Chapter 2: Working with Creational Patterns

In this chapter, we'll cover how classic **creational patterns** are implemented using **Kotlin**. These patterns deal with *how* and *when* you *create* your objects. For each design pattern, we will discuss what it aims to achieve and how Kotlin accommodates those needs.

We will cover the following topics in this chapter:

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

Mastering these design patterns will allow you to manage your objects better, adapt well to changes, and write code that is easy to maintain.

Technical requirements

For this chapter, you will need to install the following:

- **IntelliJ IDEA Community Edition** (https://www.jetbrains.com/idea/download/)
- **OpenJDK 11** (or higher) (https://openjdk.java.net/install/)

You can find the code files for this chapter on **GitHub** at https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter02.

Singleton

Singleton – the most popular bachelor in town. Everybody knows him, everybody talks about him, and everybody knows where to look for him.

Even people who don't like using design patterns will know Singleton by name. At one point, it was even proclaimed an **anti-pattern**, but only because of its wide popularity.

So, for those who are encountering it for the first time, what is this design pattern all about?

Usually, if you have a class, you can create as many instances of it as you want. For example, let's say that we both are asked to list all of our favorite movies:

```
val myFavoriteMovies = listOf("Black Hawk Down",
"Blade   Runner")
val yourFavoriteMovies = listOf(...)
```

Note that we can create as many instances of **`List`** as we want, and there's no problem with that. Most classes can have multiple instances.

Next, what if we both want to list the best movies in the Quick and Angry series?

```
val myFavoriteQuickAndAngryMovies = listOf()
val yourFavoriteQuickAndAngryMovies = listOf()
```

Note that these two lists are exactly the same because they are empty. And they will stay empty because they are immutable and because the *Quick and Angry* series is simply horrendous. I hope you would agree with that.

Since these two instances of a class are exactly the same, according to the **equals method**, it doesn't make much sense to keep them in memory multiple times. It would be great if all references to an empty list pointed

to the same instance of an object. And in fact, that's what happens with null, if you think about it. All nulls are the same.

That's the main idea behind the Singleton design pattern.

There are a couple of requirements for the Singleton design pattern:

- We should have exactly one instance in our system.
- This instance should be accessible from any part of our system.

In **Java** and some other languages, this task is quite complex. First, you need to forbid new instances of an object being created by making a constructor for the **`private`** class. Then, you also need to make sure that instantiation is preferably lazy, thread-safe, and performant, with the following requirements:

- **Lazy**: We might not want to instantiate a singleton object when our program starts, as this may be an expensive operation. We would like to instantiate it only when it's needed for the first time.
- **Thread-safe**: If two threads are trying to instantiate a singleton object at the same time, they both should receive the same instance and not two different instances. If you're not familiar with this concept, we'll cover it in *Chapter 5*, *Introducing Functional Programming*.
- **Performant**: If many threads are trying to instantiate a singleton object at the same time, we shouldn't block them for a long period of time, as this will be halting their execution.

Meeting all of these requirements in Java or **C++** is quite difficult, or at least very verbose.

Kotlin makes creating singletons easy by introducing a keyword called **`object`**. You may recognize this keyword from **Scala**. By using this keyword, we'll get an implementation of a singleton object, which accommodates all of our requirements.

IMPORTANT NOTE:

The **`object`** *keyword is used for more than just creating singletons. We'll discuss this in depth later in this chapter.*

We declare objects just like a regular class but with no constructor, as a singleton object cannot be instantiated by us:

```
object NoMoviesList
```

From now on, we can access **`NoMoviesList`** from anywhere in our code, and there will be exactly one instance of it:

```
val myFavoriteQuickAndAngryMovies = NoMoviesList
val yourFavoriteQuickAndAngryMovies = NoMoviesList
println(myFavoriteQuickAndAngryMovies ===
    yourFavoriteQuickAndAngryMovies) // true
```

Take note of the referential equality sign that checks that two variables point to the same object in memory. *Is this really a list though?*

Let's create a function that prints the list of our movies:

```
fun printMovies(movies: List<String>) {
    for (m in movies) {
        println(m)
    }
}
```

When we pass an initial list of movies, the code compiles just fine:

```
// Prints each movie on a newline
printMovies(myFavoriteMovies)
```

But if we pass it our empty movie list, the code won't compile:

```
printMovies(myFavoriteQuickAndAngryMovies)
// Type mismatch: inferred type is NoMoviesList but
// List<String> was expected
```

The reason for this is that our function only accepts arguments of the *list of strings* type, while there's nothing to tell the function that **`NoMoviesList`** is of this type (even though its name suggests it).

