

Marcin Moskała

Advanced Kotlin



Advanced Kotlin

Marcin Moskała

This book is for sale at http://leanpub.com/advanced_kotlin

This version was published on 2023-12-07



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2022 - 2023 Marcin Moskała

*For those who helped and inspired me on my path of learning
programming craftsmanship, especially Kamil Kędzia, Rafał
Trzeciak, Bartek Wilczyński, Michał Malanowicz, and Mateusz
Mikulski.*

Contents

Introduction	1
Who is this book for?	1
That will be covered?	1
The structure of the book	3
The Kotlin for Developers series	3
Conventions	4
Code conventions	4
Acknowledgments	6
Generic variance modifiers	8
List variance	10
Consumer variance	12
Function types	15
The Covariant Nothing Object	17
The Covariant Nothing Class	23
Variance modifier limitations	25
UnsafeVariance annotation	31
Variance modifier positions	33
Star projection	35
Summary	35
Exercise: Generic types usage	37
Exercise: Generic Response	38
Exercise: Generic Consumer	39
Interface delegation	42
The delegation pattern	42
Delegation and inheritance	43
Kotlin interface delegation support	45
Wrapper classes	49
The decorator pattern	50
Intersection types	53

CONTENTS

Limitations	55
Conflicting elements from parents	56
Summary	57
Exercise: ApplicationScope	58
Property delegation	59
How property delegation works	60
Other <code>getValue</code> and <code>setValue</code> parameters	64
Implementing a custom property delegate	67
Provide a delegate	70
Property delegates in Kotlin stdlib	73
The <code>NotNull</code> delegate	73
Exercise: <code>Lateinit</code> delegate	75
The <code>Lazy</code> delegate	76
Exercise: Blog Post Properties	88
The <code>Observable</code> delegate	89
The <code>Vetoable</code> delegate	94
A map as a delegate	96
Review of how variables work	98
Summary	105
Exercise: Mutable lazy delegate	105
Kotlin Contracts	107
The meaning of a contract	108
How many times do we invoke a function from an argument?	109
Implications of the fact that a function has returned a value	113
Using contracts in practice	115
Summary	116
Exercise: Coroutine time measurement	117
Java interoperability	119
Nullable types	119
Kotlin type mapping	122
JVM primitives	123
Collection types	125
Annotation targets	129
Static elements	135
<code>JvmField</code>	137
Using Java accessors in Kotlin	139
<code>JvmName</code>	140

CONTENTS

JvmMultifileClass	144
JvmOverloads	145
Unit	148
Function types and function interfaces	149
Tricky names	152
Throws	153
JvmRecord	156
Summary	157
Exercise: Adjust Kotlin for Java usage	157
Using Kotlin Multiplatform	161
Multiplatform module configuration	161
Expect and actual elements	164
Possibilities	167
Multiplatform libraries	171
A multiplatform mobile application	179
Summary	185
Exercise: Multiplatform LocalDateTime	185
JavaScript interoperability	187
Setting up a project	188
Using libraries available for Kotlin/JS	190
Using Kotlin/JS	190
Building and linking a package	193
Distributing a package to npm	195
Exposing objects	195
Exposing Flow and StateFlow	200
Adding npm dependencies	205
Frameworks and libraries for Kotlin/JS	207
JavaScript and Kotlin/JS limitations	207
Summary	209
Exercise: Migrating a Kotlin/JVM project to KMP	209
Reflection	211
Hierarchy of classes	213
Function references	214
Parameter references	222
Property references	224
Class reference	227
Serialization example	236
Referencing types	242
Type reflection example: Random value	246

CONTENTS

Kotlin and Java reflection	252
Breaking encapsulation	253
Summary	254
Exercise: Function caller	254
Exercise: Object serialization to JSON	256
Exercise: Object serialization to XML	258
Exercise: DSL-based dependency injection library	262
Annotation processing	265
Your first annotation processor	265
Hiding generated classes	278
Summary	281
Exercise: Annotation Processing	281
Kotlin Symbol Processing	283
Your first KSP processor	284
Testing KSP	293
Dependencies and incremental processing	296
Multiple rounds processing	302
Using KSP on multiplatform projects	305
Summary	306
Exercise: Kotlin Symbol Processing	306
Kotlin Compiler Plugins	308
Compiler frontend and backend	308
Compiler extensions	310
Popular compiler plugins	314
Making all classes open	315
Changing a type	317
Generate function wrappers	319
Example plugin implementations	320
Summary	321
Static Code Analysers	322
What are Static Analysers?	323
Types of analysers	326
Kotlin Code Analysers	330
Setting up detekt	333
Writing your first detekt Rule	338
Conclusion	345
Ending	346
Exercise solutions	347

Introduction

You can be a developer - even a good one - without understanding the topics explained in this book, but at some point, you'll need it. You're likely already using tools made using features described in this book every day, such as libraries based on annotation processing or compiler plugins, classes that use variance modifiers, functions with contracts, or property delegates, but do you understand these features? Would you be able to implement similar tools yourself? Would you be able to analyze and debug them? This book will make all this possible for you. It focuses exclusively on the most advanced Kotlin topics, which are often not well understood even by senior Kotlin developers. It should equip you with the knowledge you need and show you possibilities you never before imagined. I hope you enjoy it as much as I enjoyed writing it.

Who is this book for?

This book is for experienced Kotlin developers. I assume that readers understand topics like function types and lambda expressions, collection processing, creation and usage of DSLs, and essential Kotlin types like `Any?` and `Nothing`. If you don't have enough experience, I recommend the previous books from this series: *Kotlin Essentials* and *Functional Kotlin*.

That will be covered?

The chapter titles explain what will be covered quite well, but here is a more detailed list:

- Generic variance modifiers
- The Covariant Nothing Object pattern
- Generic variance modifier limitations
- Interface delegation
- Implementing custom property delegates
- Property delegates from Kotlin stdlib

- Kotlin Contracts
- Kotlin and Java type mapping
- Annotations for Kotlin and Java interoperability
- Multiplatform development structure, concepts and possibilities
- Implementing multiplatform libraries
- Implementing Android and iOS applications with shared modules
- Essentials of Kotlin/JS
- Reflecting Kotlin elements
- Reflecting Kotlin types
- Implementing custom Annotation Processors
- Implementing custom Kotlin Symbol Processors
- KSP incremental compilation and multiple-round processing
- Defining Compiler Plugins
- Core Static Analysis concepts
- Overview of Kotlin static analyzers
- Defining custom Detekt rules

This book is full of example projects, including:

- A type-safe task update class using the Covariant Nothing Object pattern (*Generic variance modifiers chapter*)
- Logging a property delegate (*Property delegation chapter*)
- An object serializer (*Reflection chapter*)
- A random value generator for generic types (*Reflection chapter*)
- An annotation processor that generates an interface for a class (*Annotation Processing chapter*).
- A Kotlin symbol processor that generates an interface for a class (*Kotlin Symbol Processing chapter*).
- A Detekt rule that finds `System.out.println` usage.

The structure of the book

I decided that the chapters should have a flat structure; however, if you read the book from the beginning, you might notice that the order is not accidental. The first four chapters, *Generic variance modifiers*, *Interface Delegation*, *Property delegation* and *Kotlin Contracts*, explain Kotlin features that are commonly used yet are not well understood by most developers. The next three chapters, *Java interoperability*, *Using Kotlin Multiplatform* and *JavaScript interoperability*, describe the details of using different flavors of Kotlin. The rest of the book, namely the chapters *Reflection*, *Annotation Processing*, *Kotlin Symbol Processing*, *Kotlin Compiler Plugins* and *Static Code Analysers*, are about meta-programming, and each of these chapters builds on the previous ones.

The Kotlin for Developers series

This book is a part of a series of books called *Kotlin for Developers*, which includes the following books:

- *Kotlin Essentials*, which covers all the basic Kotlin features.
- *Functional Kotlin*, which is dedicated to functional Kotlin features, including function types, lambda expressions, collection processing, DSLs, and scope functions.
- *Kotlin Coroutines: Deep Dive*, which covers all the Kotlin Coroutines features, including how to use and test them, using flow, the best practices, and the most common mistakes.
- *Advanced Kotlin*, which is dedicated to advanced Kotlin features, including generic variance modifiers, delegation, multiplatform programming, annotation processing, KSP and compiler plugins.
- *Effective Kotlin: Best Practices*, which is dedicated to the best practices of Kotlin programming.

Do not worry, you do not need to read the previous books in the series to understand this one. However, if you are interested in learning more about Kotlin, I recommend considering the other books from this series.

Conventions

When I refer to a concrete element from code, I will use code-font. To name a concept, I will capitalize the word. To reference an arbitrary element of some type, I will not capitalize the word. For example:

- `Flow` is a type or an interface, so it's printed in code-font (as in "Function needs to return `Flow`"),
- `Flow` represents a concept, so it is capitalized (as in "This explains the essential difference between `Channel` and `Flow`"),
- a `flow` is an instance, like a list or a set, which is why it is not capitalized ("Every `flow` consists of a few elements").

Another example: `List` refers concretely to a list interface or type ("The type of `l` is `List`"), while `List` represents a concept, and a list is one of many lists (the `list` variable holds a list).

Code conventions

Most of the presented snippets are executable code with no import statements. In the online version of this book on the Kt. Academy website, most snippets can be executed so readers can play with the code.

Snippet results are presented using the `println` function. The result will often be placed in comments after the statement that prints it.

```
import kotlin.reflect.KType
import kotlin.reflect.typeOf

fun main() {
    val t1: KType = typeOf<Int?>()
    println(t1) // kotlin.Int?
    val t2: KType = typeOf<List<Int?>>()
    println(t2) // kotlin.collections.List<kotlin.Int?>
    val t3: KType = typeOf<() -> Map<Int, Char?>>()
    println(t3)
    // () -> kotlin.collections.Map<kotlin.Int, kotlin.Char?>
}
```

In some snippets, you might notice strange formatting. This is because the line length in this book is only 62 characters, so I adjusted the formatting to fit the page width. This is, for instance, why the result of `println(t3)` in the above example is located on the next line.

Sometimes, some parts of code or results are shortened with In such cases, you can read it as “there should be more here, but the author decided to omit it”.

```
class A {
    val b by lazy { B() }
    val c by lazy { C() }
    val d by lazy { D() }

    // ...
}
```

Acknowledgments



Owen Griffiths has been developing software since the mid-1990s and remembers the productivity of languages such as Clipper and Borland Delphi. Since 2001, he's focused on web, server-based Java, and the open-source revolution. With many years of commercial Java experience, he picked up Kotlin in early 2015. After taking detours into Clojure and Scala, he - like Goldilocks - thinks Kotlin is just right and tastes the best. Owen enthusiastically helps Kotlin developers continue to succeed.



Nicola Corti is a Google Developer Expert for Kotlin. He's been working with this language since before version 1.0 and is the maintainer of several open-source libraries and tools for mobile developers (Detekt, Chucker, AppIntro). He's currently working in the React Native core team at Meta, building one of the most popular cross-platform mobile frameworks, and he's an active member of the developer community. His involvement goes from speaking at international conferences to being a member of CFP committees and supporting developer communities across Europe. In his free time, he also loves baking, podcasting, and running.



Matthias Schenk started his career with Java over ten years ago, mainly in the Spring/Spring Boot Ecosystem. Eighteen months ago, he switched to Kotlin and has since become a big fan of working with native Kotlin frameworks like Koin, Ktor, and Exposed.

Jacek Kotorowicz graduated from UMCS and is now an Android developer based in Lublin. He wrote his Master's thesis in C++ in Vim and LaTeX. Later, he found himself in a love-hate relationship with JVM languages and the Android platform. He first used Kotlin (or, at least, tried to) before version 1.0. He's still learning how NOT to be a perfectionist and how to find time for learning and hobbies.

Endre Deak is a software architect building AI infrastructure at Disco, a market-leading legal tech company. He has 15 years of experience building complex scalable systems, and he thinks Kotlin is one of the best programming languages ever created.

I would also like to thank **Michael Timberlake**, our language reviewer, for his excellent corrections to the whole book.

Generic variance modifiers

Let's say that `Puppy` is a subtype of `Dog`, and you have a generic `Box` class to enclose them both. The question is: what is the relation between the `Box<Puppy>` and `Box<Dog>` types? In other words, can we use `Box<Puppy>` where `Box<Dog>` is expected, or vice versa? To answer these questions, we need to know what the variance modifier of this class type parameter is¹.

When a type parameter has no variance modifier (no `out` or `in` modifier), we say it is invariant and thus expects an exact type. So, if we have `class Box<T>`, then there is no relation between `Box<Puppy>` and `Box<Dog>`.

```
class Box<T>
open class Dog
class Puppy : Dog()

fun main() {
    val d: Dog = Puppy() // Puppy is a subtype of Dog

    val bd: Box<Dog> = Box<Puppy>() // Error: Type mismatch
    val bp: Box<Puppy> = Box<Dog>() // Error: Type mismatch

    val bn: Box<Number> = Box<Int>() // Error: Type mismatch
    val bi: Box<Int> = Box<Number>() // Error: Type mismatch
}
```

Variance modifiers determine what the relationship should be between `Box<Puppy>` and `Box<Dog>`. When we use the `out`

¹In this chapter, I assume that you know what a type is and understand the basics of generic classes and functions. As a reminder, the type parameter is a placeholder for a type, e.g., `T` in `class Box<T>` or `fun a<T>() {}`. The type argument is the actual type used when a class is created or a function is called, e.g., `Int` in `Box<Int>()` or `a<Int>()`. A type is not the same as a class. For a class `User`, there are at least two types: `User` and `User?`. For a generic class, there are many types, like `Box<Int>`, and `Box<String>`.

modifier, we make a **covariant** type parameter. When A is a subtype of B, the Box type parameter is covariant (`out` modifier) and the `Box<A>` type is a subtype of `Box`. So, in our example, for class `Box<out T>`, the `Box<Puppy>` type is a subtype of `Box<Dog>`.

```
class Box<out T>
open class Dog
class Puppy : Dog()

fun main() {
    val d: Dog = Puppy() // Puppy is a subtype of Dog

    val bd: Box<Dog> = Box<Puppy>() // OK
    val bp: Box<Puppy> = Box<Dog>() // Error: Type mismatch

    val bn: Box<Number> = Box<Int>() // OK
    val bi: Box<Int> = Box<Number>() // Error: Type mismatch
}
```

When we use the `in` modifier, we make a **contravariant** type parameter. When A is a subtype of B and the Box type parameter is contravariant (`in` modifier), then type `Box` is a subtype of `Box<A>`. So, in our example, for class `Box<in T>` the `Box<Dog>` type is a subtype of `Box<Puppy>`.

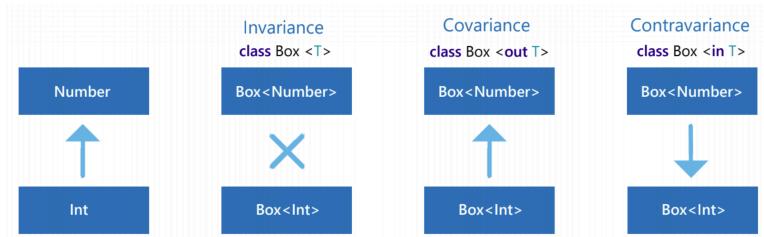
```
class Box<in T>
open class Dog
class Puppy : Dog()

fun main() {
    val d: Dog = Puppy() // Puppy is a subtype of Dog

    val bd: Box<Dog> = Box<Puppy>() // Error: Type mismatch
    val bp: Box<Puppy> = Box<Dog>() // OK

    val bn: Box<Number> = Box<Int>() // Error: Type mismatch
    val bi: Box<Int> = Box<Number>() // OK
}
```

These variance modifiers are illustrated in the diagram below:



At this point, you might be wondering how these variance modifiers are useful. In particular, contravariance might sound strange to you, so let me show you some examples.

List variance

Let's consider that you have the type `Animal` and its subclass `Cat`. You also have the standalone function `petAnimals`, which you use to pet all your animals when you get back home. You also have a list of cats that is of type `List<Cat>`. The question is: can you use your list of cats as an argument to the function `petAnimals`, which expects a list of animals?

```
interface Animal {
    fun pet()
}

class Cat(val name: String) : Animal {
    override fun pet() {
        println("$name says Meow")
    }
}

fun petAnimals(animals: List<Animal>) {
    for (animal in animals) {
        animal.pet()
    }
}
```

```
    }  
}  
  
fun main() {  
    val cats: List<Cat> =  
        listOf(Cat("Mruczek"), Cat("Puszek"))  
    petAnimals(cats) // Can I do that?  
}
```

The answer is YES. Why? Because in Kotlin, the `List` interface type parameter is covariant, so it has the `out` modifier, which is why `List<Cat>` can be used where `List<Animal>` is expected.

A generic ordered collection of elements. Methods in this interface support only read-access; read/write access is supported through the `MutableList` interface.

Params: `E` - the type of elements contained in the list. The list is covariant in its elements.

```
public interface List<out E> : Collection<E> {  
    // Query Operations  
  
    override val size: Int  
    override fun isEmpty(): Boolean
```

Covariance (`out`) is a proper variance modifier because `List` is read-only. Covariance can't be used for a mutable data structure. The `MutableList` interface has an invariant type parameter, so it has no variance modifier.

A generic ordered collection of elements that supports adding and removing elements.

Params: `E` - the type of elements contained in the list. The mutable list is invariant in its elements.

```
public interface MutableList<E> : List<E>, MutableCollection<E> {  
    // Modification Operations  
  
    override fun add(element: E): Boolean
```

Thus, `MutableList<Cat>` cannot be used where `MutableList<Animal>` is expected. There are good reasons for this which we will explore when we discuss the safety of variance modifiers. For now, I will just show you an example of what might go wrong if `MutableList` were covariant: we could use `MutableList<Cat>` where `MutableList<Animal>` is expected and then use this reference to add `Dog` to our list of cats. Someone would be really surprised to find a dog in a list of cats.

```
interface Animal
class Cat(val name: String) : Animal
class Dog(val name: String) : Animal

fun addAnimal(animals: MutableList<Animal>) {
    animals.add(Dog("Cookie"))
}

fun main() {
    val cats: MutableList<Cat> =
        mutableListOf(Cat("Mruczek"), Cat("Puszek"))
    addAnimal(cats) // COMPILATION ERROR
    val cat: Cat = cats.last()
    // If code would compile, it would break here
}
```

This illustrates why covariance, as its name `out` suggests, is appropriate for types that are only exposed and only go out of an object but never go in. So, covariance should be used for immutable classes.

Consumer variance

Let's say that you have a class that can be used to send messages of a certain type.

```

interface Sender<T : Message> {
    fun send(message: T)
}

interface Message

interface OrderManagerMessage : Message
class AddOrder(val order: Order) : OrderManagerMessage
class CancelOrder(val orderId: String) : OrderManagerMessage

interface InvoiceManagerMessage : Message
class MakeInvoice(val order: Order) : OrderManagerMessage

```

Now, you've made a class called `GeneralSender` that is capable of sending any kind of message. The question is: can you use `GeneralSender` where a class for sending some specific kind of messages is expected? You should be able to! If `GeneralSender` can send all kinds of messages, it should be able to send specific message types as well.

```

class GeneralSender(
    serviceUrl: String
) : Sender<Message> {
    private val connection = makeConnection(serviceUrl)

    override fun send(message: Message) {
        connection.send(message.toApi())
    }
}

val orderManagerSender: Sender<OrderManagerMessage> =
    GeneralSender(ORDER_MANAGER_URL)

val invoiceManagerSender: Sender<InvoiceManagerMessage> =
    GeneralSender(INVOICE_MANAGER_URL)

```

For a sender of any message to be a sender of some specific message type, we need the sender type to have a contravariant parameter, therefore it needs the `in` modifier.

```
interface Sender<in T : Message> {
    fun send(message: T)
}
```

Let's generalize this and consider a class that consumes objects of a certain type. If a class declares that it consumes objects of type `Number`, we can assume it can consume objects of type `Int` or `Float`. If a class consumes anything, it should consume strings or chars, therefore its type parameter, which represents the type this class consumes, must be contravariant, so use the `in` modifier.

```
class Consumer<in T> {
    fun consume(value: T) {
        println("Consuming $value")
    }
}

fun main() {
    val numberConsumer: Consumer<Number> = Consumer()
    numberConsumer.consume(2.71) // Consuming 2.71
    val intConsumer: Consumer<Int> = numberConsumer
    intConsumer.consume(42) // Consuming 42
    val floatConsumer: Consumer<Float> = numberConsumer
    floatConsumer.consume(3.14F) // Consuming 3.14

    val anyConsumer: Consumer<Any> = Consumer()
    anyConsumer.consume(123456789L) // Consuming 123456789
    val stringConsumer: Consumer<String> = anyConsumer
    stringConsumer.consume("ABC") // Consuming ABC
    val charConsumer: Consumer<Char> = anyConsumer
    charConsumer.consume('M') // Consuming M
}
```

It makes a lot of sense to use contravariance for the consumer or sender values as both their type parameters are only used in the `in`-position as argument types, so covariant type values are only consumed. I hope you're starting to see that the `out`

modifier is only appropriate for type parameters that are in the out-position and are thus used as a result type or a read-only property type. On the other hand, the `in` modifier is only appropriate for type parameters that are in the in-position and are thus used as parameter types.

Function types

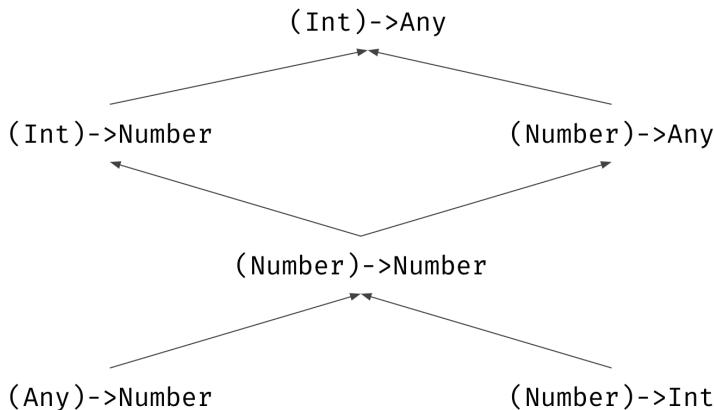
In function types, there are relations between function types with different parameters and result types. To see this in practice, think of a function that as an argument expects a function that accepts an `Int` and returns an `Any`:

```
fun printProcessedNumber(transformation: (Int) -> Any) {  
    println(transformation(42))  
}
```

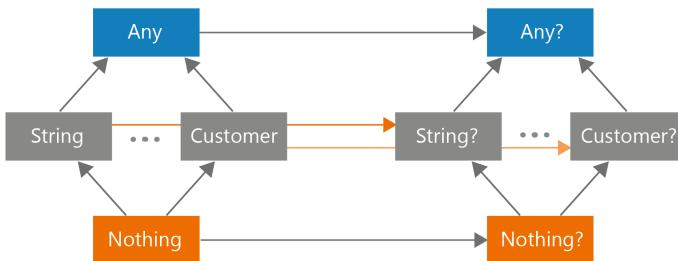
Based on its definition, such a function can accept a function of type `(Int)->Any`, but it would work with `(Int)->Number`, `(Number)->Any`, `(Number)->Number`, `(Any)->Number`, `(Number)->Int`, etc.

```
val inttoDouble: (Int) -> Number = { it.toDouble() }  
val numberAsText: (Number) -> String = { it.toString() }  
val identity: (Number) -> Number = { it }  
val numberToInt: (Number) -> Int = { it.toInt() }  
val numberHash: (Any) -> Number = { it.hashCode() }  
  
printProcessedNumber(inttoDouble)  
printProcessedNumber(numberAsText)  
printProcessedNumber(identity)  
printProcessedNumber(numberToInt)  
printProcessedNumber(numberHash)
```

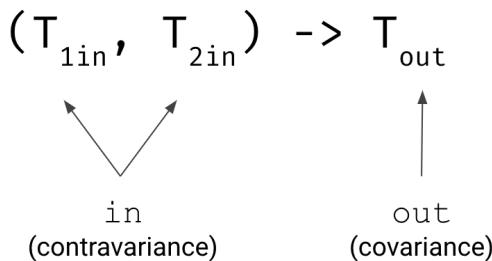
This is because there is the following relation between all these types:



Notice that when we go down in this hierarchy, the parameter type moves toward types that are higher in the typing system hierarchy, and the return type moves toward lower types.



This is no coincidence. All parameter types in Kotlin function types are contravariant, as the name of the `in` variance modifier suggests. All return types in Kotlin function types are covariant, as the name of the `out` variance modifier suggests.



In this case – as in many other cases – you don't need to understand variance modifiers to benefit from using them. You just use the function you would like to use, and it works. People rarely notice that this would not work in another language or with another implementation. This makes a good developer experience. People don't attribute this good experience to generic type modifiers, but they feel that using Kotlin or some libraries is just easier. As library creators, we use type modifiers to make a good developer experience.

The general rule for using variance modifiers is really simple: type parameters that are only used for public out-positions (function results and read-only property types) should be covariant so they have an `out` modifier. Type parameters that are only used for public in-positions (function parameter types) should be contravariant so they have an `in` modifier.

The Covariant Nothing Object

Consider that you need to define a linked list data structure, which is a type of collection constructed by two types:

- node, which represents a linked list with at least one element and includes a reference to the first element (head) and a reference to the rest of the elements (tail).
- empty, which represents an empty linked list.

In Kotlin, we would represent such a data structure with a sealed class.

```

sealed class LinkedList<T>
data class Node<T>(
    val head: T,
    val tail: LinkedList<T>
) : LinkedList<T>()
class Empty<T> : LinkedList<T>()

fun main() {
    val strs = Node("A", Node("B", Empty()))
    val ints = Node(1, Node(2, Empty()))
    val empty: LinkedList<Char> = Empty()
}

```

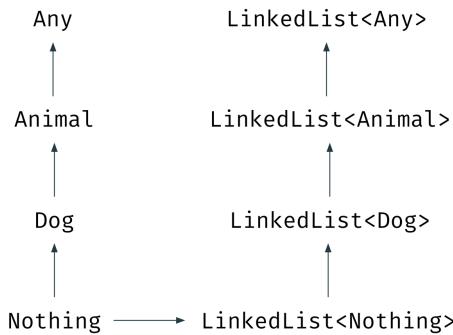
There is one problem though: for every linked list, we need a new object to represent the empty list. Every time we use `Empty()`, we create a new instance. We would prefer to have only one instance that would serve wherever we need to represent an empty linked list. For that, we use object declaration in Kotlin, but object declarations cannot have type parameters.

```

sealed class LinkedList<T>
data class Node<T>(
    val head: T,
    val tail: LinkedList<T>
) : LinkedList<T>()
object Empty<T> : LinkedList<T>() // Error

```

There is a solution to this problem. We can make the `LinkedList` type parameter covariant (by adding the `out` modifier). This is perfectly fine for a type that is only returned, i.e., for all type parameters in immutable classes. Then, we should make our `Empty` object extend `LinkedList<Nothing>`. The `Nothing` type is a subtype of all types; so, if the `LinkedList` type parameter is covariant, then `LinkedList<Nothing>` is a subtype of all linked lists.



```

sealed class LinkedList<out T>
data class Node<T>(
    val head: T,
    val tail: LinkedList<T>
) : LinkedList<T>()
object Empty : LinkedList<Nothing>()

fun main() {
    val strs = Node("A", Node("B", Empty))
    val ints = Node(1, Node(2, Empty))
    val empty: LinkedList<Char> = Empty
}
  
```

This pattern is used in many places, even in the Kotlin Standard Library. As you already know, `List` is covariant because it is read-only. When you create a list using `listOf` or `emptyList`, they both return the same object, `EmptyList`, which implements `List<Nothing>`, therefore `EmptyList` is a subtype of all lists.

```
// ...
public fun <T> emptyList(): List<T> = EmptyList

// ...
public inline fun <T> listOf(): List<T> = emptyList()

internal object EmptyList : List<Nothing>, Serializable, RandomAccess {
    private const val serialVersionUID: Long = -7390468764508069838L

    override fun equals(other: Any?): Boolean = other is List<*> && other.isEmpty()
    override fun hashCode(): Int = 1
    override fun toString(): String = "[]"

    override val size: Int get() = 0
    override fun isEmpty(): Boolean = true
    override fun contains(element: Nothing): Boolean = false
    override fun containsAll(elements: Collection<Nothing>): Boolean = elements.isEmpty()
}
```

Every empty list created with the `listOf` or `emptyList` functions from Kotlin stdlib is actually the same object.

```
fun main() {
    val empty: List<Nothing> = emptyList()
    val strs: List<String> = empty
    val ints: List<Int> = empty

    val other: List<Char> = emptyList()
    println(empty === other) // true
}
```

This pattern occurs in many places; for instance, when we define generic messages and some of them don't need to include any parameters, we should make them objects.

```
sealed interface ChangesTrackerMessage<out T>
class Change<T>(val newValue: T) : ChangesTrackerMessage<T>
object Reset : ChangesTrackerMessage<Nothing>
object UndoChange : ChangesTrackerMessage<Nothing>

sealed interface TaskSchedulerMessage<out T>

class Schedule<T>(
    val task: Task<T>
): TaskSchedulerMessage<T>
```

```
class Update<T>(
    val taskUpdate: TaskUpdate<T>
) : TaskSchedulerMessage<T>

class Delete(
    val taskId: String
) : TaskSchedulerMessage<Nothing>

object StartScheduled : TaskSchedulerMessage<Nothing>

object Reset : TaskSchedulerMessage<Nothing>
```

Even though this pattern repeats in Kotlin projects, I couldn't find a name that would describe it. I decided to name it the "Covariant Nothing Object". This is not a precise name; a precise description would be "pattern in which the object declaration implements a generic class or interface with the Nothing type argument used in the covariant type argument position". Nevertheless, the name needs to be short, and "Covariant Nothing Object" is clear and catchy.

Another example of a Covariant Nothing Object comes from a library my team co-created. It was used to schedule tasks in a microservice environment. For simplicity, you could assume that each task can be modeled as follows:

```
class Task<T>(
    val id: String,
    val scheduleAt: Instant,
    val data: T,
    val priority: Int,
    val maxRetries: Int? = null
)
```

We needed to implement a mechanism to change scheduled tasks. We also needed to represent change within a configuration in order to define updates and pass them around conveniently. The old-school approach is to make a `TaskUpdate` class

which uses `null` as a marker which indicates that a specific property should not change.

```
class TaskUpdate<T>(
    val id: String? = null,
    val scheduleAt: Instant? = null,
    val data: T? = null,
    val priority: Int? = null,
    val maxRetries: Int? = null
)
```

This approach is very limiting. Since the `null` value is interpreted as “do not change this property”, there is no way to express that you want to set a particular value to `null`. Instead, we used a **Covariant Nothing Object** in our project to represent a property change. Each property might be either kept unchanged or changed to a new value. We can represent these two options with a sealed hierarchy, and thanks to generic types we might expect specific types of values.

```
class TaskUpdate<T>(
    val id: TaskPropertyUpdate<String> = Keep,
    val scheduleAt: TaskPropertyUpdate<Instant> = Keep,
    val data: TaskPropertyUpdate<T> = Keep,
    val priority: TaskPropertyUpdate<Int> = Keep,
    val maxRetries: TaskPropertyUpdate<Int?> = Keep
)

sealed interface TaskPropertyUpdate<out T>
object Keep : TaskPropertyUpdate<Nothing>
class ChangeTo<T>(val newValue: T) : TaskPropertyUpdate<T>

val update = TaskUpdate<String>(
    id = ChangeTo("456"),
    maxRetries = ChangeTo(null), // we can change to null

    data = ChangeTo(123), // COMPILE TIME ERROR
    // type mismatch, expecting String
```

```

    priority = ChangeTo(null), // COMPILE TIME ERROR
    // type mismatch, property is not nullable
)

```

This way, we achieved a type-safe and expressive way of representing task changes. What is more, when we use the Covariant Nothing Object pattern, we can easily express other kinds of changes as well. For instance, if our library supports default values or allows a previous value to be restored, we could add new objects to represent these property changes.

```

class TaskUpdate<T>(
    val id: TaskPropertyUpdate<String> = Keep,
    val scheduleAt: TaskPropertyUpdate<Instant> = Keep,
    val data: TaskPropertyUpdate<T> = Keep,
    val priority: TaskPropertyUpdate<Int> = Keep,
    val maxRetries: TaskPropertyUpdate<Int?> = Keep
)

sealed interface TaskPropertyUpdate<out T>
object Keep : TaskPropertyUpdate<Nothing>
class ChangeTo<T>(val newValue: T) : TaskPropertyUpdate<T>
object RestorePrevious : TaskPropertyUpdate<Nothing>
object RestoreDefault : TaskPropertyUpdate<Nothing>

val update = TaskUpdate<String>(
    data = ChangeTo("ABC"),

    maxRetries = RestorePrevious,
    priority = RestoreDefault,
)

```

The Covariant Nothing Class

There are also cases where we want a **class** to implement a class or interface which uses the `Nothing` type argument as a covariant type parameter. This is a pattern I call the **Covariant Nothing Class**. For example, consider the `Either`

class, which can be either `Left` or `Right` and must have two type parameters that specify what data types it expects on the `Left` and on the `Right`. However, both `Left` and `Right` should each have only one type parameter to specify what type they expect. To make this work, we need to fill the missing type argument with `Nothing`.

```
sealed class Either<out L, out R>
class Left<out L>(val value: L) : Either<L, Nothing>()
class Right<out R>(val value: R) : Either<Nothing, R>()
```

With such definitions, we can create `Left` or `Right` without specifying type arguments.

```
val left = Left(Error())
val right = Right("ABC")
```

Both `Left` and `Right` can be up-casted to `Left` and `Right` with supertypes of the types of values they hold.

```
val leftError: Left<Error> = Left(Error())
val leftThrowable: Left<Throwable> = leftError
val leftAny: Left<Any> = leftThrowable

val rightInt = Right(123)
val rightNumber: Right<Number> = rightInt
val rightAny: Right<Any> = rightNumber
```

They can also be used wherever a result with the appropriate `Left` or `Right` type is expected.

```
val leftError: Left<Error> = Left(Error())
val rightInt = Right(123)

val el: Either<Error, Int> = leftError
val er: Either<Error, Int> = rightInt

val etnl: Either<Throwable, Number> = leftError
val etnr: Either<Throwable, Number> = rightInt
```

This, in simplification, is how `Either` is implemented in the Arrow library.

Variance modifier limitations

In Java, arrays are reified and covariant. Some sources state that the reason behind this decision was to make it possible to create functions like `Arrays::sort` that make generic operations on arrays of every type.

```
Integer[] numbers= {1, 4, 2, 3};
Arrays.sort(numbers); // sorts numbers

String[] lettrs= {"B", "C", "A"};
Arrays.sort(lettrs); // sorts letters
```

However, there is a big problem with this decision. To understand it, let's analyze the following Java operations, which produce no compilation time errors but throw runtime errors:

```
// Java
Integer[] numbers= {1, 4, 2, 3};
Object[] objects = numbers;
objects[2] = "B"; // Runtime error: ArrayStoreException
```

As you can see, casting `numbers` to `Object[]` didn't change the actual type used inside the structure (it is still `Integer`); so,

when we try to assign a value of type `String` to this array, an error occurs. This is clearly a Java flaw, but Kotlin protects us from it by making `Array` (as well as `IntArray`, `CharArray`, etc.) invariant (so upcasting from `Array<Int>` to `Array<Any>` is not possible).

To understand what went wrong in the above snippet, we should understand what in-positions and out-positions are.

A type is used in an in-position when it is used as a parameter type. In the example below, the `Dog` type is used in an in-position. Note that every object type can be up-casted; so, when we expect a `Dog`, we might actually receive any of its subtypes, e.g., a `Puppy` or a `Hound`.

```
open class Dog
class Puppy : Dog()
class Hound : Dog()

fun takeDog(dog: Dog) {}

takeDog(Dog())
takeDog(Puppy())
takeDog(Hound())
```

In-positions work well with contravariant types, including the `in` modifier, because they allow a type to be transferred to a lower one, e.g., from `Dog` to `Puppy` or `Hound`. This only limits class use, so it is a safe operation.

```
open class Dog
class Puppy : Dog()
class Hound : Dog()

class Box<in T> {
    private var value: T? = null

    fun put(value: T) {
        this.value = value
    }
}
```

```

        }
    }

fun main() {
    val dogBox = Box<Dog>()
    dogBox.put(Dog())
    dogBox.put(Puppy())
    dogBox.put(Hound())

    val puppyBox: Box<Puppy> = dogBox
    puppyBox.put(Puppy())

    val houndBox: Box<Hound> = dogBox
    houndBox.put(Hound())
}

```

However, public in-positions cannot be used with covariance, including the `out` modifier. Just think what would happen if you could upcast `Box<Dog>` to `Box<Any?>`. If this were possible, you could literally pass any object to the `put` method. Can you see the implications of this? That is why it is prohibited in Kotlin to use a covariant type (`out` modifier) in public in-positions.

```

class Box<out T> {
    private var value: T? = null

    fun set(value: T) { // Compilation Error
        this.value = value
    }

    fun get(): T = value ?: error("Value not set")
}

val dogHouse = Box<Dog>()
val box: Box<Any> = dogHouse
box.set("Some string")
// Is this were possible, we would have runtime error here

```

This is actually the problem with Java arrays. They should not be covariant because they have methods, like `set`, that allow their modification.

