplicate Java algorithms nearly word for word. As we proclaimed in the preceding chapter, though, the true power of Kotlin lies in its extensible functional programming capabilities. It's not uncommon for folks to start out writing "Java in Kotlin." As they start to feel more comfortable, they tend to gravitate toward more idiomatic Kotlin, and much of that involves applying the power of FP.

Functional Versus Procedural: A Simple Example

The following code shows a procedural way of working with a collection:

```
fun forAll() {
    for (x in collection) { doSomething(x) }
}
```

In the example, a `for` loop iterates over a list. It selects an element from `collection` and assigns it to the variable `x`. It then calls the method `doSomething` on the element. It does this for each element in the list.

The only constraint on the collection is that there must be a way to fetch each of its elements exactly once. That capability is precisely what is encapsulated by the type `Iterable<T>`.

The functional paradigm is certainly less complicated: no extra variables and no special syntax. Just a single method call:

```
fun forAll() = collection.forEach(::doSomething)
```

The `forEach` method takes a function as its argument. That argument, `doSomething` in this case, is a function that takes a single parameter of the type contained in `collection`. In other words, if `collection` is a list of `String`s, `doSomething` must be `doSomething(s: String)`. If `collection` is a `Set<Freeptootsie>`, then `doSomething` must be `doSomething(ft: Freeptootsie)`. The `forEach` method calls its argument (`doSomething`) with each element in `collection` as its parameter.

This might seem like an insignificant difference. It is not. The `forEach` method is a much better separation of concerns.

An `Iterable<T>` is stateful, ordered, and time dependent. Anyone who has ever had to deal with a `ConcurrentModificationException` knows it is entirely possible that the state of an iterator may not match the state of the collection over which it is iterating. While Kotlin's `forEach` operator is not completely immune to `ConcurrentModificationException`, those exceptions occur in code that is actually concurrent.

More importantly, the mechanism that a collection uses to apply a passed function to each of its elements is entirely the business of the collection

itself. In particular, there is no intrinsic contract about the order in which the function will be evaluated on the collection's elements.

A collection could, for instance, divide its elements into groups. It could farm each of these groups out to a separate processor and then reassemble the results. This approach is particularly interesting at a time when the number of cores in a processor is increasing rapidly. The `Iterator<T>` contract cannot support this kind of parallel execution.

Functional Android

Android has a quirky history with functional programming. Because its virtual machine has nothing to do with Java's, improvements in the Java language have not necessarily been available to Android developers. Some of the most important changes in Java, including lambdas and method references, were not supported in Android for quite a while after they appeared in Java 8.

Although Java could compile these new features and DEX (Android's bytecode) could even represent them (though, perhaps, not efficiently), the Android toolchain couldn't convert the representations of these features —the compiled Java bytecode—into the DEX code that could be run on an Android system.

The first attempt to fill the gap was a package called *RetroLambda*. Other add-on library solutions followed, sometimes with confusing rules (e.g., with the Android Gradle Plugin [AGP] 3.0+, if you wanted to use the Java Streams API you had to target, at a minimum, Android API 24).

All of these constraints are now gone with Kotlin on Android. Recent versions of the AGP will support functional programming even on older versions of Android. You can now use the full Kotlin collection package on any supported platform.

Kotlin Transformation Functions

In this section, you will see how Kotlin brings functional capabilities to collections to provide elegant and safe ways of manipulating them. Just as in the previous chapter we didn't visit all of Kotlin's syntax, we will not in

this chapter attempt to visit all of Kotlin's library functions. It isn't necessary to memorize them all. It is essential, though, for idiomatic and effective use of Kotlin, to get comfortable with a few key transforms and to get a feel for how they work.

