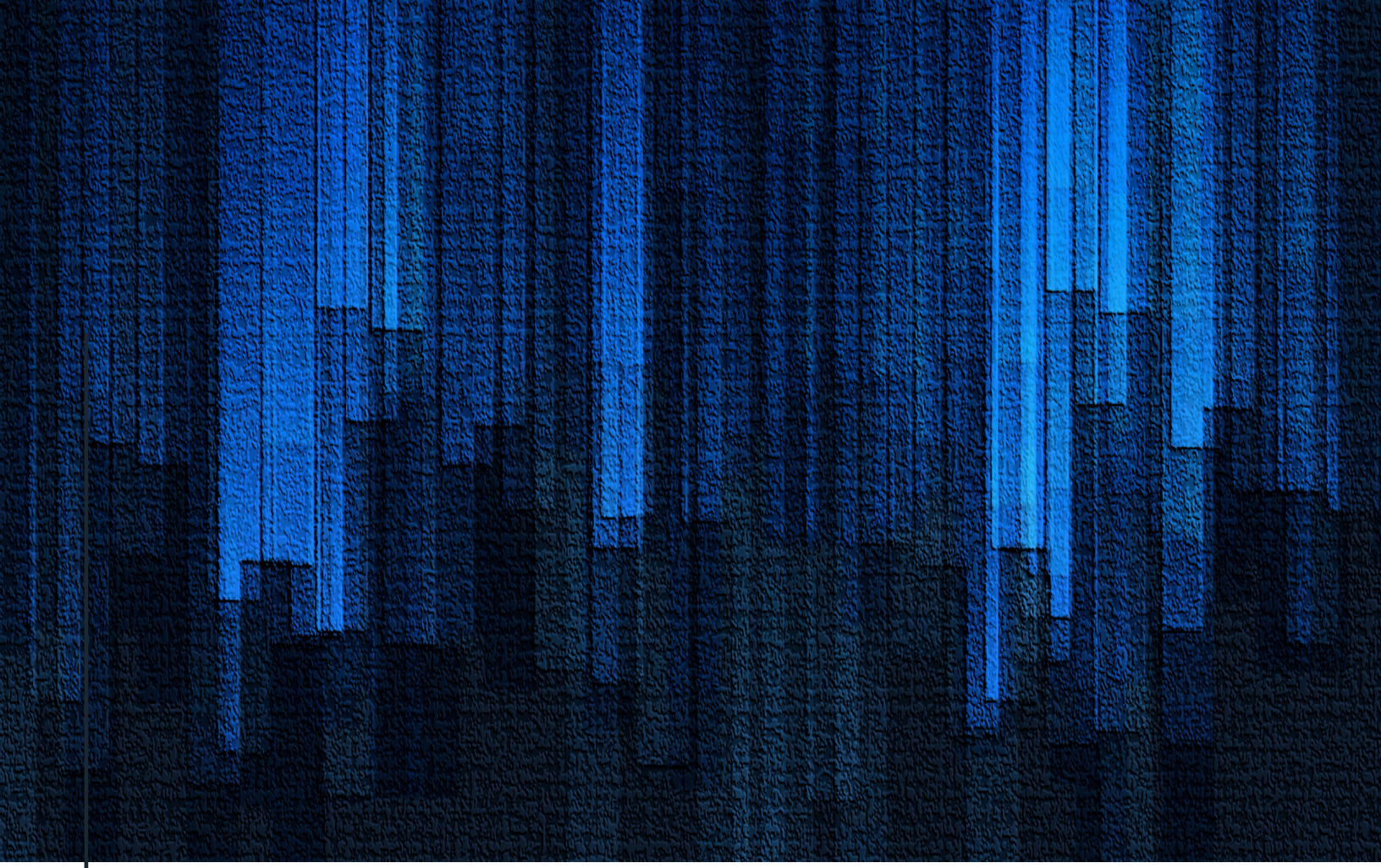

The Managed Runtime Environment

Diving into the JVM with Kotlin

Miguel Gamboa

The Managed Runtime Environment: Diving into the JVM with Kotlin

Miguel Gamboa

This book is available at https://leanpub.com/kotlinonjvm

This version was published on 2025-06-05 ISBN 978-989-33-6322-5

Contents

Preface

This book reflects my journey through software development across academia and industry, as well as my occasional pursuit of an academic degree.

My first encounter with a **managed runtime environment** (MRE) was with the *msvbvm* DLL (*dynamic-link library*) in 1997, during my first job developing software in Visual Basic. This environment, known as the "*Microsoft Visual Basic virtual machine*," provided a runtime for Visual Basic applications. Compared to the JVM, the *msvbvm* lacked some runtime services that are now considered essential in all modern MREs.

Today, these services are standard in runtimes such as .NET, JavaScript engines (e.g., V8), Ruby, Python, and many others. Yet, the JVM was perhaps the first widely accepted runtime to gather all essential features in a single platform, including dynamic loading, metadata, bytecode, verifier, just-in-time compiler, reflection, exception handling, garbage collection, and more.

I recall the significant impact of the JVM on the software development industry and the news when IBM first announced that the AS/400 (a family of midrange computers) would integrate a Java Virtual Machine, enabling it to run Java applications. This marked a major shift in midrange computer software development. Since then, many alternatives to the JVM have appeared, but none have achieved the same level of widespread acceptance. The fact that so many high-level programming languages now compile to JVM bytecode is a testament to its broad adoption.

Years later, I began my teaching career at ISEL (part of the Polytechnic Institute of Lisbon), where I had the opportunity to deepen my knowledge of MREs, particularly in .NET and JVM. From the beginning, I was fascinated by the concept, architecture, and principles governing MREs. Although my professional experience exposed me to these technologies, the pressures of deadlines and revenue goals often prevented me from fully exploring the intricacies of the JVM. Teaching at ISEL gave me the chance to explore the inner mechanics of the JVM, marking the start of my exciting journey into the world of MREs.

Later, I decided to pursue my PhD focusing on MREs. During this period, I explored various methods of interacting with the JVM. Specifically, I made custom modifications to the open-source JVM implementation, Jikes RVM. Subsequently, I shifted my approach to use instrumentation techniques with ASM to achieve a technology-independent solution that could be integrated with any JVM.

After completing my PhD, I contributed to a couple of open-source projects in Java, .NET, and JavaScript, which provided a foundation for starting my own project. I am the author and primary contributor of the HtmlFlow Java library, which has been my platform for exploring emerging trends. Since its launch, HtmlFlow has allowed me to experiment with Java 8 streams, lambdas, and CompletableFuture, as well as *progressive server-side rendering* (PSSR) and reactive data models such as *reactive streams*. More recently, it has been extended to include an idiomatic Kotlin DSL and

to explore alternative web templating asynchronous idioms, including CPS and Kotlin suspending functions.

The knowledge necessary for developing HtmlFlow is covered in this book, including topics such as metadata (Chapter 2), reflection (Chapter 3), performance analysis and microbenchmarking (Chapter 4), bytecode (Chapter 5), metaprogramming (Chapter 6), and sequence pipeline (Chapter 7). These subjects are integral components of the Languages and Managed Runtimes[1] course in the Computer Science and Computer Engineering BSc at ISEL.

Author's Note

All the texts in this book are my original work, **created entirely by me**. However, I have **used AI assistance for proofreading**, correcting typos, and improving grammatical clarity. The content, ideas, and structure remain solely my own.

[1] https://cc.isel.pt/academia/leic-en/lae-en/

Chapter 1: Introduction to the JVM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

1.1: Managed Runtime Environment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

1.2: Java Ecosystem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

1.3: JVM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

1.4: Class Loader, CLASSPATH and Interoperability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

1.5: Central Repository and Dependency Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

1.6: Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

Chapter 2: Type Basics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

2.1: Type Fundamentals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

2.1.1: Primitive Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

2.1.2: Reference Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

2.1.3: Members

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

2.1.3.1: Fields

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

2.1.4: Boxing and Unboxing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

2.1.5: Nested Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

2.2: Abstract and Base Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

2.3: Anonymous classes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

2.4: Kotlin to Java type system

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

2.4.1: Kotlin Class Members

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

2.4.2: Object declarations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

2.4.3: Function types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

2.5: Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

Chapter 3: Reflection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

3.1: Kotlin Reflection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

3.2: Java Reflection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

3.3: NaiveMapper: automatic object to object mapper

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

3.3.1: NaiveMapper Motivation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

3.3.2: NaiveMapper Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

3.4: Annotations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

3.5: Reflection of generic types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

3.6: Reflection Overhead and the Enhancement of NaiveMapper

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

3.6.1: Reflection Overhead

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

3.6.2: Enhancing the NaiveMapper

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

3.7: Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

Chapter 4: Performance testing and Microbenchmarking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

4.1: Performance Measurement Pitfalls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

4.2: JMH

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

Chapter 5: Bytecode and the Just-in-time Compiler

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

5.1: Evaluation Stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

5.2: Constant Pool and Descriptors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

5.3: Managing Instances and Members

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

Chapter 6: Metaprogramming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

6.1: Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

6.2: Class-File API

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

6.3: Dynamic Mapper

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

6.4: Cojen Maker

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

6.4.1: Cojen Maker Features

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

6.4.2: Cojen Maker Examples

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

6.4.3: Dynamic Mapper with Cojen Maker

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

Chapter 7: Sequences, Generators and Suspending functions

Sequences, also known as **streams** in some programming languages, are a key feature for software development. In the Java ecosystem, there are many libraries available as alternatives to the standard Java API, including Vavr, JOOλ, StreamEx, Eclipse Collections, Guava, among others.