Covariant type parameters can be safely used in private in-positions.

```
class Box<out T> {
    private var value: T? = null

    private fun set(value: T) { // OK
        this.value = value
    }

    fun get(): T = value ?: error("Value not set")
}
```

Covariance (`out` modifier) is perfectly safe with public out-positions, therefore these positions are not limited. This is why we use covariance (`out` modifier) for types that are produced or only exposed, and the `out` modifier is often used for producers or immutable data holders. Thus, `List` has the covariant type parameter, but `MutableList` must have the invariant type parameter.

There is also a symmetrical problem (or co-problem, as some like to say) for contravariance and out-positions. Types in out-positions are function result types and read-only property types. These types can also be up-casted to any upper type; however, since we are on the other side of an object, we can expect types that are above the expected type. In the example below, `Amphibious` is in an out-position; when we might expect it to be `Amphibious`, we can also expect it to be `Car` or `Boat`.

```

open class Car
interface Boat
class Amphibious : Car(), Boat

fun getAmphibious(): Amphibious = Amphibious()

val amphibious: Amphibious = getAmphibious()
val car: Car = getAmphibious()
val boat: Boat = getAmphibious()

```

Out positions work well with covariance, i.e., the `out` modifier. Upcasting `Producer<Amphibious>` to `Producer<Car>` or `Producer<Boat>` limits what we can expect from the `produce` method, but the result is still correct.

```

open class Car
interface Boat
class Amphibious : Car(), Boat

class Producer<out T>(val factory: () -> T) {
    fun produce(): T = factory()
}

fun main() {
    val producer: Producer<Amphibious> =
        Producer { Amphibious() }
    val amphibious: Amphibious = producer.produce()
    val boat: Boat = producer.produce()
    val car: Car = producer.produce()

    val boatProducer: Producer<Boat> = producer
    val boat1: Boat = boatProducer.produce()

    val carProducer: Producer<Car> = producer
    val car2: Car = carProducer.produce()
}

```

Out-positions do not get along with contravariant type parameters (`in` modifier). If `Producer` type parameters

were contravariant, we could up-cast `Producer<Amphibious>` to `Producer<Nothing>` and then expect `produce` to produce literally anything, which this method cannot do. That is why contravariant type parameters cannot be used in public out-positions.

```
open class Car
interface Boat
class Amphibious : Car(), Boat

class Producer<in T>(val factory: () -> T) {
    fun produce(): T = factory() // Compilation Error
}

fun main() {
    val carProducer = Producer<Amphibious> { Car() }
    val amphibiousProducer: Producer<Amphibious> = carProducer
    val amphibious = amphibiousProducer.produce()
    // If not compilation error, we would have runtime error

    val producer = Producer<Amphibious> { Amphibious() }
    val nothingProducer: Producer<Nothing> = producer
    val str: String = nothingProducer.produce()
    // If not compilation error, we would have runtime error
}
```

You cannot use contravariant type parameters (`in` modifier) in public out-positions, such as a function result or a read-only property type.

```
class Box<in T>(
    val value: T // Compilation Error
) {

    fun get(): T = value // Compilation Error
        ?: error("Value not set")
}
```

Again, it is fine when these elements are private:

```
class Box<in T>(
    private val value: T
) {

    private fun get(): T = value
        ?: error("Value not set")
}
```

This way, we use contravariance (`in` modifier) for type parameters that are only consumed or accepted. A well-known example is `kotlin.coroutines.Continuation`:

```
public interface Continuation<in T> {
    public val context: CoroutineContext
    public fun resumeWith(result: Result<T>)
}
```

Read-write property types are invariant, so public read-write properties support neither covariant nor contravariant types.

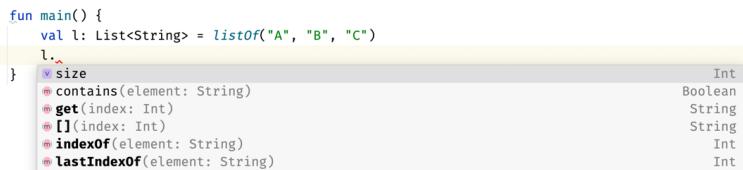
```
class Box<in T1, out T2> {
    var v1: T1 // Compilation error
    var v2: T2 // Compilation error
}
```

UnsafeVariance annotation

Every good rule must have some exceptions. In general, using covariant type parameters (`out` modifier) in public in-positions is considered unsafe, therefore such a situation blocks code compilation. Still, there are situations where we would like to do this anyway because we know we will do it safely. A good example is `List`.

As we have already explained, the type parameter in the `List` interface is covariant (`out` modifier), and this is conceptually correct because it is a read-only interface. However, it uses

this type of parameter in some public in-positions. Just consider the `contains` or `indexOf` methods: they use covariant type parameters in a public in-position, which is a clear violation of the rules we just explained.



How is that possible? According to the previous section, it should not be possible. The answer is the `@UnsafeVariance` annotation, which is used to turn off the aforementioned limitations. It is like saying, “I know it is unsafe, but I know what I’m doing and I will use this type safely”.

```
public interface List<out E> : Collection<E> {
    // Query Operations

    override val size: Int
    override fun isEmpty(): Boolean
    override fun contains(element: @UnsafeVariance E): Boolean
    override fun iterator(): Iterator<E>

    // Bulk Operations
    override fun containsAll(elements: Collection<@UnsafeVariance E>): Boolean

    // Positional Access Operations
    // Returns the element at the specified index in the list.
    public operator fun get(index: Int): E

    // Search Operations
    // Returns the index of the first occurrence of the specified element in the list, or -1 if the specified
    // element is not contained in the list.
    public fun indexOf(element: @UnsafeVariance E): Int

    // Returns the index of the last occurrence of the specified element in the list, or -1 if the specified
    // element is not contained in the list.
    public fun lastIndexOf(element: @UnsafeVariance E): Int
```

It is ok to use `@UnsafeVariance` for methods like `contains` or `indexOf` because their parameters are only used for comparison, and their arguments are not set anywhere or returned by any public functions. They could also be of type `Any?`, and

the type of those parameters is only specified so that a user of these methods knows what kind of value should be used as an argument.

Variance modifier positions

Variance modifiers can be used in two positions². The first one, the declaration side, is more common and is a modifier on the class or interface declaration. It will affect all the places where the class or interface is used.

```
// Declaration-side variance modifier
class Box<out T>(val value: T)

val boxStr: Box<String> = Box("Str")
val boxAny: Box<Any> = boxStr
```

The other position is the use-site, which is a variance modifier for a particular variable.

```
class Box<T>(val value: T)

val boxStr: Box<String> = Box("Str")
// Use-site variance modifier
val boxAny: Box<out Any> = boxStr
```

We use use-site variance when, for some reason, we cannot provide variance modifiers for all the types generated by a class or an interface, yet we need some variance for one specific type. For instance, `MutableList` cannot have the `in` modifier because then its method's result types would return `Any?` instead of the actual element type. Still, for a single parameter type we can make its type contravariant (`in` modifier) to allow any collections that can accept a type:

²This is also called mixed-site variance.

```
interface Dog
interface Pet
data class Puppy(val name: String) : Dog, Pet
data class Wolf(val name: String) : Dog
data class Cat(val name: String) : Pet

fun fillWithPuppies(list: MutableList<in Puppy>) {
    list.add(Puppy("Jim"))
    list.add(Puppy("Beam"))
}

fun main() {
    val dogs = mutableListOf<Dog>(Wolf("Pluto"))
    fillWithPuppies(dogs)
    println(dogs)
    // [Wolf(name=Pluto), Puppy(name=Jim), Puppy(name=Beam)]

    val pets = mutableListOf<Pet>(Cat("Felix"))
    fillWithPuppies(pets)
    println(pets)
    // [Cat(name=Felix), Puppy(name=Jim), Puppy(name=Beam)]
}
```

Note that some positions are limited when we use variance modifiers. When we have `MutableList<out T>`, we can use `get` to get elements, and we receive an instance typed as `T`, but we cannot use `set` because it expects us to pass an argument of type `Nothing`. This is because a list with any subtype of `T` might be passed there, including the subtype of every type that is `Nothing`. When we use `MutableList<in T>`, we can use both `get` and `set`; however, when we use `get`, the returned type is `Any?` because there might be a list with any supertype of `T`, including the supertype of every type that is `Any?`. Therefore, we can freely use `out` when we only read from a generic object, and we can freely use `in` when we only modify that generic object.

Star projection

On the use-site, we can also use the star `*` instead of a type argument to signal that it can be any type. This is known as **star projection**.

```
if (value is List<*>) {
    ...
}
```

Star projection should not be confused with the `Any?` type. It is true that `List<*>` effectively behaves like `List<Any?>`, but this is only because the associated type parameter is covariant. It might also be said that `Consumer<*>` behaves like `Consumer<Nothing>` if the `Consumer` type parameter is contravariant. However, the behavior of `Consumer<*>` is nothing like `Consumer<Any?>`, and the behavior of `List<*>` is nothing like `List<Nothing>`. The most interesting case is `MutableList`. As you might guess, `MutableList<Any?>` returns `Any?` as a result in methods like `get` or `removeAt`, but it also expects `Any?` as an argument for methods like `add` or `set`. On the other hand, `MutableList<*>` returns `Any?` as a result in methods like `get` or `removeAt`, but it expects `Nothing` as an argument for methods like `add` or `set`. This means `MutableList<*>` can return anything but accepts (literally) nothing.

Use-site type	In-position type	Out-position type
<code>T</code>	<code>T</code>	<code>T</code>
<code>out T</code>	<code>Nothing</code>	<code>T</code>
<code>in T</code>	<code>T</code>	<code>Any?</code>
<code>*</code>	<code>Nothing</code>	<code>Any?</code>

Summary

Every Kotlin type parameter has some variance:

- The default variance behavior of a type parameter is invariance. If, in `Box<T>`, type parameter `T` is invariant and `A` is a subtype of `B`, then there is no relation between `Box<A>` and `Box`.
- The `out` modifier makes a type parameter covariant. If, in `Box<T>`, type parameter `T` is covariant and `A` is a subtype of `B`, then `Box<A>` is a subtype of `Box`. Covariant types can be used in public out-positions.
- The `in` modifier makes a type parameter contravariant. If, in `Box<T>`, type parameter `T` is contravariant and `A` is a subtype of `B`, then `Cup` is a subtype of `Cup<A>`. Contravariant types can be used in public in-positions.

In Kotlin, it is also good to know that:

- Type parameters of `List` and `Set` are covariant (`out` modifier). So, for instance, we can pass any `List` where `List<Any>` is expected. Also, the type parameter representing the value type in `Map` is covariant (`out` modifier). Type parameters of `Array`, `MutableList`, `MutableSet`, and `MutableMap` are invariant (no variance modifier).
- In function types, parameter types are contravariant (`in` modifier), and the return type is covariant (`out` modifier).
- We use covariance (`out` modifier) for types that are only returned (produced or exposed).
- We use contravariance (`in` modifier) for types that are only accepted (consumed or set).

Exercise: Generic types usage

The below code will not compile due to the type mismatch.
Which lines will show compilation errors?

```
fun takeIntList(list: List<Int>) {}
takeIntList(listOf<Any>())
takeIntList(listOf<Nothing>())

fun takeIntMutableList(list: MutableList<Int>) {}
takeIntMutableList(mutableListOf<Any>())
takeIntMutableList(mutableListOf<Nothing>())

fun takeAnyList(list: List<Any>) {}
takeAnyList(listOf<Int>())
takeAnyList(listOf<Nothing>())

class BoxOut<out T>
fun takeBoxOutInt(box: BoxOut<Int>) {}
takeBoxOutInt(BoxOut<Int>())
takeBoxOutInt(BoxOut<Number>())
```

```

takeBoxOutInt(BoxOut<Nothing>())

fun takeBoxOutNumber(box: BoxOut<Number>) {}
takeBoxOutNumber(BoxOut<Int>())
takeBoxOutNumber(BoxOut<Number>())
takeBoxOutNumber(BoxOut<Nothing>())

fun takeBoxOutNothing(box: BoxOut<Nothing>) {}
takeBoxOutNothing(BoxOut<Int>())
takeBoxOutNothing(BoxOut<Number>())
takeBoxOutNothing(BoxOut<Nothing>())

fun takeBoxOutStar(box: BoxOut<*>) {}
takeBoxOutStar(BoxOut<Int>())
takeBoxOutStar(BoxOut<Number>())
takeBoxOutStar(BoxOut<Nothing>())

class BoxIn<in T>
fun takeBoxInInt(box: BoxIn<Int>) {}
takeBoxInInt(BoxIn<Int>())
takeBoxInInt(BoxIn<Number>())
takeBoxInInt(BoxIn<Nothing>())
takeBoxInInt(BoxIn<Any>())

```

Exercise: Generic Response

You need to model a response from a server that can be represented as a success or a failure. Both those options can keep data of a generic type. This is how you modelled it:

```

sealed class Response<R, E>
class Success<R, E>(val value: R) : Response<R, E>()
class Failure<R, E>(val error: E) : Response<R, E>()

```

However, you found that this implementation is problematic. To create a `Success` object, you need to provide two generic types, but you only need one. To create a `Failure` object, you need to provide two generic types, but you only need one. Your task is to fix this problem.

```
val rs1 = Success(1) // Compilation error
val rs2 = Success("ABC") // Compilation error
val re1 = Failure(Error()) // Compilation error
val re2 = Failure("Error") // Compilation error
```

You need to define `Success` in a way that it can be created with only one generic type, and `Failure` in a way that it can be created with only one generic type. You want to be able to use `Success` and `Failure` without specifying generic types, so you can write `Success(1)`.

You also want to allow upcasting `Success<Int>` to `Success<Number>` or `Success<Any>`, and `Failure<Error>` to `Failure<Throwable>` or `Failure<Any>`. You want to be able to use `Success<Int>` as `Response<Int, Throwable>`.

```
val rs1 = Success(1)
val rs2 = Success("ABC")
val re1 = Failure(Error())
val re2 = Failure("Error")

val rs3: Success<Number> = rs1
val rs4: Success<Any> = rs1
val re3: Failure<Throwable> = re1
val re4: Failure<Any> = re1

val r1: Response<Int, Throwable> = rs1
val r2: Response<Int, Throwable> = re1
```

Starting code and usage examples can be found in the [MarcinMoskala/kotlin-exercises](#) project on GitHub in the file `advanced/generics/Response.kt`. You can clone this project and solve this exercise locally.

Exercise: Generic Consumer

In your project, you use a class that represents a consumer of some type. You have two implementations of this class:

Printer and Sender. A printer that can accept Number, should also accept Int and Double. A sender that can accept Int, should also accept Number and Any. In general, a consumer that can accept τ , should also accept s if s is a subtype of τ . Update Consumer, Printer and Sender classes to achieve this.

Starting code:

```
abstract class Consumer<T> {
    abstract fun consume(elem: T)
}

class Printer<T> : Consumer<T>() {
    override fun consume(elem: T) {
        // ...
    }
}

class Sender<T> : Consumer<T>() {
    override fun consume(elem: T) {
        // ...
    }
}
```

Usage example:

```
val p1 = Printer<Number>()
val p2: Printer<Int> = p1
val p3: Printer<Double> = p1

val s1 = Sender<Any>()
val s2: Sender<Int> = s1
val s3: Sender<String> = s1

val c1: Consumer<Number> = p1
val c2: Consumer<Int> = p1
val c3: Consumer<Double> = p1
```

Starting code and usage examples can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file

advanced/generics/Consumer.kt. You can clone this project and solve this exercise locally.

Interface delegation

Kotlin has a feature called *interface delegation*. This is special support for the delegation pattern, so let's discuss this first.

The delegation pattern

Imagine you have a class that implements an interface. In the example below, this is the `GenericCreature` class, which implements the `Creature` interface. If you want another class, `Goblin`, to behave just like `GenericCreature`, you can achieve this with composition by creating an instance of `GenericCreature`, keeping it as a property, and using its methods. This new class can also implement the `Creature` interface.

```
interface Creature {
    val attackPower: Int
    val defensePower: Int
    fun attack()
}

class GenericCreature(
    override val attackPower: Int,
    override val defensePower: Int,
) : Creature {
    override fun attack() {
        println("Attacking with $attackPower")
    }
}

class Goblin : Creature {
    private val delegate = GenericCreature(2, 1)
    override val attackPower: Int = delegate.attackPower
    override val defensePower: Int = delegate.defensePower

    override fun attack() {
        delegate.attack()
    }
}
```

```
    }

    // ...

}

fun main() {
    val goblin = Goblin()
    println(goblin.defensePower) // 1
    goblin.attack() // Attacking with 2
}
```

This is an example of a delegation pattern. The `Goblin` class delegates methods defined by the `Creature` interface to an object of type `GenericCreature`. The `delegate` property in the example above is a delegate, and the `attack` method and `attackPower` and `defensePower` properties are delegated.

To reuse the same method implementation, it is enough to use composition, which means keeping a property with an object of another type and using it on its own methods. The delegation pattern also involves implementing the same interface as the class we delegate to, thus introducing polymorphic behavior. The `GenericCreature` and `Goblin` classes both implement the `Creature` interface, therefore they can be used interchangeably in some cases.

Delegation and inheritance

Delegation is an old pattern and has always been presented as an alternative to inheritance. Truth be told, similar behavior can be achieved if we make `GenericCreature` open and we make `Goblin` extend it.

```
interface Creature {
    val attackPower: Int
    val defensePower: Int
    fun attack()
}

open class GenericCreature(
    override val attackPower: Int,
    override val defensePower: Int,
) : Creature {
    override fun attack() {
        println("Attacking with $attackPower")
    }
}

class Goblin : GenericCreature(2, 1) {
    // ...
}

fun main() {
    val goblin = Goblin()
    println(goblin.defensePower) // 1
    goblin.attack() // Attacking with 2
}
```

Using inheritance seems easier, but it has some consequences and limitations that make us choose delegation anyway:

- We can inherit from only one class, but we can delegate to many objects.
- Inheritance is a very strong relationship, but this is often not what we want. Maybe we don't want `Goblin` to be a `GenericCreature` because we are not able to guarantee that the former will always behave exactly like the latter.
- Most classes are not designed for inheritance as they are either closed or we just should not inherit from them.
- Inheritance breaks encapsulation, thus causing safety threats (see Item 36: *Prefer composition over inheritance from Effective Kotlin*).

All in all, this is why we often prefer to use delegation instead of inheritance. Fortunately, the creators of Kotlin introduced special support to help us with this.

Kotlin interface delegation support

Kotlin introduced special support for the interface delegation pattern, which makes it as easy to use as inheritance. After specifying an interface you want your class to implement, you can use the `by` keyword and specify the object that should be used as a delegate. This removes the “writing additional code” overhead. This is how this could be used in `Goblin`:

```
interface Creature {
    val attackPower: Int
    val defensePower: Int
    fun attack()
}

class GenericCreature(
    override val attackPower: Int,
    override val defensePower: Int,
) : Creature {
    override fun attack() {
        println("Attacking with $attackPower")
    }
}

class Goblin : Creature by GenericCreature(2, 1) {
    // ...
}

fun main() {
    val goblin = Goblin()
    println(goblin.defensePower) // 1
    goblin.attack() // Attacking with 2
}
```

On the right side of the `by` keyword, there is a constructor call that creates an instance of `GenericCreature` that is used as a delegate. Under the hood, this delegate will be stored in a property, and all methods from the `Creature` interface will be implemented such that they call appropriate methods from the delegate.

```
public final class Goblin implements Creature {  
    // $FF: synthetic field  
    private final GenericCreature $$delegate_0 = new GenericCreature( attackPower: 2, defencePower: 1 );  
  
    public int getAttackPower() { return this.$$delegate_0.getAttackPower(); }  
  
    public int getDefencePower() { return this.$$delegate_0.getDefencePower(); }  
  
    public void attack() { this.$$delegate_0.attack(); }  
}
```

The object used as a delegate can be created using primary constructor parameters, or we can use a primary constructor parameter as a delegate. We can also use a variable from the outer scope as a delegate.

```
class Goblin : Creature by GenericCreature(2, 1)  
  
// or  
class Goblin(  
    att: Int,  
    def: Int  
) : Creature by GenericCreature(att, def)  
  
// or  
class Goblin(  
    creature: Creature  
) : Creature by creature  
  
// or  
class Goblin(  
    val creature: Creature = GenericCreature(2, 1)  
) : Creature by creature  
  
// or  
val creature = GenericCreature(2, 1)
```

```
class Goblin : Creature by creature
```

We can use interface delegation multiple times in the same class.

```
class Goblin : Attack by Dagger(), Defense by LeatherArmour()
```

```
class Amphibious : Car by SimpleCar(), Boat by MotorBoat()
```

When we use interface delegation, we can still override some methods from the interface ourselves. In such cases, these methods will not be generated automatically and will not call delegates by themselves.

```
interface Creature {
    val attackPower: Int
    val defensePower: Int
    fun attack()
}

class GenericCreature(
    override val attackPower: Int,
    override val defensePower: Int,
) : Creature {
    override fun attack() {
        println("Attacking with $attackPower")
    }
}

class Goblin : Creature by GenericCreature(2, 1) {
    override fun attack() {
        println("Special Goblin attack $attackPower")
    }
}

fun main() {
    val goblin = Goblin()
```

```
    println(goblin.defensePower) // 1
    goblin.attack() // Special Goblin attack 2
}
```

The problem is that there is currently no way to reference the delegate's implicit property. So, if we need to do this, we typically make a primary constructor property that we delegate to and that we use when we need to reference the delegate.

```
interface Creature {
    val attackPower: Int
    val defensePower: Int
    fun attack()
}

class GenericCreature(
    override val attackPower: Int,
    override val defensePower: Int,
) : Creature {
    override fun attack() {
        println("Attacking with $attackPower")
    }
}

class Goblin(
    private val creature: Creature = GenericCreature(2, 1)
) : Creature by creature {
    override fun attack() {
        println("It will be special Goblin attack!")
        creature.attack()
    }
}

fun main() {
    val goblin = Goblin()
    goblin.attack()
    // It will be a special Goblin attack!
```

```
// Attacking with 2
}
```

Wrapper classes

An interesting usage of interface delegation is to make a simple wrapper over an interface that adds something we could not add otherwise. I don't mean a method because this can be added using an extension function. I mean an annotation that might be needed by a library. Consider the following example: for Jetpack Compose, you need to use an object that has the `Immutable` and `Keep` annotations, but you want to use the `List` interface, which is read-only but does not have these annotations. The simplest solution is to make a simple wrapper over `List` and use interface delegation to easily make our wrapper class also implement the `List` interface. In this way, all the methods that we can invoke on `List` can also be invoked on the wrapper.

```
@Keep
@Immutable
data class ComposeImmutableList<T>(
    val innerList: List<T>
) : List<T> by innerList
```

Another example of a wrapper class is something that is used in some multiplatform mobile Kotlin projects. The problem is that in View Model classes, we like to expose observable properties of type `stateFlow` that can be observed easily in Android but not so easily in iOS. To make it easier to observe them, one solution is to define the following wrapper for them that specifies the `collect` method, which can be used easily from Swift. More about this case in the *Using Kotlin Multiplatform* section.

```
class StateFlow<T>(
    source: StateFlow<T>,
    private val scope: CoroutineScope
) : StateFlow<T> by source {
    fun collect(onEach: (T) -> Unit) {
        scope.launch {
            collect { onEach(it) }
        }
    }
}
```

The decorator pattern

Beyond a simple wrapper, there is also the decorator pattern, which uses a class to decorate another class with new capabilities but still implements the same interface (or extends the same class). So, for instance, when we make a `FileInputStream` to read a file, we decorate it with `BufferedInputStream` to add buffering, then we decorate it with `ZipInputStream` to add unzipping capabilities, and finally we decorate it with `ObjectInputStream` to read an object.

```
var fis: InputStream = FileInputStream("/someFile.gz")
var bis: InputStream = BufferedInputStream(fis)
var gis: InputStream = ZipInputStream(bis)
var ois = ObjectInputStream(gis)
var someObject = ois.readObject() as SomeObject
```

The decorator pattern is used by many libraries. Consider how sequence or flow processing works: each transformation decorates the previous sequence or flow with a new operation.

```
fun <T> Sequence<T>.filter(  
    predicate: (T) -> Boolean  
) : Sequence<T> {  
    return FilteringSequence(this, true, predicate)  
}
```

The decorator pattern uses delegation. Each decorator class needs to access the decorated object, use it as a delegate, and add behavior to some of its methods.

```
interface AdFilter {  
    fun showToPerson(user: User): Boolean  
    fun showOnPage(page: Page): Boolean  
    fun showOnArticle(article: Article): Boolean  
}  
  
class ShowOnPerson(  
    val authorKey: String,  
    val prevFilter: AdFilter = ShowAds  
) : AdFilter {  
    override fun showToPerson(user: User): Boolean =  
        prevFilter.showToPerson(user)  
  
    override fun showOnPage(page: Page) =  
        page is ProfilePage &&  
        page.userKey == authorKey &&  
        prevFilter.showOnPage(page)  
  
    override fun showOnArticle(article: Article) =  
        article.authorKey == authorKey &&  
        prevFilter.showOnArticle(article)  
}  
  
class ShowToLoggedIn(  
    val prevFilter: AdFilter = ShowAds  
) : AdFilter {  
    override fun showToPerson(user: User): Boolean =  
        user.isLoggedIn
```

```
override fun showOnPage(page: Page) =
    prevFilter.showOnPage(page)

override fun showOnArticle(article: Article) =
    prevFilter.showOnArticle(article)
}

object ShowAds : AdFilter {
    override fun showToPerson(user: User): Boolean = true
    override fun showOnPage(page: Page): Boolean = true
    override fun showOnArticle(article: Article): Boolean =
        true
}

fun createWorkshopAdFilter(workshop: Workshop) =
    ShowOnPerson(workshop.trainerKey)
        .let(::ShowToLoggedIn)
```

Interface delegation can help us avoid implementing methods that only call similar decorated object methods. This makes our implementation clearer and lets a reader focus on what is essential.

```
class Page
class Article(val authorKey: String)
class User(val isLoggedIn: Boolean)

interface AdFilter {
    fun showToPerson(user: User): Boolean
    fun showOnPage(page: Page): Boolean
    fun showOnArticle(article: Article): Boolean
}

class ShowOnPerson(
    val authorKey: String,
    val prevFilter: AdFilter = ShowAds
) : AdFilter by prevFilter {
    override fun showOnPage(page: Page) =
        page is ProfilePage &&
```

```
page.userKey == authorKey &&
prevFilter.showOnPage(page)

override fun showOnArticle(article: Article) =
    article.authorKey == authorKey &&
        prevFilter.showOnArticle(article)
}

class ShowToLoggedIn(
    val prevFilter: AdFilter = ShowAds
) : AdFilter by prevFilter {
    override fun showToPerson(user: User): Boolean =
        user.isLoggedIn
}

object ShowAds : AdFilter {
    override fun showToPerson(user: User): Boolean = true
    override fun showOnPage(page: Page): Boolean = true
    override fun showOnArticle(article: Article): Boolean =
        true
}

fun createWorkshopAdFilter(workshop: Workshop) =
    ShowOnPerson(workshop.trainerKey)
        .let(::ShowToLoggedIn)
```

Intersection types

It is important to know that interface delegation can be used multiple times in the same class. Thanks to this, you can have a class that implements two interfaces, each of which is delegated to a different delegate. Such a class can aggregate the types and behavior of multiple other classes, so we call such a class represents an “intersection type”. For example, in the Arrow library there is a `ScopedRaise` class that is a decorator for both `EffectScope` and `CoroutineScope`.

```
public class ScopedRaise<E>(
    raise: EffectScope<E>,
    scope: CoroutineScope
) : CoroutineScope by scope, EffectScope<E> by raise
```

The `ScopedRaise` class can represent both interfaces it implements at the same time; so, when we use it as a receiver, methods from both `EffectScope` and `CoroutineScope` can be used implicitly.

```
public suspend fun <E, A, B> Iterable<A>.parMapOrAccumulate(
    context: CoroutineContext = EmptyCoroutineContext,
    transform: suspend ScopedRaise<E>.A -> B
): Either<NonEmptyList<E>, List<B>> =
    coroutineScope {
        map {
            async(context) {
                either {
                    val scope = this@coroutineScope
                    transform(ScopedRaise(this, scope), it)
                }
            }
        }.awaitAll().flattenOrAccumulate()
    }

suspend fun test() {
    listOf("A", "B", "C")
        .parMapOrAccumulate { v ->
            this.launch { } // We can do that,
            // because receiver is CoroutineContext
            this.ensure(v in 'A'..'Z') { error("Not letter") }
            // We can do that, because receiver is EffectScope
        }
}
```

Another example comes from the Ktor framework's integration tests. In this framework, you can use the `testApplication` function, which helps start a server for

testing. It provides the `ApplicationTestBuilder` receiver, which extends `TestApplicationBuilder` and implements `clientProvider`. For my integration tests, I define my own function because I want to have a similar receiver, but I often lack a couple of other properties. I would prefer to have a receiver that aggregates methods from multiple classes or interfaces, as well as useful properties like a background scope or a created test application reference. I often define such a receiver class using interface delegation.

```
class IntegrationTestScope(
    applicationTestBuilder: ApplicationTestBuilder,
    val application: Application,
    val backgroundScope: CoroutineScope,
) : TestApplicationBuilder(),
    ClientProvider by applicationTestBuilder
```

Limitations

The biggest limitation of the interface delegation pattern is that objects we delegate to must have an interface, and only methods from this interface will be delegated. You should also note that - unlike when we use inheritance - the meaning of the `this` reference does not change.

```
interface Creature {
    fun attack()
}

class GenericCreature : Creature {
    override fun attack() {
        // this.javaClass.name is always GenericCreature
        println("${this::class.simpleName} attacks")
    }

    fun walk() {}
}
```

```
class Goblin : Creature by GenericCreature()

open class WildAnimal : Creature {
    override fun attack() {
        // this.javaClass.name depends on actual class
        println("${this::class.simpleName} attacks")
    }
}

class Wolf : WildAnimal()

fun main() {
    GenericCreature().attack() // GenericCreature attacks
    Goblin().attack() // GenericCreature attacks
    WildAnimal().attack() // WildAnimal attacks
    Wolf().attack() // Wolf attacks

    // Goblin().walk() COMPILATION ERROR, no such method
}
```

Conflicting elements from parents

There might be a situation in which two interfaces that our class uses for interface delegation define the same method or property, so we must resolve this conflict by overriding this element in the class. In the example below, both the `Attack` and `Defense` interfaces define the `defense` property, so we **must** override it in the `Goblin` class and specify how it should behave.

```
interface Attack {
    val attack: Int
    val defense: Int
}
interface Defense {
    val defense: Int
}
class Dagger : Attack {
    override val attack: Int = 1
    override val defense: Int = -1
}
class LeatherArmour : Defense {
    override val defense: Int = 2
}
class Goblin(
    private val attackDelegate: Attack = Dagger(),
    private val defenseDelegate: Defense = LeatherArmour(),
) : Attack by attackDelegate, Defense by defenseDelegate {
    // We must override this property
    override val defense: Int =
        defenseDelegate.defense + attackDelegate.defense
}
```

Summary

Interface delegation is not a very popular Kotlin feature, but it has use cases where it helps us avoid repetition of boilerplate code. It is very simple: Kotlin implicitly generates methods and properties defined in an interface, and their implementations call similar methods from the delegate objects. Nevertheless, this feature can help us make our code more clear and concise. Interface delegation can be used when we need to make a simple wrapper over an interface, implement the decorator pattern, or make a class that collects methods from two interfaces. The biggest limitation of interface delegation is that objects we delegate to must have an interface, and only methods from this interface will be delegated.

Exercise: ApplicationScope

You need to create a class `ApplicationScope`, that implements interfaces `COROUTINE_SCOPE`, `APPLICATION_CONTROL_SCOPE` and `LOGGING_SCOPE`. It should expect primary constructor properties of the same types, and use them as delegates. Use interface delegation to implement that.

Starting code and unit tests can be found in the [MarcinMoskala/kotlin-exercises](#) project on GitHub in the file `advanced/delegates/ApplicationScope.kt`. You can clone this project and solve this exercise locally.

Property delegation

In programming, there are a number of patterns that use properties, like lazy properties, property value injection, property binding, etc. In most languages, there is no easy way to extract such patterns, therefore developers tend to repeat the same patterns again and again, or they need to depend on complex libraries. In Kotlin, we extract repeatable property patterns using a feature that is (currently) unique to Kotlin: property delegation. This feature's trademark is the `by` keyword, which is used between a property definition and a delegate specification. Together with some functions from Kotlin stdlib, here is an example usage of property delegation that we use to implement lazy or observable properties:

```
val value by lazy { createValue() }

var items: List<Item> by
    Delegates.observable(listOf()) { _, _, _ ->
        notifyDataSetChanged()
    }

var key: String? by
    Delegates.observable(null) { _, old, new ->
        Log.e("key changed from $old to $new")
    }
```

Later in this chapter, we will discuss both `lazy` and `observable` functions in detail. For now, all you need to know is that delegates are just functions; they are not Kotlin keywords (just like `lazy` in Scala or Swift). We can implement our own `lazy` function in a few simple lines of code. Just like many Kotlin libraries, we can also implement our own property delegates. Some good examples are View and Resource Binding, Dependency Injection³, and Data Binding.

³This example use of Koin formally presents service location, not dependency injection.

```
// View and resource binding example in Android
private val button: Button by bindView(R.id.button)
private val textSize by bindDimension(R.dimen.font_size)
private val doctor: Doctor by argExtra(DOCTOR_ARG)

// Dependency Injection using Koin
private val presenter: MainPresenter by inject()
private val repository: NetworkRepository by inject()
private val viewModel: MainViewModel by viewModel()

// Data binding
private val port by bindConfiguration("port")
private val token: String by preferences.bind(TOKEN_KEY)
```

How property delegation works

To understand how we can extract other common behaviors using property delegation, let's start with a very simple property delegate. Let's say we need to define properties with custom getters and setters that print their value changes⁴:

```
var token: String? = null
    get() {
        println("token getter returned $field")
        return field
    }
    set(value) {
        println("token changed from $field to $value")
        field = value
    }

var attempts: Int = 0
    get() {
        println("attempts getter returned $field")
```

⁴I assume you are familiar with custom getters and setters. I explain them in the previous book from this series, Kotlin Essentials, in the Classes chapter.

```
        return field
    }
    set(value) {
        println("attempts changed from $field to $value")
        field = value
    }
}

fun main() {
    token = "AAA" // token changed from null to AAA
    val res = token // token getter returned AAA
    println(res) // AAA
    attempts++
    // attempts getter returned 0
    // attempts changed from 0 to 1
}
```

Even though `token` and `attempts` are of different types, the behavior of these two properties is nearly identical and can be extracted using property delegation.