The Boolean Functions

A convenient set of collection functions return a `Boolean` to indicate whether the collection has—or does not have—a given attribute. The function `any()`, for instance, will return `true` when a collection contains at least one element. If used with a predicate, as in `any { predicate(it) }`, `any` will return `true` if the predicate evaluates true for any element in the collection:

```
val nums = listOf(10, 20, 100, 5)
val isAny = nums.any()                  // true
val isAnyOdd = nums.any { it % 1 > 0 } // true
val isAnyBig = nums.any { it > 1000}   // false
```

NOTE

When a lambda takes only a single argument and the Kotlin compiler can figure that out using type inferencing (it usually can), you can omit the parameter declaration and use the implicit parameter named `it`. The preceding example uses this shortcut twice, in the definitions of the predicates to the `any` method.

Another boolean function, `all { predicate }`, returns `true` only if every element in the list matches the predicate:

```
val nums = listOf(10, 20, 100, 5)
val isAny = nums.all { it % 1 > 0 } // false
```

The opposite of `any` is `none`. Without a predicate, `none()` returns `true` only if there are no elements in a collection. With a predicate, `none { predicate }` returns `true` only if the predicate evaluates to true for none of the elements in the collection. For example:

```
val nums = listOf(10, 20, 100, 5)
val isAny = nums.none()                 // false
```

```
val isAny4 = nums.none { it == 4 }   // true
```

Filter Functions

The basic `filter` function will return a new collection containing only the elements of the original collection that match the given predicate. In this example, for instance, the variable `numbers` will contain a list with the single value `100`:

```
val nums = listOf(10, 20, 100, 5)
val numbers = nums.filter { it > 20 }
```

The `filterNot` function is the reverse. It returns elements that do *not* match the predicate. In this example, for instance, the variable `numbers` will contain three elements, 10, 20, and 5: the elements of `nums` that are not greater than 20:

```
val nums = listOf(10, 20, 100, 5)
val numbers = nums.filterNot { it > 20 }
```

A beautifully convenient special case of `filterNot` is the function `filterNotNull`. It removes all of the `null`s from a collection:

```
val nums = listOf(null, 20, null, 5)
val numbers = nums.filterNotNull() // { 20, 5 }
```

In this example, the variable `numbers` will be a list containing two elements, 20 and 5.

Map

The *map* function applies its argument to each element in a collection and returns a collection of the resulting values. Note that it does not mutate the collection to which it is applied; it returns a new, resulting, collection.

Here is the definition of the `map` function, for the `Array` type:

```
inline fun <T, R> Array<out T>.map(transform: (T) -> R): List<R>
```

Let's unpack this.

Starting at the left, `map` is an inline function. The "fun" part should be clear by now. But what about "inline."

The keyword `inline` tells the Kotlin compiler to copy the bytecode for a function directly into the binary whenever the method is called, instead of generating a transfer to a single compiled version. When the number of instructions necessary to call a function is a substantial percentage of the total number necessary to run it, an `inline` function makes sense as a trade-off of space for time. Sometimes, too, it can remove the overhead of the extra object allocation that some lambda expressions require.

Next, `<T, R>` are the two, free, type variables used in the function definition. We'll get back to them.

Next is the description of the receiver, `Array<out T>`. This `map` function is an extension function on the `Array` type: it is a function on an array whose elements are of type `T` (or one of `T`'s superclasses, e.g., `Any`).

Next is the `map`'s parameter. The parameter is a function named *transform*. Transform is a function `transform: (T) -> R`: it takes as its argument something of type `T` and returns something of type `R`. Well! That's interesting! The array to which the function will be applied is full of objects of type `T`! The function can be applied to the elements of the array.

Finally, there is `map`'s return. It is a `List<R>`, a list whose elements are of type `R`. An `R` is what you get if you apply `transform` to an elements of the array (a `T`).

It all works out. Calling `map` on an array with a function that can be applied to the elements of the array will return a new `List` that contains the results of the application of the function to each of the elements in the array.

Here's an example that returns a list of starting dates for employee records that have those starting dates stored as strings:

```
data class Hire(
    val name: String,
```

```
    val position: String,
    val startDate: String
)

fun List<Hire>.getStartDates(): List<Date> {
    val formatter
        = SimpleDateFormat("yyyy-MM-d", Locale.getDefault())
    return map {
        try {
            formatter.parse(it.startDate)
        } catch (e: Exception) {
            Log.d(
                "getStartDates",
                "Unable to format first date. $e")
            Date()
        }
    }
}
```

Perhaps you're wondering: "What happens if the transform function doesn't return a value?" Ah! But Kotlin functions *always* have a value!