Luckily, in Kotlin, singleton objects can implement interfaces, and a generic **`List`** interface is available:

```
object NoMoviesList : List<String>
```

Now, our compiler will prompt us to implement the required functions. We'll do that by adding a body to **`object`**:

```
object NoMoviesList : List<String> {
    override val size = 0
    override fun contains(element: String) = false
    ... /
}
```

We'll leave it to you to implement the other functions if you wish. This should be a good exercise of everything you've learned about Kotlin until now. However, you don't have to do this. Kotlin already provides a function to create empty lists of any type:

```
printMovies(emptyList())
```

If you're curious, this function returns a singleton object that implements a **`List`**. You can see the complete implementation in the Kotlin source code using your IntelliJ IDEA or on GitHub (https://github.com/JetBrains/kotlin/blob/master/libraries/stdlib/src/kotlin/collections/Collections.kt). This is an excellent example of how design patterns are still actively applied in modern software.

A Kotlin **`object`** has one major difference from a class – it can't have constructors. If you need to implement initialization for your Singleton, such as loading data from a configuration file for the first time, you can use the **`init`** block instead:

```
object Logger {
    init {
        println("I was accessed for the first time")
        // Initialization logic goes here
    }
```

```
        // More code goes here
    }
```

Note that if a Singleton is never invoked, it won't run its initialization logic at all, thereby saving resources. This is called **lazy initialization**.

Now that we have learned how to limit object creation, let's discuss how to create objects without using a constructor directly.

Factory Method

The **Factory Method** design pattern is all about creating objects.

But why do we need a method to create objects? Isn't that what constructors are for?

Well, constructors have limitations.

As an example, imagine we're building a game of chess. We would like to allow our players to save the state of the game into a text file and then restore the game from that position.

Since the size of the board is predetermined, we only need to record the position and type of each piece. We'll use algebraic notation for this – for example, the Queen piece at C3 will be stored in our file as **`qc3`**, the pawn piece at A8 will be stored as **`pa8`**, and so on.

Let's assume that we already read this file into a list of strings (which, by the way, would be an excellent application of the Singleton design pattern we discussed earlier).

Given the list of notations, we would like to populate our board with them:

```
// More pieces here
val notations = listOf("pa8", "qc3", ...)
```

```
val pieces = mutableListOf<ChessPiece>()
for (n in notations) {
    pieces.add(createPiece(n))
}
println(pieces)
```

Before we can implement our **`createPiece`** function, we need to decide what's common to all chess pieces. We'll create an interface for that:

```
interface ChessPiece {
    val file: Char
    val rank: Char
}
```

Note that interfaces in Kotlin can declare properties, which is a very powerful feature.

Each chess piece will be a **`data class`** that implements our interface:

```
data class Pawn(
    override val file: Char,
    override val rank: Char
) : ChessPiece
data class Queen(
    override val file: Char,
    override val rank: Char
) : ChessPiece
```

The implementation of the other chess pieces is left as an exercise for you to do.

Now, what's left is to implement our **`createPiece`** function:

```
fun createPiece(notation: String): ChessPiece {
    val (type, file, rank) = notation.toCharArray()
    return when (type) {
        'q' -> Queen(file, rank)
        'p' -> Pawn(file, rank)
```

```
            // ...
            else -> throw RuntimeException("Unknown
piece: $type")
        }
}
```

Before we can discuss what this function achieves, let's cover three new syntax elements we haven't seen before.

First, the **`toCharArray`** function splits a string into an array of characters. Since we assume that all of our notations are three characters long, the element at the **`0`** position will represent the *type* of the chess piece, the element at the **`1`** position will represent its vertical column – also known as **`file`** – and the last element will represent its horizontal column – also known as **`rank`**.