Property delegation is based on the idea that a property is defined by its accessors: for `val`, it is a getter; for `var`, it is a getter and a setter. These functions can be delegated to another object's methods: the getter will be delegated to the `getValue` function, and the setter to the `setValue` function. An object with these methods needs to be created and placed on the right side of the `by` keyword. To make our properties behave the same way as in the example above, we can create the following delegate:

```
import kotlin.reflect.KProperty

private class LoggingProperty<T>(var value: T) {
    operator fun getValue(
        thisRef: Any?,
        prop: KProperty<*>
    ): T {
        println("${prop.name} getter returned $value")
        return value
    }

    operator fun setValue(
        thisRef: Any?,
        value: T
    ) {
        println("setter received $value")
        this.value = value
    }
}
```

```
}

operator fun setValue(
    thisRef: Any?,
    prop: KProperty<*>,
    newValue: T
) {
    val name = prop.name
    println("$name changed from $value to $newValue")
    value = newValue
}
}

var token: String? by LoggingProperty(null)
var attempts: Int by LoggingProperty(0)

fun main() {
    token = "AAA" // token changed from null to AAA
    val res = token // token getter returned AAA
    println(res) // AAA
    attempts++
    // attempts getter returned 0
    // attempts changed from 0 to 1
}
```

To fully understand how property delegation works, take a look at what `by` is compiled to. The above `token` property will be compiled to something similar to the following code:

```
// Code in Kotlin:
var token: String? by LoggingProperty(null)

// What it is compiled to when a property is top-level
@JvmField
private val `token$delegate` = LoggingProperty<String?>(null)
var token: String?
    get() = `token$delegate`.getValue(null, ::token)
    set(value) {
        `token$delegate`.setValue(null, ::token, value)
```

```
}
```

```
// What it is compiled to when a property is in a class
@JvmField
private val `token$delegate` = LoggingProperty<String?>(null)
var token: String?
    get() = `token$delegate`.getValue(this, this::token)
    set(value) {
        `token$delegate`.setValue(this, this::token, value)
    }
```

To make sure we understand this, let's look under the hood and check out what our Kotlin property delegate usage is compiled to.

```
// Kotlin code:
var token: String? by LoggingProperty(null)

fun main() {
    token = "AAA" // token changed from null to AAA
    val res = token // token getter returned AAA
    println(res) // AAA
}

// Java representation of this code:
@Nullable
private static final LoggingProperty token$delegate =
    new LoggingProperty((Object)null);

@Nullable
public static final String getToken() {
    return (String)token$delegate
        .getValue((Object)null, $$delegatedProperties[0]);
}

public static final void setToken(@Nullable String var0) {
    token$delegate
```

```
.setValue((Object)null, $$delegatedProperties[0], var0);  
}  
  
public static final void main() {  
    setToken("AAA");  
    String res = getToken();  
    System.out.println(res);  
}
```

Let's analyze this step by step. When you get a property value, you call this property's getter; property delegation delegates this getter to the `getValue` function. When you set a property value, you are calling this property's setter; property delegation delegates this setter to the `setValue` function. This way, each delegate fully controls this property's behavior.

Other `getValue` and `setValue` parameters

You might also have noticed that the `getValue` and `setValue` methods not only receive the value that was set to the property and decide what its getter returns, but they also receive a bounded reference to the property as well as a context (`this`). The reference to the property is most often used to get its name and sometimes to get information about annotations. The parameter referencing the receiver gives us information about where the function is used and who can use it.

The `KProperty` type will be better covered later in the `Reflection` chapter.

```
import kotlin.reflect.KProperty

private class LoggingProperty<T>(
    var value: T
) {

    operator fun getValue(
        thisRef: Any?,
        prop: KProperty<*>
    ): T {
        println(
            "${prop.name} in $thisRef " +
                "getter returned $value"
        )
        return value
    }

    operator fun setValue(
        thisRef: Any?,
        prop: KProperty<*>,
        newValue: T
    ) {
        println(
            "${prop.name} in $thisRef " +
                "changed from $value to $newValue"
        )
        value = newValue
    }
}

var token: String? by LoggingProperty(null)

object AttemptsCounter {
    var attempts: Int by LoggingProperty(0)
}

fun main() {
    token = "AAA" // token in null changed from null to AAA
    val res = token // token in null getter returned AAA
```

```
    println(res) // AAA

    AttemptsCounter.attempts = 1
    // attempts in AttemptsCounter@XYZ changed from 0 to 1
    val res2 = AttemptsCounter.attempts
    // attempts in AttemptsCounter@XYZ getter returned 1
    println(res2) // 1
}
```

When we have multiple `getValue` and `setValue` methods but with different context types, different definitions of the same method will be chosen in different situations. This fact can be used in clever ways. For instance, we might need a delegate that can be used in different kinds of views, but it should behave differently with each of them based on what is offered by the context:

```
class SwipeRefreshBinderDelegate(val id: Int) {
    private var cache: SwipeRefreshLayout? = null

    operator fun getValue(
        activity: Activity,
        prop: KProperty<*>
    ): SwipeRefreshLayout {
        return cache ?: activity
            .findViewById<SwipeRefreshLayout>(id)
            .also { cache = it }
    }

    operator fun getValue(
        fragment: Fragment,
        prop: KProperty<*>
    ): SwipeRefreshLayout {
        return cache ?: fragment.view
            .findViewById<SwipeRefreshLayout>(id)
            .also { cache = it }
    }
}
```

Implementing a custom property delegate

To make it possible to use an object as a property delegate, all it needs is the `getValue` operator for `val` and the `getValue` and `setValue` operators for `var`. Both `getValue` and `setValue` are operators, so they need the `operator` modifier. They need parameters for `thisRef` of any type (most likely `Any?`) and `property` of type `KProperty<*>`. `setValue` should additionally have a property for a value whose type should be the same type or a supertype of the type used by the property. The `getValue` result type should be the same type or a subtype of the type used by the property.

```
class EmptyPropertyDelegate {
    operator fun getValue(
        thisRef: Any?,
        property: KProperty<*>
    ): String {
        return ""
    }
    operator fun setValue(
        thisRef: Any?,
        property: KProperty<*>,
        value: String
    ) {
        // no-op
    }
}

val p1: String by EmptyPropertyDelegate()
var p2: String by EmptyPropertyDelegate()
```

These methods can be member functions, but they can also be extension functions. For instance, `Map<String, *>` can be used as a property delegate thanks to the extension function below, which is defined in the standard library. We will discuss using `Map<String, *>` as a delegate later in this chapter.

```
// Function from Kotlin stdlib
inline operator fun <V, V1 : V> Map<in String, V>
    .getValue(thisRef: Any?, property: KProperty<*>): V1 =
        getOrImplicitDefault(property.name) as V1

fun main() {
    val map: Map<String, Any> = mapOf(
        "name" to "Marcin",
        "kotlinProgrammer" to true
    )
    val name: String by map
    val kotlinProgrammer: Boolean by map
    print(name) // Marcin
    print(kotlinProgrammer) // true

    val incorrectName by map
    println(incorrectName) // Exception
}
```

When we define a delegate, it might be helpful to implement the `ReadOnlyProperty` interface (when we define a property delegate for `val`) or the `ReadWriteProperty` interface (when we define a property delegate for `var`) from Kotlin stdlib. These interfaces specify `getValue` and `setValue` with the correct parameters.

```
fun interface ReadOnlyProperty<in T, out V> {
    operator fun getValue(
        thisRef: T,
        property: KProperty<*>
    ): V
}

interface ReadWriteProperty<in T, V>: ReadOnlyProperty<T,V>{
    override operator fun getValue(
        thisRef: T,
        property: KProperty<*>
    ): V
    operator fun setValue(
```

```
    thisRef: T,  
    property: KProperty<*>,  
    value: V  
)  
}
```

Notice how those interfaces use generic variance modifiers. Type parameter `T` is only used in in-positions, so it has the contravariant `in` modifier. The `ReadOnlyProperty` interface uses the type parameter `V` only in out-positions, so it has the covariant `out` modifier.

Both `ReadOnlyProperty` and `ReadWriteProperty` require two type arguments. The first is for the receiver type and is typically `Any?`, which allows our property delegate to be used in any context. The second argument should be the property's value type. If we define a property delegate for properties of a certain type, we set this type here. We can also use a generic type parameter in this type argument position.

```
private class LoggingProperty<T>(  
    var value: T  
) : ReadWriteProperty<Any?, T> {  
  
    override fun getValue(  
        thisRef: Any?,  
        property: KProperty<*>  
    ): T {  
        println("${property.name} getter returned $value")  
        return value  
    }  
  
    override fun setValue(  
        thisRef: Any?,  
        property: KProperty<*>,  
        value: T  
    ) {
```

```
    val name = property.name
    println("$name changed from ${this.value} to $value")
    this.value = value
}
}
```

Provide a delegate

This is a story as old as time. You delegate a task to another person, who, instead of doing it, delegates it to someone else. Objects in Kotlin can do the same. An object can define the `provideDelegate` method, which returns another object that will be used as a delegate.

```
import kotlin.reflect.KProperty

class LoggingProperty<T>(var value: T) {
    operator fun getValue(
        thisRef: Any?,
        prop: KProperty<*>
    ): T {
        println("${prop.name} getter returned $value")
        return value
    }
    operator fun setValue(
        thisRef: Any?,
        prop: KProperty<*>,
        newValue: T
    ) {
        val name = prop.name
        println("$name changed from $value to $newValue")
        value = newValue
    }
}

class LoggingPropertyProvider<T>(
    private val value: T
) {
```

```

operator fun provideDelegate(
    thisRef: Any?,
    property: KProperty<*>
): LoggingProperty<T> = LoggingProperty(value)
}

var token: String? by LoggingPropertyProvider(null)
var attempts: Int by LoggingPropertyProvider(0)

fun main() {
    token = "AAA" // token changed from null to AAA
    val res = token // token getter returned AAA
    println(res) // AAA
}

```

The power of `provideDelegate` is that the object that is used on the right side of the `by` keyword does not need to be used as a delegate. So, for instance, an immutable object can provide a mutable delegate.

Let's see an example. Let's say you implement a library to make it easier to operate on values stored in preference files⁵. This is how you want your library to be used:

```

object UserPref : PreferenceHolder() {
    var splashScreenShown: Boolean by bindToPreference(true)
    var loginAttempts: Int by bindToPreference(0)
}

```

The `bindToPreference` function returns an object that can be used as a property delegate. But what if we want to make it possible to use `Int` or `Boolean` as delegates in the scope of `PreferenceHolder`?

⁵Before you do this, consider the fact that there are already many similar libraries, such as `PreferenceHolder`, which I published years ago.

```
object UserPref : PreferenceHolder() {
    var splashScreenShown: Boolean by true
    var loginAttempts: Int by 0
}
```

This way, using delegates is similar to assigning some values; however, because delegates are used, some additional operations might be performed, such as binding these property values to preference file values.

It is problematic to use `Int` or `Boolean` as delegates because they are not implemented to be used this way. We could implement `getValue` and `setValue` to operate on preference files, but it would be harder to cache values. The simple solution is to define the `provideDelegate` extension function on `Int` and `Boolean`. This way, `Int` or `Boolean` can be used on the right side of `by`, but they can't be used as delegates themselves.

```
abstract class PreferenceHolder {
    operator fun Boolean.provideDelegate(
        thisRef: Any?,
        property: KProperty<*>
    ) = bindToPreference(this)

    operator fun Int.provideDelegate(
        thisRef: Any?,
        property: KProperty<*>
    ) = bindToPreference(this)

    inline fun <reified T : Any> bindToPreference(
        default: T
    ): ReadWriteProperty<PreferenceHolder, T> = TODO()
}
```

We can also use the `PropertyDelegateProvider` interface, which specifies the `provideDelegate` function with appropriate arguments and result types. The two type parameters of `PropertyDelegateProvider` represent the receiver reference type and the type of the property we delegate.

```
class LoggingPropertyProvider<T>(
    private val value: T
) : PropertyDelegateProvider<Any?, LoggingProperty<T>> {

    override fun provideDelegate(
        thisRef: Any?,
        property: KProperty<*>
    ): LoggingProperty<T> = LoggingProperty(value)
}
```

Personally, I am not a fan of using raw values as delegates because I believe that additional function names improve readability. However, this is the best example of using `provideDelegate` I could find.

Property delegates in Kotlin stdlib

Kotlin provides the following standard property delegates:

- `Delegates.notNull`
- `lazy`
- `Delegates.observable`
- `Delegates.vetoable`
- `Map<String, T>` and `MutableMap<String, T>`

Let's discuss them individually and present their use cases.

The `notNull` delegate

I will start with the simplest property delegate, which is created using the `notNull` method from the `Delegates` object that is defined in Kotlin stdlib. It is an alternative to `lateinit`, so the property delegated to `notNull` behaves like a regular property but has no initial value. Therefore, if you try to get a value before setting it, this results in an exception.

```
import kotlin.properties.Delegates

var a: Int by Delegates.notNull()
var b: String by Delegates.notNull()

fun main() {
    a = 10
    println(a) // 10
    a = 20
    println(a) // 20

    println(b) // IllegalStateException:
    // Property b should be initialized before getting.
}
```

Wherever possible, we should use the `lateinit` property instead of the `notNull` delegate for better performance because `lateinit` properties are faster. Currently, however, Kotlin does not support `lateinit` properties with types that associate with primitives, like `Int` or `Boolean`.

```
lateinit var i: Int // Compilation Error
lateinit var b: Boolean // Compilation Error
```

In such cases, we use the `notNull` delegate.

```
var i: Int by Delegates.notNull()
var b: Boolean by Delegates.notNull()
```

I often see this delegate used as part of DSL builders or for properties whose values are injected.

```
abstract class IntegrationTest {  
  
    @Value("${server.port}")  
    var serverPort: Int by Delegates.notNull()  
  
    // ...  
}  
  
// DSL builder  
fun person(block: PersonBuilder.() -> Unit): Person =  
    PersonBuilder().apply(block).build()  
  
class PersonBuilder() {  
    lateinit var name: String  
    var age: Int by Delegates.notNull()  
    fun build(): Person = Person(name, age)  
}  
  
// DSL use  
val person = person {  
    name = "Marc"  
    age = 30  
}
```

Exercise: Lateinit delegate

Implement `Lateinit` delegate that makes a property behave like a `lateinit` property, so that it should be able to keep set values, but should not require an initial value. If the getter is called before the property is set, it should throw `IllegalStateException` exception with the message “Uninitialized lateinit property”.

```
val a by Lateinit<Int>()
a = 1
println(a) // 1

val b by Lateinit<String>()
b = "ABC"
println(b) // ABC

val c by Lateinit<String>()
println(c) // IllegalStateException:
// Uninitialized lateinit property c
```

Property should support nullable types.

```
val a by Lateinit<Int?>()
a = 1
println(a) // 1
a = null
println(a) // null
```

Unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file advanced/delegates/Lateinit.kt. You can clone this project and solve this exercise locally.

Hint: You can make your class implement `ReadWriteProperty<Any?, T>` interface to make it a property delegate.

The `lazy` delegate

The most popular property delegate is `lazy`. It implements the lazy property pattern, so it postpones read-only value initialization until this value is needed for the first time. This is what an example usage of `lazy` looks like, and this is what we would need to implement if we didn't have this delegate⁶:

⁶The actual `lazy` implementation is more complicated as it has better synchronization, which secures it for concurrent use.

```
val userRepo by lazy { UserRepository() }

// Alternative code not using lazy
private var _userRepo: UserRepository? = null
private val userRepoLock = Any()
val userRepo: UserRepository
    get() {
        synchronized(userRepoLock) {
            if (_userRepo == null) {
                _userRepo = UserRepository()
            }
            return _userRepo!!
        }
    }
}
```

The lazy delegate is often used as a performance optimization for objects that are expensive to calculate. Let's start with an abstract example. Consider class `A`, which composes classes `B`, `C`, and `D`, each of which is heavy to initialize. This makes the instance of `A` really heavy to initialize because it requires the initialization of multiple heavy objects.

```
class A {
    val b = B()
    val c = C()
    val d = D()

    // ...
}
```

We can make the initialization of `A` lighter by making `b`, `c`, and `d` lazy properties. Thanks to that, initialization of their values is postponed until their first use; if these properties are never used, the associated class instances will never be initialized. Such an operation improves class creation time, which benefits application startup time and test execution time.

```
class A {  
    val b by lazy { B() }  
    val c by lazy { C() }  
    val d by lazy { D() }  
  
    // ...  
}
```

To make this example more practical, `A` might be `FlashcardsParser`, which is used to parse files defined in a language we have invented; `B`, `C`, and `D` might be some complex regex parsers which are heavy to initialize.

```
class OurLanguageParser {  
    val cardRegex by lazy { Regex("...") }  
    val questionRegex by lazy { Regex("...") }  
    val answerRegex by lazy { Regex("...") }  
  
    // ...  
}
```

Regex is a really useful notation when we need to process text. However, regex definitions can be very complex, and their parsing is a heavy operation. This reminds me of another example I have famously shown in the Effective Kotlin book. Consider a function in which we use regex to determine whether a string contains a valid IP address:

```
fun String.isValidIpAddress(): Boolean {  
    return this.matches(  
        ("\\A(?:\\d{1,3}\\.){3}\\d{1,3}" +  
         "[\\d]{1,3}[\\d]{1,3}[\\d]{1,3}[\\d]{1,3}\\.){3}(?:\\d{1,3}\\.){3}\\d{1,3}" +  
         "[\\d]{1,3}[\\d]{1,3}[\\d]{1,3}[\\d]{1,3}\\.){3}\\d{1,3}").toRegex()  
    )  
}  
  
// Usage  
print("5.173.80.254".isValidIpAddress()) // true
```

The problem with this function is that the `Regex` object needs to be created every time we use it. This is a serious disadvantage since regex pattern compilation is complex, therefore this function is unsuitable for repeated use in performance-constrained parts of our code. However, we can improve it by lifting the regex up to the top level:

```
private val IS_VALID_IP_REGEX = "\\\A(?:(:25[0-5]|2[0-4]" +
    "[0-9]| [01]?[0-9][0-9]?)\\.){3}(?:25[0-5]|2[0-4][0-9]|"+
    "[01]?[0-9][0-9]?)\\z".toRegex()

fun String.isValidIpAddress(): Boolean =
    matches(IS_VALID_IP_REGEX)
```

The problem now is that this regex is initialized whenever we initialize this file for the first time; so, if this file contains more elements (like other functions or properties), this regex might slow down the usage of these elements for no good reason. This is why it is better to make the `IS_VALID_IP_REGEX` property lazy.

```
private val IS_VALID_IP_REGEX by lazy {
    ("\\\\A(?:(:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\\.){3}" +
        "(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\\z").toRegex()
}

fun String.isValidIpAddress(): Boolean =
    matches(IS_VALID_IP_REGEX)
```

This technique⁷ can be used in a variety of situations. Consider a data class that stores a computed property⁸ that is not trivial to calculate. For example, consider the `User` class with the `fullDisplay` property, which is calculated from other properties but includes some significant logic as it depends

⁷Also known as memoization.

⁸By computed property, I mean a property whose value is determined by other properties and therefore can always be recalculated.

on multiple properties and project configurations. If we define `fullDisplay` as a regular property, it will be calculated whenever a new instance of `User` is created, which might be an unnecessary cost.

```
data class User(  
    val name: String,  
    val surname: String,  
    val pronouns: Pronouns,  
    val gender: Gender,  
    // ...  
) {  
    val fullDisplay: String = produceFullDisplay()  
  
    fun produceFullDisplay(): String {  
        println("Calculating...")  
        // ...  
        return "XYZ"  
    }  
}  
  
fun test() {  
    val user = User(...) // Calculating...  
    val copy = user.copy() // Calculating...  
    println(copy.fullDisplay) // XYZ  
    println(copy.fullDisplay) // XYZ  
}
```

If we define the `fullDisplay` property using a getter, it will be re-calculated whenever it is used, which also might be an unnecessary cost.

```
data class User(  
    val name: String,  
    val surname: String,  
    val pronouns: Pronouns,  
    val gender: Gender,  
    // ...  
) {  
    val fullDisplay: String  
        get() = produceFullDisplay()  
  
    fun produceFullDisplay(): String {  
        println("Calculating...")  
        // ...  
        return "XYZ"  
    }  
}  
  
fun test() {  
    val user = User(...)  
    val copy = user.copy()  
    println(copy.fullDisplay) // Calculating... XYZ  
    println(copy.fullDisplay) // Calculating... XYZ  
}
```

Defining the `fullDisplay` property as lazy is a sweet spot for properties that:

- are read-only (`lazy` can only be used for `val`),
- are non-trivial to calculate (otherwise, using `lazy` is not worth the effort),
- might not be used for all instances (otherwise, use a regular property),
- might be used more than once by one instance (otherwise, define a property with a getter).

```
data class User(  
    val name: String,  
    val surname: String,  
    val pronouns: Pronouns,  
    val gender: Gender,  
    // ...  
) {  
    val fullDisplay: String by lazy { produceFullDisplay() }  
  
    fun produceFullDisplay() {  
        println("Calculating...")  
        // ...  
    }  
}  
  
fun test() {  
    val user = User(...)  
    val copy = user.copy()  
    println(copy.fullDisplay) // Calculating... XYZ  
    println(copy.fullDisplay) // XYZ  
}
```

When we consider using a `lazy` delegate as a performance optimization, we should also consider which thread safety mode we want it to use. It can be specified in an additional `mode` function argument, which accepts the enum `LazyThreadSafetyMode`. The following options are available:

- `SYNCHRONIZED` is the default and safest option and uses locks to ensure that only a single thread can initialize this delegate instance. This option is also the slowest because synchronization mechanisms introduce some performance costs.
- `PUBLICATION` means that the initializer function can be called several times on concurrent access to an uninitialized delegate instance value, but only the first returned value will be used as the value of this delegate instance. If a delegate is used only by a single thread, this option will be slightly faster than `SYNCHRONIZED`; however, if a

delegate is used by multiple threads, we need to cover the possible cost of recalculation of the same value before the value is initialized.

- `NONE` is the fastest option and uses no locks to synchronize access to the delegate instance value; so, if the instance is accessed from multiple threads, its behavior is undefined. This mode should not be used unless this instance is guaranteed to never be initialized from more than one thread.

```
val v1 by lazy { calculate() }  
val v2 by lazy(LazyThreadSafetyMode.PUBLICATION){calculate()}  
val v3 by lazy(LazyThreadSafetyMode.NONE) { calculate() }
```

In all these examples, we used `lazy` as a performance optimization, but there are other reasons to use it. In this context, I'd like to share a story from my early days of using Kotlin on Android. It was around 2015. Kotlin was still before the stable release, and the Kotlin community was inventing ways of using its features to help us with everyday tasks. You see, in Android we have a concept of Activity, which is like a window that defines its view, traditionally using XML files. Then, the class representing an Activity often modifies this view by programmatically changing text or some of its properties. To change a particular view element, we need to reference it, but we cannot reference it before the view is set with the `setContentView` function. Back then, this is why it was standard practice to define references to view elements as `lateinit` properties and associate appropriate view elements with them immediately after setting up the content view.

```
class MainActivity : Activity() {  
    lateinit var questionLabelView: TextView  
    lateinit var answerLabelView: EditText  
    lateinit var confirmButtonView: Button  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        questionLabelView =  
            findViewById(R.id.main_question_label)  
        answerLabelView =  
            findViewById(R.id.main_answer_label)  
        confirmButtonView =  
            findViewById(R.id.main_button_confirm)  
    }  
}
```

This pattern was used in nearly all Android applications (and I can still see it nowadays in some projects), even though it is far from perfect. We need to define multiple `lateinit` properties, which are all read-write even though they are initialized only once. Property definition and assignment are separated. If a property is not used, it will not be marked by the IDE because an assignment is considered a usage. Overall, there is a lot of space for improvement. The solution turned out to be extremely simple: we can define property initialization next to its definition if we make it lazy.

```
class MainActivity : Activity() {  
    val questionLabelView: TextView by  
        lazy { findViewById(R.id.main_question_label) }  
    val answerLabelView: TextView by  
        lazy { findViewById(R.id.main_answer_label) }  
    val confirmButtonView: Button by  
        lazy { findViewById(R.id.main_button_confirm) }  
  
    override fun onCreate(savedInstanceState: Bundle) {  
        super.onCreate(savedInstanceState)
```

```
        setContentView(R.layout.main_activity)
    }
}
```

Thanks to the `lazy` delegate, finding the view by id will be postponed until the property is used for the first time, which we can assume will happen after the content view is set. We made such an assumption in the previous solution, so why not make it in this one too? We also have some important improvements: properties that keep view references are read-only, initialization and definition are kept together, and unused properties will be marked. We also have performance benefits: a view reference that is never used will never be associated with the view element, which is good because the `findViewById` execution can be expensive.

What is more, we can push this pattern further and extract a function that will both define a lazy delegate and find a view by id. In a project I co-created, we named this delegate `bindView`, and it helped us make our view references really clear, consistent, and readable.

```
class MainActivity : Activity() {
    var questionLabelView: TextView by
        bindView(R.id.main_question_label)
    var answerLabelView: TextView by
        bindView(R.id.main_answer_label)
    var confirmButtonView: Button by
        bindView(R.id.main_button_confirm)

    override fun onCreate(savedInstanceState: Bundle) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.main_activity)
    }
}

// ActivityExt.kt
fun <T : View> Activity.bindView(viewId: Int) =
    lazy { this.findViewById<T>(viewId) }
```

We started using this pattern to bind other references as well, like strings or colors. But my favorite part was binding Activity arguments, which I still use today in some of my projects because it's very convenient to have all activity properties defined together with their keys at the top of this activity definition.

```
class SettingsActivity : Activity() {
    private val titleString by bindString(R.string.title)
    private val blueColor by bindColor(R.color.blue)

    private val doctor by extra<Doctor>(DOCTOR_KEY)
    private val title by extraString(TITLE_KEY)

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.settings_activity)
    }
}

// ActivityExt.kt
fun <T> Activity.bindString(@IdRes id: Int): Lazy<T> =
    lazy { this.getString(id) }
fun <T> Activity.bindColor(@IdRes id: Int): Lazy<T> =
    lazy { this.getColor(id) }
fun <T : Parcelable> Activity.extra(key: String) =
    lazy { this.intent.extras.getParcelable(key) }
fun Activity.extraString(key: String) =
    lazy { this.intent.extras.getString(key) }
```

In the backend application, you can also find lazy delegates that are used to initialize properties that should be initialized when they are needed for the first time, but not during project setup. A simple example is a connection to a local database.

As you can see, `lazy` is a powerful delegate that is mainly used for performance optimization but can also be used to simplify our code. I would like to finish this section with a small puzzle.

Consider the following code⁹:

```
class Lazy {
    var x = 0
    val y by lazy { 1 / x }

    fun hello() {
        try {
            print(y)
        } catch (e: Exception) {
            x = 1
            print(y)
        }
    }
}

fun main(args: Array<String>) {
    Lazy().hello()
}
```

What will be printed? Try to answer this question before reading any further.

To understand the answer, we need to consider three things:

1. Delegate exceptions are propagated out of accessors.
2. A lazy delegate first tries to return a previously calculated value; if no value is already stored, it uses a lambda expression to calculate it. If the calculation process is disturbed with an exception, no value is stored; when we ask for the lazy value the next time, processing starts again.
3. Kotlin's lambda expressions automatically capture variable references; so, when we use `x` inside a lambda expression, every time we use its reference inside a lambda

⁹I first heard about this puzzle from Anton Keks' presentation and it can be found in his repository, so I assume he is the author.

expression we will receive the current value of `x`; when we call the lambda expression for the second time, `x` is 1, so the result should be 1.

So, the answer to this puzzle is “1”.

Exercise: Blog Post Properties

In your project, you’re working with a `BlogPost` class that represents a blog post, including its title, content, and author details. Your task is to enhance the `BlogPost` class by adding and implementing the following properties:

- `authorName` - a string that combines the author’s name and surname. For example, if the author’s name is “John” and surname is “Smith”, the value of this property should be “John Smith”. You can assume that you need to use this property many times per blog post object.
- `wordCount` - an integer that represents the number of words in the blog post. You can assume that you might need this property more than once per blog post object, and that it is expensive to calculate.
- `isLongRead` - a boolean that indicates whether the blog post is longer than 1000 characters. You can assume that you need to use this property at most once per blog post object.
- `summary` - a string that is calculated using the `generateSummary` function. This object is very heavy to calculate, and you do not want to calculate it more than once.

You need to decide how to implement each of these properties. Your options are:

- `val` property defined by value
- `val` property defined by getter
- Lazy property

Starting code:

```
data class BlogPost(  
    val title: String,  
    val content: String,  
    val author: Author,  
) {  
    // TODO: Add properties here  
  
    private fun generateSummary(content: String): String =  
        content.take(100) + "..."  
}  
  
data class Author(  
    val key: String,  
    val name: String,  
    val surname: String,  
)
```

Unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file advanced/delegates/Lazy.kt. You can clone this project and solve this exercise locally.

The observable delegate

The next important delegate from Kotlin stdlib is `observable` from the `Delegates` object, which makes a property behave like a regular property but also specifies a function that will be executed whenever the property's setter is called.

```
var name: String by  
    Delegates.observable("Empty") { prop, old, new ->  
        println("$old -> $new")  
    }  
  
fun main() {  
    name = "Martin" // Empty -> Martin  
    name = "Igor" // Martin -> Igor  
    name = "Igor" // Igor -> Igor  
}
```

This lambda expression has three parameters: a reference to the property, the value before the change, and the new value. An observable delegate can be replaced with the following setter¹⁰:

```
var prop: SomeType by Delegates.observable(initial, operation)

// Alternative to
var prop: SomeType = initial
    set(newValue) {
        field = newValue
        operation(::prop, field, newValue)
    }
```

Note that elements from object declarations can be imported directly (known as a static import) and then used directly.

```
import kotlin.properties.Delegates.observable

val prop: SomeType by observable(initial, operation)
```

We use the `observable` delegate if we want to take some action whenever a property value changes, e.g., when we want to log each property change:

```
val status by observable(INITIAL) { _, old, new ->
    log.info("Status changed from $old to $new")
}
```

This way, all property changes will be displayed in logs, and we can easily track them. On Android, the `observable` delegate is often used when a property change should lead to a view update. For instance, when we implement a list adapter (a class that decides how and what elements should be displayed

¹⁰Not in all cases because the `observable` delegate has better synchronization, but this simplified code can help us understand how `observable` works in general.

in a list), we should redraw the view whenever the list of elements changes. I often use an observable delegate to do this automatically.

```
var elements by observable(elements) { _, old, new ->
    if (new != old) notifyDataSetChanged()
}
```

We use an observable delegate to invoke some action when a property changes. We might use it to implement a class that invokes observers whenever an observable property changes.

```
import kotlin.properties.Delegates.observable

class ObservableProperty<T>(initial: T) {
    private var observers: List<(T) -> Unit> = listOf()

    var value: T by observable(initial) { _, _, new ->
        observers.forEach { it(new) }
    }

    fun addObserver(observer: (T) -> Unit) {
        observers += observer
    }
}

fun main() {
    val name = ObservableProperty("")
    name.addObserver { println("Changed to $it") }
    name.value = "A"
    // Changed to A
    name.addObserver { println("Now it is $it") }
    name.value = "B"
    // Changed to B
    // Now it is B
}
```

Using an observable delegate, one property change can influence other properties or our application state. I've used this

to implement unidirectional data binding when, for instance, I wanted to define a property whose state change influenced changes in a view. This is like the `drawerOpen` property in the example below, which opens and closes the drawer when set to `true` or `false`¹¹.

```
var drawerOpen by observable(false) { _, _, open ->
    if (open) drawerLayout.openDrawer(GravityCompat.START)
    else drawerLayout.closeDrawer(GravityCompat.START)
}
```

Note that a property delegate can be extracted into a separate function that is reusable between components.

```
var drawerOpen by bindToDrawerOpen(false) { drawerLayout }

fun bindToDrawerOpen(
    initial: Boolean,
    lazyView: () -> DrawerLayout
) = observable(initial) { _, _, open ->
    if (open) drawerLayout.openDrawer(GravityCompat.START)
    else drawerLayout.closeDrawer(GravityCompat.START)
}
```

Another example might be when we write an application for reading books, and we have properties to represent the book id and page number. Let's assume that changing the book the user is reading means resetting the page number, which we can do using an `observable` delegate.

¹¹I pushed this pattern further in the `KotlinAndroidView-Bindings` library, which makes little sense in modern Android development because it now has good support for `LiveData` and `StateFlow`.

```
import kotlin.properties.Delegates.observable

var book: String by observable("") { _, _, _ ->
    page = 0
}
var page = 0

fun main() {
    book = "TheWitcher"
    repeat(69) { page++ }
    println(book) // TheWitcher
    println(page) // 69
    book = "Ice"
    println(book) // Ice
    println(page) // 0
}
```

We can also use an `observable` delegate to interact with the property value itself. For instance, for one project we decided that a presenter should have sub-presenters, each of which should have its own lifecycle, therefore the `onCreate` and `onDestroy` methods should be called when a presenter is added or removed. In order to never forget about these function calls, we can invoke them whenever the list of presenters is changed. After each change, we call `onCreate` on the new presenters (those that are now but were not before), and we call `onDestroy` on the removed presenters (those that were before and are not now).

```
var presenters: List<Presenter> by
observable(emptyList()) { _, old, new ->
    (new - old).forEach { it.onCreate() }
    (old - new).forEach { it.onDestroy() }
}
```

As you can see, there are many practical ways in which an `observable` delegate can be used. Now, let's talk about this delegate's brother, which also seems useful but is used much less often in practice.

The vetoable delegate

“Veto” is a Latin word meaning “I forbid”¹². The `vetoable` delegate is very similar to `observable`, but it can forbid property value changes. That is why the `vetoable` lambda expression is executed before the property value changes and returns the Boolean type, which determines if the property value should change or not. If this function returns `true`, the property value will change; if it returns `false`, the value will not change.

```
var prop: SomeType by Delegates.vetoable(initial, operation)

// Alternative to
var prop: SomeType = initial
    set(newValue) {
        if (operation(::prop, field, newValue)) {
            field = newValue
        }
    }
}
```

Here is a complete usage example:

```
import kotlin.properties.Delegates.vetoable

var smallList: List<String> by
vetoable(emptyList()) { _, _, new ->
    println(new)
    new.size <= 3
}

fun main() {
    smallList = listOf("A", "B", "C") // [A, B, C]
    println(smallList) // [A, B, C]
    smallList = listOf("D", "E", "F", "G") // [D, E, F, G]
    println(smallList) // [A, B, C]
```

¹²This word is well-known in Poland for historical reasons. You can read about Liberum Veto.

```
    smallList = listOf("H") // [H]
    println(smallList) // [H]
}
```

The `vetoable` delegate can be used when we have a property with some requirements on its value; whenever someone tries to modify this value, we first need to validate the new value. We might also invoke some actions when a new value is valid (like displaying it) or when it is not (like logging an error). So, this is a conceptual presentation of how `vetoable` could be used:

```
var name: String by vetoable("") { _, _, new ->
    if (isValid(new)) {
        showNewName(new)
        true
    } else {
        showNameError()
        false
    }
}
```

In practice, the `vetoable` delegate is not used very often, but some practical examples might include allowing only specific state changes in an application, or requiring only valid values.

```
import kotlin.properties.Delegates.vetoable

val state by vetoable(Initial) { _, _, newState ->
    if (newState is Initial) {
        log.e("Cannot set initial state")
        return@vetoable false
    }
    // ...
    return@vetoable true
}

val email by vetoable(email) { _, _, newEmail ->
```

```
    emailRegex.matches(newEmail)
}
```

A map as a delegate

The last delegate from Kotlin standard library is `Map`, which has keys of type `String`. When we use it as a delegate and we ask for a property value, the result is the value associated with this property name.

```
fun main() {
    val map: Map<String, Any> = mapOf(
        "name" to "Marcin",
        "kotlinProgrammer" to true
    )
    val name: String by map
    val kotlinProgrammer: Boolean by map
    println(name) // Marcin
    println(kotlinProgrammer) // true
}
```

How can `Map` be a delegate? To be able to use an object as a read-only property delegate, this object must have a `getValue` function. For `Map`, it is defined as an extension function in Kotlin stdlib.

```
operator fun <V, V1 : V> Map<String, V>.getValue(
    thisRef: Any?,
    property: KProperty<*>
): V1 {
    val key = property.name
    val value = get(key)
    if (value == null && !containsKey(key)) {
        throw NoSuchElementException(
            "Key ${property.name} is missing in the map."
        )
    } else {
        return value as V1
    }
}
```

```
    }  
}
```

So what are some use cases for using `Map` as a delegate? In most applications, you should not need it. However, you might be forced by an API to treat objects as maps that have some expected keys and some that might be added dynamically in the future. I mean situations like “This endpoint will return an object representing a user, with properties `id`, `displayName`, etc., and on the profile page you need to iterate over all these properties, including those that are not known in advance, and display an appropriate view for each of them”. For such a requirement, we need to represent an object using `Map`, then we can use this map as a delegate in order to more easily use properties we know we can expect.

```
class User(val map: Map<String, Any>) {  
    val id: Long by map  
    val name: String by map  
}  
  
fun main() {  
    val user = User(  
        mapOf<String, Any>(  
            "id" to 1234L,  
            "name" to "Marcin"  
        )  
    )  
    println(user.name) // Marcin  
    println(user.id) // 1234  
    println(user.map) // {id=1234, name=Marcin}  
}
```

`Map` can only be used for read-only properties because it is a read-only interface. For read-write properties, use `MutableMap`. Any delegated property change is a change in the map it delegates to, and any change in this map leads to a different property value. Delegating to a `MutableMap` map is like accessing the shared state of a read/write data source.

```
class User(val map: MutableMap<String, Any>) {  
    var id: Long by map  
    var name: String by map  
}  
  
fun main() {  
    val user = User(  
        mutableMapOf(  
            "id" to 123L,  
            "name" to "Alek",  
        )  
    )  
  
    println(user.name) // Alek  
    println(user.id) // 123  
  
    user.name = "Bolek"  
    println(user.name) // Bolek  
    println(user.map) // {id=123, name=Bolek}  
  
    user.map["id"] = 456  
    println(user.id) // 456  
    println(user.map) // {id=456, name=Bolek}  
}
```

Review of how variables work

A few years ago, the following puzzle was circulating in the Kotlin community. A few people asked me to explain it before I even included it in my Kotlin workshop.

```
class Population(var cities: Map<String, Int>) {  
    val sanFrancisco by cities  
    val tallinn by cities  
    val kotlin by cities  
}  
  
val population = Population(  
    mapOf(  
        "sanFrancisco" to 864_816,  
        "tallinn" to 413_782,  
        "kotlin" to 43_005  
    )  
)  
  
fun main(args: Array<String>) {  
    // Years has passed,  
    // now we all live on Mars  
    population.cities = emptyMap()  
    println(population.sanFrancisco)  
    println(population.tallinn)  
    println(population.kotlin)  
}
```

Before going any further, try to guess the answer.

