Chapter 9: Idioms and Anti-Patterns

In the previous chapters, we discussed the different aspects of the Kotlin programming language, the benefits of functional programming, and concurrent design patterns.

This chapter discusses the best and worst practices in Kotlin. You'll learn what idiomatic Kotlin code should look like and which patterns to avoid. This chapter contains a collection of best practices spanning those different topics.

In this chapter, we will cover the following topics:

- Using the scope functions
- Type checks and casts
- An alternative to the try-with-resources statement
- Inline functions
- Implementing algebraic data types
- Reified generics
- Using constants efficiently
- Constructor overload
- Dealing with nulls
- Making asynchronicity explicit
- Validating input
- Preferring sealed classes over enums

After completing this chapter, you should be able to write more readable and maintainable Kotlin code, as well as avoid some common pitfalls.

Technical requirements

In addition to the requirements from the previous chapters, you will also need a **Gradle**-enabled **Kotlin** project to be able to add the required dependencies.

You can find the source code for this chapter here: https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter09.

Using the scope functions

Kotlin has the concept of **scoping functions**, which are available on any object and can replace the need to write repetitive code. Among other benefits, these scoping functions help us simplify single-expression functions. They are considered higher-order functions since each scoping function receives a lambda expression as an argument. In this section, we'll discuss all the necessary functions and execute their code blocks using objects as their *scope*. In this section, we'll use the terms *scope* and *context object* interchangeably to describe the objects that those functions operate on.

Let function

We can use the `let()` function to invoke a function on a nullable object, but only if the object is not null.

Let's take, as an example, the following map of quotes (we discussed this in *Chapter 1*, *Getting Started with Kotlin*):

```
val clintEastwoodQuotes = mapOf(
    "The Good, The Bad, The Ugly" to "Every gun makes
its       own tune.",
    "A Fistful Of Dollars" to "My mistake: four
coffins."
)
```

Now, let's fetch a quote from a movie that may not exist in the collection and print it, but only if it's not null:

```
val quote = clintEastwoodQuotes["Unforgiven"]
if (quote != null) {
    println(quote)
}
```

The same code can we rewritten using the **`let`** scoping function:

```
clintEastwoodQuotes["Unforgiven"]?.let {
    println(it)
}
```

One common mistake is forgetting to use the safe navigation operator before **`let`**, because **`let()`** by itself also works on nulls:

```
clintEastwoodQuotes["Unforgiven"].let {
    println(it)
}
```

This code will print **`null`** to the console. Make sure that you don't forget the question mark (**`?`**) when you use **`let()`** for null checks.

Apply function

We have discussed **`apply()`** in previous chapters. It returns the same object it operates on and sets the context to **`this`**. You can use **`apply()`** if you need to initialize a mutable object.

Think of how many times you had to create a class that has an empty constructor, and then call a lot of setters, one after another. Let's look at the following class as an example. This may be a class that comes from a library, for example:

```
class JamesBond {
    lateinit var name: String
    lateinit var movie: String
```

```
    lateinit var alsoStarring: String
}
```

When we need to create a new instance of such a class, we could do so in a procedural manner:

```
val agent = JamesBond()
agent.name = "Sean Connery"
agent.movie = "Dr. No"
```

Alternatively, we can only set **`name`** and **`movie`**, and leave **`alsoStarring`** blank, using the **`apply()`** function:

```
val `007` = JamesBond().apply {
    this.name = "Sean Connery"
    this.movie = "Dr. No"
}
println(`007`.name)
```

Since the context of the block is set to **`this`**, we can simplify the preceding code even further:

```
val `007` = JamesBond().apply {
    name = "Sean Connery"
    movie = "Dr. No"
}
```

Using the **`apply()`** function is especially good when you're working with Java classes that usually have a lot of setters and a default empty constructor.