Next, we can see three values: **`type`**, **`file`**, and **`rank`**, surrounded by parentheses. This is called a **destructuring declaration**, and you may be familiar with them from JavaScript, for example. Any **`data class`** can be destructured.

The previous code example is similar to the following, much more verbose code:

```
val type = notation.toCharArray()[0]
val file = notation.toCharArray()[1]
val rank = notation.toCharArray()[2]
```

Now, let's focus on the **`when`** expression. Based on the letter representing the type, it instantiates one of the implementations of the **`ChessPiece`** interface. Remember, this is what the Factory Method design pattern is all about.

To make sure you grasp this design pattern well, feel free to implement the classes and logic for other chess pieces as an exercise.

Finally, let's look at the bottom of our function, where we see the first use of a **`throw`** expression.

This expression, as the name suggests, *throws* an exception, which will stop the normal execution of our simple program. We'll discuss how to handle exceptions in *Chapter 5*, *Introducing Functional Programming*.

In the real world, the Factory Method design pattern is often used by libraries that need to parse configuration files – be they of the XML, JSON, or YAML format – into runtime objects.

Static Factory Method

There is a similarly named design pattern (which has a slightly different implementation) that is often confused with the Factory Method design pattern, and it is described in the *Gang of Four* book – the **Static Factory Method** design pattern.

The Static Factory Method design pattern was popularized by Joshua Bloch in his book, *Effective Java*. To understand this better, let's look at some examples from the Java standard library: the **`valueOf()`** methods. There are at least two ways to construct a **`Long`** (that is, a 64-bit integer) from a string:

```
Long l1 = new Long("1"); // constructor
Long l2 = Long.valueOf("1"); // static factory method
```

Both the constructor and the **`valueOf()`** method receive string as input and produce **`Long`** as output.

So, why should we prefer the Static Factory Method design pattern to a simple constructor?

Here are some of the advantages of using the Static Factory Method compared to constructors:

- It provides an opportunity to explicitly name different object constructors. This is especially useful when your class has multiple constructors.
- We usually don't expect exceptions from a constructor. That doesn't mean that the instantiation of a class can't fail. Exceptions from a regular method, on the other hand, are much more accepted.
- Speaking of expectations, we expect the constructor to be fast. But construction of some objects is inherently slow. Consider using the Static Factory Method instead.

These are mostly style advantages; however, there are also technological advantages to this approach.

Caching

The Static Factory Method design pattern may provide **caching**, as `Long` actually does. Instead of always returning a new instance for any value, `valueOf()` checks in the cache whether this value was already parsed. If it was, it returns a cached instance. Repeatedly calling the Static Factory Method with the same values may produce less garbage for collection than using constructors all the time.

Subclassing

When calling the constructor, we always instantiate the class we specify. On the other hand, calling a Static Factory Method is less restrictive and may produce either an instance of the class itself or one of its subclasses. We'll come to this after discussing the implementation of this design pattern in Kotlin.

Static Factory Method in Kotlin

We discussed the `object` keyword earlier in this chapter in the *Singleton* section. Now, we'll see another use of it as a **companion object**.

In Java, Static Factory Methods are declared **`static`**. But in Kotlin, there's no such keyword. Instead, methods that don't belong to an instance of a class can be declared inside **`companion object`**:

```
class Server(port: Long) {
    init {
        println("Server started on port $port")
    }
    companion object {
        fun withPort(port: Long) = Server(port)
  }
}
```

IMPORTANT NOTE:

Companion objects may have a name – for example, **`companion object`** *parser. But this is only to provide clarity about what the goal of the object is.*

As you can see, this time, we have declared an object that is prefixed by the **`companion`** keyword. Also, it's located inside a class, and not at the package level in the way we saw in the Singleton design pattern.