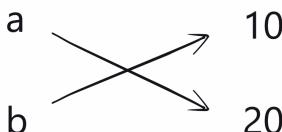
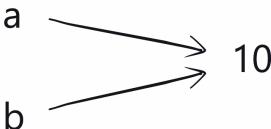
The behavior presented by this puzzle might be counterintuitive, but it is certainly correct. I will present the answer after I give a proper step-by-step rationale.

Let's start with a simpler example. Take a look at the following code.

```
fun main() {  
    var a = 10  
    var b = a  
    a = 20  
    println(b)  
}
```

What will be printed? The answer is 10 because variables are

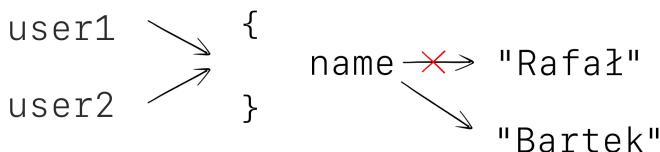
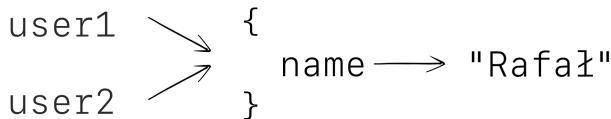
always assigned to values, never other variables. First, `a` is assigned to `10`, then `b` is assigned to `10`, then `a` changes and is assigned to `20`. This does not change the fact that `b` is assigned to `10`.



This picture gets more complicated when we introduce mutable objects. Take a look at the following snippet.

```
fun main() {
    val user1 = object {
        var name: String = "Rafał"
    }
    val user2 = user1
    user1.name = "Bartek"
    println(user2.name)
}
```

What will be printed? The answer is “Bartek”. Here both `user1` and `user2` reference the same object, and then this object changes internally.

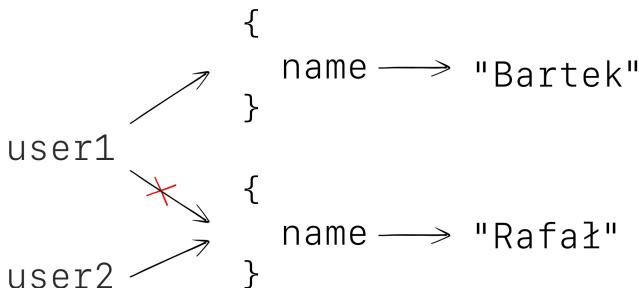
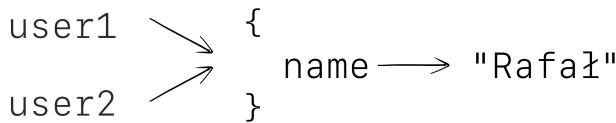


The situation would be different if we changed what `user1` references instead of changing the value of `name`.

```
interface Nameable {
    val name: String
}

fun main() {
    var user1: Nameable = object : Nameable {
        override var name: String = "Rafał"
    }
    val user2 = user1
    user1 = object : Nameable {
        override var name: String = "Bartek"
    }
    println(user2.name)
}
```

What is the answer? It is “Rafał”.



This is particularly confusing if we compare mutable lists with read-only lists referenced with `var`, especially since both can be changed with the `+=` sign. First, take a look at the following snippet:

```
fun main() {  
    var list1 = listOf(1, 2, 3)  
    var list2 = list1  
    list1 += 4  
    println(list2)  
}
```

What will be printed? The answer is `[1, 2, 3]`. It has to be this way because `list1` references a read-only list. This means that `list1 += 4` in this case means `list1 = list1 + 4`, and so a new list object is returned. Now, let's take a look at the following snippet:

```
fun main() {  
    val list1 = mutableListOf(1, 2, 3)  
    val list2 = list1  
    list1 += 4  
    println(list2)  
}
```

What will be printed? The answer is [1, 2, 3, 4] because `list1` references a mutable list. This means that `list1 += 4` in this case means `list1.plusAssign(4)`, i.e., `list1.add(4)`, and the same list object is returned. Now, consider using `Map` as a delegate:

```
fun main() {  
    var map = mapOf("a" to 10)  
    val a by map  
    map = mapOf("a" to 20)  
    println(a)  
}
```

Can you see that the answer should be 10? On the other hand, if the map were mutable, the answer would be different:

```
fun main() {  
    val mmap = mutableMapOf("a" to 10)  
    val a by mmap  
    mmap["a"] = 20  
    println(a)  
}
```

Can you see that the answer should be 20? This is consistent with the behavior of the other variables and with what properties are compiled to.

```
var map = mapOf("a" to 10)
// val a by map
// is compiled to
val `a$delegate` = map
val a: Int
    get() = `a$delegate`.getValue(null, ::a)

val mmap = mutableMapOf("b" to 10)
// val b by mmap
// is compiled to
val `b$delegate` = mmap
val b: Int
    get() = `b$delegate`.getValue(null, ::b)

fun main() {
    map = mapOf("a" to 20)
    println(a) // 10

    mmap["b"] = 20
    println(b) // 20
}
```

Finally, let's get back to our puzzle again. I hope you can see now that changing the `cities` property should not influence the value of `sanFrancisco`, `tallinn`, or `kotlin`. In Kotlin, we delegate to a delegate, not to a property, just like we assign a property to a value, not another property.

```
class Population(var cities: Map<String, Int>) {
    val sanFrancisco by cities
    val tallinn by cities
    val kotlin by cities
}

val population = Population(
    mapOf(
        "sanFrancisco" to 864_816,
        "tallinn" to 413_782,
        "kotlin" to 43_005
    )
)
```

```
)  
)  
  
fun main(args: Array<String>) {  
    // Years has passed,  
    // now we all live on Mars  
    population.cities = emptyMap()  
    println(population.sanFrancisco)  
    println(population.tallinn)  
    println(population.kotlin)  
}
```

To clear the populations, the `cities` map would have to be mutable, and using `population.cities.clear()` would cause `population.sanFrancisco` to fail.

Summary

In this chapter, you've learned how property delegation works and how to define custom property delegates. You've also learned about the most important property delegates from the Kotlin standard library: `Delegates.notNull`, `lazy`, `Delegates.observable`, `Delegates.vetoable`, `Map<String, T>` and `MutableMap<String, T>`.

I hope you will find this knowledge useful in your programming practice.

Exercise: Mutable lazy delegate

Delegate `lazy` can only be used to `val` variables. Implement `MutableLazy` delegate that can be used to `var` variables. It should behave like `lazy` delegate, but should allow to set the value. If getter is called before the property is set, it should initialize the property using lambda expression. If getter is called after the property is set, it should return the set value. If setter is called, it should set the value.

```
fun calculate(): Int {
    print("Calculating... ")
    return 42
}

var a by mutableLazy { calculate() }
println(a) // Calculating... 42
println(a) // 42
a = 1
println(a) // 1

var b by mutableLazy { calculate() }
b = 2
println(b) // 2
```

Starting code:

```
fun <T> mutableLazy(
    initializer: () -> T
): ReadWriteProperty<Any?, T> = TODO()
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file advanced/delegates/MutableLazy.kt. You can clone this project and solve this exercise locally.

Kotlin Contracts

One of Kotlin's most mysterious features is Kotlin Contracts. Most developers use them without even knowing how they work, or even how to define them. Kotlin Contracts make the Kotlin compiler smarter and allow us to do things that would otherwise be impossible. But before we see them in action, let's start with a contract definition which can be found in some Kotlin stdlib functions.

```
@kotlin.internal.InlineOnly
public inline fun CharSequence?.isNullOrEmpty(): Boolean {
    contract {
        returns(false) implies (this@isNullOrEmpty != null)
    }

    return this == null || this.isBlank()
}

public inline fun measureTimeMillis(block: () -> Unit): Long {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    val start = System.currentTimeMillis()
    block()
    return System.currentTimeMillis() - start
}
```

A Kotlin Contract is defined using the `contract` function, which for now can only be used as the start of the body of a top-level function. The `contract` function starts the DSL builder. This is a peculiar function because it is an inline function with an empty body.

```
@ContractsDsl
@ExperimentalContracts
@InlineOnly
@SinceKotlin("1.3")
@Suppress("UNUSED_PARAMETER")
inline fun contract(builder: ContractBuilder.() -> Unit) {  
}
```

Inline function calls are replaced with the body of these functions¹³. If this body is empty, it means that such a function call is literally replaced with nothing, so it is gone. So, why might we want to call a function if its call is gone during code compilation? Kotlin Contracts are a way of communicating with the compiler; therefore, it's good that they are replaced with nothing, otherwise they would only disturb and slow down our code after compilation. Inside Kotlin Contracts, we specify extra information that the compiler can utilize to improve the Kotlin programming experience. In the above example, the `isNullOrEmpty` contract specifies when the function returns `false`, thus the Kotlin compiler can assume that the receiver is not `null`. This information is used for smart-casting. The contract of `measureTimeMillis` specifies that the `block` function will be called in place exactly once. Let's see what this means and how exactly we can specify a contract.

The meaning of a contract

In the world of programming, a contract is a set of expectations on an element, library, or service. By “contract”, we mean what is “promised” by the creators of this solution in documentation, comments, or by explicit code structures¹⁴.

¹³I described inline functions in the *Inline functions* chapter in *Functional Kotlin*.

¹⁴This is known as *Design by contract*, the advantage of which is that it frees a function from having to handle cases outside of the precondition. Bertrand Meyer coined this term in connection with his design of the Eiffel programming language.

It's no wonder that this name is also used to describe language structures that are used to specify expectations on some code elements. For instance, in C++ there is a structure known as a contract that is used to demand certain conditions for the execution of a function:

```
// C++
int mul(int x, int y)
  [[expects: x > 0]]
  [[expects: y > 0]]
  [[ensures audit res: res > 0]]{
    return x * y;
}
```

In Kotlin, we use the `require` and `check` functions to specify expectations on arguments and states.

```
fun mul(x: Int, y: Int): Int {
    require(x > 0)
    require(y > 0)
    return x * y
}
```

The problem is with expressing messages that are directed to the compiler. For this, we could use annotations (which was actually considered when Kotlin Contracts were being designed), but their expressiveness is much more limited than DSL. So, the creators of Kotlin defined a DSL that starts function definitions. Let's see what we can express using Kotlin Contracts.

How many times do we invoke a function from an argument?

As part of a contract, we can use `callsInPlace` to guarantee that a parameter with a functional type is called in place during function execution. Using a value from the `InvocationKind` enum, this structure can also be used to specify how many times this function is executed.

```
@kotlin.internal.InlineOnly
public inline fun <R> run(block: () -> R): R {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    return block()
}
```

There are four possible kinds of invocation:

- EXACTLY_ONCE - specifies that this function will be called exactly once.
- AT_MOST_ONCE - specifies that this function will be called at most once.
- AT_LEAST_ONCE - specifies that this function will be called at least once.
- UNKNOWN - does not specify the number of function invocations.

Execution of `callsInPlace` is information to the compiler, which can use this information in a range of situations. For instance, a read-only variable cannot be reassigned, but it can be initialized separately from the definition.

```
fun main() {
    val i: Int
    i = 42
    println(i) // 42
}
```

When a functional parameter is specified to be called exactly once, a read-only property can be defined outside a lambda expression and initialized inside it.

```
fun main() {
    val i: Int
    run {
        i = 42
    }
    println(i) // 42
}
```

Consider the `result` variable, which is defined in the `forceExecutionTimeMillis` function and initialized in the lambda expression in the `measureTimeMillis` function. This is possible only because the `measureTimeMillis` parameter block is defined to be called in place exactly once.

```
suspend fun <T, R> forceExecutionTimeMillis(
    timeMillis: Long, block: () -> R
): R {
    val result: R
    val timeTaken = measureTimeMillis {
        result = block()
    }
    val timeLeft = timeMillis - timeTaken
    delay(timeLeft)
    return result
}

@OptIn(ExperimentalContracts::class)
inline fun measureTimeMillis(block: () -> Unit): Long {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    val start = System.currentTimeMillis()
    block()
    return System.currentTimeMillis() - start
}
```

Contracts are still an experimental feature. To define contracts in your own functions, you need to

use `@OptIn(ExperimentalContracts::class)` annotation with `ExperimentalContracts`. It is extremely unlikely Kotlin will drop this functionality, but its API might change.

The `EXACTLY_ONCE` invocation kind is the most popular because it offers the most advantages. With this invocation kind, read-only properties can only be initialized in blocks. Read-write properties can also be initialized and re-initialized in blocks whose invocation kind is `AT_LEAST_ONCE`.

```
@OptIn(ExperimentalContracts::class)
fun checkTextEverySecond(callback: (String) -> Unit) {
    contract {
        callsInPlace(callback, InvocationKind.AT_LEAST_ONCE)
    }
    val task = object : TimerTask() {
        override fun run() {
            callback(getCurrentText())
        }
    }
    task.run()
    Timer().schedule(task, 1000, 1000)
}

fun main() {
    var text: String
    checkTextEverySecond {
        text = it
    }
    println(text)
}
```

In the above code, only the first call of `callback` is called in place, so this contract is not completely correct.

Another example of how the Kotlin Compiler uses the information specified by the `callsInPlace` function is when a

statement inside a lambda is terminal for function execution (declares the `Nothing` result type¹⁵), therefore everything after it is unreachable. Thanks to the contract, this might include statements outside this lambda.

```
fun main() {
    run {
        println("A")
        return
        println("B") // unreachable
    }
    println("C") // unreachable
}
```

I've seen this used in many projects to return from a lambda expression.

```
fun makeDialog(): Dialog {
    DialogBuilder().apply {
        title = "Alert"
        setPositiveButton("OK") { /*...*/ }
        setNegativeButton("Cancel") { /*...*/ }
        return create()
    }
}

fun readFirstLine(): String? = File("XYZ")
    .useLines {
        return it.firstOrNull()
    }
```

Implications of the fact that a function has returned a value

Another kind of contract statement includes the `returns` function and the infix `implies` function to imply some value-type

¹⁵For details, see *The beauty of the Kotlin type system* chapter in *Kotlin Essentials*.

implications based on the result of the function. The compiler uses this feature for smart-casting. Consider the `isNullOrEmpty` function, which returns `true` if the receiver collection is `null` or empty. Its contract states that if this function returns `false`, the compiler can infer that the receiver is not `null`.

```
inline fun <T> Collection<T>?.isNullOrEmpty(): Boolean {
    contract {
        returns(false) implies (this@isNullOrEmpty != null)
    }

    return this == null || this.isEmpty()
}

fun printEachLine(list: List<String>?) {
    if (!list.isNullOrEmpty()) {
        for (e in list) { //list smart-casted to List<String>
            println(e)
        }
    }
}
```

Here is a custom example in which the compiler can infer that the receiver is of type `Loading` because `startedLoading` returns `true`.

```
@OptIn(ExperimentalContracts::class)
fun VideoState.startedLoading(): Boolean {
    contract {
        returns(true) implies (this@startedLoading is Loading)
    }
    return this is Loading && this.progress > 0
}
```

Currently, the `returns` function can only use `true`, `false`, and `null` as arguments. The implication must be either a parameter (or receiver) that is of some type or is not `null`.

Using contracts in practice

Some important functions from Kotlin stdlib define their contract; we benefit from this as it makes the compiler smarter. Some developers don't even notice how they use contracts and might be surprised to see how Kotlin allows something that would not be allowed in other languages, like smart-casting a variable in an unusual case. Having said that, you don't even need to know how a contract works in order to benefit from it.

Defining contracts in our own top-level functions is extraordinarily rare, even for library creators. I've seen it done in some projects, but most projects don't even have a single custom function with a specified contract. However, it's good to understand contracts because they can be really helpful in some situations. A good example is presented in the article [Slowing down your code with Coroutines](#) by Jan Vladimir Mostert, which presents a technique to make some requests take a specific time. Consider the code below. The function `measureCoroutineTimedValue` needs to return the measured time as well as the value which is calculated during its execution. To measure time, it uses `measureCoroutineDuration`, which returns `Duration`. To store the result of the body, it needs to define a variable. The body of `measureCoroutineTimedValue` only works because `measureCoroutineDuration` is defined in its `callsInPlace` contract

with `InvocationKind.EXACTLY_ONCE`.

```
@OptIn(ExperimentalContracts::class)
suspend fun measureCoroutineDuration(
    body: suspend () -> Unit
): Duration {
    contract {
        callsInPlace(body, InvocationKind.EXACTLY_ONCE)
    }
    val dispatcher = coroutineContext[ContinuationInterceptor]
    return if (dispatcher is TestDispatcher) {
        val before = dispatcher.scheduler.currentTimeMillis
    }
}
```

```
        body()
        val after = dispatcher.scheduler.currentTime
        after - before
    } else {
        measureTimeMillis {
            body()
        }
    }.milliseconds
}

@OptIn(ExperimentalContracts::class)
suspend fun <T> measureCoroutineTimedValue(
    body: suspend () -> T
): TimedValue<T> {
    contract {
        callsInPlace(body, InvocationKind.EXACTLY_ONCE)
    }
    var value: T
    val duration = measureCoroutineDuration {
        value = body()
    }
    return TimedValue(value, duration)
}
```

Summary

Kotlin Contracts let us specify information that is useful for the compiler. They help us define functions that are more convenient to use. Kotlin Contracts are defined in some Kotlin stdlib functions, like `run`, `let`, `also`, `use`, `measureTime`, `isNullOrBlank`, and many more; this makes their usage more elastic and supports better smart-casting. We rarely define contracts ourselves, but it's good to know about them and what they offer.

Exercise: Coroutine time measurement

You defined a function to measure block execution time, that works for virtual time as well:

```
suspend fun measureCoroutine(  
    body: suspend () -> Unit  
) : Duration {  
    val dispatcher = coroutineContext[ContinuationInterceptor]  
    return if (dispatcher is TestDispatcher) {  
        val before = dispatcher.scheduler.currentTimeMillis  
        body()  
        val after = dispatcher.scheduler.currentTimeMillis  
        after - before  
    } else {  
        measureTimeMillis {  
            body()  
        }  
    }.milliseconds  
}
```

However, you found that it is not very convenient to use because it lacks contract. Define it, to make the following code compile:

```
suspend fun main() {  
    runTest {  
        val result: String  
        val duration = measureCoroutine {  
            delay(1000)  
            result = "OK"  
        }  
        println(duration) // 1000 ms  
        println(result) // OK  
    }  
  
    runBlocking {  
        val result: String
```

```
val duration = measureCoroutine {
    delay(1000)
    result = "OK"
}
println(duration) // 1000 ms
println(result) // OK
}
```

Starting code and usage examples can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file advanced/contract/MesureCoroutineTime.kt. You can clone this project and solve this exercise locally.

Java interoperability

Kotlin is derived from the JVM platform, so Kotlin/JVM is its most mature flavor, compared to, e.g., Kotlin/JS or Kotlin/Native. But Kotlin and other JVM languages, like Java, are still different programming languages, therefore some challenges are inevitable when trying to get these languages to cooperate. So, some extra effort might be needed to make them work together as smoothly as possible. Let's see some examples.

Nullable types

Java cannot mark that a type is not nullable as all its types are considered nullable (except for primitive types). In trying to correct this flaw, Java developers started using `Nullable` and `NotNull` annotations from a number of libraries that define such annotations. These annotations are helpful but do not offer the safety that Kotlin offers. Nevertheless, in order to respect this convention, Kotlin also marks its types using `Nullable` and `NotNull` annotations when compiled to JVM¹⁶.

```
class MessageSender {
    fun sendMessage(title: String, content: String?) {}
}
```

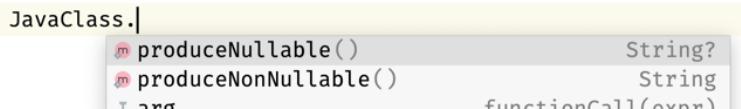
¹⁶Kotlin, when compiled to JVM, uses `Nullable` and `NotNull` annotations from the `org.jetbrains.annotations` package.

```
// Compiles to the analog of the following Java code
final class MessageSender {
    public void sendMessage(
        @NotNull String title,
        @Nullable String content
    ) {
        Intrinsics.checkNotNullParameter(title, "title");
    }
}
```

On the other hand, when Kotlin sees Java types with `Nullable` and `NotNull` annotations, it treats these types accordingly as nullable and non-nullable types¹⁷.

```
public class JavaClass {
    public static @NotNull String produceNonNullable() {
        return "ABC";
    }

    public static @Nullable String produceNullable() {
        return null;
    }
}
```



This makes interoperability between Kotlin and Java automatic in most cases. The problem is that when a Java type is not annotated, Kotlin does not know if it should be considered nullable or not. One could assume that a nullable type should be used in such a case, but this approach does not work well. Just consider a Java function that returns `Observable<List<User>>`, which in Kotlin is seen as

¹⁷Kotlin supports `Nullable` and `NotNull` annotations from a variety of libraries, including JSR-305, Eclipse, and Android.

`Observable<List<User?>?>?`. There would be so many types to unpack, even though we know none of them should actually be nullable.

```
// Java
public class UserRepo {

    public Observable<List<User>> fetchUsers() {
        /**
     }

}

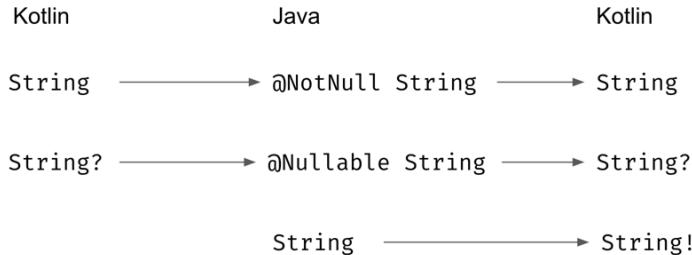
// Kotlin, if unannotated types were considered nullable
val repo = UserRepo()
val users: Observable<List<User>> = repo.fetchUsers()!!
    .map { it!! .map { it!! } }
```

This is why Kotlin introduced the concept of *platform type*, which is a type that comes from another language and has unknown nullability. Platform types are notated with a single exclamation mark ! after the type name, such as `String!`, but this notation cannot be used in code. Platform types are non-denotable, meaning that they can't be written explicitly in code. When a platform value is assigned to a Kotlin variable or property, it can be inferred, but it cannot be explicitly set. Instead, we can choose the type that we expect: either a nullable or a non-null type.

```
// Kotlin
val repo = UserRepo()
val user1 = repo.fetchUsers()
// The type of user1 is Observable<List<User!>!>!
val user2: Observable<List<User>> = repo.fetchUsers()
val user3: Observable<List<User?>?>? = repo.fetchUsers()
```

Casting platform types to non-nullable types is better than not specifying a type at all, but it is still dangerous because

something we assume is non-null might be null. This is why, for safety reasons, I always suggest being very careful when we get platform types from Java¹⁸.



Kotlin type mapping

Nullability is not the only source of differences between Kotlin and Java types. Many basic Java types have Kotlin alternatives. For instance, `java.lang.String` in Java maps to `kotlin.String` in Kotlin. This means that when a Kotlin file is compiled to Java, `kotlin.String` becomes `java.lang.String`. This also means that `java.lang.String` in a Java file is treated like it is `kotlin.String`. You could say that `kotlin.String` is type aliased to `java.lang.String`. Here are a few other Kotlin types with associated Java types:

¹⁸More about handling platform types in Effective Kotlin, Item 3: Eliminate platform types as soon as possible.

Kotlin type	Java type
kotlin.Any	java.lang.Object
kotlin.Cloneable	java.lang.Cloneable
kotlin.Comparable	java.lang.Comparable
kotlin.Enum	java.lang.Enum
kotlin.Annotation	java.lang.Annotation
kotlin.Deprecated	java.lang.Deprecated
kotlin.CharSequence	java.lang.CharSequence
kotlin.String	java.lang.String
kotlin.Number	java.lang.Number
kotlin.Throwable	java.lang.Throwable

Type mapping is slightly more complicated for primitive types and types that represent collections, so let's discuss them.

JVM primitives

Java has two kinds of values: objects and primitives. In Kotlin, all values are objects, but the `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Char`, and `Boolean` types use primitives under the hood. So, for example, when you use `Int` as a parameter, it will be `int` under the hood.

```
// KotlinFile.kt
fun multiply(a: Int, b: Int) = a * b

// Compiles to the analog of the following Java code
public final class KotlinFileKt {
    public static final int multiply(int a, int b) {
        return a * b;
    }
}
```

Kotlin uses primitives whenever possible. Here is a table of all types that are compiled into primitives.

Kotlin type	Java type
Byte	byte
Short	short
Int	int
Long	long
Float	float
Double	double
Char	char
Boolean	boolean

Primitives are not nullable on JVM, so nullable Kotlin types are always compiled into non-primitive types. For types that could be otherwise represented as primitives, we say these are *wrapped types*. Classes like `Integer` or `Boolean` are just simple wrappers over primitive values.

Kotlin type	Java type
Byte?	Byte
Short?	Short
Int?	Integer
Long?	Long
Float?	Float
Double?	Double
Char?	Char
Boolean?	Boolean

Primitives cannot be used as generic type arguments, so collections use wrapped types instead.

Kotlin type	Java type
List<Int>	List<Integer>
Set<Long>	Set<Long>
Map<Float, Double>	Map<Float, Double>

Only arrays can store primitives, so Kotlin introduced special types to represent arrays of primitives. For instance, to represent an array of primitive ints, we use `IntArray`, where `Array<Int>` represents an array of wrapped types.

Kotlin type	Java type
<code>Array<Int></code>	<code>Integer[]</code>
<code>IntArray</code>	<code>int[]</code>

Similar array types are defined for all primitive Java types.

Kotlin type	Java type
<code>ByteArray</code>	<code>byte[]</code>
<code>ShortArray</code>	<code>short[]</code>
<code>IntArray</code>	<code>int[]</code>
<code>LongArray</code>	<code>long[]</code>
<code>FloatArray</code>	<code>float[]</code>
<code>DoubleArray</code>	<code>double[]</code>
<code>CharArray</code>	<code>char[]</code>
<code>BooleanArray</code>	<code>boolean[]</code>

Using these types as an array type argument makes an array or arrays.

Kotlin type	Java type
<code>Array<IntArray></code>	<code>int[][]</code>
<code>Array<Array<LongArray>></code>	<code>long[][][]</code>
<code>Array<Array<Int>></code>	<code>Integer[][]</code>
<code>Array<Array<Array<Long>>></code>	<code>Long[][][]</code>

Collection types

Kotlin introduced a distinction between read-only and mutable collection types, but this distinction is missing in Java. For instance, in Java we have the `List` interface, which includes

methods that allow list modification. But in Java we also use immutable collections, and they implement the same interface. Their methods for collection modification, like `add` or `remove`, throw `UnsupportedOperationException` if they are called.

```
// Java
public final class JavaClass {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3);
        numbers.add(4); // UnsupportedOperationException
    }
}
```

This is a clear violation of the *Interface Segregation Principle*, an important programming principle that states that no code should be forced to depend on methods it does not use. This is an essential Java flaw, but this is hard to change due to backward compatibility.

Kotlin introduced a distinction between read-only and mutable collection types, which works perfectly when we operate purely on Kotlin projects. The problem arises when we need to interoperate with Java. When Kotlin sees a non-Kotlin file with an element expecting or returning the `List` interface, it does not know if this list should be considered mutable or not, so the result type is `(Mutable)List`, which can be used as both `List` and `MutableList`.

```
JavaClass.~
    m consumeList(list: (Mutable)List<Int!>!): Unit
    m produceList(): (Mutable)List<Int!>!
    ± arg: functionCall(expr)
```

On the other hand, when we use Kotlin code from Java, we have the opposite problem: Java does not distinguish between mutable and read-only lists at the interface level, so both these types are treated as `List`.

```
// KotlinFile.kt
fun readOnlyList(): List<Int> = listOf(1, 2, 3)
fun mutableListOf(): MutableList<Int> = mutableListOf(1, 2, 3)

// Compiles to analog of the following Java code
public final class KotlinFileKt {
    @NotNull
    public static final List readOnlyList() {
        return CollectionsKt
            .listOf(new Integer[]{1, 2, 3});
    }

    @NotNull
    public static final List mutableListOf() {
        return CollectionsKt
            .mutableListOf(new Integer[]{1, 2, 3});
    }
}
```

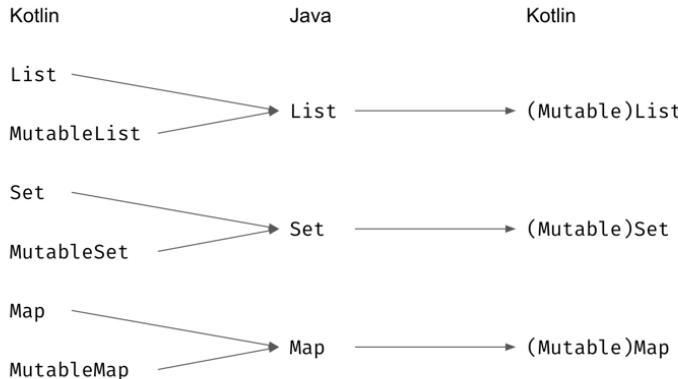


This interface has mutating methods like `add` or `remove`, so using them in Java might lead to exceptions.

```
// Java
public final class JavaClass {
    public static void main(String[] args) {
        List<Integer> integers = KotlinFileKt.readOnlyList();
        integers.add(20); // UnsupportedOperationException
    }
}
```

This fact can be problematic. A read-only Kotlin list might be mutated in Java because the transition to Java down-casts it to mutable. This is a hack that breaks the Kotlin `List` type contract¹⁹.

By default, all objects that implement a Kotlin `List` interface on JVM implement Java `List`, so methods like `add` or `remove` are generated for them. The default implementations of these methods throw `UnsupportedOperationException`. The same can be said about `Set` and `Map`.



¹⁹See Effective Kotlin, Item 32: Respect abstraction contracts.

Annotation targets

Kotlin introduced the property concept, which translates to Java getters, optional setters, fields, and delegates.

```
import kotlin.properties.Delegates.notNull

class User {
    var name = "ABC" // getter, setter, field
    var surname: String by notNull()//getter, setter, delegate
    val fullName: String // only getter
        get() = "$name $surname"
}

// Compiles to the analog of the following Java code
public final class User {
    // $FF: synthetic field
    static final KProperty[] $$delegatedProperties = ...

    @NotNull
    private String name = "ABC";

    @NotNull
    private final ReadWriteProperty surname$delegate;

    @NotNull
    public final String getName() {
        return this.name;
    }

    public final void setName(@NotNull String var1) {
        Intrinsics.checkNotNullParameter(var1, "<set->");
        this.name = var1;
    }

    @NotNull
    public final String getSurname() {
        return (String) this.surname$delegate
```

```
        .getValue(this, $$delegatedProperties[0]);  
    }  
  
    public final void setSurname(@NotNull String v) {  
        Intrinsics.checkNotNullParameter(v, "<set->");  
        this.surname$delegate  
            .setValue(this, $$delegatedProperties[0], v);  
    }  
  
    @NotNull  
    public final String getFullName() {  
        return this.name + ' ' + this.getSurname();  
    }  
  
    public User() {  
        this.surname$delegate = Delegates.INSTANCE.notNull();  
    }  
}
```

This creates a problem when we annotate a property because if a property translates to many elements under the hood, like a getter or a field, how can we annotate a concrete one on JVM? In other words, how can we annotate a getter or a field?

```
class User {  
    @SomeAnnotation  
    var name = "ABC"  
}
```

When you annotate a Kotlin element that is used to generate multiple Java elements, you can specify a use-side target for an annotation. For instance, to mark that `SomeAnnotation` should be used for a property field, we should use `@field:SomeAnnotation`.

```
class User {  
    @field:SomeAnnotation  
    var name = "ABC"  
}
```

For a property, the following targets are supported:

- `property` (annotations with this target are not visible to Java)
- `field` (property field)
- `get` (property getter)
- `set` (property setter)
- `setparam` (property setter parameter)
- `delegate` (the field storing the delegate instance for a delegated property)

```
annotation class A  
annotation class B  
annotation class C  
annotation class D  
annotation class E  
  
class User {  
    @property:A  
    @get:B  
    @set:C  
    @field:D  
    @setparam:E  
    var name = "ABC"  
}
```

```
// Compiles to the analog of the following Java code
public final class User {
    @D
    @NotNull
    private String name = "ABC";

    @A
    public static void getName$annotations() {
    }

    @B
    @NotNull
    public final String getName() {
        return this.name;
    }

    @C
    public final void setName(@E @NotNull String var1) {
        Intrinsics.checkNotNullParameter(var1, "<set->");
        this.name = var1;
    }
}
```

When a property is defined in a constructor, an additional `param` target is used to annotate the constructor parameter.

```
class User(
    @param:A val name: String
)
```

By default, the annotation target is chosen according to the `@Target` annotation of the annotation being used. If there are multiple applicable targets, the first applicable target from the following list is used:

- `param`
- `property`
- `field`

Note that property annotations without an annotation target will by default use `property`, so they will not be visible from Java reflection.

```
annotation class A

class User {
    @A
    val name = "ABC"
}

// Compiles to the analog of the following Java code
public final class User {
    @NotNull
    private String name = "ABC";

    @A
    public static void getName$annotations() {
    }

    @NotNull
    public final String getName() {
        return this.name;
    }
}
```

An annotation in front of a class annotates this class. To annotate the primary constructor, we need to use the `constructor` keyword and place the annotation in front of it.

```
annotation class A
annotation class B

@a
class User @B constructor(
    val name: String
)

// Compiles to the analog of the following Java code
@a
public final class User {
    @NotNull
    private final String name;

    @NotNull
    public final String getName() {
        return this.name;
    }

    @B
    public User(@NotNull String name) {
        Intrinsics.checkNotNullParameter(name, "name");
        super();
        this.name = name;
    }
}
```

We can also annotate a file using the `file` target and place an annotation at the beginning of the file (before the package). An example will be shown in the `JvmName` section.

When you annotate an extension function or an extension property, you can also use the `receiver` target to annotate the `receiver` parameter.

```
annotation class Positive

fun @receiver:Positive Double.log() = ln(this)

// Java alternative
public static final double log(@Positive double $this$log) {
    return Math.log($this$log);
}
```

Static elements

In Kotlin, we don't have the concept of static elements, so use object declarations and companion objects instead. Using them in Kotlin is just like using static elements in Java.

```
import java.math.BigDecimal

class Money(val amount: BigDecimal, val currency: String) {
    companion object {
        fun usd(amount: Double) =
            Money(amount.toBigDecimal(), "PLN")
    }
}

object MoneyUtils {
    fun parseMoney(text: String): Money = TODO()
}

fun main() {
    val m1 = Money.usd(10.0)
    val m2 = MoneyUtils.parseMoney("10 EUR")
}
```

However, using these objects from Java is not very convenient. To use an object declaration, we need to use the static INSTANCE field, which is called Companion for companion objects.

```
// Java
public class JavaClass {
    public static void main(String[] args) {
        Money m1 = Money.Companion.usd(10.0);
        Money m2 = MoneyUtils.INSTANCE.parseMoney("10 EUR");
    }
}
```

It's important to know that to use any Kotlin element in Java, a package must be specified. Kotlin allows elements without packages, but Java does not.

To simplify the use of these object declaration²⁰ methods, we can annotate them with `@JvmStatic`, which makes the compiler generate an additional static method to support easier calls to non-Kotlin JVM languages.