For example:

```
val doubles: List<Double?> = listOf(1.0, 2.0, 3.0, null, 5.0)
val squares: List<Double?> = doubles.map { it?.pow(2) }
```

In this example, the variable `squares` will be the list [1.0, 4.0, 9.0, null, 25.0]. Because of the conditional operator, `?.`, in the transform function, the function's value is the square of its argument, if that argument is not null. If the argument is null, however, the function has the value `null`.

There are several variations on the `map` function in the Kotlin library. One of them, `mapNotNull`, addresses situations like this:

```
val doubles: List<Double?> = listOf(1.0, 2.0, 3.0, null, 5.0)
val squares: List<Double?> = doubles.mapNotNull { it?.pow(2) }
```

The value of the variable `squares` in this example is [1.0, 4.0, 9.0, 25.0].

Another variant of `map` is `mapIndexed`. `mapIndexed` also takes a function as its argument. Unlike `map`, though, `mapIndexed`'s functional argu-

ment takes an element of the collection as its second parameter (not its first and only parameter, as did `map`'s argument). `mapIndexed`'s functional argument takes, as its first parameter, an `Int`. The `Int` is the ordinal that gives the position in the collection of the element that is its second paramter: 0 for the first element, 1 for the second, and so on.

There are mapping functions for most collection-like objects. There are even similar functions for `Map`s (though they are not subtypes of `Collection`): the functions `Map::mapKeys` and `Map::mapValues`.

flatMap

The thing that makes the `flatMap` function hard to understand is that it may seem abstract and not particularly useful. It turns out that, although it is abstract, it is quite useful.

Let's start with an analogy. Suppose you decide to reach out to the members of your old high school debate team. You don't know how to get in touch anymore. You do remember, though, that you have yearbooks for all four years you were in the school and that each yearbook has a picture of the debate team.

You decide to divide the process of contacting members into two steps. First you will examine each photo of the team and try to identify each person depicted there. You will make a list of the people you identify. You will then combine the four lists into a single list of all debate-team members.

That's flatmapping! It's all about containers. Let's generalize.

Suppose you have some kind of container of something. It is a `CON<T>`. In the yearbook example, `CON<T>` was four photographs, a `Set<Photo>`. Next you have a function that maps `T -> KON<R>`. That is, it takes an element of `CON` and turns it into a new kind of container, a `KON`, whose elements are of type `R`. In the example, this was you identifying each person in one of the photos, and producing a list of names of people in the photo. `KON` is a paper list and `R` is the name of a person.

The result of the `flatMap` function in the example is the consolidated list of names.

The flatmap on `CON<T>` is the function:

```
fun <T, R> CON<T>.flatMap(transform: (T) -> KON<R>): KON<R>
```

Note, just for comparison, how `flatMap` is different from `map`. The `map` function, for the container `CON`, using the same transform function, has a signature like this:

```
fun <T, R> CON<T>.map(transform: (T) -> KON<R>): CON<KON<R>>
```

The `flatMap` function "flattens" away one of the containers.

While we're on the subject, let's take a look at an example of the use of `flatMap` that is very common:

```
val list: List<List<Int>> = listOf(listOf(1, 2, 3, 4), listOf(5, 6))
val flatList: List<Int> = list.flatMap { it }
```

The variable `flatList` will have the value [1, 2, 3, 4, 5, 6].

This example can be confusing. Unlike the previous example, which converted a set of photographs to lists of names and then consolidated those lists, in this common example the two container types `CON` and `KON` are the same: they are `List<Int>`. That can make it difficult to see what's actually going on.

Just to prove that it works, though, let's go through the exercise of binding the quantities in this somewhat baffling example to the types in the function description. The function is applied to a `List<List<Int>>`, so `T` must be a `List<Int>`. The transform function is the identity function. In other words, it is `(List<Int>) -> List<Int>`: it returns its parameter. This means that `KON<R>` must also be a `List<Int>` and `R` must be an `Int`. The `flatMap` function, then, will return a `KON<R>`, a `List<Int>`.