Also function

As we mentioned in the introduction to this section, single-expression functions are very nice and concise. Let's look at the following simple function, which multiplies two numbers:

```
fun multiply(a: Int, b: Int): Int = a * b
```

But often, you have a single-statement function that also needs to, for example, write to a log or have another side effect. To achieve this, we could rewrite our function in the following way:

```
fun multiply(a: Int, b: Int): Int {
    val c = a * b
    println(c)
    return c
}
```

We had to make our function much more verbose here and introduce another variable. Let's see how we can use the **`also()`** function instead:

```
fun multiply(a: Int, b: Int): Int =
    (a * b).also { println(it) }
```

This function will assign the results of the expression to **`it`** and return the result of the expression. The **`also()`** function is also useful when you want to have a side effect on a chain of calls:

```
val l = (1..100).toList()
l.filter{ it % 2 == 0 }
    // Prints, but doesn't mutate the collection
    .also { println(it) }
    .map { it * it }
```

Here, you can see that we can continue our chain of calls with a **`map()`** function, even though we used the **`also()`** function to print each element of a list.

Run function

The **`run()`** function is very similar to the **`let()`** function, but it sets the context of the block to **`this`** instead of using **`it`**.

Let's look at an example to understand this better:

```
val justAString = "string"
```

```
val n = justAString.run {
    this.length
}
```

In this example, **`this`** is set to reference the **`justAString`** variable.

Usually, **`this`** could be omitted, so the code will look as follows:

```
val n = justAString.run {
    length
}
```

The **`run()`** function is mostly useful when you plan to initialize an object, much like the **`apply()`** function we discussed earlier. However, instead of returning the object itself, like **`apply()`** does, you usually like to return the result of some computation:

```
val lowerCaseName = JamesBond().run {
    name = "ROGER MOORE"
    movie = "THE MAN WITH THE GOLDEN GUN"
    name.toLowerCase() // <= Not JamesBond type
}
println(lowerCaseName)
```

The preceding code prints the following output:

```
> roger moore
```

Here, the object was initialized with **`"ROGER MOORE"`**. Note that here, we operated on the **`JamesBond`** object, but our return value was a **`String`**.

With function

Unlike the other four scoping functions, **`with()`** is not an extension function. This means you cannot do the following:

```
"scope".with { ... }
```

Instead, **`with()`** receives the object you want to scope as an argument:

```
with("scope") {
    println(this.length) // "this" set to the
argument of                          // with()
}
```

And as usual, we can omit **`this`**:

```
with("scope") {
    length
}
```

Just like **`run()`** and **`let()`**, you can return any result from **`with()`**.

In this section, we learned how the various scope functions can help reduce the amount of boilerplate code by defining a code block to be executed on the object. In the next section, we'll see how Kotlin also allows us to write fewer instance checks than other languages.

Type checks and casts

While writing your code, you may often be inclined to check what type your object is using, **`is`**, and cast it using **`as`**. As an example, let's imagine we're building a system for superheroes. Each superhero has their own set of methods:

```
interface Superhero
class Batman : Superhero {
    fun callRobin() {
        println("To the Bat-pole, Robin!")
    }
}
class Superman : Superhero {
    fun fly() {
        println("Up, up and away!")
    }
}
```

There's also a function where a superhero tries to invoke their superpower:

```
fun doCoolStuff(s: Superhero) {
    if (s is Superman) {
        (s as Superman).fly()
    }
    else if (s is Batman) {
        (a as Batman).callRobin()
    }
}
```

But as you may know, Kotlin has smart casts, so implicit casting, in this case, is not needed. Let's rewrite this function using smart casts and see how they improve our code. All we need to do is remove the explicit casts from our code:

```
fun doCoolStuff(s: Superhero) {
    if (s is Superman) {
        s.fly()
    }
    else if (s is Batman) {
        s.callRobin()
    }
}
```

Moreover, in most cases, using **`when()`** while smart casting produces cleaner code:

```
fun doCoolStuff(s : Superhero) {
    when(s) {
        is Superman -> s.fly()
        is Batman -> s.callRobin()
        else -> println("Unknown superhero")
    }
}
```

As a rule of thumb, you should avoid using casts and rely on smart casts most of the time:

```
// Superhero is clearly not a string
val superheroAsString = (s as String)
```

But if you absolutely must, there's also a safe cast operator:

```
val superheroAsString = (s as? String)
```

The **safe cast operator** will return null if the object cannot be cast, instead of throwing an exception.