This object has its own methods, and you may wonder what the benefit of this is. Just like a Java static method, calling a **`companion object`** will lazily instantiate it when the containing class is accessed for the first time:

```
Server.withPort(8080) // Server started on port 8080
```

Moreover, calling it on an instance of a class simply won't work, unlike in Java:

```
Server(8080) // Won't compile, constructor is private
```

IMPORTANT NOTE:

A class may have only one **`companion object`**.

Sometimes, we also want the Static Factory Method to be the only way to instantiate our object. In order to do that, we can declare the default constructor of our object as **`private`**:

```
class Server private constructor(port: Long) {
    ...
}
```

This means that now there's only one way of constructing an instance of our class – through our Static Factory Method:

```
val server = Server(8080))  // Doesn't compile
val server = Server.withPort(8080) // Works!
```

Let's now discuss another design pattern that is often confused with the Factory Method – Abstract Factory.

Abstract Factory

Abstract Factory is a greatly misunderstood design pattern. It has a notorious reputation for being very complex and bizarre. Actually, it's quite simple. If you understood the Factory Method design pattern, you'll understand this one in no time. This is because the Abstract Factory design pattern is a factory of factories. That's all there is to it. The *factory* is a function or class that's able to create other classes. In other words, an abstract factory is a class that wraps multiple factory methods.

You may understand this and still wonder what the use of such a design pattern may be. In the real world, the Abstract Factory design pattern is often used in frameworks and libraries that get their configuration from files. The **Spring Framework** is just one example of these.

To better understand how the design pattern works, let's assume we have a configuration for our server written in a YAML file:

```
server:
    port: 8080
```

```
environment: production
```

Our task is to construct objects from this configuration.

In the previous section, we discussed how to use Factory Method to construct objects from the same family. But here, we have two families of objects that are related to each other but are not *siblings*.

First, let's describe them as interfaces:

```
interface Property {
    val name: String
    val value: Any
}
```

Instead of a **`data class`**, we'll return an interface. You'll see how this helps us later in this section:

```
interface ServerConfiguration {
    val properties: List<Property>
}
```

Then, we can provide basic implementations to be used later:

```
data class PropertyImpl(
    override val name: String,
    override val value: Any
) : Property
data class ServerConfigurationImpl(
    override val properties: List<Property>
) : ServerConfiguration
```

The server configuration simply contains the list of properties – and a *property* is a pair comprising a **`name`** object and a **`value`** object.

This is the first time we have seen the **`Any`** type being used. The **`Any`** type is Kotlin's version of Java's **`object`**, but with one important distinction: it cannot be null.

Now, let's write our first Factory Method, which will create **`Property`** given as a string:

```
fun property(prop: String): Property {
    val (name, value) = prop.split(":")
    return when (name) {
        "port" -> PropertyImpl(name,
value.trim().toInt())
        "environment" -> PropertyImpl(name,
value.trim())
        else -> throw RuntimeException("Unknown
property:           $name")
    }
}
```

As in many other languages, **`trim()`** is a function that is declared on strings that removes any spaces in the string. Now, let's create two properties to represent the port (**`port`**) and environment (**`environment`**) of our service:

```
val portProperty = property("port: 8080")
val environment = property("environment: production")
```

There is a slight issue with this code. To understand what it is, let's try to store the value of the **`port`** property into another variable:

```
val port: Int = portProperty.value
// Type mismatch: inferred type is Any but Int was
expected
```

We already ensured that **`port`** is parsed to an **`Int`** in our Factory Method. But now, this information is lost because the type of the value is declared as **`Any`**. It can be **`String`**, **`Int`**, or any other type, for that matter. We need a new tool to solve this issue, so let's take a short detour and discuss casts in Kotlin.

Casts

Casts in typed languages are a way to try and force the compiler to use the type we specify, instead of the type it has inferred. If we are sure what type the value is, we can use an *unsafe* cast on it:

```
val port: Int = portProperty.value as Int
```

The reason it is called *unsafe* is that if the value is not of the type we expect, our program will crash without the compiler being able to warn us.

Alternatively, we could use a *safe* cast:

```
val port: Int? = portProperty.value as? Int
```

Safe casts won't crash our program, but if the type of the object is not what we expect, it will return null. Notice that our **`port`** variable now is declared as the nullable **`Int`**, so we have to explicitly deal with the possibility of not getting what we want during compilation time.