```
// Kotlin
class Money(val amount: BigDecimal, val currency: String) {
    companion object {
        @JvmStatic
        fun usd(amount: Double) =
            Money(amount.toBigDecimal(), "PLN")
    }
}

object MoneyUtils {
    @JvmStatic
    fun parseMoney(text: String): Money = TODO()
}

fun main() {
    val money1 = Money.usd(10.0)
    val money2 = MoneyUtils.parseMoney("10 EUR")
}
```

²⁰A companion object is also an object declaration, as I explained in the Kotlin Essentials book.

```
// Java
public class JavaClass {
    public static void main(String[] args) {
        Money m1 = Money.usd(10.0);
        Money m2 = MoneyUtils.parseMoney("10 EUR");
    }
}
```

JvmField

As we've discussed already, each property is represented by its accessors. This means that if you have a property `name` in Kotlin, you need to use the `getName` getter to use it in Java; if a property is read-write, you need to use the `setName` setter.

```
// Kotlin
class Box {
    var name = ""
}

// Java
public class JavaClass {
    public static void main(String[] args) {
        Box box = new Box();
        box.setName("ABC");
        System.out.println(box.getName());
    }
}
```

The `name` field is private in `Box`, so we can only access it using accessors. However, some libraries that use reflection require the use of public fields, which are provided using the `JvmField` annotation for a property. Such properties cannot have custom accessors, use a delegate, be open, or override another property.

```
// Kotlin
class Box {
    @JvmField
    var name = ""
}

// Java
public class JavaClass {
    public static void main(String[] args) {
        Box box = new Box();
        box.name = "ABC";
        System.out.println(box.name);
    }
}
```

When the `JvmField` annotation is used for an annotation in an object declaration or a companion object, its field also becomes static.

```
// Kotlin
object Box {
    @JvmField
    var name = ""
}

// Java
public class JavaClass {
    public static void main(String[] args) {
        Box.name = "ABC";
        System.out.println(Box.name);
    }
}
```

Constant variables do not need this annotation as they will always be represented as static fields.

```
// Kotlin
class MainWindow {
    // ...

    companion object {
        const val SIZE = 10
    }
}

// Java
public class JavaClass {
    public static void main(String[] args) {
        System.out.println(MainWindow.SIZE);
    }
}
```

Using Java accessors in Kotlin

Kotlin properties are represented by Java accessors. A typical accessor is just a property name capitalized with a `get` or `set` prefix. The only exception is Boolean properties prefixed with “`is`”, whose getter name is the same as the property name, and its setter skips the “`is`” prefix.

```
class User {
    var name = "ABC"
    var isAdult = true
}
```

```
// Java alternative
public final class User {
    @NotNull
    private String name = "ABC";
    private boolean isAdult = true;

    @NotNull
    public final String getName() {
        return this.name;
    }

    public final void setName(@NotNull String var1) {
        Intrinsics.checkNotNullParameter(var1, "<set->");
        this.name = var1;
    }

    public final boolean isAdult() {
        return this.isAdult;
    }

    public final void setAdult(boolean var1) {
        this.isAdult = var1;
    }
}
```

Java getters and setters can be treated as properties in Kotlin. A getter without a setter is treated like a `val` property. When there is both a getter and a setter, they are treated together like a `var` property. A setter without a getter cannot be interpreted as a property because every property needs a getter.

JvmName

JVM has some platform limitations that are a result of its implementation. Consider the following two functions:

```
fun List<Long>.average() = sum().toDouble() / size
fun List<Int>.average() = sum().toDouble() / size
```

Each of these functions is an extension on `List` with a different type argument. This is perfectly fine for Kotlin, but not when targeted for the JVM platform. Due to *type erasure*, both these functions on JVM are considered methods called `average` with a single parameter of type `List`. So, having them both defined in the same file leads to a platform name clash.

```
11 fun List<Long>.average() = sum().toDouble() / size
12 fun List<Int>.average() = sum().toDouble() / size
13
```

Platform declaration clash: The following declarations have the same JVM signature (`average(Ljava/util/List;)D`):

- `public fun List<Int>.average(): Double` defined in file `KotlinPerson.kt`
- `public fun List<Long>.average(): Double` defined in file `KotlinPerson.kt`

A simple solution to this problem is to use the `@JvmName` annotation, which changes the name that will be used for this function under the hood on the JVM platform. Usage will not change in Kotlin, but if you want to use these functions from a non-Kotlin JVM language, you need to use the names specified in the annotation.

```
@JvmName("averageLongList")
fun List<Long>.average() = sum().toDouble() / size

@JvmName("averageIntList")
fun List<Int>.average() = sum().toDouble() / size

fun main() {
    val ints: List<Int> = List(10) { it }
    println(ints.average()) // 4.5
    val longs: List<Long> = List(10) { it.toLong() }
    println(longs.average()) // 4.5
}
```

```
// Java
public class JavaClass {
    public static void main(String[] args) {
        List<Integer> ints = List.of(1, 2, 3);
        double res1 = TestKt.averageIntList(ints);
        System.out.println(res1); // 2.0
        List<Long> longs = List.of(1L, 2L, 3L);
        double res2 = TestKt.averageLongList(longs);
        System.out.println(res2); // 2.0
    }
}
```

The `JvmName` annotation is useful for resolving name conflicts, but there is another case where it is used much more often. As I explained in the Kotlin Essentials book, for all Kotlin top-level functions and properties on JVM, a class is generated whose name is the file name with the “Kt” suffix. Top-level functions and properties are compiled to static JVM functions in this class and thus can be used from Java.

```
package test

const val e = 2.71

fun add(a: Int, b: Int) = a + b

// Compiles to the analog of the following Java code
package test;

public final class TestKt {
    public static final double e = 2.71;

    public static final int add(int a, int b) {
        return a + b;
    }
}
```

```
// Usage from Java
public class JavaClass {
    public static void main(String[] args) {
        System.out.println(TestKt.e); // 2.71
        int res = TestKt.add(1, 2);
        System.out.println(res); // 3
    }
}
```

This auto-generated name is not always what we want to use. Often we would prefer to specify a custom name, in which case we should use the `JvmName` annotation for the file. As I explained in the Annotation targets section, file annotations must be placed at the beginning of the file, even before the package definition, and they must use the `file` target. The name that we will specify in the `JvmName` file annotation will be used for the class that stores all the top-level functions and properties.

```
@file:JvmName("Math")

package test

const val e = 2.71

fun add(a: Int, b: Int) = a + b

// Compiles to the analog of the following Java code
package test;

public final class Math {
    public static final double e = 2.71;

    public static final int add(int a, int b) {
        return a + b;
    }
}
```

```
// Usage from Java
public class JavaClass {
    public static void main(String[] args) {
        System.out.println(Math.e); // 2.71
        int res = Math.add(1, 2);
        System.out.println(res); // 3
    }
}
```

JvmMultifileClass

Because all functions or fields on JVM must be located in a class, it is common practice in Java projects to make huge classes like `Math` or `Collections` that are holders for static elements. In Kotlin, we use top-level functions instead, which offers us the convenience that we don't need to collect all these functions and properties in the same file. However, when we design Kotlin code for use from Java, we might want to collect elements from multiple files that define the same package in the same generated class. For that, we need to use the `JvmMultifileClass` annotation next to `JvmName`.

```
// FooUtils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package demo

fun foo() {
    // ...
}
```

```
// BarUtils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package demo

fun bar() {
    // ...
}

// Usage from Java

import demo.Utils;

public class JavaClass {
    public static void main(String[] args) {
        Utils.foo();
        Utils.bar();
    }
}
```

JvmOverloads

Another inconsistency between Java and Kotlin is a result of another Java limitation. Java, unlike most modern programming languages, does not support named optional arguments, therefore default Kotlin arguments cannot be used in Java.

```
class Pizza(
    val tomatoSauce: Int = 1,
    val cheese: Int = 0,
    val ham: Int = 0,
    val onion: Int = 0,
)

class EmailSender {
    fun send(
```

```
    receiver: String,  
    title: String = "",  
    message: String = "",  
)  
{  
    /*...*/  
}  
}
```



A screenshot of an IDE showing code completion for the 'new' keyword. The code being typed is 'Pizza pizza = **new**'. A dropdown menu shows two constructor options for the 'Pizza' class:

- new Pizza()**
- new Pizza(int tomatoSauce, int cheese, int ham, int onion)**

The first option is highlighted. The IDE interface includes a status bar at the bottom with the text 'Press ⌘ to insert, ⌘ to replace Next Tip'.



A screenshot of an IDE showing code completion for 'sender.'. The code being typed is 'EmailSender sender = **new** EmailSender(); sender.**.**'. A dropdown menu shows the 'send' method of the 'EmailSender' class:

- send(String receiver, String title, String message)** void

The method name 'send' is highlighted. The IDE interface includes a status bar at the bottom with the text 'Press ⌘ to insert, ⌘ to replace Next Tip'.

Note that when all constructor parameters are optional, two constructors are generated: one with all the parameters, and one that uses only default values.

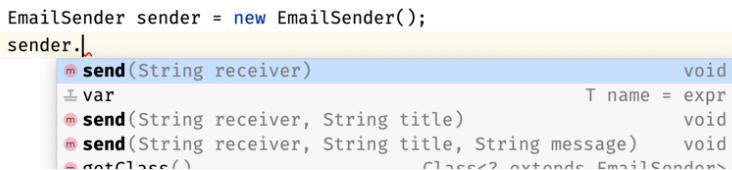
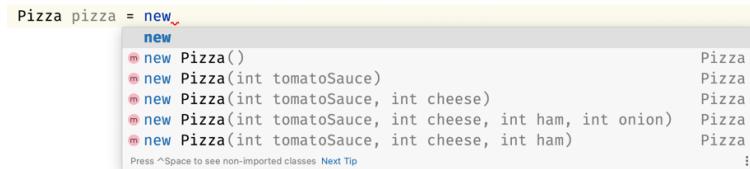
The best that can be offered for Java is an implementation of the telescoping constructor pattern so that Kotlin can generate different variants of a constructor or function for a different number of arguments. This is not done by default in order to avoid generating methods that are not used. So, you need to use the `jvmOverloads` annotation before a function to make Kotlin generate different variants with different numbers of expected arguments.

```

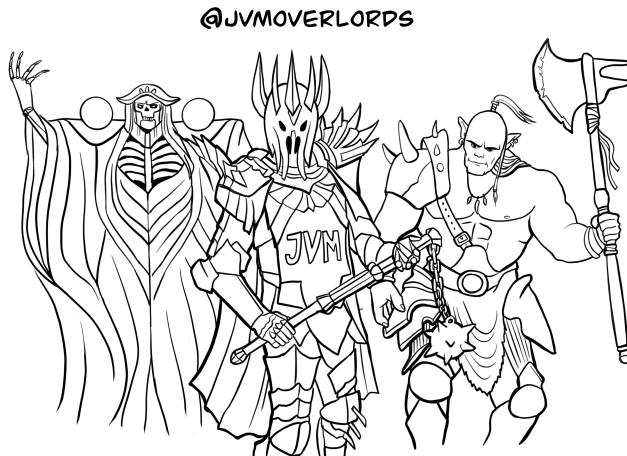
class Pizza @JvmOverloads constructor(
    val tomatoSauce: Int = 1,
    val cheese: Int = 0,
    val ham: Int = 0,
    val onion: Int = 0,
)

class EmailSender {
    @JvmOverloads
    fun send(
        receiver: String,
        title: String = "",
        message: String = "",
    ) {
        /*...*/
    }
}

```



Notice that Kotlin does not generate all possible combinations of parameters; it only generates one additional variant for each optional parameter.



Do not confuse JvmOverloads with JVM overlords.

Unit

In Kotlin, every function declares a return type, so we return `Unit` in Kotlin instead of the Java `void` keyword.

```
fun a() {}

fun main() {
    println(a()) // kotlin.Unit
}
```

Of course, practically speaking it would be inefficient to always return `Unit` even when it's not needed; so, functions with the `Unit` result type are compiled to functions with no result. When their result type is needed, it is injected on the use side.

```
// Kotlin code
fun a(): Unit {
    return Unit
}

fun main() {
    println(a()) // kotlin.Unit
}

// Compiled to the equivalent of the following Java code
public static final void a() {
}

public static final void main() {
    a();
    Unit var0 = Unit.INSTANCE;
    System.out.println(var0);
}
```

This is a performance optimization. The same process is used for Java functions without a result type, therefore they can be treated like they return `Unit` in Kotlin.

Function types and function interfaces

Using function types from Java can be problematic. Consider the following `setListItemClickListener` function, which expects a function type as an argument. Thanks to named arguments, Kotlin will provide proper suggestions and the usage will be convenient.

```

class ListAdapter {

    fun setListItemListener(
        listener: (
            position: Int,
            id: Int,
            child: View,
            parent: View
        ) -> Unit
    ) {
        // ...
    }

    // ...
}

// Usage
fun usage() {
    val a = ListAdapter()
    a.setListItemListener { position, id, child, parent ->
        // ...
    }
}

```

Using Kotlin function types from Java is more problematic. Not only are named parameters lost, but also it is expected that the `Unit` instance is returned.

```

ListAdapter adapter = new ListAdapter();
adapter.setListItemListener(Function4<? super Integer, ? super Integer, ? super View, ? super View, Unit> liste... void
    var
    T name = expr

ListAdapter adapter = new ListAdapter();
adapter.setListItemListener();
    (integer, integer2, view, view2) -> {} Function4<Integer, Integer, View, View, Unit>
    adanter
    ListAdapter

ListAdapter adapter = new ListAdapter();
adapter.setListItemListener((integer, integer2, view, view2) -> {}); Missing return statement

```

```
ListAdapter adapter = new ListAdapter();
adapter.setListItemSelectedListener((integer, integer2, view, view2) -> {
    return Unit.INSTANCE;
});
```

The solution to this problem is functional interfaces, i.e., interfaces with a single abstract method and the `fun` modifier. After using functional interfaces instead of function types, the usage of `setListItemSelectedListener` in Kotlin remains the same, but in Java it is more convenient as named parameters are understood and there is no need to return `Unit`.

```
fun interface ListItemClickListener {
    fun handle(
        position: Int,
        id: Int,
        child: View,
        parent: View
    )
}

class ListAdapter {

    fun setListItemSelectedListener(listener: ListItemClickListener) {
        // ...
    }

    // ...
}

fun usage() {
    val a = ListAdapter()
    a.setListItemSelectedListener { position, id, child, parent ->
        // ...
    }
}
```

```
ListAdapter adapter = new ListAdapter();
adapter.setListitemListener(ListItemListener listener) void
    System.out.println("new")
}

ListAdapter adapter = new ListAdapter();
adapter.setListitemListener((position, id, child, parent) -> {}); ListItemListener
    adapter ListAdapter
```



```
ListAdapter adapter = new ListAdapter();
adapter.setListitemListener((position, id, child, parent) -> {});
```

Tricky names

Keywords like `when` or `object` are reserved in Kotlin, so they cannot be used as names for functions or variables. The problem is that some of these keywords are not reserved in Java, so they might be used by Java libraries. A good example is Mockito, which is a popular mocking library, but one of its most important functions is named “`when`”. To use this function in Kotlin, we need to surround its name with backticks (`).

```
// Example Mockito usage
val mock = mock(UserService::class.java)
`when`(mock.getUser("1")).thenAnswer { aUser }
```

Backticks can also be used to define functions or variable names that would otherwise be illegal in Kotlin. They are most often used to define unit test names with spaces in order to improve their readability in execution reports. Such function names are legal only in Kotlin/JVM, and only if they are not going to be used for code that runs on Android (unit tests are executed locally, so they can be named this way).

```
class MarkdownToHtmlTest {  
  
    @Test  
    fun `Simple text should remain unchanged`() {  
        val text = "Lorem ipsum"  
        val result = markdownToHtml(text)  
        assertEquals(text, result)  
    }  
}
```

Throws

In Java, there are two types of exceptions:

- **Checked exceptions**, which need to be explicitly stated and handled in code. Checked exceptions in Java must be specified after the `throws` keyword in the declarations of functions that can throw them. When we call such functions from Java, we either need the current function to state that it might throw a specific exception type as well, or this function might catch expected exceptions. In Java, except for `RuntimeException` and `Error`, classes that directly inherit `Throwable` are checked exceptions.
- **Unchecked exceptions**, which can be thrown “at any time” and don’t need to be stated in any way, therefore methods don’t have to catch or throw unchecked exceptions explicitly. Classes that inherit `Error` or `RuntimeException` are unchecked exceptions.

```
public class JavaClass {  
    // IOException are checked exceptions,  
    // and they must be declared with throws  
    String readFirstLine(String fileName) throws IOException {  
        FileInputStream fis = new FileInputStream(fileName);  
        InputStreamReader reader = new InputStreamReader(fis);  
        BufferedReader bufferedReader =  
            new BufferedReader(reader);  
        return bufferedReader.readLine();  
    }  
  
    void checkFirstLine() {  
        String line;  
        try {  
            line = readFirstLine("number.txt");  
            // We must catch checked exceptions,  
            // or declare them with throws  
            } catch (IOException e) {  
                throw new RuntimeException(e);  
            }  
            // parseInt throws NumberFormatException,  
            // which is an unchecked exception  
            int number = Integer.parseInt(line);  
            // Dividing two numbers might throw  
            // ArithmeticException if number is 0,  
            // which is an unchecked exception  
            System.out.println(10 / number);  
    }  
}
```

In Kotlin, all exceptions are considered unchecked. This leads to a problem when we use Java to call Kotlin methods that throw exceptions that are considered checked exceptions in Java. In Java, such methods must have exceptions specified after the `throws` keyword. Kotlin does not generate these, therefore Java is confused. If you try to catch such an exception, Java will prohibit it, explaining that such an exception is not expected.

```
// Kotlin
@file:JvmName("FileUtils")

package test

import java.io.*

fun readFirstLine(fileName: String): String =
    File(fileName).useLines { it.first() }

void checkFirstLine() {
    String line;
    try {
        line = FileUtils.readFirstLine( fileName: "number.txt");
    } catch (IOException e) {
        throw new RuntimeException(e)
    }
    int number = Integer.parseInt(line);
    System.out.println(number)
}
```



To solve this issue, in all Kotlin functions that are intended to be used from Java, we should use the `Throws` annotation to specify all exceptions that are considered checked in Java.

```
// Kotlin
@file:JvmName("FileUtils")

package test

import java.io.*

@Throws(IOException::class)
fun readFirstLine(fileName: String): String =
    File(fileName).useLines { it.first() }
```

When this annotation is used, the Kotlin Compiler will specify these exceptions in the `throws` block in the generated JVM functions, therefore they are expected in Java.

```
void checkFirstLine() {
    String line;
    try {
        line = FileUtils.readFirstLine( fileName: "number.txt");
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    int number = Integer.parseInt(line);
    System.out.println(10 / number);
}
```

Using the `throws` annotation is not only useful for Kotlin and Java interoperability; it is also often used as a form of documentation that specifies which exceptions should be expected.

JvmRecord

Java 16 introduced records as immutable data carriers. In simple words, these are alternatives to Kotlin data classes. Java records can be used in Kotlin just like any other kind of class. To declare a record in Kotlin, we define a data class and use the `JvmRecord` annotation.

```
@JvmRecord
data class Person(val name: String, val age: Int)
```

Records have more restrictive requirements than data classes. Here are the requirements for the `JvmRecord` annotation to be used for a class:

- The class must be in a module that targets JVM 16 bytecode (or 15 if the `-Xjvm-enable-preview` compiler option is enabled).
- The class cannot explicitly inherit any other class (including `Any`) because all JVM records implicitly inherit `java.lang.Record`. However, the class can implement interfaces.

- The class cannot declare any properties that have backing fields, except these initialized from the corresponding primary constructor parameters.
- The class cannot declare any mutable properties that have backing fields.
- The class cannot be local.
- The class's primary constructor must be as visible as the class itself.

Summary

Kotlin and Java are two different languages, designed in two different centuries²¹, which sometimes makes it challenging when we interoperate between them. Most problems relate to important Kotlin features, like eliminating the concept of checked exceptions, or distinguishing between nullable and non-nullable types, between interfaces for read-only and mutable collections, or between shared types for primitives and wrapped primitives. Kotlin does all it can to make interoperability with Java as convenient as possible, but there are some inevitable trade-offs. For example, both the `List` and `MutableList` Kotlin types relate to the Java `List` interface. Also, Kotlin relies on Java nullability annotations and uses platform types when they are missing. We also need to know and use some annotations that determine how our code will behave when used from Java or another JVM language. These are challenges for developers, but they're definitely worth all the amazing features that Kotlin offers.

Exercise: Adjust Kotlin for Java usage

Consider the following Kotlin elements:

²¹Java's first stable release was in 1996, while Kotlin's first stable release was around 20 years later in 2016.

```
package advanced.java

data class Money(
    val amount: BigDecimal = BigDecimal.ZERO,
    val currency: Currency = Currency.EUR,
) {
    companion object {
        fun eur(amount: String) =
            Money(BigDecimal(amount), Currency.EUR)

        fun usd(amount: String) =
            Money(BigDecimal(amount), Currency.USD)

        val ZERO_EUR = eur("0.00")
    }
}

fun List<Money>.sum(): Money? {
    if (isEmpty()) return null
    val currency = this.map { it.currency }.toSet().single()
    return Money(
        amount = sumOf { it.amount },
        currency = currency
    )
}

operator fun Money.plus(other: Money): Money {
    require(currency == other.currency)
    return Money(amount + other.amount, currency)
}

enum class Currency {
    EUR, USD
}
```

This is how they can be used in Kotlin:

```
fun main() {
    val money1 = Money.eur("10.00")
    val money2 = Money.eur("29.99")

    println(listOf(money1, money2, money1).sum())
    // Money(amount=49.99, currency=EUR)

    println(money1 + money2)
    // Money(amount=39.99, currency=EUR)

    val money3 = Money.usd("10.00")
    val money4 = Money()
    val money5 = Money(BigDecimal.ONE)
    val money6 = Money.ZERO_EUR
}
```

However, Java usage is not that convenient. Your task it to add appropriate annotations to make it more Java-friendly, so that it can be used like this:

```
package advanced.java;

import java.math.BigDecimal;
import java.util.List;

public class JavaClass {

    public static void main(String[] args) {
        Money money1 = Money.eur("10.00");
        Money money2 = Money.eur("29.99");

        List<Money> moneyList =
            List.of(money1, money2, money1);

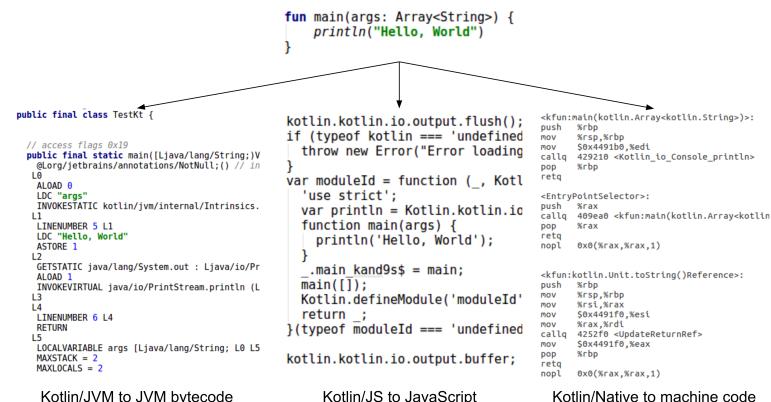
        System.out.println(MoneyUtils.plus(money1, money2));
        // Money(amount=39.99, currency=EUR)

        Money money3 = Money.usd("10.00");
        Money money4 = new Money();
```

```
    Money money5 = new Money(BigDecimal.ONE);
    Money money6 = Money.ZERO_EUR;
}
}
```

Using Kotlin Multiplatform

Kotlin is a compiled programming language, which means you can write some code in Kotlin and then use the Kotlin compiler to produce code in another language. Kotlin can currently be compiled into JVM bytecode (Kotlin/JVM), JavaScript (Kotlin/JS), or machine code (Kotlin/Native). This is why we say Kotlin is a multiplatform language: the same Kotlin code can be compiled to multiple platforms.



This is a powerful feature. Not only can Kotlin be used to write applications for multiple platforms, but we can also reuse the same code between different platforms. For instance, you can write code that will be used on a website, as well as on Android and iOS-native clients. Let's see how we can make our own multiplatform module.

Multiplatform module configuration

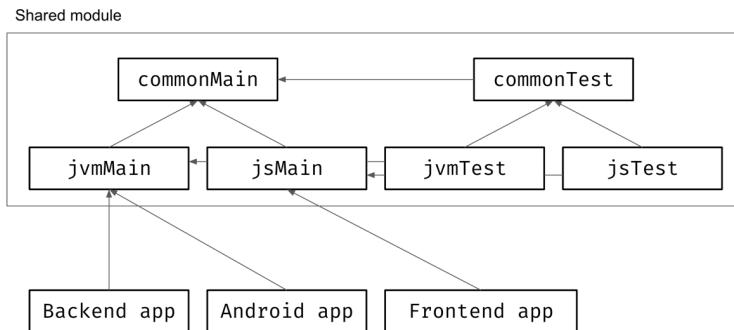
Gradle projects are divided into modules. Many projects have only one module, but they can have more than one. Each module is a different folder with its own `build.gradle(.kts)` file. Modules using Kotlin need to use the appropriate Kotlin plugin. If we want a Kotlin/JVM module, we use the

`kotlin("jvm")` plugin. To make a multiplatform plugin, we use `kotlin("multiplatform")`. In multiplatform modules, we define different source sets:

- Common source set, which contains Kotlin code that is not specific to any platform, as well as declarations that don't implement platform-dependent APIs. Common source sets can use multiplatform libraries as dependencies. By default, the common source set is called “commonMain”, and its tests are located in “commonTest”.
- Target source sets, which are associated with concrete Kotlin compilation targets. They contain implementations of platform-dependent declarations in the common source set for a specific platform, as well as other platform-dependent code. They can use platform-specific libraries, including standard libraries. Platform source sets can represent the projects we implement, like backend or Android applications, or they can be compiled into libraries. Some example target source sets are “jvmMain”, “androidMain” and “jsTest”.

A multiplatform module that is used by multiple other modules is often referred to as a “shared module”²².

²²The term “shared module” is also used for single-platform modules that are used by multiple other modules; however, in this chapter I will use the term “shared module” to specifically reference “shared multiplatform modules”.



In `build.gradle(.kts)` of multiplatform modules, we define source sets inside the `kotlin` block. We first configure each compilation target, and then for each source set we define dependencies inside the `sourceSets` block, as presented in the example below. This example shows a possible complete `build.gradle.kts` configuration, targeting JVM and JS.

```

plugins {
    kotlin("multiplatform") version "1.8.21"
}

group = "com.marcinmoskala"
version = "0.0.1"

kotlin {
    jvm {
        withJava()
    }
    js(IR) {
        browser()
        binaries.library()
    }
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlinx:

```

```
kotlinx-coroutines-core:1.6.4")
    }
}
val commonTest by getting {
    dependencies {
        implementation(kotlin("test"))
    }
}
val jvmMain by getting
val jvmTest by getting
val jsMain by getting
val jsTest by getting
}
jvmToolchain(11)
}
```

We define source set files inside folders in `src` that have the same name as the source set name. So, common files should be inside the `src/commonMain` folder, and JVM test files should be inside `src/jvmTest`.

This is how we configure common modules. Transforming a project that uses no external libraries from Kotlin/JVM to multiplatform is quite simple. The problem is that in the common module, you cannot use platform-specific libraries, so the Java stdlib and Java libraries cannot be used. You can use only libraries that are multiplatform and support all your targets, but it's still an impressive list, including Kotlin Coroutines, Kotlin Serialization, Ktor Client and many more. To deal with other cases, it's useful to define expected and actual elements.

Expect and actual elements

The common source set (`commonMain`) needs to operate on elements that, for each platform, have platform-specific implementations. Consider the fact that in `commonMain` you need to generate a random UUID string. For that, different classes are used on different platforms. On JVM,

we could use `java.util.UUID`, while on iOS we could use `platform.Foundation.NSUUID`. To use these classes to generate a UUID in the common module, we can specify the **expected** `randomUUID` function in `commonMain` and specify its **actual** implementations in each platform's source set.

```
// commonMain
expect fun randomUUID(): String

// jvmMain
import java.util.*

actual fun randomUUID() = UUID.randomUUID().toString()

// One of iOS source sets
import platform.Foundation.NSUUID

actual fun randomUUID(): String = NSUUID().UUIDString()
```

The compiler ensures that every declaration marked with the **expected** keyword in `commonMain` has the corresponding declarations marked with the **actual** keyword in all platform source sets.

All Kotlin essential elements can be expected in `commonMain`, so we can define expected functions, classes, object declarations, interfaces, enumerations, properties, and annotations.

```
// commonMain
expect object Platform {
    val name: String
}

// jvmMain
actual object Platform {
    actual val name: String = "JVM"
}

// jsMain
```

```
actual object Platform {  
    actual val name: String = "JS"  
}
```

Actual definitions can be type aliases that reference types that fulfill expected declaration expectations.

```
// commonMain  
expect class DateTime {  
    fun getHour(): Int  
    fun getMinute(): Int  
    fun getSecond(): Int  
    // ...  
}  
  
// jvmMain  
actual typealias DateTime = LocalDateTime  
  
// jsMain  
import kotlin.js.Date  
  
actual class DateTime(  
    val date: Date = Date()  
) {  
    actual fun getHour(): Int = date.getHours()  
    actual fun getMinute(): Int = date.getMinutes()  
    actual fun getSecond(): Int = date.getSeconds()  
}
```

More examples of expected and actual elements are presented later in this chapter.

In many cases, we don't need to define expected and actual elements as we can just define interfaces in `commonMain` and inject platform-specific classes that implement them. An example will be shown later in this chapter.

Expected classes are essential for multiplatform development because they specify elements with platform-specific imple-

mentations that are used as foundations for elements' implementations. Kotlin's Standard Library is based on expected elements, without which it would be hard to implement any serious multiplatform library.

Now let's review the possibilities that Kotlin's multiplatform capabilities offer us.

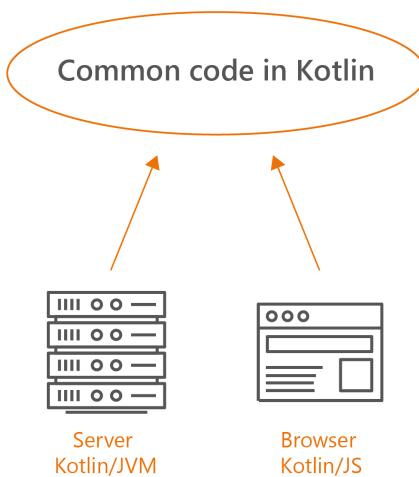
Possibilities

Companies rarely write applications for only a single platform²³. They would rather develop a product for two or more platforms because products often rely on several applications running on different platforms. Think of client and server applications communicating through network calls. As they need to communicate, there are often similarities that can be reused. Implementations of the same product for different platforms generally have even more similarities, especially their business logic, which is often nearly identical. Projects like this can benefit significantly from sharing code.

Lots of companies are based on web development. Their products are websites, but in most cases these products need a backend application (also called server-side). On websites, JavaScript is king and practically has a monopoly. On the backend, a very popular option is Java. Since these languages are very different, it is common for backend and web development to be separated, but this might change now that Kotlin is becoming a popular alternative to Java for backend development. For instance, Kotlin is a first-class citizen with Spring, the most popular Java framework. Kotlin can be used as an alternative to Java in every framework, and there are also many Kotlin backend frameworks, such as Ktor. This is why many backend projects are migrating from Java to Kotlin. A great thing about Kotlin is that it can also be compiled into JavaScript. There are already many Kotlin/JS libraries, and

²³In Kotlin, we view the JVM, Android, JavaScript, iOS, Linux, Windows, Mac, and even embedded systems like STM32 as separate platforms.

we can use Kotlin to write different kinds of web applications. For instance, we can write a web frontend using the React framework and Kotlin/JS. This allows us to write both the backend and the website in Kotlin. Even better, **we can have parts that compile to both JVM bytecode and JavaScript**. These are shared parts where we can put, for instance, universal tools, API endpoint definitions, common abstractions, etc.

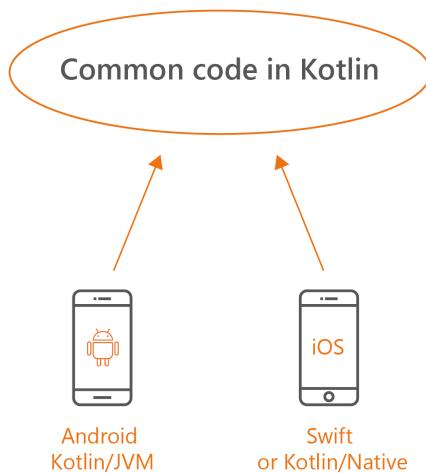


This capability is even more important in the mobile world as we rarely build only for Android. Sometimes we can live without a server, but we generally also need to implement an iOS application. Each application is written for a different platform using different languages and tools. In the end, the Android and iOS versions of the same application are very similar. They might be designed differently, but they nearly always have the same logic inside. Using Kotlin's multiplatform capabilities, we can implement this logic only once and reuse it between these two platforms. We can make a shared module in which we implement business logic, which should be independent of frameworks and platforms anyway (Clean Architecture). Such common logic can be written in pure

Kotlin or using other multiplatform modules, and it can then be used on different platforms.

Shared modules can be used directly in Android. The experience of working with multiplatform and JVM modules is great because both are built using Gradle. The experience is similar to having these common parts in our Android project.

For iOS, we compile these common parts to an Objective-C framework using Kotlin/Native, which is compiled into native code²⁴ using LLVM²⁵. We can then use the resulting code from Swift in Xcode or AppCode. Alternatively, we can implement our whole application using Kotlin/Native.



We can use all these platforms together. Using Kotlin, we can develop for nearly all kinds of popular devices and platforms,

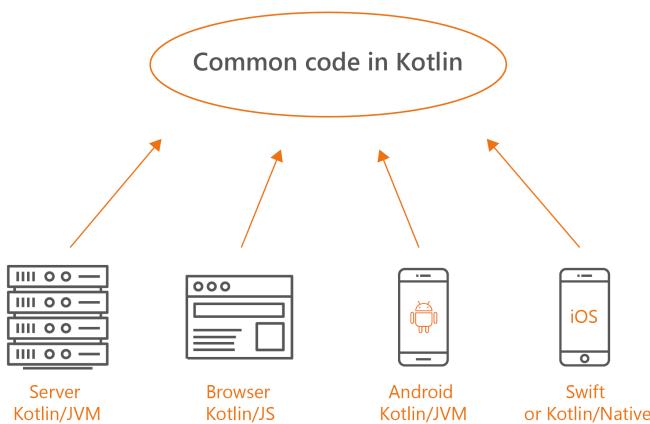
²⁴Native code is code that is written to run on a specific processor. Languages like C, C++, Swift, and Kotlin/Native are native because they are compiled into machine code for each processor they need to run on.

²⁵Like Swift or Rust.

and code can be reused between them however we want. Here are just a few examples of what we can write in Kotlin:

- Backend in Kotlin/JVM, for instance, in Spring or Ktor
- Website in Kotlin/JS, for instance, in React
- Android in Kotlin/JVM, using the Android SDK
- iOS Frameworks that can be used from Objective-C or Swift using Kotlin/Native
- Desktop applications in Kotlin/JVM, for instance, in TornadoFX
- Raspberry Pi, Linux, or macOS programs in Kotlin/Native

Here is a visualization of a typical application:



Defining shared modules is also a powerful tool for libraries. In particular, libraries that are not highly platform-dependent can easily be moved to a shared module, thus developers can use them from all languages running on the JVM, or from JavaScript, or natively (so from Java, Scala, JavaScript, CoffeeScript, TypeScript, C, Objective-C, Swift, Python, C#, etc.).

Writing multiplatform libraries is harder than writing libraries for just a single platform as they often require platform-specific parts for all platforms. However, there are already plenty of multiplatform libraries for network communication (like Ktor client), serialization (like kotlinx.serialization), date and time (like kotlinx-datetime), database communication (like SQLDelight), dependency injection (like Kodein-DI) and much more. Even more importantly, there are already libraries that let us implement UI elements in shared modules, like Jetpack Compose. With all these, you can implement completely functional applications for multiple platforms using only shared modules.

This is the theory, but let's get into the practice. Let's see some practical examples of multiplatform projects.

Multiplatform libraries

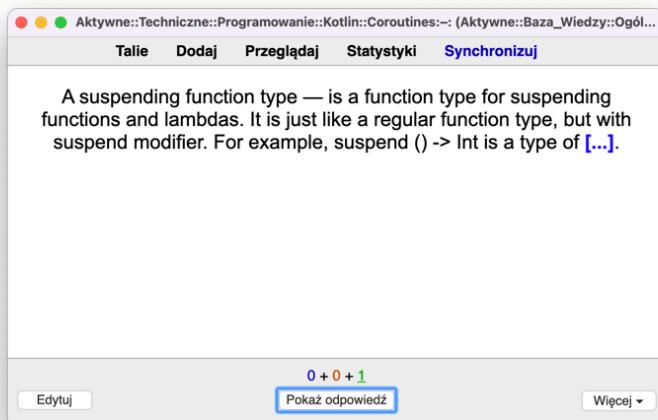
As a young and ambitious student, I started using the Anki flashcards application for learning. I still use it from time to time, but one of my biggest problems with it is the redundancy between the flashcards I create and my notes. To fix this, I started working on my AnkiMarkdown project. I invented a syntax to mark specific kinds of flashcards in the Markdown I use for making notes, then I implemented a program that understands this syntax. It has two modes of synchronization: update flashcards based on notes, or update notes based on flashcards.

The screenshot shows a window titled "Coroutines" from the "Notatki" application. The main content area displays a note with the title "Coroutines". The note contains several paragraphs of text, some of which are preceded by a code-like identifier (e.g., @1516805777754). The text discusses the concept of coroutines in Kotlin, mentioning suspending function types, continuations, and their execution across threads.