An alternative to the try-with-resources statement

Java 7 added the notion of **`AutoCloseable`** and the try-with-resources statement.

This statement allows us to provide a set of resources that will be automatically closed once the code is done with them. So, there will be no more risk (or at least less risk) of forgetting to close a file.

Before Java 7, this was a total mess, as shown in the following code:

```
BufferedReader br = null; // Nulls are bad, we know
that
try {
    br = new BufferedReader(new FileReader
      ("./src/main/kotlin/7_TryWithResource.kt "));
    System.out.println(br.readLine());
}
finally {
    if (br != null) { // Explicit check
        br.close(); // Boilerplate
    }
```

```
}
```

After **Java** 7 was released, the preceding code could be written as follows:

```
try (BufferedReader br = new BufferedReader(new
  FileReader("/some/path"))) {
    System.out.println(br.readLine());
}
```

Kotlin doesn't support this syntax. Instead, the try-with-resource statement is replaced with the **`use()`** function:

```
val br = BufferedReader(FileReader("./src/main
  /kotlin/7_TryWithResource.kt"))
br.use {
    println(it.readLines())
}
```

An object must implement the **`Closeable`** interface for the **`use()`** function to be available. The **`Closeable`** object will be closed as soon as we exit the **`use{}`** block.

Inline functions

You can think of **`inline`** functions as instructions for the compiler to copy and paste your code. Each time the compiler sees a call to a function marked with the **`inline`** keyword, it will replace the call with the concrete function body.

It makes sense to use the **`inline`** function if it's a higher-order function that receives a lambda as one of its arguments. This is the most common use case where you would like to use **`inline`**.

Let's look at such a higher-order function and see what pseudocode the compiler will output.

First, here is the function definition:

```
inline fun logBeforeAfter(block: () -> String) {
    println("Before")
    println(block())
    println("After")
}
```

Here, we pass a lambda, or a **`block`**, to our function. This **`block`** simply returns the word **`"Inlining"`** as a **`String`**:

```
logBeforeAfter {
    "Inlining"
}
```

If you were to view the Java equivalent of the decompiled bytecode, you'd see that there's no call to our **`makesSense`** function at all. Instead, you'd see the following:

```
String var1 = "Before"; <- Inline function call
System.out.println(var1);
var1 = "Inlining";
System.out.println(var1);
var1 = "After";
System.out.println(var1);
```

Since the **`inline`** function is a copy/paste of your code, you shouldn't use it if you have more than a few lines of code. It would be more efficient to have it as a regular function. But if you have single-expression functions that receive a lambda, it makes sense to mark them with the **`inline`** keyword to optimize performance. In the end, it's a trade-off between the size of your application and its performance.

Implementing Algebraic Data Types

Algebraic Data Types, or **ATDs** for short, is a concept from functional programming and is very similar to the **Composite design pattern** we discussed in *Chapter 3*, *Understanding Structural Patterns*.

To understand how ADTs work and what their benefits are, let's discuss how we can implement a simple binary tree in Kotlin.

First, let's declare an interface for our tree. Since a tree data structure can contain any type of data, we can parameterize it with a type (**`T`**):

```
sealed interface Tree<out T>
```

The type is marked with an **`out`** keyword, which means that this type is *covariant*. If you aren't familiar with this term, we'll cover it later, while implementing the interface.

The opposite of a covariant is a *contravariant*. Contravariant types should be marked using the **`in`** keyword.