Subclassing

Instead of resorting to casts, let's try another approach. Instead of using a single implementation with a value of the **`Any`** type, we'll use two separate implementations:

```
data class IntProperty(
    override val name: String,
    override val value: Int
) : Property
data class StringProperty(
    override val name: String,
    override val value: String
) : Property
```

Our Factory Method will have to change a little to be able to return one of the two classes:

```
fun property(prop: String): Property {
    val (name, value) = prop.split(":")
```

```
    return when (name) {
        "port" -> IntProperty(name,
value.trim().toInt())
        "environment" -> StringProperty(name,
value.trim())
        else -> throw RuntimeException("Unknown
property:           $name")
    }
}
```

This looks fine, but if we try to compile our code again, it still won't work:

```
val portProperty = Parser.property("port: 8080")
val port: Int = portProperty.value
```

Although we now have two concrete classes, the compiler doesn't know if the parsed property is **`IntProperty`** or **`StringProperty`**. All it knows is that it's **`Property`**, and the type of the value is still **`Any`**:

```
> Type mismatch: inferred type is Any but Int was
expected
```

We need another trick, and that trick is called **smart casts**.

Smart casts

We can check if an object is of a given type by using the **`is`** keyword:

```
println(portProperty is IntProperty) // true
```

However, the Kotlin compiler is very smart. *If we performed a type check on an* **`if`** *expression, it would mean that* **`portProperty`** *was indeed* **`IntProperty`***, right?* So, it could be safely cast.

The Kotlin compiler will do just that for us:

```
if (portProperty is IntProperty) {
    val port: Int = portProperty.value // works!
}
```

There is no compilation error anymore, and we also do not have to deal with nullable values.

Smart casts also work on nulls. In Kotlin's type hierarchy, the non-nullable **`Int`** type is a subclass of a nullable type, **`Int?`**, and this is true for all types. Previously, we mentioned that a *safe* cast will return **`null`** if it fails:

```
val port: Int? = portProperty.value as? Int
```

We could check if **`port`** is null, and if it isn't, it will be smartly cast to a non-nullable type:

```
if (port != null) {
    val port: Int = port
}
```

Nice! But wait, what's going on in this code?

In the previous chapter, we said that values cannot be reassigned. But here, we defined the **`port`** value twice. *How is this possible?* This is not a bug, but another Kotlin feature, and it is called **variable shadowing**.

Variable shadowing

First, let's consider how our code would look if there was no shadowing. We would have to declare two variables with different names:

```
val portOrNull: Int? = portProperty.value as? Int
if (portOrNull != null) {
    val port: Int = portOrNull // works
}
```

However, this is a waste, for two reasons. First, the variable names become quite verbose. Second, the **`portOrNull`** variable would most probably never be used past this point because null is not a very useful value to begin with. Instead, we can declare values with the same names in different scopes, denoted by curly brackets (**`{}`**).

Please note that variable shadowing may confuse you, and it is error-prone by nature. However, it is important to be aware that it exists, but the recommendation is to name your variables explicitly whenever possible.

Collection of Factory Methods

Now that we've had our detour into casts and variable shadowing, let's go back to the previous code example and implement a second Factory Method, that will create a **`server`** configuration object:

```
fun server(propertyStrings: List<String>):
  ServerConfiguration {
    val parsedProperties = mutableListOf<Property>()
    for (p in propertyStrings) {
        parsedProperties += property(p)
    }
    return ServerConfigurationImpl(parsedProperties)
}
```

This method takes the lines from our configuration file and converts them into **`Property`** objects using the **`property()`** Factory Method that we've already implemented.

We can test that our second Factory Method works as well:

```
println(server(listOf("port: 8080", "environment:
  production")))
> ServerConfigurationImpl(properties=
[IntProperty(name=port, value=8080),
StringProperty(name=environment, value=production)])
```

Since these two methods are related, it would be good to put them together under the same class. Let's call this class **`Parser`**. Although we didn't parse any actual file and agreed that we get its contents line by line already, by the end of this book, you would probably agree that implementing the actual reading logic is quite trivial.