In common, these libraries provide the capability of building a **collection pipeline**, as defined in Martin Fowler's renowned article [Fowler, 2015]:

> *Collection pipelines are a programming pattern where you organize some computation as a sequence of operations which compose by taking a collection as output of one operation and feeding it into the next. (Common operations are filter, map, and reduce.)*

DISCLAIMER: In this context, we are **excluding reactive sequences**, such as *reactive streams*, Kotlin Flow, and similar asynchronous APIs. Our focus is solely on synchronous and blocking access to sequences.

Most sequence libraries are constructed solely with programming language features. However, for the purpose of enhancing extensibility, many environments offer the **generators** feature and its **`yield` primitive**, which require some level of support from the managed runtime environment.

This idea was first introduced in the CLU programming language in 1975 and was key to expression evaluation in the Icon programming language in 1977. Its recent popularity may be attributed to its use in C# 2.0 and later in Ruby 1.9. In CLU and C#, **generators** are known as **iterators**, and in Ruby, they are called **enumerators**. Python, PHP, JavaScript, Scala, Dart, and Kotlin provide variants of the yield operator.

In the Kotlin programming language, `yield` is implemented through **suspending functions**. Suspending functions have a broader scope beyond generators, particularly in concurrent programming and providing coroutine support at the language level. Coroutines offer a structured approach to managing concurrent tasks within a specific scope and are a fundamental aspect of **structured concurrency**. There is extensive literature in the field of concurrent programming exploring the intricate details of structured concurrency, and we have chosen to omit this topic from the scope of the book. Instead, we will focus on utilizing only a small fraction of the capabilities of **suspend functions** to demonstrate how they enable the implementation of the `yield` functionality.

7.1: Sequence Alternatives

Sequence pipelines let programmers compose transformations over data, where the result of **each computation serves as the input for the next transformation in the pipeline**. One of the

advantages of this programming idiom is its **readability**, as each operation in the pipeline clearly expresses its specific step and purpose.

For example, given a variable `pastWeather` that refers to a list of `Weather` objects (i.e., `List<Weather>`), each containing daily weather information, we can filter for sunny days, create a sequence of temperatures, and finally select the first five elements. This is achieved through a chain of *filter*, *map*, and *take* operations, resulting in the pipeline shown in Figure 7.1 in Kotlin. In this case, we need to provide certain operations (i.e., *filter* and *map*) with functions that specify what we want to do with each `Weather` object (i.e., `Weather::isSunny` and `Weather::celsius`).

Figure 7.1. Example of sequence pipeline to take 5 temperatures in sunny days.

```
val top5temps = pastWeather
        .filter(Weather::isSunny)
        .map(Weather::celsius)
        .take(5)
```

On the other hand, the example in Figure 7.2 demonstrates an alternative implementation that avoids using auxiliary functions like *filter*, *map*, or *take*. Instead, it leverages the control flow constructs provided by the host language. This imperative approach manually processes the items from the data source, checking their properties (i.e., `isSunny` and `celsius`) to validate and transform the items into a new sequence.

Figure 7.2. Imperative approach to take 5 temperatures in sunny days.

```
val top5temps = mutableListOf<Int>()
for (w in pastWeather) {
    if (w.isSunny) {                // ~ filter
        top5temps.add(w.celsius)    // ~ map
        if (top5temps.size >= 5) { // ~ take
            break
        }
    }
}
```

Ignoring performance or efficiency issues for now and focusing solely on readability, in Figure 7.2, we need to mentally parse the control flow to understand what is happening. In contrast, Figure 7.1 is not only less verbose but also clearly conveys the purpose of each line, with the name of each operation corresponding directly to the action performed on the data.

Despite having similar goals, sequence APIs may vary in different characteristics depending on the technological environment, such as:

- Operation names.

- Composability: *method chaining* versus *nested functions*.
- *Eager* versus *lazy* evaluation.
- Access approach: *pull* versus *push*.
- Extensibility.

7.1.1: Operation names

Although **operation names** like *filter* and *map* are fairly consistent for sequence operations, different environments may use varying terminology. For example, in C#, operations similar to *filter* and *map* are called `Where` and `Select`, respectively, while `flatMap` is known as `SelectMany` in C# and `expand` in Dart. The equivalent example to Figure 7.1 in C# is shown in Figure 7.3. So, maybe in other technological environments, you will need to look up the correct name for the operation you are seeking.

Figure 7.3. Example of sequence pipeline in C#.

```
var top5temps = pastWeather
            .Where(weather => weather.IsSunny)
            .Select(weather => weather.Celsius)
            .Take(5)
```

7.1.2: Composability: *method chaining* versus *nested functions*

The technique of composing a pipeline, as shown in Listings 7.1 and 7.3, is known as **method chaining**. In this approach, the **receiver object** (the object on which the method is called) is implicitly passed as an argument to each method call, allowing subsequent methods to be invoked on the result of the previous method. While this idiom may appear to be the most natural way to chain operations into a pipeline, it might not be intuitive for all developers, particularly those not familiar with object-oriented programming.

For example, Scheme or Clojure developers might prefer using the **nested function** idiom, where functions are combined by making function calls the arguments of higher-level function calls. In this approach, the sequence of nested functions is evaluated from the innermost function outward, meaning that arguments are evaluated before the function is called. Consequently, operations in a pipeline need to be composed in the reverse order of their execution. However, this is typically not an issue for functional programmers, who are accustomed to writing such pipelines in functional programming languages like Clojure, as demonstrated in 7.4.

Figure 7.4. Example of sequence pipeline in Clojure.

```
(take 5
    (map :celsius
        (filter :isSunny pastWeather)))
```

7.1.3: *Eager* versus *lazy* evaluation

Another design difference between the technologies and idioms presented in the previous listings relates to **evaluation time**. The operations used in the approach of Figure 7.3 (e.g., `Where`, `Select`, and `Take`) and in Figure 7.4 (e.g., `map`, `filter`, and `take`) are characterized by **lazy evaluation**. This means they do not process the source elements (e.g., `pastWeather`) immediately when called. Instead, calling these methods merely adds another step to the sequence pipeline. It's important to note that sequences are **immutable**, so each query method returns a new sequence that results from composing the previous sequence with an additional operation.

This contrasts with **collections**, which are **in-memory data structures** that store all values they contain. In the imperative approach shown in Figure 7.2, we are **eagerly** instantiating the resulting list `top5temps`, which holds the outcome of the eager pipeline processing. Although it may not be immediately obvious, the pipeline in Figure 7.1 also processes eagerly. Each Kotlin function in Figure 7.1 generates a new list containing the intermediate elements resulting from each operation, as depicted in Figure 7.5.

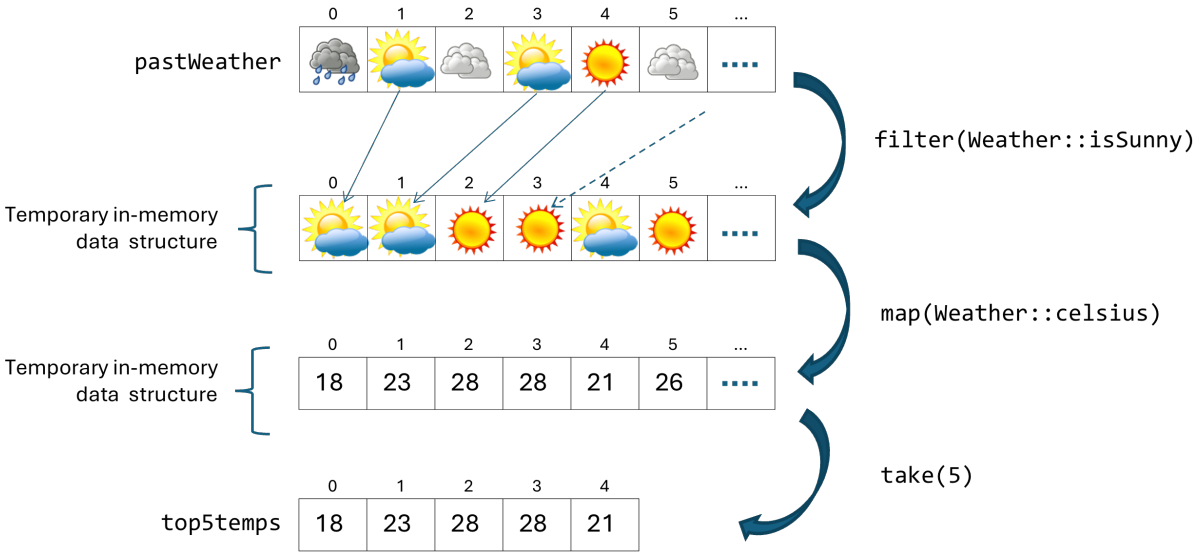

Figure 7.5. Eager evaluation of pastWeather data items.