We can also mark this interface with a **`sealed`** keyword. We saw this keyword applied to regular classes in *Chapter 4*, *Getting Familiar with Behavioral Patterns*, while discussing the **Visitor pattern**. But **`sealed`** interfaces are a relatively new feature and were introduced in **Kotlin 1.5**.

The meaning is the same, though: only the owner of the interface can implement it. This means that all the implementations of the interface are known at compile time.

Next, let's declare what an empty tree looks like:

```
object Empty : Tree<Nothing> {
    override fun toString() = "Empty"
}
```

Since all empty trees are the same, we declare it as an object. This is another use of the **Singleton design pattern**, which we discussed in *Chapter 2*, *Working with Creational Patterns*. We can also use **`Nothing`** as the type of an empty tree. This is a special class in Kotlin's object hierarchy.

IMPORTANT NOTE:

There is some confusion between **`Any`***, which represents any class and is similar to* **`Object`** *in Java, and* **`Nothing`***, which represents no class. We'll see why* **`Any`** *wouldn't work in this case later in this chapter.*

Next, let's define a non-empty node of a tree:

```
data class Node<T>(
    val value: T,
    val left: Tree<T> = Empty,
    val right: Tree<T> = Empty
) : Tree<T>
```

`Node` also implements the **`Tree`** interface, but it is a data class and not an object since every node is different. The type of the value of a **`Node`** is **`T`**, which means it can contain any type of value, but all the nodes in the same tree will contain the same type of value. This is the real power of generics.

A node also has two children, left and right, since it's a binary tree. By default, both of them are empty.

We can specify the default values for the children of a node thanks to the fact that the type is covariant and **`Empty`** is of the **`Nothing`** type. **`Nothing`** is at the bottom of the class hierarchy, while **`Any`** is at the very top.

When we declared the type of our **`Tree`** as **`out T`**, we meant that our **`Tree`** could contain values of type **`T`** or anything that inherits from that type.

Since **`Nothing`** is at the bottom of a class hierarchy, it *inherits* from all types.

Now that everything has been set, let's learn how to create a new instance of the tree we just defined:

```
val tree = Node(
    1,
    Empty,
```

```
    Node(
        2,
        Node(3)
    )
)
println(tree)
```

Here, we created a tree with **1** as the value of the root node and a right node with a value of **2**. The right node has a left child with a value of **3**. This is what our tree looks like:

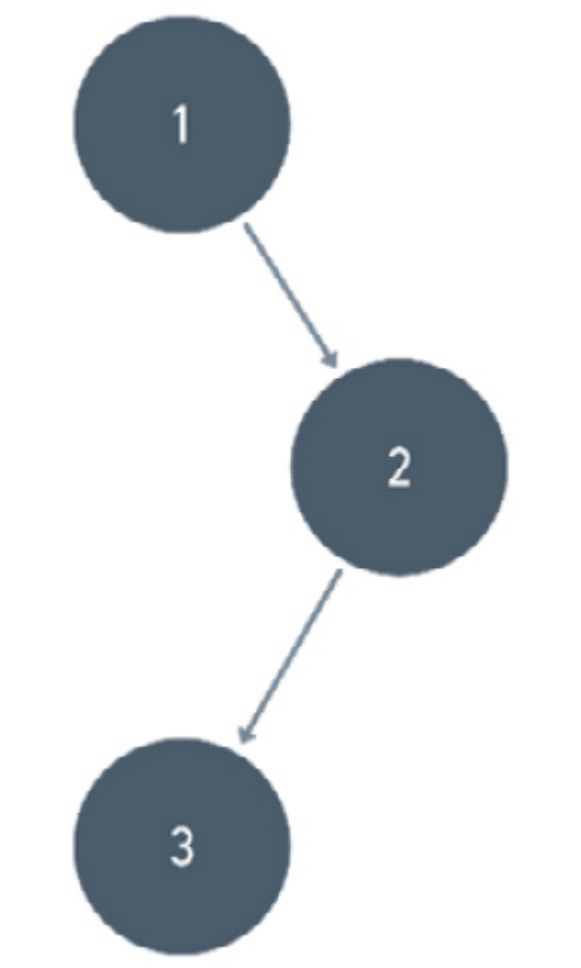

Figure 9.1 – Tree diagram

The preceding code outputs the following:

```
> Node(value=1, left=Empty, right=Node(value=2,
left=Node(value=3, left=Empty, right=Empty),
right=Empty))
```

However, printing the tree in such a form is not very interesting. So, let's implement a function that will summarize all the nodes of a tree if it's numeric:

```
fun Tree<Int>.sum(): Long = when (this) {
    Empty -> 0
    is Node -> value + left.sum() + right.sum()
}
```

This is also called an **operation** on an ADT. This is an extension function that is declared only on trees that contain integers.

For each node, we check whether it's **`Empty`** or **`Node`**. That's the beauty of **`sealed`** classes and interfaces. Since the compiler knows that the **`Tree`** interface has exactly two implementations, we don't need an **`else`** block in our **`when`** expression.

If it's an **`Empty`** node, we use **`0`** as a neutral value. If it's not empty, then we sum its values with the left and right children.

This function is also another example of a recursive algorithm, which we discussed in *Chapter 5*, *Introducing Functional Programming*.

Now, let's discuss another topic related to generics in Kotlin.

Reified generics

Previously in this chapter, we mentioned **`inline`** functions. Since **`inline`** functions are copied, we can get rid of one of the major JVM limitations: **type erasure**. After all, inside the function, we know exactly what type we're getting.

Let's look at the following example. We would like to create a generic function that will receive a **`Number`** (**`Number`** can either be **`Int`** or **`Long`**), but will only print it if it's of the same type as the function type.

We'll start with a naïve implementation, simply trying the instance check on the type directly:

```
fun <T> printIfSameType(a: Number) {
    if (a is T) { // <== Error
        println(a)
    }
}
```

However, this code won't compile and we'll get the following error:

```
> Cannot check for instance of erased type: T
```

What we usually do in Java, in this case, is pass the class as an argument. We can try a similar approach in Kotlin. If you've worked with Android before, you'll recognize this pattern immediately, since it's used a lot in the standard library:

```
fun <T : Number> printIfSameType(clazz: KClass<T>, a:
  Number) {
    if (clazz.isInstance(a)) {
        println("Yes")
    } else {
        println("No")
    }
}
```

We can check that the code works correctly by running the following lines:

```
printIfSameType(Int::class, 1) // Prints yes, as 1 is
Int
printIfSameType(Int::class, 2L) // Prints no, as 2 is
Long
printIfSameType(Long::class, 3L) // Prints yes, as 3
is Long
```

This code works but has a few downsides:

- We cannot use the **`is`** operator and must use the **`isInstance()`** function instead.
- We must pass the correct class; that is, **`clazz: KClass<T>`**.

This code could be improved by using a **`reified`** type:

```
inline fun <reified T : Number> printIfSameReified(a:
  Number) {
    if (a is T) {
        println("Yes")
    } else {
        println("No")
    }
}
```

This function works the same as the previous one but doesn't need a class as input to work. A function that uses a **`reified`** type must be declared as **`inline`**. This is due to type erasure on the JVM.

We can test that our code still works as expected:

```
printIfSameReified<Int>(1) // Prints yes, as 1 is Int
printIfSameReified<Int>(2L) // Prints no, as 2 is
Long
printIfSameReified<Long>(3L) // Prints yes, as 3 is
Long
```

Notice that now, we specify the type that the function operates on, such as **`Int`** or **`Long`**, between *angular brackets*, instead of passing a class to it as an argument. We get the following benefits from using the **`reified`** functions:

- A clear method signature, without the need to pass a class as an argument.
- The ability to use the **`is`** construct inside the function.
- It's type-inference friendly, which means that if the *type* parameter can be inferred by the compiler, it can be completely omitted.