It works.

Grouping

In addition to filtering, the Kotlin Standard Library provides another small set of transformation extension functions that group elements of a collection. The signature for the `groupBy` function, for instance, looks like this:

```
inline fun <T, K> Array<out T>
    .groupBy(keySelector: (T) -> K): Map<K, List<T>>
```

As is often the case, you can intuit this function's behavior just by looking at the type information. `groupBy` is a function that takes an `Array` of things (`Array` in this case: there are equivalents for other container types). For each of the things, it applies the `keySelector` method. That method, somehow, labels the thing with a value of type `K`. The return from the `groupBy` method is a map of each of those labels to a list of the things to which the `keySelector` assigned that label.

An example will help:

```
val numbers = listOf(1, 20, 18, 37, 2)
val groupedNumbers = numbers.groupBy {
    when {
        it < 20 -> "less than 20"
        else -> "greater than or equal to 20"
    }
}
```

The variable `groupedNumbers` now contains a `Map<String, List<Int>>`. The map has two keys, "less than 20" and "greater than or equal to 20." The value for the first key is the list [1, 18, 2]. The value for the second is [20, 37].

Maps that are generated from grouping functions will preserve the order of the elements in the original collection, in the lists that are the values of the keys of the output map.

Iterators Versus Sequences

Suppose you are going to paint your desk. You decide that it will look much nicer if it is a nice shade of brown instead of that generic tan. You

head down to the paint store and discover that there are around 57 colors that might be just the thing.

What you do next? Do you buy samples of each of the colors to take home? Almost certainly not! Instead, you buy samples of two or three that seem promising and try them. If they turn out not to be all your heart desires, you go back to the store and buy three more. Instead of buying samples of all the candidate colors and iterating over them, you create a process that will let you get the next candidate colors, given the ones you have already tried.

A sequence differs from an iterator in a similar way. An iterator is a way of getting each element from an existing collection exactly once. The collection exists. All the iterator needs to do is order it.

A sequence, on the other hand, is not necessarily backed by a collection. Sequences are backed by *generators*. A generator is a function that will provide the next item in the sequence. In this example, if you need more paint samples, you have a way of getting them: you go back to the store and buy more. You don't have to buy them all and iterate over them. You just need to buy a couple because you know how to get more. You can stop when you find the right color, and with luck, that will happen before you pay for samples of all of the possible colors.

In Kotlin, you might express your search for desk paint like this:

```
val deskColor = generateSequence("burnt umber") {
    buyAnotherPaintSample(it)
}.first { looksGreat(it) }

println("Start painting with ${deskColor}!")
```

This algorithm is efficient. On average, desk painters using it will buy only 28 paint samples instead of 57.

Because sequences are lazy—only generating the next element when it is needed—they can be very, very useful in optimizing operations, even on collections with fixed content. Suppose, for instance, that you have a list of URLs, and you want to know which one is a link to a page that contains an image of a cat. You might do it like this:

```
val catPage = listOf(
    "http://ragdollies.com",
    "http://dogs.com",
    "http://moredogs.com")
    .map { fetchPage(it) }
    .first { hasCat(it) }
```

That algorithm will download all of the pages. If you do the same thing using a sequence:

```
val catPage = sequenceOf(
    "http://ragdollies.com",
    "http://dogs.com",
    "http://moredogs.com")
    .map { fetchPage(it) }
    .first { hasCat(it) }
```

only the first page will be downloaded. The sequence will provide the first URL, the `map` function will fetch it, and the `first` function will be satisfied. None of the other pages will be downloaded.

Be careful, though! Don't ask for all of the elements of an infinite collection! This code, for instance, will eventually produce an `OutOfMemory` error:

```
val nums = generateSequence(1) { it + 1 }
    .map { it * 7 }                  // that's fine
    .filter { it mod 10000 = 0 }     // still ok
    .asList()                        // FAIL!
```

An Example

Let's make all this concrete with an example.