We can also use Static Factory Method and the **`companion object`** syntax we learned about in the previous section.

The resulting implementation will look like this:

```
class Parser {
    companion object {
        fun property(prop: String): Property {
           ...
        }
        fun server(propertyStrings: List<String>):
...{
           ...
        }
    }
}
```

This pattern allows us to create *families* of objects – in this case, **`ServerConfig`** is the *parent* of a property.

The previous code is just one way to implement an Abstract Factory. You may find some implementations that rely on implementing an interface instead:

```
interface Parser {
    fun property(prop: String): Property
    fun server(propertyStrings: List<String>):
      ServerConfiguration
}
class YamlParser : Parser {
    // Implementation specific to YAML files
}
class JsonParser : Parser {
    // Implementation specific to JSON files
}
```

This approach may be better if your Factory Methods grow to contain lots of code.

One last question you may have is where we can see Abstract Factory used in real code. One example is the **`java.util.Collections`** class. It has methods such as **`emptyMap`**, **`emptyList`**, and **`emptySet`**, which each generate a different class. However, what is common to all of them is that they are all collections.

Builder

Sometimes, our objects are very simple and have only one constructor, be it an empty or non-empty one. But sometimes, their creation is very complex and based on a lot of parameters. We've seen one pattern already that provides *a better constructor* – the Static Factory Method design pattern. Now, we'll discuss the **Builder** design pattern, which will help us create complex objects.

As an example of such an object, imagine we need to design a system that sends emails. We won't implement the actual mechanism of sending them, we will just design a class that represents it.

An email may have the following properties:

- An address (at least one is mandatory)
- CC (optional)
- Title (optional)
- Body (optional)
- Important flag (optional)

We can describe an email in our system as a **`data class`**:

```
data class Mail_V1(
    val to: List<String>,
    val cc: List<String>?,
```

```
    val title: String?,
    val message: String?,
    val important: Boolean,
)
```

IMPORTANT NOTE:

Look at the definition of the last argument in the preceding code. This comma is not a typo. It is called a **trailing comma***, and these were introduced in* **Kotlin 1.4***. This is done so you can easily change the order of the arguments.*

Next, let's attempt to create an email addressed to our manager:

```
val mail = Mail_V1(
    listOf("manager@company.com"),    // To
    null,                             // CC
    "Ping ",                          // Title
    null,                             // Message,
    true))                            // Important
```

Note that we have defined *carbon copy* (that's what **`CC`** stands for) as nullable so that it can receive either a list of emails or null. Another option would be to define it as **`List<String>`** and force our code to pass **`listOf()`**.

Since our constructor receives a lot of arguments, we had to put in some comments in order not to get confused.

But what happens if we need to change this class now?

First, our code will stop compiling. Second, we need to keep track of the comments. In short, constructors with a long list of arguments quickly become a mess.

This is the problem the Builder design pattern sets out to solve. It decouples the assigning of arguments from the creation of objects and allows the creation of complex objects one step at a time. In this section, we'll see a number of approaches to this problem.

Let's start by creating a new class, **`MailBuilder`**, which will wrap our **`Mail`** class:

```
class MailBuilder {
    private var to: List<String> = listOf()
    private var cc: List<String> = listOf()
    private var title: String = ""
    private var message: String = ""
    private var important: Boolean = false
    class Mail internal constructor(
        val to: List<String>,
        val cc: List<String>?,
        val title: String?,
        val message: String?,
        val important: Boolean
    )
    ... // More code will come here soon
}
```

Our builder has exactly the same properties as our resulting class. But these properties are all mutable.

Note that the constructor is marked using the **`internal`** visibility modifier. This means that our **`Mail`** class will be accessible to any code inside our module.

To finalize the creation of our class, we'll introduce the **`build()`** function:

```
fun build(): Mail {
    if (to.isEmpty()) {
        throw RuntimeException("To property is
empty")
    }
    return Mail(to, cc, title, message, important)
}
```

And for each property, we'll have another function to be able to set it:

```
fun message(message: String): MailBuilder {
    this.message = message
    return this
}
// More functions for each of the properties
```

Now, we can use our builder to create an email in the following way:

```
val email =
MailBuilder("hello@hello.com").title("What's
  up?").build()
```

After setting a new value, we return a reference to our object by using **`this`**, which provides us with access to the next setter to allow us to perform chaining (please refer to the *Fluent setters* section in this chapter for an explanation of this).