Of course, the same rules for the regular **`inline`** functions apply here. This code would be replicated, so it shouldn't be too large.

Now, let's consider another case for **`reified`** types – **function overloading**. We'll try to define two functions with the same name that differ only in terms of the types they operate on:

```
fun printList(list: List<Int>) {
    println("This is a list of Ints")
    println(list)
}
fun printList(list: List<Long>) {
    println("This is a list of Longs")
    println(list)
}
```

This won't compile because there's a platform declaration clash. Both have the same signature in terms of JVM: **`printList(list: List)`**. This is because types are erased during compilation.

But with **`reified`**, we can achieve this easily:

```
inline fun <reified T : Any> printList(list: List<T>)
{
    when {
        1 is T -> println("This is a list of Ints")
        1L is T -> println("This is a list of Longs")
        else -> println("This is a list of something
else")
    }
    println(list)
}
```

Since the entire function is *inlined*, we can check the actual type of the list and output the correct result.

Using constants efficiently

Since everything in Java is an object (unless it's a primitive type), we're used to putting all the constants inside our objects as static members.

And since Kotlin has **`companion`** objects, we usually try putting them there:

```
class Spock {
    companion object {
        val SENSE_OF_HUMOR = "None"
    }
}
```

This will work, but you should remember that **`companion object`** is an object, after all.

So, this will be translated into the following code, more or less:

```
public class Spock {
    private static final String SENSE_OF_HUMOR =
"None";
    public String getSENSE_OF_HUMOR() {
        return Spock.SENSE_OF_HUMOR;
    }
    ...
}
```

In this example, the Kotlin compiler generates a getter for our constant, which adds another level of indirection.

If we look at the code using the constant, we'll see the following:

```
String var0 = Spock.Companion.getSENSE_OF_HUMOR();
System.out.println(var0);
```

We can invoke a method to get the constant value, which is not very efficient.

Now, let's mark this value as constant and see how the code produced by the compiler changes:

```
class Spock {
    companion object {
        const val SENSE_OF_HUMOR = "None"
    }
}
```

Here are the bytecode changes:

```
public class Spock {
   public static final String SENSE_OF_HUMOR =
"None";
   ...
}
```

And here is the call:

```
String var1 = "None";
System.out.println(var1);
```

Notice that there's no reference for our **`Spock`** class in the code anymore. The compiler has already *inlined* its value for us. After all, it's constant, so it will never change and can be safely *inlined*.

If all you need is a constant, you can also set it up outside of any class:

```
const val SPOCK_SENSE_OF_HUMOR = "NONE"
```

And if you need namespacing, you can wrap it in an object:

```
object SensesOfHumor {
    const val SPOCK = "None"
}
```

Now that we've learned how to use constants more efficiently, let's learn how to work with constructors in an idiomatic manner.

Constructor overload

In Java, we're used to having overloaded constructors. For example, let's look at the following Java class, which requires the **`a`** parameter and de-

faults the value of **b** to **1**:

```
class User {
    private final String name;
    private final boolean resetPassword;
    public User(String name) {
        this(name, true);
    }
    public User(String name, boolean resetPassword) {
        this.name = name;
        this.resetPassword = resetPassword;
    }
}
```

We can simulate the same behavior in Kotlin by defining multiple constructors using the **constructor** keyword:

```
class User(val name: String, val resetPassword:
Boolean) {
    constructor(name: String) : this(name, true)
}
```

The secondary constructor, as defined in the class body, will invoke the primary constructor, providing **1** as the default value for the second argument.