We just met several of the handy functions that Kotlin's Standard Library provides for manipulating collections. Using those functions, you can create robust implementations of complex logic. To illustrate that, we'll take an example inspired by a real application used in an aircraft engine factory.

The Problem

Bandalorium Inc. builds aircraft engines. Each engine part is uniquely identifiable by its serial number. Each part goes through a rigorous quality control process that records numerical measurements for several of the part's critical attributes.

An attribute for an engine part is any measurable feature. For example, the outside diameter of a tube might be an attribute. The electrical resistance of some wire might be another. A third might be a part's ability to reflect a certain color of light. The only requirement is that measuring the attribute must produce a single numerical value.

One of the things that Bandalorium wants to track is the precision of its production process. It needs to track the measurements of the parts it produces and whether they change over time.

The challenge, then, is:

Given a list of measurements for attributes of parts produced during a certain interval (say, three months), create a CSV (comma-separated value) report similar to the one shown in Figure 2-2. As shown, the report should be sorted by the time that the measurement was taken.

	A	B	C	D	E
1	**Date**	**Serial**	**AngleOfAttack**	**ChordLength**	**PaintColor**
2	27-07-2020 15:15:00	HC14	15.08	0.71	7,951,688.0
3	27-07-2020 15:25:00	HC13	15.11	0.69	-
4	27-07-2020 15:35:00	HC12	15.05	0.7	-
5	27-07-2020 15:45:00	HC11	15.1	0.68	2201331.0

Figure 2-2. Example of CSV ouput.

If we might make a suggestion—now would be a great time to put this book aside for a moment and consider how you would approach this problem. Maybe just sketch enough high-level code to feel confident that you can solve it.

The Implementation

In Kotlin, we might represent an attribute like this:

```
data class Attr(val name: String, val tolerance: Tolerance)

enum class Tolerance {
    CRITICAL,
    IMPORTANT,
    REGULAR
}
```

The name is a unique identifier for the attribute. An attribute's tolerance indicates the significance of the attribute to the quality of the final product: critical, important, or just regular.

Each attribute probably has lots of other associated information. There is, surely, a record of the units of measurement (centimeters, joules, etc.), a description of its acceptable values, and perhaps the procedure used to measure it. We will ignore those features for this example.

A measurement of an attribute for a specific engine part includes the following:

- The serial number of the part being measured
- A timestamp giving the time at which the measurement was made
- The measured value

A measurement, then, might be modeled in Kotlin like this:

```
data class Point(
    val serial: String,
    val date: LocalDateTime,
    val value: Double)
```

Finally, we need a way to connect a measurement to the attribute it measures. We model the relationship like this:

```
data class TimeSeries(val points: List<Point>, val attr: Attr)
```

The `TimeSeries` relates a list of measurements to the `Attr`s that they measure.

First, we build the header of the CSV file: the column titles that comprise the first line (see Example 2-2). The first two columns are named `date` and `serial`. The other column names are the distinct names of the attributes in the dataset.

Example 2-2. Making the header

```
fun createCsv(timeSeries: List<TimeSeries>): String {
    val distinctAttrs = timeSeries
        .distinctBy { it.attr }
        .map { it.attr }          ❷
        .sortedBy { it.name }     ❸

    val csvHeader = "date;serial;" +
        distinctAttrs.joinToString(";") { it.name } +
        "\n"

    /* Code removed for brevity */
}
```

❶ Use the `distinctBy` function to get a list of `TimeSeries` instances that have distinct values for the `attr` attribute.

❷ We have a list of distinct `TimeSeries` from the previous step and we only want the `attr`, so we use the `map` function.

❸ Finally, we sort alphabetically using `sortedBy`. It wasn't required but why not?

Now that we have the list of distinct characteristics, formatting the header is straightforward using the `joinToString` function. This function transforms a list into a string by specifying a string separator to insert between each element of the list. You can even specify a prefix and/or a postfix if you need to.