This is a working approach. But it has a couple of downsides:

- The properties of our resulting class must be repeated insider the builder.
- For every property, we need to declare a function to set its value.

Kotlin provides two other ways that you may find even more useful.

Fluent setters

The approach using **fluent setters** is a bit more concise. Here, we won't construct any additional classes. Instead, our **`data class`** constructor will contain only the mandatory fields. All other fields will become **`private`**, and we'll provide setters for these fields:

```
data class Mail_V2(
    val to: List<String>,
    private var _message: String? = null,
    private var _cc: List<String>? = null,
    private var _title: String? = null,
```

```
    private var _important: Boolean? = null
) {
    fun message(message: String) = apply {
        _message = message
    }
    // Pattern repeats for every other field
    //...
}
```

IMPORTANT NOTE:

Using underscores for **`private`** *variables is a common convention in Kotlin. It allows us to avoid repeating* **`this.message = message`** *and mistakes such as* **`message = message`**.

In this code example, we used the **`apply`** function. This is part of the family of scoping functions that can be invoked on every Kotlin object, and we'll cover them in detail in *Chapter 9*, *Idioms and Anti-Patterns*. The **`apply`** function returns the reference to an object after executing the block. So, it's a shorter version of the setter function from the previous example:

```
fun message(message: String): MailBuilder {
    this.message = message
    return this
}
```

This provides us with the same API as the previous example:

```
val mailV2 =
Mail_V2(listOf("manager@company.com")).message("Ping"
)
```

However, we may not need setters at all. Instead, we can use the **`apply()`** function we previously discussed on the object itself. This is one of the extension functions that every object in Kotlin has. This approach will work only if all of the optional fields are *variables* instead of *values*.

Then, we can create our email like this:

```
val mail = Mail_V2("hello@mail.com").apply {
    message = "Something"
    title = "Apply"
}
```

This is a nice approach, and it requires less code to implement. However, there are a few downsides to this approach too:

- We had to make all of the optional arguments mutable. Immutable fields should always be preferred to mutable ones, as they are thread-safe and easier to reason about.
- All of our optional arguments are also nullable. Kotlin is a null-safe language, so every time we access them, we first have to check that their value was set.
- This syntax is very verbose. For each field, we need to repeat the same pattern over and over again.

Now, let's discuss the last approach to this problem.

Default arguments

In Kotlin, we can specify default values for constructor and function parameters:

```
data class Mail_V3(
    val to: List<String>,
    val cc: List<String> = listOf(),
    val title: String = "",
    val message: String = "",
    val important: Boolean = false
)
```

Default arguments are set using the `=` operator after the type. This means that although our constructor still has all the arguments, we don't need to provide them any.

So, if you would like to create an email without a body, you can do it like this:

```
val mail = Mail_V3(listOf("manager@company.com"),
listOf(), "Ping")
```

However, note that we had to specify that we don't want anyone in the CC field by providing an empty list, which is a bit inconvenient.

What if we wanted to send an email that is only flagged as important?

Not having to specify order with fluent setters was very handy. Kotlin has *named arguments* for that:

```
val mail = Mail_V3(title = "Hello", message =
"There", to = listOf("my@dear.cat"))
```

Combining default parameters with named arguments makes creating complex objects in Kotlin rather easy. For that reason, you will rarely need the Builder design pattern at all in Kotlin.

In real applications, you'll often see the Builder design pattern used to construct instances of servers. A server would accept an optional host and an optional port and so on, and then when all of the arguments were set, you'd invoke a listen method to start it.

Prototype

The **Prototype** design pattern is all about customization and creating objects that are similar but slightly different. To understand it better, Let's look at an example.