Examining the implementation of such operations in Kotlin, we find extension functions for the `Iterable<T>` type, similar to those shown in Figure 7.6, albeit with some simplifications. All these functions begin by creating a destination list to store the elements resulting from the computation. In the example of selecting the first 5 temperatures on sunny days, the `filter` and `map` operations generate intermediate results in the pipeline that are eventually discarded and later reclaimed by garbage collection.

Figure 7.6. Sample of Kotlin standard library extensions for Iterable.

```
public inline fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T> {
    val destination = ArrayList<T>()
    for (element in this)
        if (predicate(element))
            destination.add(element)
    return destination
}
public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
    val destination = ArrayList<R>(collectionSizeOrDefault(10))
    for (item in this)
        destination.add(transform(item))
    return destination
}
public inline fun <T> Iterable<T>.take(n: Int): List<T> {
    val destination = ArrayList<T>(n)
    var count = 0
    for (item in this) {
        destination.add(item)
        if (++count == n)
            break
    }
    return destination.optimizeReadOnlyList()
}
```

Note that all operations in Figure 7.6 are extension functions of the `Iterable<T>` interface, which is the base type for all collections in Kotlin and on the JVM. The `Iterable<T>` interface provides the minimal functionality necessary for traversing and accessing elements, which is required for implementing these operations. This design allows the use of these operations with any collection type, as all collections either directly or indirectly inherit from `Iterable<T>`.

In Kotlin, to differentiate between *eager* and *lazy* processing, there is the `Sequence<T>` interface, which has similar behavior to `Iterable<T>`, but with different semantics. The documentation for `Sequence<T>` states:

> The values are evaluated lazily, and the sequence is potentially infinite.

To take advantage of lazy processing in Kotlin, we can convert any collection into a sequence using the utility function `asSequence()`. From this point, all sequence operations will use lazy evaluation. These operations have the same names as the `Iterable` extensions but with different implementations. Instead of creating an auxiliary destination list, these lazy operations return an instance of a new implementation of `Sequence<T>`. When sequence operations are invoked, the only computation performed is the instantiation of a new object that encapsulates the operation's logic. In Figure 7.7, we present an example of such an implementation of the `map` operation, taken from the Kotlin standard library for JVM. This function instantiates the `TransformingSequence` class, which implements the `Sequence` interface by providing an `iterator()`. The actual instantiation of an anonymous `Iterator` happens only when the `iterator()` method is called. This `Iterator` then enables the traversal and transformation of each element from the original sequence.

Figure 7.7. Lazy implementation of map in Kotlin.

```
public fun <T, R> Sequence<T>.map(transform: (T) -> R): Sequence<R> {
    return TransformingSequence(this, transform)
}
internal class TransformingSequence<T, R>
constructor(
    private val sequence: Sequence<T>,
    private val transformer: (T) -> R)
: Sequence<R> {
    override fun iterator(): Iterator<R> = object : Iterator<R> {
        val iterator = sequence.iterator()

        override fun next() = transformer(iterator.next())

        override fun hasNext() = iterator.hasNext()
    }
}
```

If we interleave `asSequence()` between `pastWeather` and the first operation of the pipeline, i.e., `filter`, we obtain a lazy pipeline where nothing is produced until a *terminal operation* is chained. In contrast to *intermediate operations* like `filter`, `map`, or `take`, which produce a new sequence, a *terminal operation* either returns void (i.e. Kotlin `unit`) or something different from a sequence, preventing further chaining of operations in the pipeline. In this case, the elements of the resulting sequence (i.e., 18, 23, 28, 28, and 21) are computed on demand lazily, only when the sequence is iterated, for example, by a `forEach` operation, as depicted in Figure 7.8.

Figure 7.8. Example of lazy pipeline to take 5 temperatures in sunny days.

```
pastWeather
    .asSequence()
    .filter(Weather::isSunny)
    .map(Weather::celsius)
    .take(5)
    .forEach { print("$it, ") }
```

The processing resulting from the *lazy evaluation* of a sequence pipeline is called **vertical processing** in some contexts, in contrast to **horizontal processing** performed during *eager evaluation*. When a pipeline is processed *eagerly*, **all the items of a sequence are traversed by each operation**. Conversely, with *lazy* evaluation, **each item traverses all operations of the pipeline**. If data items are laid out horizontally within a sequence, as depicted in Figure 7.5, then processing all items *eagerly* means that each operation traverses all items horizontally, hence the term **horizontal processing**. On the other hand, when the operations are arranged vertically, as shown in Figure 7.9, and each item in a *lazy* evaluation traverses all operations, we refer to it as **vertical processing**.

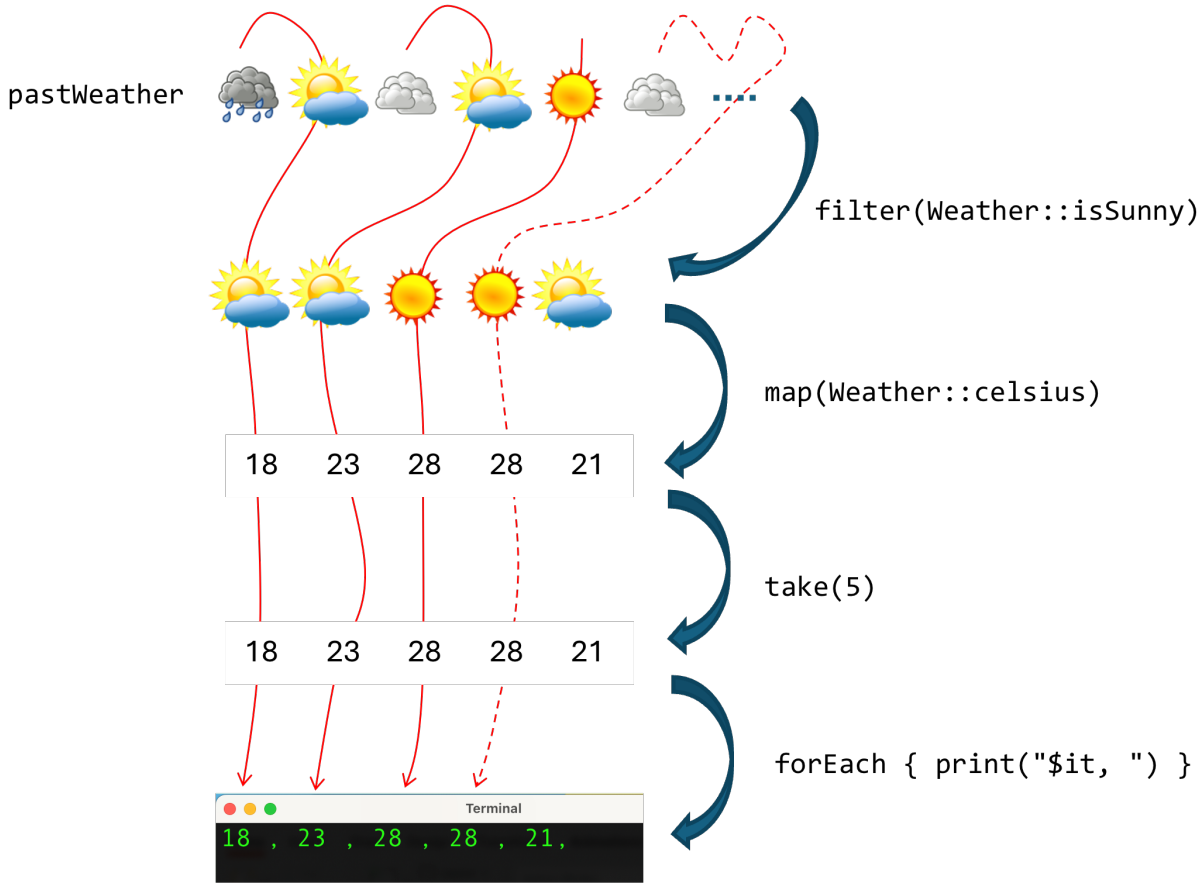

Figure 7.9. Lazy evaluation of pastWeather data items.

There is no universal rule that dictates lazy evaluation is always more efficient than an eager approach, or vice versa. However, for certain use cases, such as the example in this section, the lazy version may perform fewer data accesses and computations than the eager version. Consider `pastWeather` has 31 items, representing the 31 days of May, with 9 sunny days. In the eager approach, the `filter` operation calls the predicate 31 times, and the `map` operation performs 9 transformations, totaling 40 data accesses. If the 5th sunny day occurs at the 12th position in `pastWeather` (index 11), a lazy evaluation would perform only 12 data accesses for `filter` and 5 transformations for `map`, as only the 5 items requested by the `take` operation are processed, totaling 17 data accesses.