NOTE

It is often useful to be able to find the types of the returns from collection transformation functions. In Example 2-2, for instance, if you activate type hints, you'll only get the inferred type of the whole chain (the type of the variable `distinctAttrs`). There is a nice IntelliJ/Android Studio feature that can help!

1. Click on `distinctCharacs` in the source code.
2. Hit Ctrl + Shift + P. You'll see a drop-down window appear.

3. Select the step you want and the inferred type will appear before your eyes!

After building the header, we build the content of the CSV file. This is the most technical and interesting part.

The rest of the CSV file that we are trying to reproduce sorts the data by date. For each given date, it gives a part's serial number and then that part's measurement for each attribute of interest. That's going to take some thought because, in the model we've created, those things are not directly related. A `TimeSeries` contains only data for a single attribute and we will need data for multiple attributes.

A common approach in this situation is to merge and flatten the input data into a more convenient data structure, as shown in Example 2-3.

Example 2-3. Merge and flatten the data

```
fun createCsv(timeSeries: List<TimeSeries>): String {
    /* Code removed for brevity */

    data class PointWithAttr(val point: Point, val attr:

    // First merge and flatten so we can work with a lis
    val pointsWithAttrs = timeSeries.flatMap { ts ->
        ts.points.map { point -> PointWithAttr(point, ts
```

```
    /* Code removed for brevity */
}
```

In this step, we associate each `Point` with its corresponding `Attr`, in a single `PointAndAttr` object. This is much like joining two tables in SQL.

The `flatMap` function transforms a list of `TimeSeries` objects. Internally, the function applied by `flatMap` uses the `map` function, `series.points.map { ... }`, to create a list of `PointAndAttr`s for each point in the `TimeSeries`. If we had used `map` instead of `flatMap`, we would have produced a `List<List<PointAndAttr>>`. Remember, though, that `flatMap` flattens out the top layer of the container, so the result here is a `List<PointAndAttr>`.

Now that we have "spread" the attribute information into every `Point`, creating the CSV file is fairly straightforward.

We'll group the list of `pointWithAttrs` by date to create a `Map<LocalDate, List<PointWithAttr>`. This map will contain a list of `pointWithAttrs` for each date. Since the example seems to have a secondary sort (by the part's serial number), we'll have to group each of the lists in the previously grouped `Map` by serial number. The rest is just string formatting, as shown in Example 2-4.

Example 2-4. Create data rows

```
fun createCsv(timeSeries: List<TimeSeries>): String {
    /* Code removed for brevity */

    val rows = importantPointsWithAttrs.groupBy { it.poi
    .toSortedMap()                                          ❷
    .map { (date, ptsWithAttrs1) ->
        ptsWithAttrs1
            .groupBy { it.point.serial }                    ❸
            .map { (serial, ptsWithAttrs2) ->
                listOf(
                    date.format(DateTimeFormatter.ISO_LO
                    serial
                ) + distinctAttrs.map { attr ->
                    val value = ptsWithAttrs2.firstOrNul
                    value?.point?.value?.toString() ?: "
                }
```

```
            }.joinToString(separator = "") {
                it.joinToString(separator = ";", postfix
            }
    }.joinToString(separator = "")


    return csvHeader + rows
}
```

Group by date, using the `groupBy` function.

2. Sort the map (by date). It's not mandatory, but a sorted CSV is easier to read.

3. Group by serial number.

   Build the list of values for each line.

   Format each line and assemble all those lines using the `joinToString` function.

   Finally, return the header and the rows as a single `String`.

Now, let's suppose that you get an additional request to report only on attributes that are `CRITICAL` or `IMPORTANT`. You just have to use the `filter` function, as shown in Example 2-5.

Example 2-5. Filter critical and important samples

```
fun createCsv(timeSeries: List<TimeSeries>): String {
    /* Code removed for brevity */

    val pointsWithAttrs2 = timeSeries.filter {
        it.attr.tolerance == Tolerance.CRITICAL
                || it.attr.tolerance == Tolerance.IMPORTA
    }.map { series ->
        series.points.map { point ->
            PointWithAttr(point, series.attr)
        }
    }.flatten()

    /* Code removed for brevity */
```

```
    return csvHeader + rows
}
```

That's it!