Imagine we have a system that manages users and their permissions. A **`data class`** representing a user might look like this:

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

```
    val role: Role,
    val permissions: Set<String>,
) {
    fun hasPermission(permission: String) =
permission in       permissions
}
```

Each user must have a role, and each role has a set of permissions.

We'll describe a role as an **enum** class:

```
enum class Role {
    ADMIN,
    SUPER_ADMIN,
    REGULAR_USER
}
```

The **enum** classes are a way to represent a collection of constants. This is more convenient than representing a role as a string, for example, as we check at compile time that such an object exists.

When we create a new *user*, we assign them permissions that are similar to another user with the same *role*:

```
// In real application this would be a database of
users
val allUsers = mutableListOf<User>()
fun createUser(name: String, role: Role) {
    for (u in allUsers) {
        if (u.role == role) {
            allUsers += User(name, role,
u.permissions)
            return
        }
    }
    // Handle case that no other user with such a
role exists
```

```
}
```

Let's imagine that we now need to add a new field to the **`User`** class, which we will name **`tasks`**:

```
data class User(
    val name: String,
    val role: Role,
    val permissions: Set<String>,
    val tasks: List<String>,
) {
   ...
}
```

Our **`createUser`** function will stop compiling. We'll have to change it by copying the value of this newly added field to the new instance of our class:

```
allUsers += User(name, role, u.permissions, u.tasks)
```

This work will have to be repeated every time the **`User`** class is changed.

However, there's a bigger problem still: *What if a new requirement is introduced, making the* **`permissions`** *property, for example,* **`private`***?*

```
data class User(
    val name: String,
    val role: Role,
    private val permissions: Set<String>,
    val tasks: List<String>,
) {
   ...
}
```

Our code will stop compiling again, and we'll have to change it again. The constant requirement of changes to the code is a clear indication that we need another approach to solve this problem.

Starting from a prototype

The whole idea of a *prototype* is to be able to clone an object easily. There are at least two reasons you may want to do this:

- It helps in instances where creating your object is very expensive – for example, if you need to fetch it from the database.
- It helps if you need to create objects that are similar but vary slightly and you don't want to repeat similar parts over and over again.
  IMPORTANT NOTE:
  There are also more advanced reasons to use the Prototype design pattern. JavaScript, for example, uses prototypes to implement inheritance-like behavior without having classes.

Luckily, Kotlin fixes the somewhat broken Java **`clone()`** method. Data classes have a **`copy()`** method, which takes an existing **`data class`**, and creates a new copy of it, optionally changing some of its attributes in the process:

```
// Name argument is underscored here simply not to
confuse
// it with the property of the same name in the User
object
fun createUser(_name: String, role: Role) {
    for (u in allUsers) {
        if (u.role == role) {
            allUsers += u.copy(name = _name)
            return
        }
    }
    // Handle case that no other user with such a
role exists
}
```

In a similar way to what we saw with the Builder design pattern, named arguments allow us to specify attributes that we can change in any order. And we need to specify only the attributes we want to change. All of the other data will be copied for us, even the **`private`** properties.

The **`data class`** is yet another example of a design pattern that is so common that it became part of a language syntax. They are an extremely useful feature, and we will see them being used many more times in this book.

Summary

In this chapter, we have learned when and how to use creational design patterns. We started by discussing how to use the **`object`** keyword to construct a singleton class, and then we discussed the use of **`companion object`** if you need a Static Factory Method. We also covered how to assign multiple variables at once using destructuring declarations.

Then, we discussed smart casts, and how they can be applied in the Abstract Factory design pattern to create families of objects. We then moved to the Builder design pattern and learned that functions can have default parameter values. We then learned that we can refer to their arguments using not only positions but also names.

Finally, we covered the **`copy()`** function of the data classes, and how it helps us when implementing the Prototype design pattern to produce similar objects with slight changes. You should now understand how to use creational design patterns to better manage your objects.

In the next chapter, we'll cover the second family of design patterns: **structural patterns**. These design patterns will help us create extensible and maintainable object hierarchies.

Questions

1. Name two uses for the **`object`** keyword we learned about in this chapter.
2. What is the **`apply()`** function used for?
3. Provide one example of a Static Factory Method.