```
deckName: "Programmin::Kotlin::Coroutines"
---  
@1516805777754  
{{c1::A suspending function type}} — is a {{c2::function type for suspending functions and lambdas}}. It is just like a regular function type, but with {{c3::suspend modifier}}. For example, suspend () -> Int is a type of {{c4::suspending function without arguments that returns Int}}.  
  
@1516805350738  
{{c2::A coroutine}} — is {{c1::an instance of suspendable computation}}. It is conceptually similar to a thread, in the sense that it {{c3::takes a block of code to run and has a similar life-cycle — it is created and started}}, but it is not {{c4::bound to any particular thread}}. It may {{c5::suspend its execution}} in one thread and {{c5::resume}} in another one. Moreover, like a future or promise, it may complete with some {{c7::result or exception}}.  
  
@1516806072494  
{{c1::A continuation}} — is {{c2::a state of the suspended coroutine at suspension point}}. It conceptually represents {{c3::the rest of its execution after the suspension point}}.
```

At the bottom right of the note area, there is a status bar with the text "0 backlinks 3497 words 23402 characters".

Example of flashcards defined using Anki Markdown.



Flashcard of type cloze in Anki.

I initially implemented this program in Kotlin/JVM and ran it using the console, but then I started using the Obsidian program to manage my notes. I realized that I could make an Obsidian plugin for AnkiMarkdown synchronization that would need to be implemented in JavaScript, but it turned out to be really simple to turn my Kotlin/JVM module into a multiplatform module because Kotlin is a multiplatform language. I had already used the Ktor client for communication with Anki, and the only significant change was that I needed to move file management to multiplatform module platform source sets.

Complete example can be found on GitHub under the name [MarcinMoskala/AnkiMarkdown](#).

Let's see it in practice. Imagine you've decided to make a library for parsing and serializing YAML. You've implemented all the parsing and serialization using Kotlin and provided the following class with two exposed methods as your library API:

```
class YamlParser {  
    fun parse(text: String): YamlObject {  
        /*...*/  
    }  
    fun serialize(obj: YamlObject): String {  
        /*...*/  
    }  
  
    // ...  
}  
  
sealed interface YamlElement  
  
data class YamlObject(  
    val properties: Map<String, YamlElement>  
) : YamlElement  
  
data class YamlString(val value: String) : YamlElement  
  
// ...
```

Since you only use Kotlin and the Kotlin Standard Library, you can place this code in the common source set. For that, we need to set up a multiplatform module in our project, which entails defining the file where we will define our common and platform modules. It needs to have its own `build.gradle(.kts)` file with the Kotlin Multiplatform Gradle plugin (`kotlin("multiplatform")` using `kotlin dsl` syntax), then it needs to define the source sets configuration. This is where we specify which platforms we want to compile this module to, and we specify dependencies for each platform.

```
// build.gradle.kts
plugins {
    kotlin("multiplatform") version "1.8.10"
    // ...
    java
}

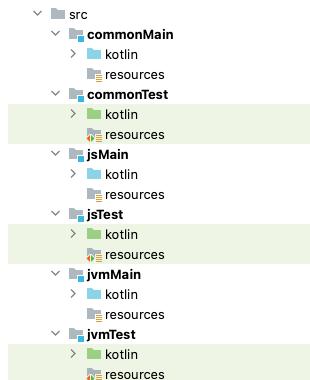
kotlin {
    jvm {
        compilations.all {
            kotlinOptions.jvmTarget = "1.8"
        }
        withJava()
        testRuns["test"].executionTask.configure {
            useJUnitPlatform()
        }
    }
    js(IR) {
        browser()
        binaries.library()
    }
    sourceSets {
        val commonMain by getting {
            dependencies {
                // ...
            }
        }
        val commonTest by getting {
            dependencies {
                // ...
            }
        }
        val jvmMain by getting {
            dependencies {
                // ...
            }
        }
        val jvmTest by getting
        val jsMain by getting
```

```

    val jsTest by getting
}
}

```

Source sets' names matter as they describe the corresponding platform. Note that tests for each platform are separate source sets with separate dependencies, therefore each source set needs an appropriately named folder that includes a “kotlin” subfolder for code and a “resources” folder for other resources.



We should place our common source set files inside the “commonMain” folder. If we do not have any expected declarations, we should now be able to generate a library in JVM bytecode or JavaScript from our shared module.

This is how we could use it from Java:

```

// Java
YamlParser yaml = new YamlParser();
System.out.println(yaml.parse("someProp: ABC"));
// YamlObject(properties={someProp=YamlString(value=ABC)})

```

If you build this code using Kotlin/JS for the browser, this is how you can use this class:

```
// JavaScript
const parser = new YamlParser();
console.log(parser.parse("someProp: ABC"))
// {properties: {someprop: "ABC"}}
```

It is more challenging when you build a NodeJS package because, in such packages, only exposed elements can be imported. To make your class functional for this target, you need to use the `@JsExport` annotation for all elements that need to be visible from JavaScript.

```
@JsExport
class YamlParser {
    fun parse(text: String): YamlObject {
        /*...*/
    }
    fun serialize(obj: YamlObject): String {
        /*...*/
    }
}

// ...
}

@JsExport
sealed interface YamlElement

@JsExport
data class YamlObject(
    val properties: Map<String, YamlElement>
) : YamlElement

@JsExport
data class YamlString(val value: String) : YamlElement
// ...
```

This code can be used not only from JavaScript but also from TypeScript, which should see proper types for classes and interfaces.

```
// TypeScript
const parser: YamlParser = new YamlParser();
const obj: YamlObject = parser.parse(text);
```

Now, let's complicate this example a bit and assume that you've decided to introduce a class in your library for reading YAML from a file or a URL. Reading files and making network requests are both platform specific, but you've already found multiplatform libraries for that. Let's say that you've decided to use Okio to read a file and the Ktor client to fetch a file from a URL.

```
// Using Okio to read file
class FileYamlReader {
    private val parser = YamlParser()

    fun read(filePath: String): YamlObject {
        val source = FileSystem.SYSTEM
            .source(filePath)
            .let(Okio::buffer)
        val fileContent = source.readUtf8()
        source.close()
        return parser.parse(fileContent)
    }
}

// Using Ktor client to read URL
class NetworkYamlReader {
    private val parser = YamlParser()

    suspend fun read(url: String): YamlObject {
        val resp = client.get(url) {
            headers {
                append(HttpHeaders.Accept, "text/yaml")
            }
        }.bodyAsText()
        return parser.parse(resp)
    }
}
```

The second problem is that to use network requests we needed to introduce a suspend function, but suspend functions cannot be used by languages other than Kotlin. To actually support using our class from other languages, like Java or JavaScript, we need to add classes that will adapt `NetworkYamlReader` for different platforms, like a blocking variant for JVM, or a variant that exposes promises on JS.

```
// jsMain module
@JsExport
@JsName("NetworkYamlReader")
class NetworkYamlReaderJs {
    private val reader = NetworkYamlReader()
    private val scope = CoroutineScope(SupervisorJob())

    fun read(url: String): Promise<YamlObject> =
        scope.promise { reader.read(url) }
}
```

I hope you have a sense of the possibilities that multiplatform modules offer, as well as the challenges that arise from these possibilities. You will see more of them in the next example.

A multiplatform mobile application

Kotlin multiplatform mobile (KMM) capabilities are often used to implement shared parts between Android and iOS. The idea is simple: we define a shared module for Android, and one for iOS; then, based on these modules, we generate Android and iOS libraries. In these libraries, thanks to numerous libraries, we can use network clients, databases, serialization, and much more, without writing any platform-specific code ourselves.

Let's see a concrete example of this in action. Let's say you implement an application for morning fitness routines on Android and iOS. You decide to utilize multiplatform Kotlin capabilities, so you define a shared module. Nowadays, it is a common practice to extract application business logic

into classes known as View Models, which include observable properties. These properties are observed by views that change when these properties change. You decide to define your `WorkoutViewModel` in the shared module. As observable properties, we can use `MutableStateFlow` from Kotlin Coroutines²⁶.

```
class WorkoutViewModel(
    private val timer: TimerService,
    private val speaker: SpeakerService,
    private val loadTrainingUseCase: LoadTrainingUseCase
    // ...
) : ViewModel() {
    private var state: WorkoutState = ...

    val title = MutableStateFlow("")
    val imgUrl = MutableStateFlow("")
    val progress = MutableStateFlow(0)
    val timerText = MutableStateFlow("")

    init {
        loadTraining()
    }

    fun onNext() {
        // ...
    }

    // ...
}
```

Let's discuss some important aspects of the development of such an application as this will teach us some important lessons about KMP development in general.

²⁶It is a popular practice to hide this property behind `StateFlow` to limit its methods' visibility, but I decided not to do this to simplify this example.

ViewModel class

Let's start with the `ViewModel` class. Android requires that classes representing view models implement `ViewModel` from `androidx.lifecycle`. iOS does not have such a requirement. To satisfy both platforms, we need to specify the expected class `ViewModel`, whose actual class on Android should extend `androidx.lifecycle.ViewModel`. On iOS, these actual classes can be empty.

```
// commonMain
expect abstract class ViewModel() {
    open fun onCleared()
}

// androidMain
abstract class ViewModel : androidx.lifecycle.ViewModel() {
    val scope = viewModelScope

    override fun onCleared() {
        super.onCleared()
    }
}

// iOS source sets
actual abstract class ViewModel actual constructor() {
    actual open fun onCleared() {
    }
}
```

We might add some other capabilities to our `ViewModel` class. For instance, we could use it to define coroutine scope. On Android, we use `viewModelScope`. On iOS, we need to construct the scope ourselves.

```
// commonMain
expect abstract class ViewModel() {
    val scope: CoroutineScope
    open fun onCleared()
}

// androidMain
abstract class ViewModel : androidx.lifecycle.ViewModel() {
    val scope = viewModelScope

    override fun onCleared() {
        super.onCleared()
    }
}

// iOS source sets
actual abstract class ViewModel actual constructor() {
    actual val scope: CoroutineScope = MainScope()

    actual open fun onCleared() {
        scope.cancel()
    }
}
```

Platform-specific classes

Now consider the parameters of the `WorkoutViewModel` constructor. Some of them can be implemented in our shared module using common libraries. `LoadTrainingUseCase` is a good example that only needs a network client. Some other dependencies need to be implemented on each platform. `SpeakerService` is a good example because I don't know of a library that would be able to use platform-specific TTS (Text-to-Speech) classes from a shared module.

We could define `SpeakerService` as an expected class, but it would be easier just to make it an interface in `commonMain` and inject different classes that implement this interface in platform source sets.

```
// commonMain
interface SpeakerService {
    fun speak(text: String)
}

// Android application
class AndroidSpeaker(context: Context) : SpeakerService {

    private var tts = TextToSpeech(context, null)

    override fun speak(text: String) {
        tts.speak(text, TextToSpeech.QUEUE_FLUSH, null, null)
    }
}

// Swift
class iOSSpeaker: Speaker {
    private let synthesizer = AVSpeechSynthesizer()

    func speak(text: String) {
        synthesizer.stopSpeaking(at: .word)
        let utterance = AVSpeechUtterance(string: text)
        synthesizer.speak(utterance)
    }
}
```

The biggest problem with classes like this is finding a common interface for both platforms. Even in this example, you can see some inconsistencies between Android `TextToSpeech` and iOS `AVSpeechSynthesizer`. On Android, we need to provide `Context`. On iOS, we need to make sure the previous speech request is stopped. What if one of these classes representing a speech synthesizer needs to be initialized before its first use? We would need to add an `initialize` method and implement it for both platforms, even though one of them will be empty. Common implementation aggregates specificities from all platforms, which can make common classes complicated.

Observing properties

Android has great support for observing `StateFlow`. On XML, we can bind values; on Jetpack Compose, we can simply collect these properties as a state:

```
val title: String by viewModel.title.collectAsState()
val imgUrl: String by viewModel.imgUrl.collectAsState()
val progress: Int by viewModel.progress.collectAsState()
val timerText: String by viewModel.timerText.collectAsState()
```

This is not so easy in Swift. There are already a few solutions that could help us, but none of them seem to be standard; hopefully, this will change over time. One solution is using a library like MOKO that helps you turn your view model into an observed object, but this approach needs some setup and modifications in your view model.

```
// iOS Swift
struct LoginScreen: View {
    @ObservedObject
    var viewModel: WorkoutViewModel = WorkoutViewModel()

    // ...
}
```

You can also turn `StateFlow` into an object that can be observed with callback functions. There are also libraries for that, or we can just define a simple wrapper class that will let you collect `StateFlow` in Swift.

```
// Swift
viewModel.title.collect(
    onNext: { value in
        // ...
    },
    onCompletion: { error in
        // ...
    }
)
```

I guess there might be more options in the future, but for now this seems to be the best approach to multiplatform Kotlin projects.

Summary

In Kotlin, we can implement code for multiple platforms, which gives us amazing possibilities for code reuse. To support common code implementation, Kotlin offers a multiplatform stdlib, and numerous libraries already support network calls, serialization, dependency injection, database usage, and much more. As library creators, we can implement libraries for multiple platforms at the same time with little additional effort. As mobile developers, we can implement the logic for Android and iOS only once and use it on both platforms. Code can also be reused between different platforms according to our needs. I hope you can see how powerful Kotlin multiplatform is.

Exercise: Multiplatform LocalDateTime

In your multiplatform project, you need to define a common type to represent time. You decided that you want it to behave just like `LocalDateTime` from `java.time` library. In fact, you decided that on Kotlin/JVM you want to use `LocalDateTime` directly as an actual type. Define a common type `LocalDateTime`, and define actual typealias for Kotlin/JVM, and an actual class for Kotlin/JS that wraps over `Date` from JavaScript.

Those are expected elements that you need to provide for each platform:

```
expect class LocalDateTime {
    fun getSecond(): Int
    fun getMinute(): Int
    fun getHour(): Int
    fun plusSeconds(seconds: Long): LocalDateTime
}

expect fun now(): LocalDateTime

expect fun parseLocalDateTime(str: String): LocalDateTime
```

Starting code and unit tests for this exercise can be found in the MarcinMoskala/kmp-exercise project on GitHub. You can clone this project and solve this exercise locally.

JavaScript interoperability

Let's say you have a Kotlin/JVM project and realize you need to use some parts of it on a website. This is not a problem: you can migrate these parts to a common module that can be used to build a package to be used from JavaScript or TypeScript²⁷. You can also distribute this package to npm and expose it to other developers.

The first time I encountered such a situation was with AnkiMarkdown²⁸, a library I built that lets me keep my flashcards in a special kind of Markdown and synchronize it with Anki (a popular program for flashcards). I initially implemented this synchronization as a JVM application I ran in the terminal, but this was inconvenient. Since I use Obsidian to manage my notes, I realized using AnkiMarkdown as an Obsidian plugin would be great. For that, I needed my synchronization code in JS. Not a problem! It took only a couple of hours to move it to the multiplatform module and distribute it to npm, and now I can use it in Obsidian.

The second time I had a similar situation was when I worked for Scanz. We had a desktop client for a complex application implemented in Kotlin/JVM. Since we wanted to create a new web-based client, we extracted common parts (services, view models, repositories) into a shared module and reused it. There were some trade-offs we needed to make, as I will show in this chapter, but we succeeded, and not only was it much faster than rewriting all these parts, but we could also use the common module for Android and iOS applications.

The last story I want to share is my Sudoku problem generator and solver²⁹. It implements all the basic sudoku techniques and uses them to generate and solve sudoku puzzles. I needed

²⁷The current implementation of Kotlin/JS compiles Kotlin code to ES5 or ES6.

²⁸Link to AnkiMarkdown repository:
github.com/MarcinMoskala/AnkiMarkdown

²⁹Link to the repository of this project:
github.com/MarcinMoskala/sudoku-generator-solver

it for a book that teaches how to solve sudoku. To present these sudoku problems and solve them myself conveniently, I implemented a React application. Then, I turned my Sugoku generator and solver into a common module and distributed it to npm.

In this chapter, I would like to share everything I learned along the way. I will show you the challenges when we transform a Kotlin/JVM project to use it conveniently from TypeScript and how to use the package produced this way in practice.

When I was writing this chapter, I was using Kotlin version 1.8.21. Technically speaking, Kotlin/JS should be stable since 1.8.0, but I would not be surprised if some details of what I am explaining in this chapter (like Gradle task names) change in the next versions.

Setting up a project

The first thing we need to do is set up a multiplatform project. We need a Gradle module using the Kotlin Multiplatform plugin, with `js` configured in the `kotlin` block. To specify the output format, we should call `browser`, `nodejs` or `useEsModules` in the `js` block, depending on where we want to run our code. If you want to run your code in browser applications, like in the examples from the introduction to this chapter, use the `browser` block. If you want to generate a NodeJS module, use `nodejs`. To generate ES6 modules, use `useEsModules`. Finally, you can call `binaries.executable()` in the `js` block, which explicitly instructs the Kotlin compiler to emit executable `.js` files. We should also specify `jsMain` (and `jsTest` if we want to write JS-specific tests) in the `sourceSets` block.

```
plugins {
    kotlin("multiplatform") version "1.8.21"
}

kotlin {
    jvm {}
    js(IR) {
        browser() // use if you need to run code in a browser
        nodejs() // use if you need to run code in a Node.js
        useEsModules() // output .mjs ES6 modules
        binaries.executable()
    }
    sourceSets {
        val commonMain by getting {
            dependencies {
                // common dependencies
            }
        }
        val commonTest by getting
        val jvmMain by getting
        val jvmTest by getting
        val jsMain by getting
        val jsTest by getting
    }
}
```

The `IR` in the `js` block means that the Intermediate Representation compiler will be used, which is a modern standard because of its source map generation and the JS optimizations it introduces; moreover, in the future it might help support Kotlin/JS debugging.

Currently, the `kotlin("js")` plugin is deprecated, and we should only use `kotlin("multiplatform")` to target JavaScript.

Now that we know how to set up a Kotlin/JS project let's see what it lets us do.

Using libraries available for Kotlin/JS

Not all dependencies are multiplatform and have support for Kotlin/JS. If a certain dependency is not multiplatform, you must find an alternative or implement platform-specific implementations yourself. To avoid this, it is best to use truly multiplatform libraries in the first place, especially those provided by the Kotlin team, like Kotlinx Serialization, Coroutines, and Ktor Client.

Using Kotlin/JS

Using Kotlin/JS is very similar to using Kotlin/JVM as we can use all Kotlin features, including standard library elements. However, a standard Java library is not available, so we have to use functions provided by JavaScript instead. For instance, we can use `console.log("Hello")` to print a message to the console.

```
fun printHello() {
    console.log("Hello")
}
```

This is possible because Kotlin/JS provides a set of declarations for JavaScript functions and objects. For instance, `console` is declared as a top-level property of type `Console`. This type has a `log` function that accepts any values as arguments.

```
@Suppress("NOT_DOCUMENTED")
public external interface Console {
    public fun dir(o: Any): Unit
    public fun error(vararg o: Any?): Unit
    public fun info(vararg o: Any?): Unit
    public fun log(vararg o: Any?): Unit
    public fun warn(vararg o: Any?): Unit
}

public external val console: Console
```

Both the `console` property and the `Console` interface are declared as external, which means they are not implemented in Kotlin, but JavaScript provides them. We can use them in our Kotlin code but not implement them. If there is a JavaScript element we want to use in Kotlin but don't have a declaration for, we can create it ourselves. For instance, if we want to use the `alert` function, we can declare it as follows:

```
fun showAlert() {
    alert("Hello")
}

@JsName("alert")
external fun alert(message: String)
```

Sometimes our external declarations need to be nested and complex, as I will show in the *Adding npm dependencies* section.

Using the `js` function, we can also call JavaScript elements from Kotlin/JS without explicit declarations. It executes JavaScript code defined as a string and returns its result. For instance, we can use it to call the `prompt` function:

```
fun main() {
    val message = js("prompt('Enter your name')")
    println(message)
}
```

The string used as an argument for the `js` function must be known at compile time and must be a raw string without any operations. However, inside it we can use local variables defined in Kotlin. For instance, we can use the `user` variable in the following code:

```
fun main() {
    val user = "John"
    val surname =
        js("prompt('What is your surname ${user}?')")
    println(surname)
}
```

The result of `js` is `dynamic`, which is a special type for Kotlin/JS that is not checked by the compiler and represents any JavaScript value. This means we can assign it to any variable, call any function on it, and cast it to any type. If a cast is not possible, it will throw an exception at runtime. For instance, we can cast `js("1")` to `Int`.

```
fun main() {
    val o: dynamic = js("{name: 'John', surname: 'Foo'}")
    println(o.name) // John
    println(o.surname) // Foo
    println(o.toLocaleString()) // [object Object]
    println(o.unknown) // undefined

    val i: Int = js("1")
    println(i) // 1
}
```

We can create JavaScript objects in Kotlin using the `json` function, which expects a vararg of pairs of `String` and `Any?` and returns an object of type `Json`. To create a nested object, use the `json` function with arguments representing key-value pairs.

```
import kotlin.js.json

fun main() {
    val o = json(
        "name" to "John",
        "age" to 42,
    )
    print(JSON.stringify(o)) // {"name":"John", "age":42}
}
```

These are the essentials of Kotlin/JS-specific structures. Now, let's focus on what we can do with them.

Building and linking a package

We have our multiplatform module configured with a Kotlin/JS target or just a Kotlin/JS module, and we want to use it in a JavaScript or TypeScript project. For that, we need to build a package. A production-ready package for the browser can be built using the `jsBrowserProductionLibraryDistribution` Gradle task, the result of which is a `.js` file and a `.d.ts` file within your common module's `build/productionLibrary` directory. The default name of these files is the name of the module, or it can be specified using the `moduleName` property in the `js` block inside the `kotlin` block.

For example, if your module is called `common`, you should use the `:common:jsBrowserProductionLibraryDistribution` task, and the result will be `common.js` and `common.d.ts` files inside the `common/build/productionLibrary` directory.

If your project defines no modules and therefore only has a top-level Gradle file, the module's name will be the project's name. In the `AnkiMarkdown` project, for instance, the result of the `jsBrowserProductionLibraryDistribution` task is `ankimarkdown.js` and `ankimarkdown.d.ts` files inside `/build/productionLibrary` directory.

The next step is to link this package to your JavaScript or TypeScript project. We could just copy-paste the generated files, but this wouldn't be very convenient because we would need to repeat this process every time the common module changed. A better way is to link the generated files.

I assume we'll use npm or yarn to manage our JavaScript or TypeScript project. In this case, we should define a dependency on a common module package in the `package.json` file. We can do this by adding a line to the `dependencies` section that defines the path to the directory containing our package, starting from "file:". Use relative paths. This is what it might look like in our example projects:

```
// A project where common module is called common
"dependencies": {
    // ...
    "common": "file:../common/build/productionLibrary"
}

// AnkiMarkdown project
"dependencies": {
    // ...
    "AnkiMarkdown": "file:../build/productionLibrary"
}
```

To avoid reinstalling our dependencies whenever we build a new package from our common module, we should now link our module using npm or yarn linking. To link a module defined in a file, go to that file location first (`productionLibrary` folder) and call `npm link` or `yarn link`. Then, go to your JavaScript or TypeScript project and call `npm link <module-name>` or `yarn link <module-name>`. When you subsequently start your project, you should see changes in the dependency immediately after building a new package using the `jsBrowserProductionLibraryDistribution` task.

Distributing a package to npm

We can also distribute our Kotlin module as a package to npm. Currently, it seems to be standard to use the `dev.petuska.npm.publish`³⁰ plugin for this. You need to configure your npm package inside the `npmPublish` block, set the name and version, and register the npm registry. This is what it currently looks like in the AnkiMarkdown project (before using this plugin, I recommend checking its documentation and alternatives):

```
npmPublish {  
    packages {  
        named("js") {  
            packageName.set("anki-markdown")  
            version.set(libVersion)  
        }  
    }  
    registries {  
        register("npmjs") {  
            uri.set(uri("https://registry.npmjs.org"))  
            authToken.set(npmSecret)  
        }  
    }  
}
```

`npmSecret` is a string containing the npm token needed to publish your package. You can get it from your npm account.

Exposing objects

All public elements from a Kotlin/JS module can be used in another Kotlin/JS module, but we need to expose them if we want to use them in JavaScript or TypeScript. For that, we need to use the `@JsExport` annotation, which can be used

³⁰Link to this plugin repository: github.com/mpetuska/npm-publish

on classes, objects, functions, properties, and top-level functions and properties. Elements that are not exported will not appear in the `.d.ts` file, and their methods and property names will be mangled in the `.js` file.

Consider the following file in Kotlin/JS:

```
@JsExport
class A(
    val b: Int
) {
    fun c() { /*...*/ }
}

@JsExport
fun d() { /*...*/ }

class E(
    val h: String
) {
    fun g() { /*...*/ }
}

fun i() { /*...*/ }
```

The `.d.ts` result of the `jsBrowserProductionLibraryDistribution` task will be:

```
type Nullable<T> = T | null | undefined
export class A {
    constructor(b: number);
    get b(): number;
    c(): void;
}
export function d(): void;
export as namespace AnkiMarkdown;
```

There is no sign of class `E` or function `i`. If elements are used in other parts of our code, they will be present in the `.js` file, but

property and function names will be mangled³¹, so we cannot use them in JavaScript or TypeScript.

This is a huge limitation because we cannot use any Kotlin class that is not exported. The simplest solution would be to mark our common module elements as `@JsExport`, but there is a problem with this. We cannot use the Kotlin standard library elements in JavaScript or TypeScript because many of its classes are not exported, including `kotlin.collections.List`. Instead, we should use arrays. Another problematic type is `Long` because all other types of numbers are transformed to JavaScript `number` type, but `Long` is non-exportable³².

As a consequence, we have two options:

1. Mark common module classes as `JsExport` and adjust them to the limitations. This, for instance, means we need to use arrays instead of lists, and we cannot use the `Long` type. Such changes might not be acceptable when JavaScript is not the primary target of our project.
2. Create wrappers for all classes we want to use in JavaScript or TypeScript. This option is considered better for projects that are not built primarily for Kotlin/JS. This is what such wrappers could look like:

³¹By name mangling, I mean that these names are replaced with some random characters in JS.

³²There are some discussions about making types like `List`, `Map`, `Set` and `Long` exportable to JavaScript. I truly support this with all my heart because interoperating between Kotlin and JavaScript/TypeScript seems to be the biggest pain at the moment.

```
@JsExport
@JsName("SudokuGenerator")
class SudokuGeneratorJs {
    private val sudokuGenerator = SudokuGenerator()

    fun generate(): SudokuJs {
        return SudokuJs(sudokuGenerator.generate())
    }
}

@JsExport
@JsName("Sudoku")
class SudokuJs internal constructor(
    private val sudoku: Sudoku
) {
    fun valueAt(position: PositionJs): Int {
        return sudoku.valueAt(position.toPosition())
    }
    fun possibilitiesAt(position: PositionJs): Array<Int> {
        return sudoku.possibilitiesAt(position.toPosition()
            .toTypedArray())
    }

    fun isSolved(): Boolean {
        return sudoku.isSolved()
    }
}

@JsExport
@JsName("Position")
class PositionJs(
    val row: Int,
    val column: Int
)

fun PositionJs.toPosition() = Position(
    row = row,
    column = column
```

```
)  
fun Position.toPositionJs() = PositionJs(  
    row = row,  
    column = column  
)
```

`JsName` annotation is used to change the name of an element in JavaScript. We often use `JsName` for wrapper classes to give them the same name as the original class, but without the JS suffix. We also sometimes use it to prevent mangling of method or property names.

This is how TypeScript declarations will look:

```
type Nullable<T> = T | null | undefined  
export class SudokuGenerator {  
    constructor();  
    generate(): Sudoku;  
}  
export class Sudoku {  
    private constructor();  
    valueAt(position: Position): number;  
    possibilitiesAt(position: Position): Array<number>;  
    isSolved(): boolean;  
}  
export class Position {  
    constructor(row: number, column: number);  
    get row(): number;  
    get column(): number;  
}  
export as namespace Sudoku;
```

It is likely that, one day, there will be KSP or compiler plugin libraries to generate such wrappers automatically, but right now we need to create them manually.

Exposing Flow and StateFlow

Another common problem with using Kotlin code from JavaScript is that types from the Kotlin Coroutines library, like `Flow` and `StateFlow`, which in MVVM architecture are used to represent state and data streams, are not exported. Consider that you have the following class in your common module:

```
class UserListViewModel(
    private val userRepository: UserRepository
) : ViewModel() {
    private val _userList: MutableStateFlow<List<User>> =
        MutableStateFlow(emptyList())
    val userList: StateFlow<List<User>> = _userList

    private val _error: MutableStateFlow<Throwable?> =
        MutableSharedFlow()
    val error: Flow<Throwable?> = _error

    fun loadUsers() {
        viewModelScope.launch {
            userRepository.fetchUsers()
                .onSuccess { _usersList.value = it }
                . onFailure { _error.emit(it) }
        }
    }
}
```

This object could not be used in JavaScript even if it were exported because it uses the `StateFlow` and `Flow` types, which are not exported, so we need to make a wrapper for them. The `Flow` type represents an observable source of values. A value that wraps it could provide a `startObserving` method to observe its events, and a `stopObserving` method to stop all observers.

```
@JsExport
interface FlowObserver<T> {
    fun stopObserving()
    fun startObserving(
        onEach: (T) -> Unit,
        onError: (Throwable) -> Unit = {},
        onComplete: () -> Unit = {},
    )
}

fun <T> FlowObserver(
    delegate: Flow<T>,
    coroutineScope: CoroutineScope
): FlowObserver<T> =
    FlowObserverImpl(delegate, coroutineScope)

class FlowObserverImpl<T>(
    private val delegate: Flow<T>,
    private val coroutineScope: CoroutineScope
) : FlowObserver<T> {
    private var observeJobs: List<Job> = emptyList()

    override fun startObserving(
        onEach: (T) -> Unit,
        onError: (Throwable) -> Unit,
        onComplete: () -> Unit,
    ) {
        observeJobs += delegate
            .onEach(onEach)
            .onCompletion { onComplete() }
            .catch { onError(it) }
            .launchIn(coroutineScope)
    }

    override fun stopObserving() {
        observeJobs.forEach { it.cancel() }
    }
}
```

The constructor must be internal because it requires the `CoroutineScope` type, which is not exported. This means the `FlowObserver` constructor can only be used in Kotlin/JS code.

The `StateFlow` type represents an observable source of values that always has a value. A value that wraps it should provide the `value` property to access the current state, a `startObserving` method to observe state changes, and a `stopObserving` method to stop all observers. Since `StateFlow` never completes, it doesn't call the `onComplete` or `onError` methods.

```
@JsExport
interface StateFlowObserver<T> : FlowObserver<T> {
    val value: T
}

fun <T> StateFlowObserver(
    delegate: StateFlow<T>,
    coroutineScope: CoroutineScope
): StateFlowObserver<T> =
    StateFlowObserverImpl(delegate, coroutineScope)

class StateFlowObserverImpl<T>(
    private val delegate: StateFlow<T>,
    private val coroutineScope: CoroutineScope
) : StateFlowObserver<T> {
    private var jobs = mutableListOf<Job>()
    override val value: T
        get() = delegate.value

    override fun startObserving(
        onEach: (T) -> Unit,
        onError: (Throwable) -> Unit = {},
        onComplete: () -> Unit = {}
    ) {
        jobs += delegate
            .onEach(onEach)
            .launchIn(coroutineScope)
    }
}
```

```
    override fun stopObserving() {
        jobs.forEach { it.cancel() }
        jobs.clear()
    }
}
```

Elements that are exposed in `UserListViewModel`, like `List<User>` or `Throwable?`, might not be understood by JavaScript, so we also need to create wrappers for them. For `Flow` this is a simple task as we can map its values using the `map` method. For `StateFlow`, we need to create a wrapper that maps objects.

```
fun <T, R> StateFlowObserver<T>.map(
    transformation: (T) -> R
): StateFlowObserver<R> =
    object : StateFlowObserver<R> {
        override val value: R
            get() = transformation(this@map.value)

        override fun startObserving(
            onEach: (T) -> Unit,
            onError: (Throwable) -> Unit = {},
            onComplete: () -> Unit = {},
        ) {
            this@map.observe { onEach(transformation(it)) }
        }

        override fun stopObserving() {
            this@map.stopObserving()
        }
    }
```

Now we can define the `UserListViewModel` class that can be used in JavaScript or TypeScript:

```
@JsExport("UserListViewModel")
class UserListViewModelJs internal constructor(
    userRepository: UserRepository
) : ViewModelJs() {
    val delegate = UserListViewModel(userRepository)

    val userList: StateFlow<List<User>> = StateFlowObserver(
        delegate.usersList,
        viewModelScope
    ).map { it.map { it.asJsUser() }.toTypedArray() }

    val error: Flow<Throwable?> = FlowObserver(
        delegate.error.map { it?.asJsError() },
        viewModelScope
    )

    fun loadUsers() {
        delegate.loadUsers()
    }
}
```

This is an example React hook that can simplify observing flow state:

```
export function useFlowState<T>(
    property: FlowObserver<T>,
): T | undefined {
    const [state, setState] = useState<T>()
    useEffect(() => {
        property.startObserving((value: T)=>setState(value))
        return () => property.stopObserving()
    }, [property])
    return state
}

// Usage
const SomeView = ({app}: { app: App }) => {
    const viewModel = useMemo(() => {
        app.createUserListViewModel()
```

```
    }, [])
  const userList = useStateFlowState(viewModel.userList)
  const error = useFlowState(viewModel.error)
  // ...
}
```

All these wrappers add a lot of boilerplate code, so I'm still hoping for a good KSP library or compiler plugin to generate them automatically. These wrappers are also not very efficient because they introduce additional objects, so it's debatable whether making a common JavaScript module available is worth it for a specific application. I would say that if common parts are heavy in logic, then it should be worth the effort; on the other hand, if common parts are not logic-heavy, it might be better to duplicate them in JavaScript.

Adding npm dependencies

Adding Kotlin/JS dependencies to a Kotlin/JS project is easy: you just define them in the dependencies of this target. However, adding npm dependencies is a bit more demanding: you should also add them to the dependency list in `build.gradle.kts`, but you need to wrap such dependencies using the `npm` function.