In addition, we must also consider the overhead and waste of creating intermediate collections through eager evaluation that are discarded at the end of the pipeline processing. This issue was highlighted by Roman Elizarov, the former project lead for Kotlin, who proposed an "*inspection to convert non-lazy chains of collection functions into sequences*[1]". He claimed that:

> Code like this will conceptually work faster and produce less garbage (thus improving overall performance) when replaced with a lazy version using `.asSequence()`.

[1] https://youtrack.jetbrains.com/issue/KTIJ-6827/Inspection-to-convert-non-lazy-chains-of-collection-functions-into-sequences

In the JVM standard library, Kotlin `Sequence` is not the only option for lazy evaluation of sequences. Before Kotlin, the Java standard library already provided a sequences API in the package `java.util.stream` through its core interface `Stream`. Therefore, it is possible to get a lazy stream over a collection and perform a pipeline, as depicted in Figure 7.10, to get the first 5 temperatures on sunny days.

Figure 7.10. Example of Java stream pipeline to take 5 temperatures in sunny days.

```
var count = 0
pastWeather
    .stream()
    .filter { count++; it.isSunny }
    .map { count++; it.celsius }
    .limit(5)
    .forEach { print("$it, ") }
assertEquals(17, count)
```

For the example where the 5th sunny day occurs at the 12th position in the `pastWeather` stream, the count of filtering and transformations in the streams pipeline will match the count observed for the Kotlin sequences version. This can be illustrated by using a count variable incremented within each lambda, as shown in Figure 7.10.

7.1.4: Extensibility

Implementing new custom operations for a sequences API requires adherence to the interface of its internal *traversal* mechanism. Each sequence technology has an interface that specifies how elements are **traversed** and **accessed**. Despite slight differences, many iterator protocols, such as `Iterator` in JVM, `Enumerator` in C#, or `Iterator` in Dart, provide at least one method to *advance* to the next element and one property to *access* the current element. In Java's standard library `Iterator`:

- The `next()` method serves both roles of advancing and accessing the next element.
- The `hasNext()` method indicates whether there are more items to iterate over.

`Enumerator` in C# has a similar API to `Iterator` in Dart, providing:

- A `moveNext()` method to advance, which returns `true` or `false` depending on whether it moves successfully or not.
- A `current` property that returns the current element.

With the exception of Java, many programming languages such as Kotlin, C#, Dart, and others, offer extension methods that allow developers to add new functionalities to existing interfaces like `Iterator`. We will demonstrate how to implement a new `interleave` method that intersperses

elements from two sequences. By chaining this `interleave` method in the pipeline, we can add a label like `($position)` next to each temperature, producing results such as `18 (1), 23 (2), 28 (3), 28 (4), 21 (5)`. Although similar outcomes can be achieved through various methods and operations, our focus here is to illustrate the use of a new custom `interleave` method, as shown in Figure 7.11.

Figure 7.11. Example of Kotlin sequence pipeline with custom interleave.

```
pastWeather
        .asSequence()
        .filter(Weather::isSunny)
        .map(Weather::celsius)
        .interleave(generateSequence(1) { it + 1 }.map { " ($it) " })
        .take(10)
        .forEach { print(it) }
```

In the pipeline of Figure 7.11, the `interleave` function is used to combine two sequences: one derived from `pastWeather` and another generated on-the-fly. It interleaves the sequence of temperatures with a sequence of positional labels generated by `generateSequence`, where each label is formatted as `" ($it) "`. The `interleave` function alternates elements from both sequences, pairing each temperature with its corresponding positional label.

The `interleave` function for Kotlin `Sequence` can be created by providing a custom implementation of the `Iterator` interface, as illustrated in Figure 7.12. Later in Section 7.2, we present a more compact implementation using generators.

Figure 7.12. Implementation of Kotlin extension function interleave.

```
fun <T> Sequence<T>.interleave(other: Sequence<T>) : Sequence<T> {
    return InterleavingSequence(this, other)
}
class InterleavingSequence<T>(self: Sequence<T>, other: Sequence<T>) : Sequence<T> {
    var selfIter = self.iterator()
    var otherIter = other.iterator()
    override fun iterator(): Iterator<T> {
        return object : Iterator<T> {
            override fun hasNext() = selfIter.hasNext()

            override fun next() = selfIter
                .next()
                .also { // Swap iterators so we pull from the other one next time.
                    selfIter = otherIter.also {  otherIter = selfIter }
                }
        }
```

```
17      }
18  }
```

The `InterleavingSequence` class of Figure 7.12 implements the `Sequence` interface and manages two iterators, `selfIter` and `otherIter`, for the respective sequences. The `iterator()` method of `InterleavingSequence` returns a custom iterator that alternates between the two sequences. It checks if there are more elements in the current iterator. After yielding an element, the `next()` method swaps the iterators to pull the next element from the other sequence, thus achieving the interleaving effect.

7.1.5: Access approach: *pull* versus *push*

The iteration protocol described in the previous section follows a **pull-based** approach, where elements are accessed from the sequence and then processed. Developers can check if the sequence has more elements by calling the `hasNext` method, and they can obtain the next element using the `next` method. Once an element is accessed, it can be processed according to the developer's code. The `Enumerator` in C# and the `Iterator` in Dart both use a similar *pull-based* approach.

Java streams use a different access approach known as **push-based**. Instead of pulling items from the sequence (i.e., requesting elements), the developer specifies what to do with the items (i.e., providing instructions). In a push-based approach, the equivalent method to `moveNext()`, such as `tryAdvance()`, takes a function that defines what to do with the next element, rather than returning the element itself. The `tryAdvance()` method returns a boolean: `false` if there are no remaining elements, and `true` if there are more elements to process.

For instance, if `pastWeather` is a Java `Stream`, we can iterate over its elements using a *push-based* approach, as demonstrated in Figure 7.13. In this example, `tryAdvance` takes a lambda that prints the given item. The `while` block remains empty as there is no additional logic required within it.

Figure 7.13. Traversing items through a Spliterator using a push-based approach.

```
1  val iter = pastWeather.spliterator()
2  while (iter.tryAdvance { w ->
3      println(w)
4  }) {
5      /* while block */
6  }
```

The `tryAdvance` method is part of the `Spliterator` interface in Java streams. Although `Spliterator` provides additional functionality for parallel processing, we are focusing only on sequential processing here. With that in mind, we can implement a custom interleave operation for Java streams through the explicit implementation of the `AbstractSpliterator` interface, as shown in Figure 7.14.

Figure 7.14. Implementation of Kotlin extension function interleave for Java streams.

```kotlin
fun <T> Stream<T>.interleave(other: Stream<T>) : Stream<T> {
    val res = object : AbstractSpliterator<T>(Long.MAX_VALUE, ORDERED) {
        var selfIter = this@interleave.spliterator()
        var otherIter = other.spliterator()

        override fun tryAdvance(cons: Consumer<in T>): Boolean {
            return selfIter
                .tryAdvance(cons)
                .also { // Swap iterators so we pull from the other one next time.
                    selfIter = otherIter.also {  otherIter = selfIter }
                }
        }
    }
    return StreamSupport.stream(res, false)
}
```

The logic behind the implementation in Figure 7.14 is quite similar to the Figure 7.12. The spliterator alternates between two internal spliterators, `selfIter` and `otherIter`, to push elements from the respective streams to given consumer `cons`. It does so by creating an anonymous `AbstractSpliterator` with an effectively unlimited size (`Long.MAX_VALUE`) and ordered characteristics. The `tryAdvance` method attempts to advance the current iterator and, after yielding an element, swaps the iterators to alternate the sequence of elements. Finally, the resulting interleaved elements are provided through the `StreamSupport.stream` method, returning a new stream that reflects the interleaving behavior.

Note that a similar implementation for Java streams can be achieved by implementing a **pull-based** iterator such as `Iterator<T>`. However, for Java streams extensibility, it is recommended to extend `Spliterator` rather than `Iterator`. As Brian Goetz (Architect for the Java Language at Oracle Corporation) stated in response to the Stack Overflow question Iterator versus Stream of Java 8[2]:

> ***Spliterator has fundamentally lower per-element access costs than Iterator, even sequentially.***

7.2: Generators and yield pattern

Simply put, a **generator is like a function that generates a sequence of values**. However, instead of building the entire sequence at once (like an array or list), a **generator yields values one at a time**, returning a "new" value each time it is iterated over or called.

A **generator** refers to a computation that:

[2] https://stackoverflow.com/a/31212695/1140754

1. **yields values** to the caller.
2. **is resumed** after the yielded value has been consumed by the caller.

When a generator yields, it pauses its execution and does not need to be resumed by its caller, even though it may have pending computations to perform.