However, it's usually better to have default parameter values and named arguments instead:

```
class User(val name: String, val resetPassword:
Boolean =   true)
```

Note that all the secondary constructors must delegate to the primary constructor using the **this** keyword. The only exception is when you have a default primary constructor:

```
class User {
    val resetPassword: Boolean
```

```
    val name: String
    constructor(name: String, resetPassword: Boolean
=
      true) {
        this.name = name
        this.resetPassword = resetPassword
    }
}
```

Next, let's discuss how to efficiently handle nulls in Kotlin code.

Dealing with nulls

Nulls are unavoidable, especially if you work with Java libraries or get data from a database. We've already discussed that there are different ways to check whether a variable contains **`null`** in Kotlin; for example:

```
// Will return "String" half of the time and null the
other
// half
val stringOrNull: String? = if (Random.nextBoolean())
  "String" else null
// Java-way check
if (stringOrNull != null) {
    println(stringOrNull.length)
}
```

We could rewrite this code using the **`Elvis`** operator (**`?:`**):

```
val alwaysLength = stringOrNull?.length ?: 0
```

If the length is not **`null`**, this operator will return its value. Otherwise, it will return the default value we supplied, which is **`0`** in this case.

If you have a nested object, you can chain those checks. For example, let's have a **`Response`** object that contains a **`Profile`**, which, in turn, contains the first name and last name fields, which can be nullable:

```
data class Response(
    val profile: UserProfile?
)
data class UserProfile(
    val firstName: String?,
    val lastName: String?
)
```

This chaining will look like this:

```
val response: Response? = Response(UserProfile(null,
null))
println(response?.profile?.firstName?.length)
```

If any of the fields in the chain are null, our code won't crash. Instead, it will print **`null`**.

Finally, you can use the **`let()`** block for null checks, as we briefly mentioned in the *Using the scope functions* section. The same code, but using the **`let()`** function instead, will look like this:

```
println(response?.let {
    it.profile?.let {
        it.firstName?.length
    }
})
```

If you want to get rid of **`it`** everywhere, you can use another scoping function, **`run()`**:

```
println(response?.run {
    profile?.run {
        firstName?.length
    }
})
```

Try to avoid using the unsafe `!!` null operator in production code:

```
println(json!!.User!!.firstName!!.length)
```

This will result in **`KotlinNullPointerException`**. However, during tests, the `!!` operator may prove useful, as it will help you spot null-safety issues faster.

Making asynchronicity explicit

As you saw in the previous chapter, it is very easy to create an asynchronous function in Kotlin. Here is an example:

```
fun CoroutineScope.getResult() = async {
   delay(100)
   "OK"
}
```

However, this asynchronicity may be an unexpected behavior for the user of the function, as they may expect a simple value.

What do you think the following code prints?

```
println("${getResult()}")
```

For the user, the preceding code somewhat unexpectedly prints the following instead of **`"OK"`**:

```
> Name: DeferredCoroutine{Active}@...
```

Of course, if you have read *Chapter 6*, *Threads and Coroutines*, you will know that what's missing here is the **`await()`** function:

```
println("${getResult().await()}")
```

But it would have been a lot more obvious if we'd named our function accordingly, by adding an **`async`** suffix:

```
fun CoroutineScope.getResultAsync() = async {
   delay(100)
   "OK"
}
```

Kotlin's convention is to distinguish asynchronous functions from regular ones by adding **`Async`** to the end of the function's name. If you're working with **IntelliJ IDEA**, it will even suggest you that rename it.

Now, let's talk about some built-in functions for validating the user's input.