```
// build.gradle.kts
kotlin {
    // ...

    sourceSets {
        // ...
        val jsMain by getting {
            dependencies {
                implementation(npm("@js-joda/timezone", "2.18.0"))
                implementation(npm("@oneidentity/zstd-js", "1.0.3"))
                implementation(npm("base-x", "4.0.0"))
            }
        }
    }
}
```

```
// ...
}
}
```

Kotlin's dependencies have Kotlin types and can be used in Kotlin directly. JavaScript elements are not visible in Kotlin because they don't have Kotlin types, so you need to define them in Kotlin if you want to use them. For that, define an external object, functions, classes, and interfaces. If these classes need to be imported from a library, you need to use the `@JsModule` annotation in front of the object representing the whole dependency. Here is an example of such definitions for the `@oneidentity/zstd-js` and `base-x` libraries:

```
@JsModule("@oneidentity/zstd-js")
external object zstd {
    fun ZstdInit(): Promise<ZstdCodec>

    object ZstdCodec {
        val ZstdSimple: ZstdSimple
        val ZstdStream: ZstdStream
    }

    class ZstdSimple {
        fun decompress(input: Uint8Array): Uint8Array
    }

    class ZstdStream {
        fun decompress(input: Uint8Array): Uint8Array
    }
}

@JsModule("base-x")
external fun base(alphabet: String): BaseConverter

external interface BaseConverter {
    fun encode(data: Uint8Array): String
    fun decode(data: String): Uint8Array
}
```

There is a library called Dukat³³ that generates Kotlin declarations based on TypeScript definition files.

The Kotlin/JS Gradle plugin includes a dead code elimination tool that reduces the size of the resulting JavaScript code by removing unused properties, functions, and classes, including those from external libraries.

Frameworks and libraries for Kotlin/JS

Most of the problems I've described in this book result from interoperability between Kotlin and JavaScript, but most of them disappear if you limit or eliminate this interoperability. Instead of using an npm package, you can nearly always find a Kotlin/JS or multiplatform library that is easier to use and does not need any additional wrappers or special kinds of dependency declaration. I recommend starting your library search from the [Kotlin Wrappers](#) library, created by JetBrains. It contains an astonishing number of wrappers for different browser APIs and a variety of popular JavaScript libraries.

Instead of exporting elements so they can be used in JavaScript, you can write your client in pure Kotlin as well. For instance, the Kotlin Wrappers library offers methods for DOM manipulation or defining views using HTML DSL, or React Kotlin can define complete React applications using only Kotlin. There are also frameworks designed to be used in Kotlin/JS, like KVision, and JetPack Compose can be used to write websites.

JavaScript and Kotlin/JS limitations

When you consider targeting JavaScript, you should also consider its general limitations. JavaScript is an essentially different platform than JVM, therefore it has different types, dif-

³³Dukat can be found at github.com/Kotlin/dukat

ferent memory management, and different threading models. JavaScript runs on a single thread, and it is impossible to run blocking operations on JavaScript³⁴. This means that you cannot use `Dispatchers.IO` in Kotlin/JS code.

You should also consider browser limitations. In one project, we established a WebSocket connection with a specific header in the handshake in our JVM client. However, it turned out that it's not possible to do this in browsers due to their limitations. It's a similar case with cookie headers: on JVM programs, this header can be set to whatever you want, but in browsers, you can only send actual cookies that are set for a specific domain.

The web is an extremely powerful platform and you would likely be surprised by web browsers' capabilities, which include using databases, offline mode, background sync, shared workers and much more. However, the web also has its limits, so if you write a common module that should be used in a browser, you'd better know these limitations in advance.

On the other hand, it is also worth mentioning that Kotlin is much more limited than TypeScript when it comes to type systems: we cannot express type literals, union or intersection types, and much more. TypeScript is extremely expressive, and JavaScript, as a dynamic language, allows much more to be done with its objects than JVM, therefore Kotlin is more limited in the area of TypeScript API design.

```
// Example type that cannot be expressed in Kotlin
type ProcessResult = number | "success" | Error
```

³⁴It is possible to start processes in other threads in browsers that use Web Workers, and there is a KEEP (a document evaluated as part of the Kotlin Evolution and Enhancement Process) that proposes introducing support for a `worker` block that behaves similarly to the `thread` block on JVM.

Summary

As you can see, exposing Kotlin code to JavaScript is not that simple, but it's possible, and it makes a lot of sense in some cases. In my AnkiMarkdown and SudokuSolver libraries, it was practically effortless because these libraries are heavy in logic and use very few platform-specific features. In the case of reusing the common logic of a complex application, it is much harder. We need additional wrappers, and we need to adjust our code to the limitations of JavaScript. I think it's still worth it, but before you make a similar decision yourself, you should first consider all the pros and cons.

Exercise: Migrating a Kotlin/JVM project to KMP

Clone the following project:

<https://github.com/MarcinMoskala/sudoku-generator-exercise>

It is a Kotlin/JVM project implementing a logic of generating and solving sudoku. However, you need to use those capabilities in your React project written in TypeScript. Transform your project to a Kotlin Multiplatform project, and generate a JavaScript library named “sudoku-generator” from it. Then use it in your React project.

In the folder `web-client` you can find a React project. It is already set to use the generated library, assuming it is going to be named “sudoku-generator” and located in the default path “`build/productionLibrary`”. You can run it using `npm start` command. It is a simple project that displays an unsolved and solved sudoku. To make it work, you need to transform Kotlin parts to Kotlin multiplatform, and define Kotlin/JS wrapper over sudoku generator, that is exported to JavaScript, and that exposes objects that can be used in TypeScript.

The exported class should be named `SudokuGenerator`, should have no package, an empty constructor, and a single method `generateSudoku` that takes no arguments and returns an object `Sudoku` with a random sudoku and its solution. The

Sudoku object should have two properties: `sudoku` and `solved`. Both should be of type `Array<Array<Int?>>`. Sudoku should be generated using `SudokuGenerator` and `SudokuSolver`. Both those classes can be created using `SudokuGenerator()` and `SudokuSolver()` constructors, and the `sudoku` should be generated using `generate` method from `SudokuGenerator` class, with `solver` from `SudokuSolver` class used as an argument.

Hint: IntelliJ often has troubles with recognizing change from Kotlin/JVM to Kotlin Multiplatform. If you have problems with it, try to invalidate cache and restart IntelliJ. Also, give it a moment.

Hint: To set the name of the generated module, you need to set `moduleName` property in `js(IR)` block.

Hint: If you have problems with generating TypeScript definitions, try to call `generateTypeScriptDefinitions` in `js(IR)` block.

Hint: Generate the library using Gradle task `jsBrowserProductionLibraryDistribution`, then remember to use `npm install` in web-client.

Hint: To transform `SudokuState` to `Array<Array<Int?>>` you can use the following function:

```
fun SudokuState.toJs(): Array<Array<Int?>> = List(9) { row ->
    List(9) { col ->
        val cell = this.cells[SudokuState.Position(row, col)]
        when (cell) {
            is SudokuState.CellState.Filled -> cell.value
            is SudokuState.CellState.Empty, null -> null
        }
    }.toTypedArray()
}.toTypedArray()
```

Reflection

Reflection in programming is a program's ability to introspect its own source code symbols at runtime. For example, it might be used to display all of a class's properties, like in the `displayPropertiesAsList` function below.

```
import kotlin.reflect.full.memberProperties

fun displayPropertiesAsList(value: Any) {
    value::class.memberProperties
        .sortedBy { it.name }
        .map { p -> " * ${p.name}: ${p.call(value)}" }
        .forEach(::println)
}

class Person(
    val name: String,
    val surname: String,
    val children: Int,
    val female: Boolean,
)

class Dog(
    val name: String,
    val age: Int,
)

enum class DogBreed {
    HUSKY, LABRADOR, PUG, BORDER_COLLIE
}

fun main() {
    val granny = Person("Esmeralda", "Weatherwax", 0, true)
    displayPropertiesAsList(granny)
    // * children: 0
    // * female: true
    // * name: Esmeralda
```

```
// * surname: Weatherwax

val cookie = Dog("Cookie", 1)
displayPropertiesAsList(cookie)
// * age: 1
// * name: Cookie

displayPropertiesAsList(DogBreed.BORDER_COLLIE)
// * name: BORDER_COLLIE
// * ordinal: 3
}
```

Reflection is often used by libraries that analyze our code and behave according to how it is constructed. Let's see a few examples.

Libraries like Gson use reflection to serialize and deserialize objects. These libraries often reference classes to check which properties they require and which constructors they offer in order to then use these constructors. Later in this chapter, we will implement our own serializer.

```
data class Car(val brand: String, val doors: Int)  
  
fun main() {  
    val json = "{\"brand\":\"Jeep\", \"doors\": 3}\"\n    val gson = Gson()  
    val car: Car = gson.fromJson(json, Car::class.java)  
    println(car) // Car(brand=Jeep, doors=3)  
    val newJson = gson.toJson(car)  
    println(newJson) // {"brand":"Jeep", "doors": 3}  
}
```

As another example, we can see the Koin dependency injection framework, which uses reflection to identify the type that should be injected and to create and inject an instance appropriately.

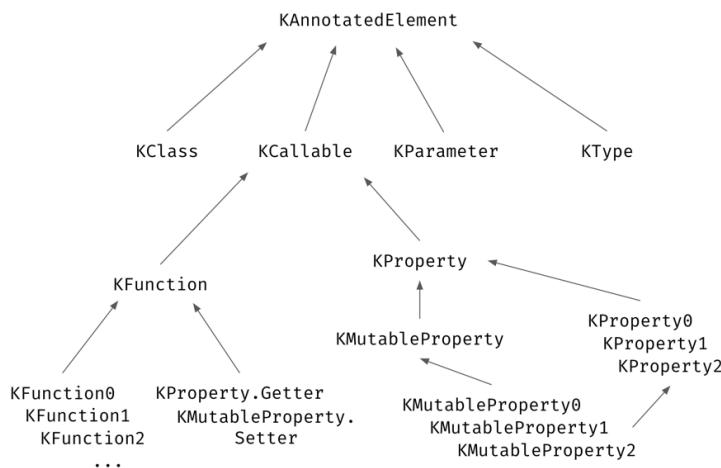
```
class MyActivity : Application() {  
    val myPresenter: MyPresenter by inject()  
}
```

Reflection is extremely useful, so let's start learning how we can use it ourselves.

To use Kotlin reflection, we need to add the `kotlin-reflect` dependency to our project. It is not needed to reference an element, but we need it for the majority of operations on element references. If you see `KotlinReflectionNotSupportedException`, it means the `kotlin-reflect` dependency is required. In the code examples in this chapter, I assume this dependency is included.

Hierarchy of classes

Before we get into the details, let's first review the general type hierarchy of element references.



Notice that all the types in this hierarchy start with the `K` prefix. This indicates that this type is part of Kotlin Reflection and differentiates these classes from Java Reflection. The type `Class` is part of Java Reflection, so Kotlin called its equivalent `KClass`.

At the top of this hierarchy, you can find `KAnnotatedElement`. `Element` is a term that includes classes, functions, and properties, so it includes everything we can reference. All elements can be annotated, which is why this interface includes the `annotations` property, which we can use to get element annotations.

```
interface KAnnotatedElement {  
    val annotations: List<Annotation>  
}
```

The next confusing thing you might have noticed is that there is no type to represent interfaces. This is because **interfaces in reflection API nomenclature are also considered classes, so their references are of type `KClass`**. This might be confusing, but it is really convenient.

Now we can get into the details, which is not easy because everything is connected to nearly everything else. At the same time, using the reflection API is really intuitive and easy to learn. Nevertheless, I decided that to help you understand this API better I'll do something I generally avoid doing: we will go through the essential classes and discuss their methods and properties. In between, I will show you some practical examples and explain some essential reflection concepts.

Function references

We reference functions using a double colon and a function name. Member references start with the type specified before colons.

```
import kotlin.reflect.*

fun printABC() {
    println("ABC")
}

fun double(i: Int): Int = i * 2

class Complex(val real: Double, val imaginary: Double) {
    fun plus(number: Number): Complex = Complex(
        real = real + number.toDouble(),
        imaginary = imaginary
    )
}

fun Complex.double(): Complex =
    Complex(real * 2, imaginary * 2)

fun Complex?.isNullOrZero(): Boolean =
    this == null ||
        (this.real == 0.0 && this.imaginary == 0.0)

class Box<T>(var value: T) {
    fun get(): T = value
}

fun <T> Box<T>.set(value: T) {
    this.value = value
}

fun main() {
    val f1 = ::printABC
    val f2 = ::double
    val f3 = Complex::plus
    val f4 = Complex::double
    val f5 = Complex?::isNullOrZero
    val f6 = Box<Int>::get
    val f7 = Box<String>::set
}
```

The result type from the function reference is an appropriate `KFunctionX`, where X indicates the number of parameters. This type also includes type parameters for each function parameter and the result. For instance, the `printABC` reference type is `KFunction0<Unit>`. For method references, a receiver is considered another parameter, so the `Complex::double` type is `KFunction1<Complex, Complex>`.

```
// ...
```

```
fun main() {
    val f1: KFunction0<Unit> =
        ::printABC
    val f2: KFunction1<Int, Int> =
        ::double
    val f3: KFunction2<Complex, Number, Complex> =
        Complex::plus
    val f4: KFunction1<Complex, Complex> =
        Complex::double
    val f5: KFunction1<Complex?, Boolean> =
        Complex?::isNullOrZero
    val f6: KFunction1<Box<Int>, Int> =
        Box<Int>::get
    val f7: KFunction2<Box<String>, String, Unit> =
        Box<String>::set
}
```

Alternatively, you can reference methods on concrete instances. These are so-called *bounded function references*, and they are represented with the same type but without additional parameters for the receiver.

```
// ...  
  
fun main() {  
    val c = Complex(1.0, 2.0)  
    val f3: KFunction1<Number, Complex> = c::plus  
    val f4: KFunction0<Complex> = c::double  
    val f5: KFunction0<Boolean> = c::isNullOrZero  
    val b = Box(123)  
    val f6: KFunction0<Int> = b::get  
    val f7: KFunction1<Int, Unit> = b::set  
}
```

All the specific types representing function references implement the `KFunction` type, with only one type parameter representing the function result type (because every function must have a result type in Kotlin).

```
// ...  
  
fun main() {  
    val f1: KFunction<Unit> = ::printABC  
    val f2: KFunction<Int> = ::double  
    val f3: KFunction<Complex> = Complex::plus  
    val f4: KFunction<Complex> = Complex::double  
    val f5: KFunction<Boolean> = Complex?::isNullOrZero  
    val f6: KFunction<Int> = Box<Int>::get  
    val f7: KFunction<Unit> = Box<String>::set  
    val c = Complex(1.0, 2.0)  
    val f8: KFunction<Complex> = c::plus  
    val f9: KFunction<Complex> = c::double  
    val f10: KFunction<Boolean> = c::isNullOrZero  
    val b = Box(123)  
    val f11: KFunction<Int> = b::get  
    val f12: KFunction<Unit> = b::set  
}
```

Now, what can we do with function references? In a previous book from this series, *Functional Kotlin*, I showed that they

can be used instead of lambda expressions where a function type is expected, like in the example below, where function references are used as arguments to `filterNot`, `map`, and `reduce`.

```
// ...  
  
fun nonZeroDoubled(numbers: List<Complex?>): List<Complex?> =  
    numbers  
        .filterNot(Complex::isNullOrZero)  
        .filterNotNull()  
        .map(Complex::double)
```

Using function references where a function type is expected is not “real” reflection because, under the hood, Kotlin transforms these references to lambda expressions. We use function references like this only for our own convenience.

```
fun nonZeroDoubled(numbers: List<Complex?>): List<Complex?> =  
    numbers  
        .filterNot { it.isNullOrZero() }  
        .filterNotNull()  
        .map { it.double() }
```

Formally, this is possible because types that represent function references, like `KFunction2<Int, Int, Int>`, implement function types; in this example, the implemented type is `(Int, Int) -> Int`. So, these types also include the `invoke` operator function, which lets the reference be called like a function.

```
fun add(i: Int, j: Int) = i + j  
  
fun main() {  
    val f: KFunction2<Int, Int, Int> = ::add  
    println(f(1, 2)) // 3  
    println(f.invoke(1, 2)) // 3  
}
```

The `KFunction` by itself includes only a few properties that let us check some function-specific characteristics:

- `isInline`: Boolean - true if this function is `inline`.
- `isExternal`: Boolean - true if this function is `external`.
- `isOperator`: Boolean - true if this function is `operator`.
- `isInfix`: Boolean - true if this function is `infix`.
- `isSuspend`: Boolean - true if this is a suspending function.

```
import kotlin.reflect.KFunction

inline infix operator fun String.times(times: Int) =
    this.repeat(times)

fun main() {
    val f: KFunction<String> = String::times
    println(f.isInline) // true
    println(f.isExternal) // false
    println(f.isOperator) // true
    println(f.isInfix) // true
    println(f.isSuspend) // false
}
```

`KCallable` has many more properties and a few functions. Let's start with the properties:

- `name`: `String` - The name of this callable as declared in the source code. If the callable has no name, a special invented name is created. Here are some atypical cases:
 - * constructors have the name “`<init>`”,
 - * property accessors: the getter for a property named “`foo`” will have the name “`<get-foo>`”; similarly the setter will have the name “`<set-foo>`”.
- `parameters`: `List<KParameter>` - a list of references to the parameters of this callable. We will discuss parameter references in a dedicated section.
- `returnType`: `KType` - the type that is expected as a result of this callable call. We will discuss the `KType` type in a dedicated section.

- `typeParameters: List<KTypeParameter>` - a list of generic type parameters of this callable. We will discuss the `KTypeParameter` type in the section dedicated to class references.
- `visibility: KVisibility?` - visibility of this callable, or `null` if its visibility cannot be represented in Kotlin. `KVisibility` is an enum class with values `PUBLIC`, `PROTECTED`, `INTERNAL`, and `PRIVATE`.
- `isFinal: Boolean` - true if this callable is `final`.
- `isOpen: Boolean` - true if this function is `open`.
- `isAbstract: Boolean` - true if this function is `abstract`.
- `isSuspend: Boolean` - true if this is a suspending function (it is defined in both `KFunction` and `KCallable`).

```
import kotlin.reflect.KCallable

operator fun String.times(times: Int) = this.repeat(times)

fun main() {
    val f: KCallable<String> = String::times
    println(f.name) // times
    println(f.parameters.map { it.name }) // [null, times]
    println(f.returnType) // kotlin.String
    println(f.typeParameters) // []
    println(f.visibility) // PUBLIC
    println(f.isFinal) // true
    println(f.isOpen) // false
    println(f.isAbstract) // false
    println(f.isSuspend) // false
}
```

`KCallable` also has two methods that can be used to call it. The first one, `call`, accepts a vararg number of parameters of type `Any?` and the result type `R`, which is the only `KCallable` type parameter. When we call the `call` method, we need to provide a proper number of values with appropriate types, otherwise, it throws `IllegalArgumentException`. Optional arguments must also have a value specified when we use the `call` function.

```
import kotlin.reflect.KCallable

fun add(i: Int, j: Int) = i + j

fun main() {
    val f: KCallable<Int> = ::add
    println(f.call(1, 2)) // 3
    println(f.call("A", "B")) // IllegalArgumentException
}
```

The second function, `callBy`, is used to call functions using named arguments. As an argument, it expects a map from `KParameter` to `Any?` that should include all non-optional arguments.

```
import kotlin.reflect.KCallable

fun sendEmail(
    email: String,
    title: String = "",
    message: String = ""
) {
    println(
        """
            Sending to $email
            Title: $title
            Message: $message
        """.trimIndent()
    )
}

fun main() {
    val f: KCallable<Unit> = ::sendEmail

    f.callBy(mapOf(f.parameters[0] to "ABC"))
    // Sending to ABC
    // Title:
    // Message:
```

```
val params = f.parameters.associateBy { it.name }
f.callBy(
    mapOf(
        params["title"]!! to "DEF",
        params["message"]!! to "GFI",
        params["email"]!! to "ABC",
    )
)
// Sending to ABC
// Title: DEF
// Message: GFI

f.callBy(mapOf()) // throws IllegalArgumentException
}
```

Parameter references

The `KCallable` type has the `parameters` property, with a list of references of type `KParameter`. This type includes the following properties:

- `index: Int` - the index of this parameter.
- `name: String?` - a simple parameter name, or `null` if the parameter has no name or its name is not available at runtime. Examples of nameless parameters include a `this` instance for member functions, an extension receiver for extension functions or properties, and parameters of Java methods compiled without debug information.
- `type: KType` - the type of this parameter.
- `kind: Kind` - the kind of this parameter, which can be one of the following:
 - * `VALUE` for regular parameters.
 - * `EXTENSION_RECEIVER` for extension receivers.
 - * `INSTANCE` for dispatch receivers, so instances needed to make member callable calls.
- `isOptional: Boolean` - true if this parameter is optional, therefore it has a default argument specified.
- `isVararg: Boolean` - true if this parameter is vararg.

As an example of how the `parameters` property can be used, I created the `callWithFakeArgs` function, which can be used to call a function reference with some constant values for the non-optional parameters of supported types. As you can see in the code below, this function takes parameters; it uses `filterNot` to keep only parameters that are not optional, and it then associates a value with each of them. A constant value is provided by the `fakeValueFor` function, which for `Int` always returns 123; for `String`, it constructs a fake value that includes a parameter name (the `typeOf` function will be described later in this chapter). The resulting map of parameters with associated values is used as an argument to `callBy`. You can see how this `callWithFakeArgs` can be used to execute different functions with the same arguments.

```
import kotlin.reflect.KCallable
import kotlin.reflect.KParameter
import kotlin.reflect.typeOf

fun callWithFakeArgs(callable: KCallable<*>) {
    val arguments = callable.parameters
        .filterNot { it.isOptional }
        .associateWith { fakeValueFor(it) }
    callable.callBy(arguments)
}

fun fakeValueFor(parameter: KParameter) =
    when (parameter.type) {
        typeOf<String>() -> "Fake ${parameter.name}"
        typeOf<Int>() -> 123
        else -> error("Unsupported type")
    }

fun sendEmail(
    email: String,
    title: String,
    message: String = ""
) {
    println(
```

```
    """
    Sending to $email
    Title: $title
    Message: $message
    """.trimIndent()
)
}

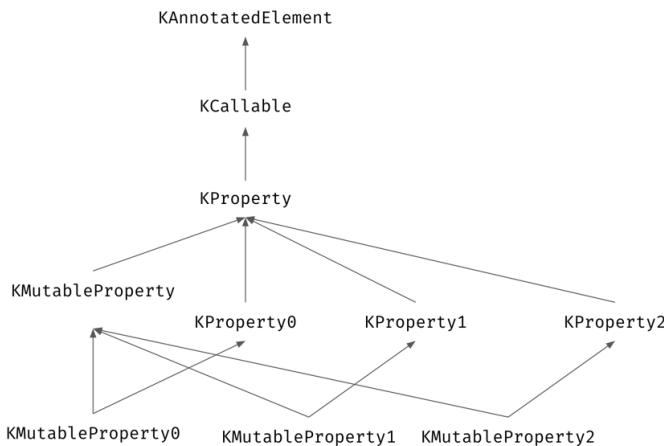
fun printSum(a: Int, b: Int) {
    println(a + b)
}

fun Int.printProduct(b: Int) {
    println(this * b)
}

fun main() {
    callWithFakeArgs(::sendEmail)
    // Sending to Fake email
    // Title: Fake title
    // Message:
    callWithFakeArgs(::printSum) // 246
    callWithFakeArgs(Int::printProduct) // 15129
}
```

Property references

Property references are similar to function references, but they have a slightly more complicated type hierarchy.



All property references implement `KProperty`, which implements `KCallable`. Calling a property means calling its getter. Read-write property references implement `KMutableProperty`, which implements `KProperty`. There are also specific types like `KProperty0` or `KMutableProperty1`, which specify how many receivers property calls require:

- Read-only top-level properties implement `KProperty0` because their getter can be called without any receiver.
- Read-only member or extension properties implement `KProperty1` because their getter needs a single receiver object.
- Read-only member extension properties implement `KProperty2` because their getter needs two receivers: a dispatch receiver and an extension receiver.
- Read-write top-level properties implement `KMutableProperty0` because their getter can be called without any receiver.
- Read-write member or extension properties implement `KMutableProperty1` because their getter and setter need a single receiver object.
- Read-write member extension properties implement `KMutableProperty2` because their getter and setter need

two receivers: a dispatch receiver and an extension receiver.

Properties are referenced just like functions: use two colons before their name and an additional class name for member properties. There is no syntax to reference member extension functions or member extension properties; so, to show the `KProperty2` example in the example below, I needed to find it in the class reference, which will be described in the next section.

```
import kotlin.reflect.*  
import kotlin.reflect.full.memberExtensionProperties  
  
val lock: Any = Any()  
var str: String = "ABC"  
  
class Box(  
    var value: Int = 0  
) {  
    val Int.addedToBox  
        get() = Box(value + this)  
}  
  
fun main() {  
    val p1: KProperty0<Any> = ::lock  
    println(p1) // val lock: kotlin.Any  
    val p2: KMutableProperty0<String> = ::str  
    println(p2) // var str: kotlin.String  
    val p3: KMutableProperty1<Box, Int> = Box::value  
    println(p3) // var Box.value: kotlin.Int  
    val p4: KProperty2<Box, *, *> = Box::class  
        .memberExtensionProperties  
        .first()  
    println(p4) // val Box.(kotlin.Int.)addedToBox: Box  
}
```

The `KProperty` type has a few property-specific properties:

- `isLateinit`: Boolean - true if this property is `lateinit`.
- `isConst`: Boolean - true if this property is `const`.
- `getter`: `Getter<V>` - a reference to an object representing a property getter.

`KMutableProperty` only adds a single property:

- `setter`: `Setter<V>` - a reference to an object representing a property setter.

Types representing properties with a specific number of receivers additionally provide this property getter's `get` function, and mutable variants also provide the `set` function for this property setter. Both `get` and `set` have an appropriate number of additional parameters for receivers. For instance, in `KMutableProperty1`, the `get` function expects a single argument for the receiver, and `set` expects one argument for the receiver and one for a value. Additionally, more specific types that represent properties provide more specific references to getters and setters.

```
import kotlin.reflect.*

class Box(
    var value: Int = 0
)

fun main() {
    val box = Box()
    val p: KMutableProperty1<Box, Int> = Box::value
    println(p.get(box)) // 0
    p.set(box, 999)
    println(p.get(box)) // 999
}
```

Class reference

To reference a class, we use the class name or instance, the double colon, and the `class` keyword. The result is `KClass<T>`,

where τ is the type representing a class.

```
import kotlin.reflect.KClass

class A

fun main() {
    val clazz1: KClass<A> = A::class
    println(clazz1) // class A

    val a: A = A()
    val clazz2: KClass<out A> = a::class
    println(clazz2) // class A
}
```

Note that the reference on the variable is covariant because a variable of type `A` might contain an object of type `A` or any of its subtypes.

```
import kotlin.reflect.KClass

open class A
class B : A()

fun main() {
    val a: A = B()
    val clazz: KClass<out A> = a::class
    println(clazz) // class B
}
```

Since `class` is a reserved keyword in Kotlin and cannot be used as the variable name, it is a popular practice to use “`clazz`” instead.

A class has two types of names:

- Simple name is just the name used after the `class` keyword. We can read it using the `simpleName` property.

- The fully qualified name is the name that includes the package and the enclosing classes. We can read it using the `qualifiedName` property.

```
package a.b.c

class D {
    class E
}

fun main() {
    val clazz = D.E::class
    println(clazz.simpleName) // E
    println(clazz.qualifiedName) // a.b.c.D.E
}
```

Both `simpleName` and `qualifiedName` return `null` when we reference an object expression or any other nameless object.

```
fun main() {
    val o = object {}
    val clazz = o::class
    println(clazz.simpleName) // null
    println(clazz.qualifiedName) // null
}
```

`KClass` has only a few properties that let us check some class-specific characteristics:

- `isFinal`: Boolean - true if this class is `final`.
- `isOpen`: Boolean - true if this class has the `open` modifier. Abstract and sealed classes, even though they are generally considered abstract, will return `false`.
- `isAbstract`: Boolean - true if this class has the `abstract` modifier. Sealed classes, even though they are generally considered abstract, will return `false`.

- `isSealed`: Boolean - true if this class has the `sealed` modifier.
- `isData`: Boolean - true if this class has the `data` modifier.
- `isInner`: Boolean - true if this class has the `inner` modifier.
- `isCompanion`: Boolean - true if this class is a companion object.
- `isFun`: Boolean - true if this class is a Kotlin functional interface and so has the `fun` modifier.
- `isValue`: Boolean - true if this class is a value class and so has the `value` modifier.

Just like for functions, we can check classes' visibility using the `visibility` property.

```
sealed class UserMessages

private data class UserId(val id: String) {
    companion object {
        val ZERO = UserId("")
    }
}

internal fun interface Filter<T> {
    fun check(value: T): Boolean
}

fun main() {
    println(UserMessages::class.visibility) // PUBLIC
    println(UserMessages::class.isSealed) // true
    println(UserMessages::class.isOpen) // false
    println(UserMessages::class.isFinal) // false
    println(UserMessages::class.isAbstract) // false

    println(UserId::class.visibility) // PRIVATE
    println(UserId::class.isData) // true
    println(UserId::class.isFinal) // true

    println(UserId.Companion::class.isCompanion) // true
```

```
    println(UserId.Companion::class.isInner) // false

    println(Filter::class.visibility) // INTERNAL
    println(Filter::class.isFun) // true
}
```

Functions and properties defined inside a class are known as members. This category does not include extension functions defined outside the class, but it does include elements defined by parents. Members defined by a particular class are called *declared members*. Since referencing a class to list its members is quite popular, it is good to know the following properties we can use:

- `members: Collection<KCallable<*>>` - returns all class members, including those declared by parents of this class.
- `functions: Collection<KFunction<*>>` - returns all class member functions, including those declared by parents of this class.
- `memberProperties: Collection<KProperty1<*>>` - returns all class member properties, including those declared by parents of this class.
- `declaredMembers: Collection<KCallable<*>>` - returns members declared by this class.
- `declaredFunctions: Collection<KFunction<*>>` - returns functions declared by this class.
- `declaredMemberProperties: Collection<KProperty1<*>>` - returns member properties declared by this class.

```
import kotlin.reflect.full.*

open class Parent {
    val a = 12
    fun b() {}
}

class Child : Parent() {
    val c = 12
    fun d() {}
}

fun Child.e() {}

fun main() {
    println(Child::class.members.map { it.name })
    // [c, d, a, b, equals, hashCode, toString]
    println(Child::class.functions.map { it.name })
    // [d, b, equals, hashCode, toString]
    println(Child::class.memberProperties.map { it.name })
    // [c, a]

    println(Child::class.declaredMembers.map { it.name })
    // [c, d]
    println(Child::class.declaredFunctions.map { it.name })
    // [d]
    println(
        Child::class.declaredMemberProperties.map { it.name }
    ) // [c]
}
```

A class constructor is not a member, but it is not considered a function either. We can get a list of all constructors using the `constructors` property.

```
package playground

import kotlin.reflect.KFunction

class User(val name: String) {
    constructor(user: User) : this(user.name)
    constructor(json: UserJson) : this(json.name)
}

class UserJson(val name: String)

fun main() {
    val constructors: Collection<KFunction<User>> =
        User::class.constructors

    println(constructors.size) // 3
    constructors.forEach(::println)
    // fun <init>(playground.User): playground.User
    // fun <init>(playground.UserJson): playground.User
    // fun <init>(kotlin.String): playground.User
}
```

We can get superclass references using the `superclasses` property, which returns `List<KClass<*>>`. In reflection API nomenclature, remember that interfaces are also considered classes, so their references are of type `KClass` and they are returned by the `superclasses` property. We can also get the types of the same direct superclass and directly implemented interfaces using the `supertypes` property, which returns `List<KType>`. This property actually returns a list of superclasses, not supertypes, as it doesn't include nullable types, but it includes `Any` if there is no other direct superclass.

```
import kotlin.reflect.KClass
import kotlin.reflect.full.superclasses

interface I1
interface I2
open class A : I1
class B : A(), I2

fun main() {
    val a = A::class
    val b = B::class
    println(a.superclasses) // [class I1, class kotlin.Any]
    println(b.superclasses) // [class A, class I2]
    println(a.supertypes) // [I1, kotlin.Any]
    println(b.supertypes) // [A, I2]
}
```

You can use a class reference to check if a specific object is a subtype of this class (or interface).

```
interface I1
interface I2
open class A : I1
class B : A(), I2

fun main() {
    val a = A()
    val b = B()
    println(A::class.isInstance(a)) // true
    println(B::class.isInstance(a)) // false
    println(I1::class.isInstance(a)) // true
    println(I2::class.isInstance(a)) // false

    println(A::class.isInstance(b)) // true
    println(B::class.isInstance(b)) // true
    println(I1::class.isInstance(b)) // true
    println(I2::class.isInstance(b)) // true
}
```

Generic classes have type parameters that are represented with the `KTypeParameter` type. We can get a list of all type parameters defined by a class using the `typeParameters` property.

```
fun main() {
    println(List::class.typeParameters) // [out E]
    println(Map::class.typeParameters) // [K, out V]
}
```

If a class includes some nested classes, we can get a list of them using the `nestedClasses` property.

```
class A {
    class B
    inner class C
}

fun main() {
    println(A::class.nestedClasses) // [class A$B, class A$C]
}
```

If a class is a sealed class, we can get a list of its subclasses using `sealedSubclasses: List<KClass<out T>>`.

```
sealed class LinkedList<out T>

class Node<out T>(
    val head: T,
    val next: LinkedList<T>
) : LinkedList<T>()

object Empty : LinkedList<Nothing>()

fun main() {
    println(LinkedList::class.sealedSubclasses)
    // [class Node, class Empty]
}
```

An object declaration has only one instance, and we can get its reference using the `objectInstance` property of type `T?`, where `T` is the `KClass` type parameter. This property returns `null` when a class does not represent an object declaration.

```
import kotlin.reflect.KClass

sealed class LinkedList<out T>

class Node<out T>(
    val head: T,
    val next: LinkedList<T>
) : LinkedList<T>()

object Empty : LinkedList<Nothing>()

fun printInstance(c: KClass<*>) {
    println(c.objectInstance)
}

fun main() {
    printInstance(Node::class) // null
    printInstance(Empty::class) // Empty@XYZ
}
```

Serialization example

Let's use our knowledge now on a practical example. Our goal is to define a `toJson` function which will serialize objects into JSON format.

```
class Creature(  
    val name: String,  
    val attack: Int,  
    val defence: Int,  
)  
  
fun main() {  
    val creature = Creature(  
        name = "Cockatrice",  
        attack = 2,  
        defence = 4  
    )  
    println(creature.toJson())  
    // {"attack": 2, "defence": 4, "name": "Cockatrice"}  
}
```

To help us implement `toJson`, I will define a couple of helper functions, starting with `objectToJson`, which is responsible for serializing objects to JSON and assumes that its argument is an object. Objects in JSON format are surrounded by curly braces containing property-value pairs separated with commas. In each pair, first there is a property name in quotes, then a colon, and then a serialized value. To implement `objectToJson`, we first need to have a list of object properties. For that, we will reference this object, and then we can either use `memberProperties` (including all properties in this object, including those inherited from the parent) or `declaredMemberProperties` (including properties declared by the class constructing this object). Once we have a list of properties, we can use `joinToString` to create a string with property-value pairs. We specify `prefix` and `postfix` parameters to surround the result string with curly brackets. We also define `transform` to specify how property-value pairs should be transformed to a string. Inside them, we take property names using the `name` property; we get property value by calling the `call` method from this property reference, and we then transform the result value to a string using the `valueToJson` function.

```
fun Any.toJson(): String = objectToJson(this)

private fun objectToJson(any: Any) = any::class
    .memberProperties
    .joinToString(
        prefix = "{",
        postfix = "}",
        transform = { prop ->
            "\"${prop.name}\": ${valueToJson(prop.call(any))}"
        }
    )
)
```

The above code needs the ‘valueToJson’ function to serialize JSON values. JSON format supports a number of values, but most of them can just be serialized using the Kotlin string template. This includes the `null` value, all numbers, and enums. An important exception is strings, which need to be additionally wrapped with quotes[^9_2]. All non-basic types will be treated as objects and serialized with the `objectToJson`‘function.

```
private fun valueToJson(value: Any?): String = when (value) {
    null, is Number -> "$value"
    is String, is Enum<*> -> "\"$value\""
    // ...
    else -> objectToJson(value)
}
```

This is all we need to make a simple JSON serialization function. To make it more functional, I also added some methods to serialize collections.