In summary a generator acts like a subroutine encompassing a special computation, that is restricted to communicate with its caller through the **yield** primitive. During its computation, a generator can yield out many values.

This characteristic makes generators useful for simplifying the implementation of lazy iterators. As we have seen, extending sequences with new custom operations in a lazy manner is not as straightforward as implementing them in an eager way. The verbosity of the lazy map operation implementation in Figure 7.7 is evident when compared to the eager counterpart in Figure 7.6, which only required four lines of code. This is especially true in object-oriented languages where following the iterator design pattern substantially increases the complexity and verbosity of those implementations, often requiring the implementation of two interfaces: `Iterable`/`Sequence` and `Iterator`.

With generators, programming languages provide a way to implement operations in a concise manner similar to eager approaches, but without incurring their drawbacks and maintaining a lazy behavior. The compiler then translates that implementation into a lazy form by implementing the required traversal interfaces.

In Kotlin, generators and the `yield` primitive are implemented through a lambda with a receiver of type `SequenceScope`. To avoid delving into these internal details immediately, we will first introduce the concept of generators using JavaScript. As a widely understood language, JavaScript uses native keywords for generators, making it easier to explain the concept without getting into the complexities behind the scenes.

In JavaScript, a generator function is defined with `function*` rather than just `function`. The asterisk signifies that the function can yield multiple values, as opposed to returning a single value. Within the generator, using `return` terminates the sequence, while `yield` produces an item in the sequence.

In Figure 7.15, the generator `foo` creates an iterator representing a sequence of three items: 19, 7, and 11. Note that the last yielded item, 5, is not included in the sequence because it occurs after the generator body has completed and returned.

Figure 7.15. JavaScript generator foo yielding numbers.

```
function* foo() {
  yield 19
  yield 7
  yield 11
  return
  yield 5
}
```

In Figure 7.16, we illustrate an example using the generator `foo`. When `foo()` is called, none of its body's statements are executed immediately. Instead, it simply creates an instance of an iterator that encapsulates those statements. Execution of the statements begins only when we start advancing through the iterator by calling `next()`, at which point the generator's body will execute until it reaches a `yield` instruction. On subsequent call to `next()`, the generator resumes execution from the point where it was last suspended. In JavaScript, the `next()` method returns an object with two properties: `value`, which holds the current item, and `done`, which is `true` if the iterator has reached the end of the sequence.

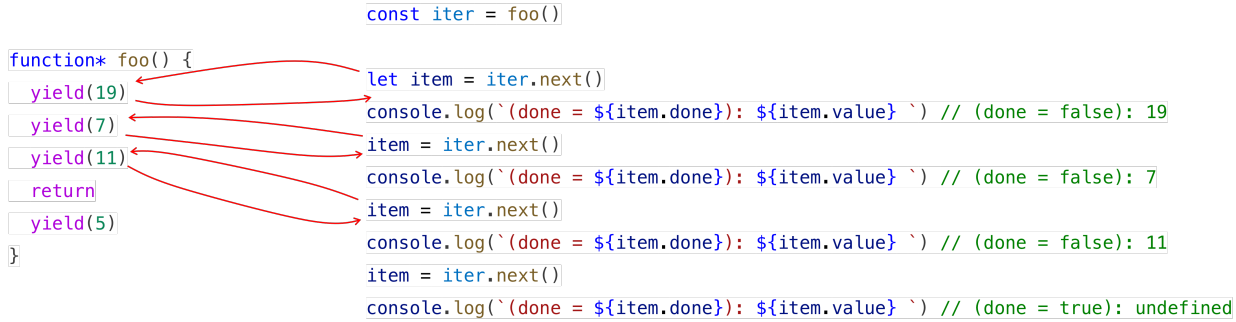

Figure 7.16. Example of Javascript generator.

In Kotlin, a generator is created using the `sequence` function, which takes a lambda defining the generator's body. The Kotlin version of the `foo` generator of Figure 7.17 returns a `Sequence` produced by calling `sequence`.

Figure 7.17. Kotlin generator foo yielding numbers.

```
fun foo() : Sequence<Int> {
    return sequence {
        yield(19)
        yield(7)
        yield(11)
        return@sequence
        yield(5)
    }
}
```

The `sequence` function receives a lambda where items are emitted using the `yield` function. Unlike JavaScript, `yield` in Kotlin is not a keyword but a function. It is a method of the `SequenceScope` instance, which is passed as the receiver of the lambda argument to `sequence`, as depicted in its signature:

```
fun <T> sequence(block: suspend SequenceScope<T>.() -> Unit): Sequence<T>
```

Just like in JavaScript, the behavior of `yield` is similar in Kotlin. When we call `foo`, it only creates an instance of a `Sequence` that encapsulates the function provided in the lambda; none of its instructions are executed at that time. Instead, execution starts only when we advance through the iterator by calling the `next()` method. The function runs until it reaches a `yield` statement, which pauses execution and returns a value. This process alternates between calls to `next()` from the caller and `yield` from the generator. In the following sections, we will explain how `yield` in Kotlin suspends execution and how it resumes.

Thus, the generator and `yield` functionality simplify the implementation of lazy iterators by abstracting away the details of interface implementations. This approach consolidates the iterator's logic into a single method, rather than requiring separate methods like `next()` and `hasNext()`. In Figure 7.18, we present the equivalent lazy interleave implementation using Kotlin's `sequence` utility.

Figure 7.18. Kotlin interleave implementation with sequence and yield.

```
fun <T> Sequence<T>.interleave(other: Sequence<T>) : Sequence<T> {
    return sequence {
        var selfIter = this@interleave.iterator()
        var otherIter = other.iterator()
        while(selfIter.hasNext()) {
            yield(selfIter.next())
            selfIter = otherIter
                .also {  otherIter = selfIter }
        }
    }
}
```

7.3: Suspend functions

A suspend function differs from a "normal" function (i.e., non-suspend function) by potentially having zero or more **suspension points**. Whenever a suspend function calls another suspend function, it creates a **suspension point** — statements in its body that may **pause** the function's execution to be **resumed** at a later moment in time.

In the JVM, a **suspend function** is translated by the Kotlin compiler to a method with an **additional parameter of type `Continuation<T>`**. This continuation parameter serves two purposes:

1. It receives the result of the suspend function being invoked, conforming to the **continuation-passing style**.
2. It manages the suspension implementation, effectively **transforming the suspension point into a continuation that can later resume the execution flow**.

CPS, or Continuation-Passing Style, contrasts with the **direct style**, where the **result of a function aligns with its returned value**. Conversely, in **CPS**, the **result is handed off to a continuation function**.

CPS initially gained prominence within the Scheme programming language community. This technique later found widespread adoption in JavaScript due to its suitability for handling asynchronous operations in the language's event-driven, single-threaded environment.

CPS may manifest in various forms depending on the implementation technology. In JavaScript, a continuation may often appear as a callback function, typically defined with the signature `(error, result) => void`, where `error`represents an error object if an error occurred during the operation. It is usually `null` or `undefined` if no error occurred. On the other hand, `result` contains the outcome or result of the asynchronous operation.

In Kotlin, the continuation is represented by the following interface:

```
1  interface Continuation<in T>
2    abstract val context: CoroutineContext
3    abstract fun resumeWith(result: Result<T>)
```

Ignoring the `context` property for the moment, the `resumeWith` function is equivalent to a callback, handling the return value of the invoked suspending function, whether it completes successfully or with an error. Instead of receiving two arguments, the `resumeWith` function has a single parameter of type `Result<T>`, which encapsulates either a successful outcome with a value of type `T` or a failure with a `Throwable` exception.

It is possible to explicitly cast a suspend function to its normal version counterpart, which includes an additional parameter `Continuation`. For example, consider the following suspending function with the signature: `suspend fun fetchSuspend(url: String): String` that performs an HTTP request for the given URL and returns the response body as a String. We call it `fetchSuspend` to distinguish it from a different `fetch` function that we will use later in this chapter. We can get a reference to that function of type `(String, Continuation<String>) -> Any` using the following statement:

```
1  val fetchHandle = ::fetchSuspend as (String, Continuation<String>) -> Any
```

Note that `Continuation` is parametrized with `String`, which is the type of the result of the `suspendFetch` function. The `Any` returned by the `fetchHandle` does not represent the result of the function but rather the execution state of that function. We will revisit this topic later. Given the `fetchHandle`, we can invoke it using the continuation-passing style. In the example of Figure 7.19, we perform a fetch to `https://github.com` and print the response body.

Figure 7.19. Calling a suspend function using continuation-passing style.

```
val fetchHandle = ::fetchSuspend as (String, Continuation<String>) -> Any
fetchHandle("https://github.com", object : Continuation<String> {
    override val context = EmptyCoroutineContext
    override fun resumeWith(result: Result<String>) {
        println(result.getOrThrow())
    }
})
```

The coroutine context encompasses various elements, such as the coroutine's job and its dispatcher, which determine the thread or threads used for execution by the corresponding coroutine. Given that we are using these functions synchronously and not engaging in concurrent programming, we leave those details out of the scope of this book. Therefore, we initialize the context with an `EmptyCoroutineContext`.