To test that code, we can use a predefined input and check that the output matches your expectations. We won't show a full-blown set of unit tests here—just an example of CSV output, as shown in Example 2-6.

Example 2-6. Demonstrate the application

```
fun main() {
    val dates = listOf<LocalDateTime>(
        LocalDateTime.parse("2020-07-27T15:15:00"),
        LocalDateTime.parse("2020-07-27T15:25:00"),
        LocalDateTime.parse("2020-07-27T15:35:00"),
        LocalDateTime.parse("2020-07-27T15:45:00")
    )
    val seriesExample = listOf(
        TimeSeries(
            points = listOf(
                Point("HC11", dates[3], 15.1),
                Point("HC12", dates[2], 15.05),
                Point("HC13", dates[1], 15.11),
                Point("HC14", dates[0], 15.08)
            ),
            attr = Attr("AngleOfAttack", Tolerance.CRITI
        ),
        TimeSeries(
            points = listOf(
                Point("HC11", dates[3], 0.68),
                Point("HC12", dates[2], 0.7),
                Point("HC13", dates[1], 0.69),
                Point("HC14", dates[0], 0.71)
            ),
            attr = Attr("ChordLength", Tolerance.IMPORTA
        ),
        TimeSeries(
            points = listOf(
                Point("HC11", dates[3], 0x2196F3.toDoubl
                Point("HC14", dates[0], 0x795548.toDoubl
            ),
            attr = Attr("PaintColor", Tolerance.REGULAR)
        )
    )
```

```
    val csv = createCsv(seriesExample)
    println(csv)
}
```

If you use the `csv` string as the content of a file with the ".csv" extension, you can open it using your favorite spreadsheet tool. Figure 2-3 shows what we got using FreeOffice.

	A	B	C	D	E
1	**Date**	**Serial**	**AngleOfAttack**	**ChordLength**	**PaintColor**
2	27-07-2020 15:15:00	HC14	15.08	0.71	7,951,688.0
3	27-07-2020 15:25:00	HC13	15.11	0.69	-
4	27-07-2020 15:35:00	HC12	15.05	0.7	-
5	27-07-2020 15:45:00	HC11	15.1	0.68	2201331.0

Figure 2-3. Final output.

Using functional programming to transform data, as in this example, is particularly robust. Why? By combining Kotlin's null safety and functions from the Standard Library, you can produce code which has either few or no side effects. Throw in any list of `PointWithAttr` you can imagine. If even one `Point` instance has a `null` value, the code won't even compile. Anytime the result of transformation returns a result which can be null, the language forces you to account for that scenario. Here we did this in step 4, with the `firstOrNull` function.

It's always a thrill when your code compiles and does exactly what you expect it to do on the first try. With Kotlin's null safety and functional programming, that happens a lot.

Summary

As a functional language, Kotlin employs great ideas like mapping, zipping, and other functional transformations. It even allows you to create your own data transformations with the power of higher-order functions and lambdas:

- Kotlin collections include the entire Java collections API. In addition, the library provides all the common functional transformations like mapping, filtering, grouping, and more.
- Kotlin supports inline functions for more performant data transformations.

- The Kotlin collections library supports sequences, a way of working with collections that are defined by intention instead of extension. Sequences are appropriate when getting the next element is very expensive, or even on collections of unbounded size.

If you've ever used languages like Ruby, Scala, or Python, perhaps some of this feels familiar to you. It should! Kotlin's design is based on many of the same principles that drove the development of those languages.

Writing your Android code in a more functional way is as easy as using data transformation operations offered with the Kotlin Standard Library. Now that you are familiar with Kotlin syntax and the spirit of functional programming in Kotlin, the next chapter focuses on the Android OS and other programming fundamentals. Android development turned toward Kotlin as an official language back in 2017, so Kotlin has heavily influenced Android's evolution in recent years. It will continue to do so in the coming years.

1 Roman Elizarov; email interview on Kotlin Collections Immutable Library. Oct. 8, 2020.