Validating input

Input validation is a necessary but very tedious task. *How many times did you have to write code like the following?*

```
fun setCapacity(cap: Int) {
    if (cap < 0) {
        throw IllegalArgumentException()
    }
    ...
}
```

Instead, you can check arguments with the **`require()`** function:

```
fun setCapacity(cap: Int) {
    require(cap > 0)
}
```

This makes the code a lot more fluent. You can use **`require()`** to check for nulls:

```
fun printNameLength(p: Profile) {
    require(p.firstName != null)
}
```

But there's also **`requireNotNull()`** for that:

```
fun printNameLength(p: Profile) {
    requireNotNull(p.firstName)
}
```

Use **`check()`** to validate the state of your object. This is useful when you are providing an object that the user may not have set up correctly:

```
class HttpClient {
    var body: String? = null
    var url: String = ""
    fun postRequest() {
        check(body != null) {
            "Body must be set in POST requests"
        }
    }
}
```

And again, there's a shortcut for this as well: **`checkNotNull()`**.

The difference between the **`require()`** and **`check()`** functions is that **`require()`** throws **`IllegalArgumentException`**, implying that the input that was provided to the function was incorrect. On the other hand, **`check()`** throws **`IllegalStateException`**, which means that the state of the object is corrupted.

Consider using functions such as **`require()`** and **`check()`** to improve the readability of your code.

Finally, let's discuss how to efficiently represent different states in Kotlin.

Preferring sealed classes over enums

Coming from Java, you may be tempted to overload your **`enum`** with functionality.

For example, let's say you build an application that allows users to order a pizza and track its status. We can use the following code for this:

```
// Java-like code that uses enum to represent State
enum class PizzaOrderStatus {
    ORDER_RECEIVED, PIZZA_BEING_MADE,
OUT_FOR_DELIVERY,      COMPLETED;
    fun nextStatus(): PizzaOrderStatus {
        return when (this) {
            ORDER_RECEIVED -> PIZZA_BEING_MADE
            PIZZA_BEING_MADE -> OUT_FOR_DELIVERY
            OUT_FOR_DELIVERY -> COMPLETED
            COMPLETED -> COMPLETED
        }
    }
}
```

Alternatively, you can use the **`sealed`** class:

```
sealed class PizzaOrderStatus(protected val orderId:
Int) {
    abstract fun nextStatus(): PizzaOrderStatus
}
class OrderReceived(orderId: Int) :
  PizzaOrderStatus(orderId) {
    override fun nextStatus() =
PizzaBeingMade(orderId)
}
class PizzaBeingMade(orderId: Int) :
  PizzaOrderStatus(orderId) {
    override fun nextStatus() =
OutForDelivery(orderId)
}
class OutForDelivery(orderId: Int) :
  PizzaOrderStatus(orderId) {
    override fun nextStatus() = Completed(orderId)
}
class Completed(orderId: Int) :
PizzaOrderStatus(orderId) {
```

```
    override fun nextStatus() = this
}
```

Here, we created a separate class for each object state, extending the **`PizzaOrderStatus`** sealed class.

The benefit of this approach is that we can now store the state, along with its **`status`**, more easily. In our example, we can store the ID of the order:

```
var status: PizzaOrderStatus = OrderReceived(123)
while (status !is Completed) {
    status = when (status) {
        is OrderReceived -> status.nextStatus()
        is PizzaBeingMade -> status.nextStatus()
        is OutForDelivery -> status.nextStatus()
        is Completed -> status
    }
}
```

In general, **`sealed`** classes are good if you want to have data associated with a state, and you should prefer them over enums.

Summary

In this chapter, we reviewed the best practices in Kotlin, as well as some of the caveats of the language. Now, you should be able to write more idiomatic code that is also performant and maintainable.

You should make use of the scoping functions where necessary, but make sure not to overuse them as they may make the code confusing, especially for those newer to the language.

Be sure to handle nulls and type casts correctly, with **`let()`**, the **`Elvis`** operator, and the smart casts that the language provides. Finally, generics and **`sealed`** classes and interfaces are powerful tools that help describe complex relationships and behaviors between different classes.

In the next chapter, we'll put those skills to use by writing a real-life microservice Reactive design pattern.

Questions

1. What is the alternative to Java's try-with-resources in Kotlin?
2. What are the different options for handling nulls in Kotlin?
3. Which problem can be solved by reified generics?