The code snippet in Figure 7.19 is equivalent to a portion of the work performed by the Kotlin compiler when we invoke a suspend function like `fetchSuspend`. This code could be the result of the translation of a suspending function call, such as the snippet of Figure 7.20:

Figure 7.20. Calling a suspend function.

```
val result = fetchSuspend("https://github.com")
println(result)
```

When we call `fetch("https://github.com")`, the Kotlin compiler transforms it into the continuation-passing style similar to what we've shown with Figure 7.19. The code snippet of Figure 7.20 presents a direct style despite being translated to a continuation-passing style. Some IDEs, such as IntelliJ, mark suspension points with a distinctive icon, indicating that invocations may be paused and resumed later.

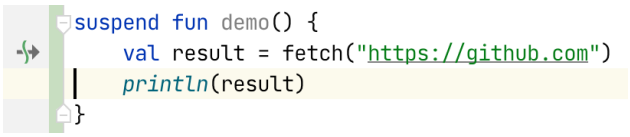

Figure 7.21. IntelliJ highlighting suspension points.

We can also make the reverse transformation by casting a normal function that uses the continuation-passing style to a suspending function. For instance, consider the following definition of `fetchCps`:

```
fun fetchCps(url: String, onCompletion: Continuation<String>) { ... }
```

In this case, `fetchCps` is a function that takes a URL and a continuation as parameters, and later invokes that continuation with the response body of the HTTP request. If we cast this function to its suspending counterpart, we can then invoke it as a suspend function, such as the snippet of Figure 7.22.

Figure 7.22. Calling CPS function as a suspend function

```
val fetchCpsHandle = ::fetchCps as (suspend (String) -> String)
val result = fetchCpsHandle("https://github.com")
println(result)
```

Since `fetchCpsHandle` is a suspend function, the call `fetchCpsHandle("https://github.com")` corresponds to a suspension point. Additionally, **note that this code is only valid within a suspend function, as only suspend functions may call other suspend functions**.

7.4: Deconstructing yield and generators

Generators and **yield** in Kotlin are implemented by means of **suspend functions**. The semantics of **suspending functions** and **suspension points** align closely with the concepts of **generators** and the **yield primitive**. In this analogy, a **generator** is akin to a suspending function, and a **call to yield** creates a **suspension point that can be resumed later**.

7.4.1: Translating a Suspend Function into a State Machine

The **Kotlin compiler translates a suspending function into a state machine**, where **each state represents a continuation**—i.e., the code to be executed after a suspension point.

To illustrate this translation process, we will manually implement a function with suspension points supported by the explicit implementation of a `Continuation<T>`. In our use case, we will implement a `fetchMany` function that receives a list of URLs and returns a list of strings (`List<String>`) containing the HTTP response body of each request for the given URLs.

We will incrementally evolve this example until we reach the final version, where we will fully utilize the existing suspend function mechanism. This development proceeds through the following phases:

1. `fetchMany`
2. `fetchManyCps`
3. `fetchManyCpsLazy`
4. `fetchManySuspendingLazy`

In the first phase, we implement `fetchMany` as a 'normal' (i.e., non-suspend) function, which will be transformed in the second phase into `fetchManyCps` to adhere to continuation-passing style. Additionally, `fetchManyCps` breaks the original algorithm into states, with each state fetching one of the given URLs. In the third phase, instead of returning a `List<String>`, the `fetchManyCpsLazy` function returns a `Sequence<String>`, which fetches each URL lazily on demand only when we advance the resulting iterator (i.e., call `next()`). This implementation might not fetch all provided URLs if the resulting iterator is not traversed to the end. Finally, the `fetchManySuspendingLazy` function implements the same behavior as `fetchManyCpsLazy` but with the support of suspending functions.

At some points in the intermediate stages, the code may seem more complex than necessary. However, it's important to keep in mind that we're essentially simulating the behavior of the compiler-generated code. Ultimately, in the final version, this complexity will be abstracted away by the use of suspending functions.

While this may seem limiting, instead of receiving a `vararg` parameter or any other form of list of strings, we will restrict the arguments to three URLs: `url1`, `url2`, and `url3`. This approach allows us to more easily illustrate the state machine and reason about each state in the generated bytecode.

---

1. fetchMany

The `fetchMany` function utilizes an auxiliary `fetch` function responsible for performing the HTTP request and returning the response body. This `fetch`is a normal function (i.e. non-suspending). To observe the eager versus lazy behavior of each implementation, we include a print message in the `fetch` function. In Figure 7.23, we provide the implementations of both the `fetchMany` and `fetch` functions.

Figure 7.23. fetchMany implementation in a "normal" function.

```
fun fetch(path: String): String {
    println("Fetching $path")
    return URI(path).toURL().readText()
}
fun fetchMany(url1: String, url2: String, url3: String): List<String> {
    val bodies = mutableListOf<String>()
    bodies.add(fetch(url1))
    bodies.add(fetch(url2))
    bodies.add(fetch(url3))
    return bodies
}
```

The `fetchMany` function begins by initializing an empty mutable list called `bodies` to store the response bodies. It then calls the `fetch` function for each URL (`url1`, `url2`, and `url3`) and adds the

returned response body to the `bodies` list. Finally, it returns the `bodies` list containing the response bodies fetched from all three URLs.

---

2. fetchManyCps

At this stage, we will adapt the previous `fetch` and `fetchMany` functions to the continuation-passing style in two new implementations, named `fetchCps` and `fetchManyCps`. This new implementation with CPS behaves in the same manner as the former direct style and offers no advantages beyond increased complexity. Nevertheless, it serves as a necessary step to comprehend the suspension points generated by the invocation of suspending functions, such as `yield`.

First, we will introduce a new function `fetchCps` (Figure 7.24) adhering to the continuation-passing style. In comparison to the previous `fetch` function, `fetchCps` receives an additional parameter of type `Continuation<String>`. This continuation is responsible for handling the result of the `fetchCps` function. It will either resume with the response body upon successful completion of the HTTP request or with an exception in case of failure.

Figure 7.24. fetchCps adhering to the continuation-passing style.

```
fun fetchCps(path: String, onComplete: Continuation<String>) {
    println("Fetching $path")
    try {
        val body = URI(path).toURL().readText()
        onComplete.resume(body)
    } catch(err: Throwable) {
        onComplete.resumeWithException(err)
    }
}
```

Now, we will break down the `fetchMany` algorithm into a state machine within the implementation of a `Continuation<String?>`. In this setup, the `resumeWith` function will sequentially handle the results of the calls to `fetchCps`, following the definition outlined in Section 7.4.1.

```
override fun resumeWith(result: Result<String?>) {
   result.getOrThrow()?.let { bodies.add(it) }
   when(state++) {
      0 -> fetchCps(url1, this)
      1 -> fetchCps(url2, this)
      2 -> fetchCps(url3, this)
      3 -> onComplete.resume(bodies)
   }
}
```

In this implementation, the list `bodies`, the `state`, and the `onComplete` are properties within the class `FetchManyCps`, which implements the `Continuation` interface. On each `fetchCps` call, we reuse the same `Continuation` object (`this`), ultimately resuming to the `resumeWith` function, but in a different state. The `fetchManyCps` function instantiates the `FetchManyCps` class and initiates the flow by invoking the `resumeWith` method with `null`, as depicted in Figure 7.25.

Figure 7.25. fetchManyCps function starting the execution flow.

```
fun fetchManyCps(
   url1: String,
   url2: String,
   url3: String,
   onComplete: Continuation<List<String>>
) {
    FetchManyCps(url1, url2, url3, onComplete)
        .resume(null)
}
```

Beyond the implementation of `resumeWith` and the previously mentioned properties, the `FetchManyCps` class also overrides the `context` property of the `Continuation` interface, in accordance with the definition provided in Figure 7.26.

Figure 7.26. FetchManyCps class

```
class FetchManyCps(
    private val url1: String,
    private val url2: String,
    private val url3: String,
    private val onComplete: Continuation<List<String>>
)
    : Continuation<String?>
{
    private val bodies = mutableListOf<String>()
    var state = 0
```

```
11
12      override val context = EmptyCoroutineContext
13
14      override fun resumeWith(result: Result<String?>) {...}
15  }
```

The implementation of `fetchManyCps` follows CPS but lacks the ability to be paused and resumed. When `fetchManyCps` is invoked, all HTTP requests are executed for the given URLs.

For example, when making the call depicted in the snippet from Figure 7.27, it promptly prints the following output to the terminal:

```
1  Fetching https://stackoverflow.com/
2  Fetching https://github.com/
3  Fetching https://developer.mozilla.org/
```

Figure 7.27. Calling fetchManyCps with 3 urls

```
1  val res = fetchManyCps(
2      "https://stackoverflow.com/",
3      "https://github.com/",
4      "https://developer.mozilla.org/")
```

---

3. fetchManyCpsLazy

To make it lazy, we will modify the function to return a `Sequence<String>` instead of a list. This sequence will progress through the state machine only when the `next()` function is invoked.

To achieve this, the `FetchManyCpsLazy` class will implement the `Iterator` interface. It will deliver each item whenever the `next()` function is called, rather than collecting all items into a mutable list `bodies` that is later delivered to the `onComplete` continuation. Therefore, in the `FetchManyCpsLazy` class, the `bodies` and `onComplete` properties are replaced with the `nextItem` property. This property will hold the next element delivered by the `next()` method. In the code provided in Figure 7.28, only the new parts that differ from the former implementation of the `FetchManyCps` class are included.

Figure 7.28. FetchManyCpsLazy implementing Continuation and Iterator.

```
class FetchManyCpsLazy(...) : Continuation<String?>, Iterator<String> {
    var state = 0
    var nextItem: String? = null
    ...
    override fun resumeWith(result: Result<String?>) {
        nextItem = result.getOrThrow()
    }
    private fun block() {
        when(state++) {
            0 -> fetchCps(url1, this)
            1 -> fetchCps(url2, this)
            2 -> fetchCps(url3, this)
            else -> throw NoSuchElementException()
        }
    }

    override fun hasNext() = state <= 2

    override fun next(): String {
        block()
        return nextItem as String
    }
}
```

It's worth noting that the original state machine, initially presented in Section 7.4.1, is now encapsulated within the `block` function. Additionally, the `resumeWith` implementation of `Continuation` solely stores the next item. In essence, the `next()` function progresses through the state machine by invoking the `block()` function, which may pause during a `fetchCps` call and resume during the execution of the `resumeWith` function.

The new implementation of `fetchManyCpsLazy` simply needs to construct a sequence that returns a new instance of `FetchManyCpsLazy` when the iterator is called, as demonstrated in the following listing:

```
fun fetchManyCpsLazy(url1: String, url2: String, url3: String)
  : Sequence<String> {
    return object : Sequence<String> {
        override fun iterator() = FetchManyCpsLazy(url1, url2, url3)
    }
}
```

Now, invoking the `fetchManyCpsLazy` function will not trigger any actions until we iterate over the resulting sequence. Running the snippet of Figure 7.29 will only print the messages `"Fetching"` after the `"Call to fetchManyCpsLazy finished!"` message appears in the terminal, as shown in the following output:

```
Call to fetchManyCpsLazy finished!
Fetching https://developer.mozilla.org/
Fetching https://stackoverflow.com/
```

Figure 7.29. Getting only 2 items of the Iterator from the fetchManyCpsLazy.

```
val seq = fetchManyCpsLazy(
    "https://stackoverflow.com/",
    "https://github.com/",
    "https://developer.mozilla.org/")
val iter = seq.iterator()
println("Call to fetchManyCpsLazy finished!")
iter.next()
iter.next()
```

Additionally, we are not obligated to fetch all given URLs unless we traverse the iterator to the end. In this scenario, if we do not consume the last item, **we leave the iterator in a suspended state indefinitely**.

---

4. fetchManySuspendLazy

At this stage, we are approaching the final implementation of the `fetchManySuspend` function, defined as follows:

```
fun fetchManySuspend(url1: String, url2: String, url3: String) : Sequence<String> {
    return sequence {
        val body1 = fetch(url1)
        yield(body1)
        val body2 = fetch(url2)
        yield(body2)
        val body3 = fetch(url3)
        yield(body3)
    }
}
```

The lambda passed to the `sequence` builder function will be translated by the Kotlin compiler into an anonymous function that implements a state machine similar to the one presented in Figure 7.28 and can be denoted as:

```
private fun block() {
    when (state++) {
        0 -> fetchCps(url1, this)
        1 -> fetchCps(url2, this)
        2 -> fetchCps(url3, this)
        else -> throw NoSuchElementException()
    }
}
```

Thus, we will leverage Kotlin's native support for suspending functions and suspension points to automatically generate an equivalent state machine. In the new version of the `FetchManySuspendLazy` class, the `block` function will be a suspending function defined as depicted in Figure 7.30:

Figure 7.30. block function in FetchManySuspendLazy class.

```
private suspend fun block() {
    val body1 = fetch(url1)
    yield(body1)
    val body2 = fetch(url2)
    yield(body2)
    val body3 = fetch(url3)
    yield(body3)
}
```

Note that `yield` is a suspending function too, and each call creates a suspension point. Additionally, we replaced the usage of `fetchCps` with a non-suspending call to the normal `fetch` function. Otherwise, we would have three more suspension points compared to the former state machine example.

In Figure 7.31, we present the bytecode resulting from the compilation of the `block` function described in Figure 7.30. Here, local variable 2 will store the result of the invocation of `fetch()`. Local variable 6 refers to an instance of the anonymous inner class `FetchManySuspendLazy$block$1`, which implements `Continuation`. This object serves as the continuation for the calls to the suspending function `yield`. Additionally, `FetchManySuspendLazy$block$1` contains an integer field `label`, equivalent to our `state` property of Figure 7.26, which holds the current state in the state machine. It has also a field `this$0` that is a reference to an instance of the outer class `FetchManySuspendLazy`.

Figure 7.31. Bytecode resulting from compilation of suspending block function.

```
private final block(Lkotlin/coroutines/Continuation;)Ljava/lang/Object;
...
    /*
     * loc2: String
     * loc6: FetchManySuspendLazy$block$1
     */
    ALOAD 6
    GETFIELD pt/isel/FetchManySuspendLazy$block$1.label : I
    TABLESWITCH
      0: L4
      1: L5
      2: L6
      3: L7
      default: L8
  L4:
    /*
     * <=> loc2 = FetchKt.fetch(this.url1)
     */
    ALOAD 0
    GETFIELD pt/isel/FetchManySuspendLazy.url1 : Ljava/lang/String;
    INVOKESTATIC pt/isel/FetchKt.fetch (Ljava/lang/String;)Ljava/lang/String;
    ASTORE 2
    /*
     * <=> loc6.label = 1
     */
    ALOAD 6
    ICONST_1
    PUTFIELD pt/isel/FetchManySuspendLazy$block$1.label : I
    /*
     * <=> this.yield(loc2, loc6)
     */
    ALOAD 0
    ALOAD 2
    ALOAD 6
    INVOKESPECIAL pt/isel/FetchManySuspendLazy.yield (
        Ljava/lang/String;Lkotlin/coroutines/Continuation;)Ljava/lang/Object;
```

The bytecode presented in Figure 7.31 offers a simplified version, focusing on the core instructions. It begins by evaluating the `label` field using `TABLESWITCH` and jumps to one of the corresponding states. Label `L7` represents the final state reached after the last `yield`, terminating the execution of

the suspending function `block`. Label `L8` indicates an illegal state that should never occur, except in exceptional cases.

In Figure 7.31, we present bytecode for label `L4`. However, this can be extended to cover other labels such as `L5` and `L6` with minimal changes. These changes include updating the `GETFIELD` instruction to access `url2` and `url3`, as well as updating the `label` field to the values `2` and `3` respectively.

Each state begins by executing the code corresponding to the current step, which involves fetching the given URL and storing the response body in local variable `2`. Subsequently, it updates the state variable (the `label` field) and concludes with a call to the suspending function `yield`. The continuation, in this case, is represented by local variable `6` of type `FetchManySuspendLazy$block$1`, which is passed as the last argument in the `yield` call.

Looking now into `FetchManySuspendLazy$block$1` of Figure 7.32 we can observe this class extends `ContinuationImpl` that redirects the `resumeWith` call to the function `invokeSuspend`. Despite other internal details, we are only highlighting that in this case the `invokeSuspend` will call again the `block` function when this continuation is resumed, but entering in a new state. The call to block receives this same instance of `FetchManySuspendLazy$block$1` as the continuation, which is equivalent to call `this$0.block(this)`.

Figure 7.32. Bytecode of continuation passed to yield that resumes to the outer function.

```
1  final class pt/isel/FetchManySuspendLazy$block$1
2    extends kotlin/coroutines/jvm/internal/ContinuationImpl
3  {
4    /*
5     * field `label` holding the current state in the state machine.
6     * field `this$0` that is a reference to the outer object.
7     */
8    I label
9    final synthetic Lpt/isel/FetchManySuspendLazy; this$0
10   ...
11   public final invokeSuspend(Ljava/lang/Object;)Ljava/lang/Object;
12     ...
13     /*
14      * <=> this$0.block(this)
15      */
16     ALOAD 0
17     GETFIELD pt/isel/FetchManySuspendLazy$block$1.this$0
18     ALOAD 0
19     INVOKESTATIC pt/isel/FetchManySuspendLazy.access$block (
20         Lpt/isel/FetchManySuspendLazy;Lkotlin/coroutines/Continuation;)
```

Now, let's create our custom implementation of the `yield` function, which will control the progress in the state machine. The `yield` function only needs to store the `nextItem` and update the `nextStep`

property with a reference to the next continuation, which is responsible for resuming the execution when invoked. To access this continuation, we will deconstruct the definition of the suspending function `yield` using the same technique presented in Section 7.3. The implementation of `yield` presented in Figure 7.33 is part of the `FetchManySuspendLazy` class.

Figure 7.33. Custom implementation of an yield suspending function.

```
private suspend fun yield(item: String) = yieldHande(item)

private val yieldHande = ::yieldCps as (suspend (String) -> Unit)

private fun yieldCps(item: String, onComplete: Continuation<Unit>) : Any {
    nextItem = item
    nextStep = onComplete
    return COROUTINE_SUSPENDED
}
```

The `COROUTINE_SUSPENDED` value returned by `fetchCps` indicates that the suspending function has suspended execution and will not return any result immediately.

The remaining implementation of the class `FetchManySuspendLazy` is presented in Figure 7.34. This class includes an auxiliary function `tryAdvance` that advances by either invoking the `nextStep` continuation (`nextStep?.resume(Unit)`) or starting the execution by invoking the `block` function (`blockHandle(this)`). Since `block` is a suspending function, we invoke it with a continuation to the same instance (i.e., `this`), where the `resumeWith` function will store the end result in the property `finish`. This property allows us to determine when the `block` function has terminated its execution.

Figure 7.34. FetchManySuspendLazy class

```
class FetchManySuspendLazy(...) : Continuation<Unit>, Iterator<String>
{
    var nextItem: String? = null
    var nextStep: Continuation<Unit>? = null
    var finish: Result<Unit>? = null
    ...
    private val blockHandle = ::block as ((Continuation<Unit>) -> Any)

    private fun tryAdvance() {
        nextStep
            ?.resume(Unit)
            ?:blockHandle(this)
    }
    override fun resumeWith(result: Result<Unit>) {
        finish = result
```

```
16      }
17      override fun hasNext() = finish == null
18
19      override fun next(): String {
20          tryAdvance()
21          finish?.getOrThrow()
22          if(finish != null) throw NoSuchElementException()
23          return nextItem as String
24      }
25      ...
```

The behavior of `FetchManySuspendLazy` closely mirrors that of the `FetchManyCpsLazy` class. However, it relies on the continuations generated by the Kotlin compiler rather than explicitly implementing a state machine.

7.4.2: sequence builder

Now, let's build our custom `buildSequence` utility function, equivalent to the existing `sequence` builder in Kotlin. Our `buildSequence` function will create an instance of the `SequenceBuilderIterator` class, which is based on the previous implementation of `FetchManySuspendLazy`. However, the `SequenceBuilderIterator` class will receive the `block` function as a constructor parameter, allowing it to remain flexible and not tied to the specific implementation logic of the `block`.

However, clients of our library who implement this `block` function should have access to the internal `yield` function, which is crucial for retaining the continuation instances. To facilitate this, the `SequenceBuilderIterator` class provides a `Yieldable` object as the receiver of the block function, as denoted in Figure 7.35. Additionally, the `SequenceBuilderIterator` class implements the `Yieldable` interface, allowing it to act as the receiver of the block function (`this.blockHandle(this)`) (line 23) and thereby receive the items yielded during the block function's execution.

Figure 7.35. SequenceBuilderIterator implementation with a block function parameter.

```
1   interface Yieldable<T> { suspend fun yield(item: T) }
2
3   enum class SequenceState { NotReady, Ready, Done, Failed}
4
5   private class SequenceBuilderIterator<T>(block: suspend Yieldable<T>.() -> Unit) :
6       Yieldable<T>,
7       Iterator<T>,
8       Continuation<Unit>
9   {
10      private var nextItem: T? = null
```

```
11      private var state = SequenceState.NotReady
12      private var finish: Result<Unit>? = null
13      private var nextStep: Continuation<Unit>? = null
14
15      override val context = EmptyCoroutineContext
16
17      private val blockHandle = block as (Yieldable<T>.(Continuation<Unit>) -> Any)
18
19      private fun tryAdvance() {
20          if(state == SequenceState.NotReady) {
21              nextStep
22                  ?.resume(Unit)
23                  ?: this.blockHandle(this)
24          }
25      }
26      override fun resumeWith(result: Result<Unit>) {
27          state =
28              if(result.isSuccess) {
29                  SequenceState.Done
30              } else {
31                  finish = result
32                  SequenceState.Failed
33              }
34      }
35      ...
```

There is also a further enhancement compared to the `FetchManyCpsLazy` regarding state control. The `hasNext()` function might need to advance the traversal depending on whether `next()` has been called before or not. Thus, we cannot simply check if the `finish` property is still null. We must improve it with a generic state machine that controls whether we have advanced or not. We follow a similar approach to the existing `SequenceBuilderIterator` of Kotlin in a simplified version, including four states: `NotReady`, `Ready`, `Done`, and `Failed`. The iterator will proceed by transitioning between `NotReady` and `Ready`, and will reach a final state at either `Done` or `Failed`.

Our `SequenceBuilderIterator` implements the `yield` function in the same way as the `FetchManyCpsLazy` class by deconstructing the suspend function `yield` into a `yieldCps` function. However, now, whenever `yieldCps` is called, it should update the state to the `Ready` value.

Finally, in the listing of Figure 7.36, we present the rest of the implementation of the `SequenceBuilderIterator`, including its `next()` and `hasNext()` methods.

Figure 7.36. SequenceBuilderIterator implementation of next() and hasNext().

```
private class SequenceBuilderIterator<T>(...) : ...{
    ...
    override fun hasNext(): Boolean {
        tryAdvance()
        return state == SequenceState.Ready
    }

    override fun next(): T {
        tryAdvance()
        finish?.getOrThrow()
        if(state != SequenceState.Ready) throw NoSuchElementException()
        state = SequenceState.NotReady
        return nextItem as T
    }
    ...
}
```

Summary

This chapter begins by discussing the Collection Pipeline idiom, a term coined by Martin Fowler, and comparing its characteristics across different technological environments. It also details the various design options considered in the implementation of sequences, including naming, composability, evaluation timing, access methods, and extensibility.

Following that, we introduce generators and the `yield` primitive—one of the most widely adopted mechanisms for lazily implementing custom sequence operations, commonly found in most high-level programming languages, with the notable exception of Java.

Afterward, we delve into the internals of suspending functions in Kotlin, which form the foundation for supporting generators and `yield` functionality in this environment. We conclude by explaining how generators in Kotlin are translated into a state machine using suspending functions.

7.5: Exercises

Exercise 1

Consider the following Kotlin code:

```
arrayOf("abc", "isel", "super")
    .map { print("$it "); it.length }
    .filter { print("$it "); it == 4 }
    .first()
```

a. What is the output of executing the given code?

b. If you used `sequenceOf` instead of `arrayOf`, would there be any difference? Justify your answer.

Exercise 2

Consider the following Kotlin code:

```
val res = arrayOf("abcdef", "super", "isel", "trio", "tri")
    .map { print("$it "); it.length }
    .filter { print("$it "); it % 2 == 0 }
    .distinct()
```

a. What is printed in the output when this code is executed, and what is stored in the `res` variable?

b. What is printed in the output when this code is executed using `sequenceOf` instead of `arrayOf`? Justify your answer.

Exercise 3

Make two distinct implementations of the generic extension function `coalesce`: one for `Iterable` (eager) and another for `Sequence` (lazy). Both implementations should combine pairs of elements using a function passed as a parameter, producing a new sequence with half the number of elements.

For example, the expression:

```
sequenceOf(1, 2, 3, 4, 5, 6, 7, 8, 9).coalesce { a, b -> (a + b) / 2.0 }
```

produces the sequence `1.5, 3.5, 5.5, 7.5`. If the sequence has an odd number of elements, the last element is not used. As in the example, the elements of the output sequence can be of a different type than those of the input sequence.

Exercise 4

The function `Sequence<String>.reversedWordsLongerThan(longSize: Int) : List<String>` returns a list of words longer than `longSize`, with their characters in reverse order. In the following example, the standard output will display: “lesi” and “repus”.

```
sequenceOf("isel", "abc", "super", "por")
    .reversedWordsLongerThan(3)
    .forEach(::println)
```

a. Given the two implementations below for the function `reversedWordsLongerThan`, indicate whether both produce the correct result. If they do, explain which one you would choose and why. If not, explain the difference in the results and identify the correct version.

- `return this.toList().map { it.reversed() }.filter { it.length > longSize }`
- `return this.toList().filter { it.length > longSize }.map { it.reversed() }`

b. In the previous examples, explain the difference between using `toList()` at the beginning or the end of the pipeline, or if it makes no difference.

Chapter 8: Garbage Collection and Cleanup Actions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

8.1: The Managed Heap

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

8.2: Garbage Collection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

8.3: Types Requiring Special Cleanup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

8.3.1: Using closeable types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.

8.3.2: Finalization and Cleaners

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/kotlinonjvm.