```
import kotlin.reflect.full.memberProperties

// Serialization function definition
fun Any.toJson(): String = objectToJson(this)

private fun objectToJson(any: Any) = any::class
    .memberProperties
    .joinToString(
        prefix = "{",
        postfix = "}",
        transform = { prop ->
            "\"${prop.name}\": ${valueToJson(prop.call(any))}"
        }
    )

private fun valueToJson(value: Any?): String = when (value) {
    null, is Number, is Boolean -> "$value"
    is String, is Enum<*> -> "\"$value\""
    is Iterable<*> -> iterableToJson(value)
    is Map<*, *> -> mapToJson(value)
    else -> objectToJson(value)
}

private fun iterableToJson(any: Iterable<*>): String = any
    .joinToString(
        prefix = "[",
        postfix = "]",
        transform = ::valueToJson
    )

private fun mapToJson(any: Map<*, *>) = any.toList()
    .joinToString(
        prefix = "{",
        postfix = "}",
        transform = {
            "\"${it.first}\": ${valueToJson(it.second)}"
        }
    )
```

```
// Example use
class Creature(
    val name: String,
    val attack: Int,
    val defence: Int,
    val traits: List<Trait>,
    val cost: Map<Element, Int>
)
enum class Element {
    FOREST, ANY,
}
enum class Trait {
    FLYING
}

fun main() {
    val creature = Creature(
        name = "Cockatrice",
        attack = 2,
        defence = 4,
        traits = listOf(Trait.FLYING),
        cost = mapOf(
            Element.ANY to 3,
            Element.FOREST to 2
        )
    )
    println(creature.toJson())
    // {"attack": 2, "cost": {"ANY": 3, "FOREST": 2},
    // "defence": 4, "name": "Cockatrice",
    // "traits": ["FLYING"]}
}
```

Before we close this topic, we might also practice working with annotations. We will define the `JsonName` annotation, which should set a different name for the serialized form, and `JsonIgnore`, which should make the serializer ignore the annotated property.

```
// Annotations
@Target(AnnotationTarget.PROPERTY)
annotation class JsonName(val name: String)

@Target(AnnotationTarget.PROPERTY)
annotation class JsonIgnore

// Example use
class Creature(
    @JsonIgnore val name: String,
    @JsonName("att") val attack: Int,
    @JsonName("def") val defence: Int,
    val traits: List<Trait>,
    val cost: Map<Element, Int>
)
enum class Element {
    FOREST, ANY,
}
enum class Trait {
    FLYING
}

fun main() {
    val creature = Creature(
        name = "Cockatrice",
        attack = 2,
        defence = 4,
        traits = listOf(Trait.FLYING),
        cost = mapOf(
            Element.ANY to 3,
            Element.FOREST to 2
        )
    )
    println(creature.toJson())
    // {"att": 2, "cost": {"ANY": 3, "FOREST": 2},
    // "def": 4, "traits": ["FLYING"]}
}
```

To respect these annotations, we need to modify our

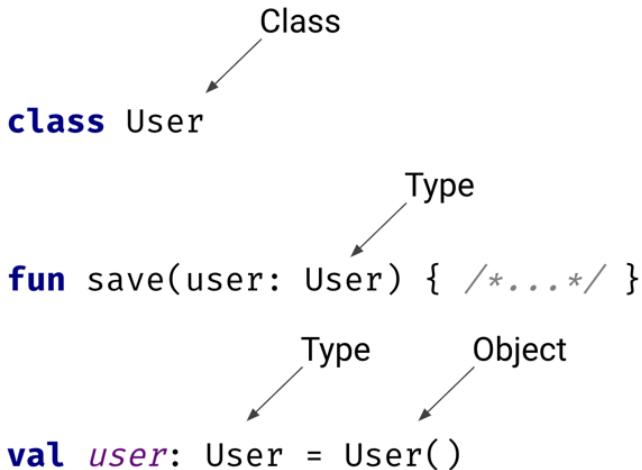
`objectToJson` function. To ignore properties, we will add a filter on the properties list. For each property, we need to check if it has the `JsonIgnore` annotation. To check if a property has this annotation, we could use the `annotations` property, but we can also use the `hasAnnotation` extension function on `KAnnotatedElement`. To respect a name change, we need to find the `JsonName` property annotation by using the `findAnnotation` extension function on `KAnnotatedElement`. This is how our function needs to be modified to respect both annotations:

```
private fun objectToJson(any: Any) = any::class
    .memberProperties
    .filterNot { it.hasAnnotation<JsonIgnore>() }
    .joinToString(
        prefix = "{",
        postfix = "}",
        transform = { prop ->
            val annotation = prop.findAnnotation<JsonName>()
            val name = annotation?.name ?: prop.name
            "\"${name}\": ${valueToJson(prop.call(any))}"
        }
    )
}
```

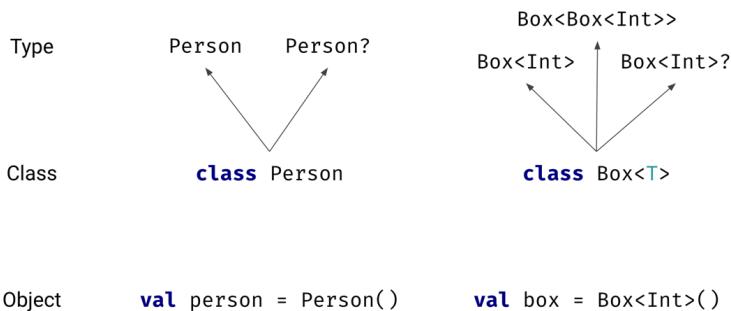
Referencing types

It's time to discuss type references of type `KType`. Types should not be confused with classes. Variables and parameters have types, not classes. Types can be nullable and can have type arguments³⁵.

³⁵I explained the differences between classes and types in the *The beauty of Kotlin's type system* chapter of *Kotlin Essentials*.



Examples of types and classes. This image was first published in my book *Kotlin Essentials*.



Relations between classes, types and objects. This image was first published in my book *Kotlin Essentials*.

To directly reference a type, we use the `typeof` function. Note that this function holds information about nullability and type arguments.

```
import kotlin.reflect.KType
import kotlin.reflect.typeOf

fun main() {
    val t1: KType = typeOf<Int?>()
    println(t1) // kotlin.Int?
    val t2: KType = typeOf<List<Int?>>()
    println(t2) // kotlin.collections.List<kotlin.Int?>
    val t3: KType = typeOf<() -> Map<Int, Char?>>()
    println(t3)
    // () -> kotlin.collections.Map<kotlin.Int, kotlin.Char?>
}
```

KType is a simple class with only three properties: isMarkedNullable, arguments, and classifier.

```
// Simplified KType definition
interface KType : KAnnotatedElement {
    val isMarkedNullable: Boolean
    val arguments: List<KTypeProjection>
    val classifier: KClassifier?
}
```

The isMarkedNullable property is simplest; it returns true if this type is marked as nullable in the source code.

```
import kotlin.reflect.typeOf

fun main() {
    println(typeOf<Int>().isMarkedNullable) // false
    println(typeOf<Int?>().isMarkedNullable) // true
}
```

Property arguments provide type arguments of this type, so the List<Int> type will have a single argument of type Int, and Map<Long, Char> will have two type arguments Long and Char. The type of these type arguments is KTypeProjection, which is a data class that includes type and a potential variance modifier, therefore Box<out String> has one type argument: out String.

```
// Simplified KTypeProjection definition
data class KTypeProjection(
    val variance: KVariance?,
    val type: KType?
)

import kotlin.reflect.typeOf

class Box<T>

fun main() {
    val t1 = typeOf<List<Int>>()
    println(t1.arguments) // [kotlin.Int]
    val t2 = typeOf<Map<Long, Char>>()
    println(t2.arguments) // [kotlin.Long, kotlin.Char]
    val t3 = typeOf<Box<out String>>()
    println(t3.arguments) // [out kotlin.String]
}
```

Finally, we have the `classifier` property, which is a way for a type to reference the associated class. Its result type is `KClassifier?`, which is a supertype of `KClass` and `KTypeParameter`. `KClass` represents a class or an interface. `KTypeParameter` represents a generic type parameter. The type classifier can be `KTypeParameter` when we reference generic class members. Also, `classifier` returns `null` when the type is not denotable in Kotlin, e.g., an intersection type.

```
import kotlin.reflect.*

class Box<T>(val value: T) {
    fun get(): T = value
}

fun main() {
    val t1 = typeOf<List<Int>>()
    println(t1.classifier) // class kotlin.collections.List
    println(t1 is KType) // true
```

```

    println(t1 is KClass<*>) // false
    val t2 = typeOf<Map<Long, Char>>()
    println(t2.classifier) // class kotlin.collections.Map
    println(t2.arguments[0].type?.classifier)
    // class kotlin.Long

    val t3 = Box<Int>::get.returnType.classifier
    println(t3) // T
    println(t3 is KTypeParameter) // true
}

// KTypeParameter definition
interface KTypeParameter : KClassifier {
    val name: String
    val upperBounds: List<KType>
    val variance: KVariance
    val isReified: Boolean
}

```

Type reflection example: Random value

To show how we can use type reference, we will implement the `ValueGenerator` class with a `randomValue` method that generates a random value of a specific type. We will specify two variants of this function: one expecting a type as a reified type argument, and another that expects a type reference as a regular argument.

```

class ValueGenerator(
    private val random: Random = Random,
) {
    inline fun <reified T> randomValue(): T =
        randomValue(typeOf<T>()) as T

    fun randomValue(type: KType): Any? = TODO()
}

```

When implementing the `randomValue` function, a philosophical problem arises: if a type is nullable, what is the probability that our random value is `null`? To solve this problem, I added a configuration that specifies the probability of a null value.

```
import kotlin.random.Random
import kotlin.reflect.KType
import kotlin.reflect.typeOf

class RandomValueConfig(
    val nullProbability: Double = 0.1,
)

class ValueGenerator(
    private val random: Random = Random,
    val config: RandomValueConfig = RandomValueConfig(),
) {

    inline fun <reified T> randomValue(): T =
        randomValue(typeOf<T>()) as T

    fun randomValue(type: KType): Any? = when {
        type.isMarkedNullable -> randomNullable(type)
        // ...
        else -> error("Type $type not supported")
    }

    private fun randomNullable(type: KType) =
        if (randomBoolean(config.nullProbability)) null
        else randomValue(type.withNullability(false))

    private fun randomBoolean(probability: Double) =
        random.nextDouble() < probability
}
```

Now we can add support for some other basic types. `Boolean` is simplest because it can be generated using `nextBoolean` from `Random`. The same can be said about `Int`, but `0` is a special value,

so I decided to specify its probability in the configuration as well.

```
import kotlin.math.ln
import kotlin.random.Random
import kotlin.reflect.KType
import kotlin.reflect.full.isSubtypeOf
import kotlin.reflect.typeOf

class RandomValueConfig(
    val nullProbability: Double = 0.1,
    val zeroProbability: Double = 0.1,
)

class ValueGenerator(
    private val random: Random = Random,
    val config: RandomValueConfig = RandomValueConfig(),
) {

    inline fun <reified T> randomValue(): T =
        randomValue(typeOf<T>()) as T

    fun randomValue(type: KType): Any? = when {
        type.isMarkedNullable &&
            randomBoolean(config.nullProbability) -> null
        type == typeOf<Boolean>() -> randomBoolean()
        type == typeOf<Int>() -> randomInt()
        // ...
        else -> error("Type $type not supported")
    }

    private fun randomInt() =
        if (randomBoolean(config.zeroProbability)) 0
        else random.nextInt()

    private fun randomBoolean() =
        random.nextInt()

    private fun randomBoolean(probability: Double) =
```

```
    random.nextDouble() < probability  
}
```

Finally, we should generate strings and lists. The biggest problem here is size. If we used random numbers as a size for random collections, these collections would be huge. A random value for a type like `List<List<List<String>>` could literally consume all our memory, not to mention the fact that the readability of such objects would be poor. But we should also make it possible for our function to generate a big collection or a string as this might be an edge case we need in some unit tests. I believe that the sizes of the collections and strings we use in real projects are described with exponential distribution: most of them are rather short, but some are huge. The exponential distribution is parametrized with a lambda value. I made this parameter configurable. By default, I decided to specify the exponential distribution parameter for strings as `0.1`, which makes our function generate an empty string with around 10% probability, and a string longer than 5 characters with around

55% probability. For lists, I specify the default parameter as `0.3`, which means that lists will be empty with around 25% probability, and they have only 16% probability of having a size greater than 5.

```
import kotlin.math.ln  
import kotlin.random.Random  
import kotlin.reflect.KType  
import kotlin.reflect.full.isSubtypeOf  
import kotlin.reflect.full.withNullability  
import kotlin.reflect.typeOf  
  
class RandomValueConfig(  
    val nullProbability: Double = 0.1,  
    val zeroProbability: Double = 0.1,  
    val stringSizeParam: Double = 0.1,  
    val listSizeParam: Double = 0.3,  
)
```

```
class ValueGenerator(
    private val random: Random = Random,
    val config: RandomValueConfig = RandomValueConfig(),
) {

    inline fun <reified T> randomValue(): T =
        randomValue(typeOf<T>()) as T

    fun randomValue(type: KType): Any? = when {
        type.isMarkedNullable -> randomNullable(type)
        type == typeOf<Boolean>() -> randomBoolean()
        type == typeOf<Int>() -> randomInt()
        type == typeOf<String>() -> randomString()
        type.isSubtypeOf(typeOf<List<*>>()) ->
            randomList(type)
        // ...
        else -> error("Type $type not supported")
    }

    private fun randomNullable(type: KType) =
        if (randomBoolean(config.nullProbability)) null
        else randomValue(type.withNullability(false))

    private fun randomString(): String =
        (1..random.exponential(config.stringSizeParam))
            .map { CHARACTERS.random(random) }
            .joinToString(separator = "")

    private fun randomList(type: KType) =
        List(random.exponential(config.listSizeParam)) {
            randomValue(type.arguments[0].type!!)
        }

    private fun randomInt() =
        if (randomBoolean(config.zeroProbability)) 0
        else random.nextInt()

    private fun randomBoolean() =
        random.nextBoolean()
```

```
private fun randomBoolean(probability: Double) =  
    random.nextDouble() < probability  
  
companion object {  
    private val CHARACTERS =  
        ('A'..'Z') + ('a'..'z') + ('0'..'9') + " "  
}  
  
private fun Random.exponential(f: Double): Int {  
    return (ln(1 - nextDouble()) / -f).toInt()  
}
```

Let's use what we've implemented so far to generate a bunch of random values:

```
fun main() {  
    val r = Random(1)  
    val g = ValueGenerator(random = r)  
    println(g.randomValue<Int>()) // -527218591  
    println(g.randomValue<Int?>()) // -2022884062  
    println(g.randomValue<Int?>()) // null  
    println(g.randomValue<List<Int>>())  
    // [-1171478239]  
    println(g.randomValue<List<List<Boolean>>>())  
    // [[true, true, false], [], [], [false, false], [],  
    // [true, true, true, true, true, true, true, false]]  
    println(g.randomValue<List<Int?>>())  
    // [-416634648, null, 382227801]  
    println(g.randomValue<String>()) // WjMNxTwDPrQ  
    println(g.randomValue<List<String?>>())  
    // [VAg, , null, AIKeGp9Q7, 1dqARHjUjee3i6XZzhQ02l, DLG, , ]  
}
```

Added 1 as a seed value to `Random` to produce predictable pseudo-random values for demonstration purposes.

We could push this project much further and make it support many more types. We could also generate random class instances thanks to the `classifier` property and `constructors`. Nevertheless, let's stop where we are.

Kotlin and Java reflection

On Kotlin/JVM, we can use the Java Reflection API, which is similar to the Kotlin Reflection API but is primarily designed to be used with Java, not Kotlin, but both type hierarchies are similar. We can transform between these two hierarchies using extension properties accessible on Kotlin/JVM. For example, we can use the `java` property to transform `KClass` to `Java Class`, or we can use `javaMethod` to transform `KFunction` to `Method`. Similarly, we can transform Java Reflection classes to Kotlin Reflection classes using extension properties which start with the `kotlin` prefix. For example, we can use the `kotlin` property to transform `Class` to `KClass`, and the `kotlinFunction` property to transform `Method` to `KFunction`.

```
import java.lang.reflect.*
import kotlin.reflect.*
import kotlin.reflect.jvm.*

class A {
    val a = 123
    fun b() {}
}

fun main() {
    val c1: Class<A> = A::class.java
    val c2: Class<A> = A().javaClass

    val f1: Field? = A::a.javaField
    val f2: Method? = A::a.javaGetter
    val f3: Method? = A::b.javaMethod

    val kotlinClass: KClass<A> = c1.kotlin
```

```
    val kotlinProperty: KProperty<*>? = f1?.kotlinProperty
    val kotlinFunction: KFunction<*>? = f3?.kotlinFunction
}
```

Breaking encapsulation

Using reflection, we can call elements we should not have access to. Each element has a specified accessibility parameter that we can check using the `isAccessible` property in the Kotlin and Java Reflection APIs. The same property can be used to change elements' accessibility. When we do that, we can call private functions or operate on private properties. Of course, you should avoid doing this if not absolutely necessary.

```
import kotlin.reflect.*
import kotlin.reflect.full.*
import kotlin.reflect.jvm.isAccessible

class A {
    private var value = 0
    private fun printValue() {
        println(value)
    }
    override fun toString(): String =
        "A(value=$value)"
}

fun main() {
    val a = A()
    val c = A::class

    // We change value to 999
    val prop: KMutableProperty1<A, Int>? =
        c.declaredMemberProperties
            .find { it.name == "value" }
            ?.as? KMutableProperty1<A, Int>
    prop?.isAccessible = true
}
```

```
prop?.set(a, 999)
println(a) // A(value=999)
println(prop?.get(a)) // 999

// We call printValue function
val func: KFunction<*>? =
    c.declaredMemberFunctions
        .find { it.name == "printValue" }
func?.isAccessible = true
func?.call(a) // 999
}
```

Summary

Reflection is a powerful tool used by many libraries to implement functionalities specific to code elements' definitions. We shouldn't use reflection too often in our applications as it is considered heavy, but it is good to know how to use it because it offers possibilities that cannot be substituted with anything else.

Exercise: Function caller

Your task is to implement methods of a class that is used to call function references with constant values specified by type. This class should have the following methods:

- `setConstant` - sets a constant value for a given type.
- `call` - calls a function reference with constant values specified by type.

If a constant value for a given type is not specified, an exception should be thrown. Unless it is an optional parameter, then its default argument should be used.

Starting code:

```
class FunctionCaller {  
    inline fun <reified T> setConstant(value: T) {  
        setConstant(typeOf<T>(), value)  
    }  
  
    fun setConstant(type: KType, value: Any?) {  
        TODO()  
    }  
  
    fun <T> call(function: KFunction<T>): T {  
        TODO()  
    }  
}
```

Usage example:

```
fun printStrIntNum(str: String, int: Int, num: Number) {  
    println("str: $str, int: $int, num: $num")  
}  
  
fun printWithOptionals(l: Long = 999, s: String) {  
    println("l: $l, s: $s")  
}  
  
fun main() {  
    val caller = FunctionCaller()  
    caller.setConstant("ABC")  
    caller.setConstant(123)  
    caller.setConstant(typeOf<Number>(), 3.14)  
    caller.call(::printStrIntNum)  
    // str: ABC, int: 123, num: 3.14  
    caller.call(::printWithOptionals)  
    // l: 999, s: ABC  
}
```

Starting code, usage examples and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file advanced/reflection/FunctionCaller.kt. You can clone this project and solve this exercise locally.

Hint: To support optional parameters, you should use `callBy` instead of `call`.

Exercise: Object serialization to JSON

Your task is to implement a function to serialize a Kotlin object to JSON. The result text should include all class member properties. You should support primitive types, iterables (present as an array), and map type (present as an object). You should also support nested objects. Do not use any external libraries. Kotlin's reflection is all you need.

You should support the following annotations:

- `@SerializationName` - can be applied to a property to change its name in the resulting JSON.
- `@SerializationIgnore` - can be applied to a property to ignore it in the resulting JSON.
- `@SerializationNameMapper` - can be applied to a class or a property to specify a custom name mapper. The mapper should implement `NameMapper` interface. This mapper can be an object declaration, or a class with a no-arg constructor.
- `@SerializationIgnoreNulls` - can be applied to a class to ignore all null properties in the resulting JSON.

```
@Target(AnnotationTarget.PROPERTY)
annotation class SerializationName(val name: String)
```

```
@Target(AnnotationTarget.PROPERTY)
annotation class SerializationIgnore
```

```
@Target(AnnotationTarget.PROPERTY, AnnotationTarget.CLASS)
annotation class SerializationNameMapper(
    val mapper: KClass<out NameMapper>
)
```

```
@Target(AnnotationTarget.CLASS)
```

```
annotation class SerializationIgnoreNulls

interface NameMapper {
    fun map(name: String): String
}
```

Starting code:

```
fun serializeToJson(value: Any): String = TODO()
```

Usage example:

```
@SerializationNameMapper(SnakeCaseName::class)
@SerializationIgnoreNulls
class Creature(
    val name: String,
    @SerializationName("att")
    val attack: Int,
    @SerializationName("def")
    val defence: Int,
    val traits: List<Trait>,
    val elementCost: Map<Element, Int>,
    @SerializationNameMapper(LowerCaseName::class)
    val isSpecial: Boolean,
    @SerializationIgnore
    var used: Boolean = false,
    val extraDetails: String? = null,
)

object LowerCaseName : NameMapper {
    override fun map(name: String): String = name.lowercase()
}

class SnakeCaseName : NameMapper {
    val pattern = "(?<=.)[A-Z]".toRegex()

    override fun map(name: String): String =
        name.replace(pattern, "_$0").lowercase()
```

```
}

enum class Element {
    FOREST, ANY,
}

enum class Trait {
    FLYING
}

fun main() {
    val creature = Creature(
        name = "Cockatrice",
        attack = 2,
        defence = 4,
        traits = listOf(Trait.FLYING),
        elementCost = mapOf(
            Element.ANY to 3,
            Element.FOREST to 2
        ),
        isSpecial = true,
    )
    println(serializeToJson(creature))
    // {"att": 2, "def": 4,
    // "element_cost": {"ANY": 3, "FOREST": 2},
    // "isspecial": true, "name": "Cockatrice",
    // "traits": ["FLYING"]}
}
```

Starting code, usage examples and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file advanced/reflection/JsonSerializer.kt. You can clone this project and solve this exercise locally.

Exercise: Object serialization to XML

Your task is to implement a function to serialize a Kotlin object to XML. The result text should include all class member

properties. You should also support nested objects. Do not use any external libraries. Kotlin's reflection is all you need.

You should support the following annotations:

- `@SerializationName` - can be applied to a property to change its name in the resulting XML.
- `@SerializationIgnore` - can be applied to a property to ignore it in the resulting XML.
- `@SerializationNameMapper` - can be applied to a class or a property to specify a custom name mapper. The mapper should implement `NameMapper` interface. This mapper can be an object declaration, or a class with a no-arg constructor.
- `@SerializationIgnoreNulls` - can be applied to a class to ignore all null properties in the resulting XML.

```
@Target(AnnotationTarget.PROPERTY)
annotation class SerializationName(val name: String)

@Target(AnnotationTarget.PROPERTY)
annotation class SerializationIgnore

@Target(AnnotationTarget.PROPERTY, AnnotationTarget.CLASS)
annotation class SerializationNameMapper(
    val mapper: KClass<out NameMapper>
)

@Target(AnnotationTarget.CLASS)
annotation class SerializationIgnoreNulls

interface NameMapper {
    fun map(name: String): String
}
```

Starting code:

```
fun serializeToXml(value: Any): String = TODO()
```

Usage example (result XML should not include indentation, it was added here for readability):

```
fun main() {
    data class SampleDataClass(
        val externalTxnId: String,
        val merchantTxnId: String,
        val reference: String
    )

    val data = SampleDataClass(
        externalTxnId = "07026984141550752666",
        merchantTxnId = "07026984141550752666",
        reference = "MERCHPAY"
    )

    println(serializeToXml(data))
    // <SampleDataClass>
    //   <externalTxnId>07026984141550752666<externalTxnId>
    //   <merchantTxnId>07026984141550752666<merchantTxnId>
    //   <reference>MERCHPAY<reference>
    // </SampleDataClass>

    @SerializationNameMapper(UpperSnakeCaseName::class)
    @SerializationIgnoreNulls
    class Book(
        val title: String,
        val author: String,
        @SerializationName("YEAR")
        val publicationYear: Int,
        val isbn: String?,
        @SerializationIgnore
        val price: Double,
    )

    @SerializationNameMapper(UpperSnakeCaseName::class)
    class Library(
```

```
    val catalog: List<Book>
)

val library = Library(
    catalog = listOf(
        Book(
            title = "The Hobbit",
            author = "J. R. R. Tolkien",
            publicationYear = 1937,
            isbn = "978-0-261-10235-4",
            price = 9.99,
        ),
        Book(
            title = "The Witcher",
            author = "Andrzej Sapkowski",
            publicationYear = 1993,
            isbn = "978-0-575-09404-2",
            price = 7.99,
        ),
        Book(
            title = "Antifragile",
            author = "Nassim Nicholas Taleb",
            publicationYear = 2012,
            isbn = null,
            price = 12.99,
        )
    )
)

println(serializeToXml(library))
// <LIBRARY>
//   <CATALOG>
//     <BOOK>
//       <AUTHOR>J. R. R. Tolkien<AUTHOR>
//       <ISBN>978-0-261-10235-4<ISBN>
//       <YEAR>1937<YEAR>
//       <TITLE>The Hobbit<TITLE>
//     </BOOK>
//     <BOOK>
```

```
//      <AUTHOR>Andrzej Sapkowski<AUTHOR>
//      <ISBN>978-0-575-09404-2<ISBN>
//      <YEAR>1993<YEAR>
//      <TITLE>The Witcher<TITLE>
//      </BOOK>
//      <BOOK>
//          <AUTHOR>Nassim Nicholas Taleb<AUTHOR>
//          <YEAR>2012<YEAR>
//          <TITLE>Antifragile<TITLE>
//          </BOOK>
//      <CATALOG>
//  </LIBRARY>
}
```

Starting code, usage examples and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file advanced/reflection/XmlSerializer.kt. You can clone this project and solve this exercise locally.

Exercise: DSL-based dependency injection library

Your task is to implement a simple dependency injection library. It should be based on the `Registry` class, that should be used to register dependencies. It should have the following methods:

- `register` - registers a normal dependency, that is created every time it is needed. It should take a type and a lambda expression that returns an instance of that type. In the scope of this lambda expression, you should be able to use `Registry` to get other dependencies. This function should have both an inline version with reified type, and a non-inline version with `KClass` parameter.
- `singletor` - registers a singleton dependency, that is created only once and then reused. It should take a type and a lambda expression that returns an instance of that type. In the scope of this lambda expression, you should

be able to use `Registry` to get other dependencies. This function should have both an inline version with reified type, and a non-inline version with `KClass` parameter.

- `get` - returns an instance of a given type. If the type is registered as a singleton, it should return the same instance every time. If the type is registered as a normal dependency, it should return a new instance every time. This function should have both an inline version with reified type, and a non-inline version with `KClass` parameter.
- `exists` - returns true if a given type is registered, false otherwise. This function should have both an inline version with reified type, and a non-inline version with `KClass` parameter.

You should also implement `registry` function to create a `Registry` instance in DSL style. It should take a lambda expression with `Registry` as a receiver, and return a `Registry` instance. In the scope of this lambda expression, you should be able to use `Registry` to register dependencies.

Usage example:

```
data class UserConfiguration(val url: String)

interface UserRepository {
    fun get(): String
}

class RealUserRepository(
    private val userConfiguration: UserConfiguration,
) : UserRepository {
    override fun get(): String =
        "User from ${userConfiguration.url}"
}

class UserService(
    private val userRepository: UserRepository,
    private val userConfiguration: UserConfiguration,
) {
```

```
fun get(): String = "Got ${userRepository.get()}"  
}  
  
fun main() {  
    val registry: Registry = registry {  
        singleton<UserConfiguration> {  
            UserConfiguration("http://localhost:8080")  
        }  
        normal<UserService> {  
            UserService(  
                userRepository = get(),  
                userConfiguration = get(),  
            )  
        }  
        singleton<UserRepository> {  
            RealUserRepository(  
                userConfiguration = get(),  
            )  
        }  
    }  
  
    val userService: UserService = registry.get()  
    println(userService.get())  
    // Got User from http://localhost:8080  
  
    val ur1 = registry.get<UserRepository>()  
    val ur2 = registry.get<UserRepository>()  
    println(ur1 === ur2) // true  
  
    val uc1 = registry.get<UserService>()  
    val uc2 = registry.get<UserService>()  
    println(uc1 === uc2) // false  
}
```

Usage example and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file advanced/reflection/DependencyInjection.kt. You can clone this project and solve this exercise locally.

Annotation processing

Java 5 introduced a new tool that completely changed how Java development looks: annotation processing. Many important Java libraries rely on annotation processing, including Spring, Hibernate, Room, Dagger, and many more. One could even say that annotation processing is essential for modern Java development and, as a result, also Kotlin/JVM development. Regardless of this, most Java and Kotlin/JVM developers don't understand how it works. This is perfectly fine as a driver doesn't need to understand how a car works, but understanding annotation processing can help us debug libraries, develop them, or implement our own tools. So, this chapter will explain how annotation processing works and show how a custom annotation processor can be implemented.

Before we start, I need to warn you that annotation processing only works for Kotlin/JVM as it still needs javac and cannot be used for other targets (Kotlin/JS or Kotlin/Native). Additionally, javac Annotation processing is costly, so Kotlin decided it needed its own annotation processor. As a result, Google introduced Kotlin Symbol Processing (KSP), which is a direct successor of annotation processing. We will discuss KSP in the next chapter, and the current chapter can be treated as an introduction or prerequisite to fully understanding KSP.

Your first annotation processor

The idea behind annotation processing is quite simple: we define classes called processors that analyze our source code and generate additional files that typically also include code; however, these processors themselves don't modify existing code. As an example, I will implement a simple library based on the idea of a friend of mine. I've never used it in any project, but it is simple enough to serve as a great example. To understand the idea, let's see the problem first. For many classes, we define interfaces primarily to make it easier to define fake variants of these classes for unit testing.

Consider the `MongoUserRepository` below, which implements the `UserRepository` interface with a fake `FakeUserRepository` for unit tests.

```
interface UserRepository {
    fun findUser(userId: String): User?
    fun findUsers(): List<User>
    fun updateUser(user: User)
    fun insertUser(user: User)
}

class MongoUserRepository : UserRepository {
    override fun findUser(userId: String): User? = TODO()
    override fun findUsers(): List<User> = TODO()
    override fun updateUser(user: User) {
        TODO()
    }
    override fun insertUser(user: User) {
        TODO()
    }
}

class FakeUserRepository : UserRepository {
    private var users = listOf<User>()

    override fun findUser(userId: String): User? =
        users.find { it.id == userId }

    override fun findUsers(): List<User> = users

    override fun updateUser(user: User) {
        val oldUsers = users.filter { it.id == user.id }
        users = users - oldUsers + user
    }

    override fun insertUser(user: User) {
        users = users + user
    }
}
```

The form of `UserRepository` is determined by the methods that we want to expose by `MongoUserRepository`; therefore, this class and interface often change together, so it might be simpler for `UserRepository` to be automatically generated based on public methods in `MongoUserRepository`³⁶. We can do this using annotation processing.

```

@GenerateInterface("UserRepository")
class MongoUserRepository : UserRepository {
    override fun findUser(userId: String): User? = TODO()
    override fun findUsers(): List<User> = TODO()
    override fun updateUser(user: User) {
        TODO()
    }
    override fun insertUser(user: User) {
        TODO()
    }
}

class FakeUserRepository : UserRepository {
    private var users = listOf<User>()

    override fun findUser(userId: String): User? =
        users.find { it.id == userId }

    override fun findUsers(): List<User> = users

    override fun updateUser(user: User) {
        val oldUsers = users.filter { it.id == user.id }
        users = users - oldUsers + user
    }
}

```

³⁶This idea goes against the practices we use in modern JVM development. Interfaces that define repositories (ports) are typically part of the domain layer, where their implementations are part of the data layer. What's more, at least in theory, we should define our repositories based on the abstraction we've specified by interfaces, not the other way around. That is why the usefulness of this annotation processor is very limited. Nevertheless, it will serve as a good example.

```
    }

    override fun insertUser(user: User) {
        users = users + user
    }
}
```

The complete project can be found on GitHub under the name [MarcinMoskala/generateinterface-ap](#).

For this, we need two things:

- Definition of the `GenerateInterface` annotation.
- Definition of the processor that generates the appropriate interfaces based on annotations.

The processor needs to be defined in a separate module because its code is not added to our source code and shipped to production; instead, it is used during compilation. An annotation is just a simple declaration and needs to be accessible in both your project and the annotation processor, so it also needs to be located in a separate module. This is why I will define two additional modules:

- `generateinterface-annotations` - which is just a regular module that includes `GenerateInterface`.
- `generateinterface-processor` - where I will define my annotation processor.

For our own convenience, I will use Kotlin in both these modules, but they could also be implemented in any other JVM language, like Java or Groovy.

We need to use these modules in our main module configuration³⁷. The module that contains your annotation should be

³⁷By “main module” I mean the module that will use annotation processing.

attached like any other dependency. To use annotation processing in Kotlin, we should use the `kapt` plugin³⁸. Assuming we use Gradle³⁹ in our project, this is how we might define our main module dependency in newly created modules.

```
// build.gradle.kts
plugins {
    kotlin("kapt") version "<your_kotlin_version>"
}

dependencies {
    implementation(project(":generateinterface-annotations"))
    kapt(project(":generateinterface-processor"))
    // ...
}
```

If we distribute our solution as a library, we need to publish both the annotations and the processor as separate packages.

All we need in the `generateinterface-annotations` module is a simple file with the following annotation:

```
package academy.kt

import kotlin.annotation.AnnotationTarget.CLASS

@Target(CLASS)
annotation class GenerateInterface(val name: String)
```

In the `generateinterface-processor` module, we need to specify the annotation processor. All annotation processors must extend the `AbstractProcessor` class.

³⁸As its documentation specifies, `kapt` is in maintenance mode, which means its creators are keeping it up-to-date with recent Kotlin and Java releases but have no plans to implement new features.

³⁹IDEA's built-in compiler does not directly support `kapt` and annotation processing.

```
package academy.kt

class GenerateInterfaceProcessor : AbstractProcessor() {
    // ...
}
```

There must also be a document that specifies that this class will be used as an annotation processor. We must create a file named `javax.annotation.processing.Processor` under the path `src/main/resources/META-INF/services`. Inside this file, you need to specify the processor using a fully qualified name:

```
academy.kt.GenerateInterfaceProcessor
```

Alternatively, one might use the Google AutoService library and just annotate the processor with `@AutoService(Processor::class.java)`.

Inside our processor, we should override the following methods:

- `getSupportedAnnotationTypes` - specifies a set of annotations our processor responds to. Should return `Set<String>`, where each value is a fully qualified annotation name (`qualifiedName` property). If this set includes "*", it means that the processor is interested in all annotations.
- `getSupportedSourceVersion` - specifies the latest Java source version this processor supports. To support the latest possible version, use `SourceVersion.latestSupported()`.
- `process` - this is where our processing and code generation will be implemented. It receives as an argument a set of annotations that are chosen based on `getSupportedAnnotationTypes`. It also receives a reference to `RoundEnvironment`, which lets us analyze the source code of the project where the processor is running. In every round, the compiler looks for more annotated

elements that could have been generated by a previous round until there are no more inputs. It returns a Boolean that determines if the annotations from the argument should be considered claimed by this processor. So, if we return true, other processors will not receive these annotations. Since we operate on custom annotations, we will return true. In our case, we will only need the RoundEnvironment reference, and I will make a separate method, generateInterfaces, which will generate interfaces.

```
class GenerateInterfaceProcessor : AbstractProcessor() {

    override fun getSupportedAnnotationTypes(): Set<String> =
        setOf(GenerateInterface::class.qualifiedName!!)

    override fun getSupportedSourceVersion(): SourceVersion =
        SourceVersion.latestSupported()

    override fun process(
        annotations: Set<TypeElement>,
        roundEnv: RoundEnvironment
    ): Boolean {
        generateInterfaces(roundEnv)
        return true
    }

    private fun generateInterfaces(
        roundEnv: RoundEnvironment
    ) {
        // ...
    }
}
```

Note that when we implement our annotation processor, we don't have access to typical class or function references from the project where the processor is running. To have these references, the project needs to be compiled, and our processor

runs before the compilation phase. The annotation processor operates on a separate type hierarchy that represents declared code elements and has some essential limitations. The annotation processor has the capability to introspect the types of your code but it cannot actually run functions or instantiate classes.

So, now let's focus on the `generateInterfaces` method implementation. We first need to find all the elements that are annotated with `GenerateInterface`. For that, we can use `getElementsAnnotatedWith` from `RoundEnvironment`, which should produce a set of element references of type `Element`. Since our annotation can only be used for classes (this is specified using the `Target` meta-annotation), we can expect that all these elements are of type `TypeElement`. To safely cast our set, I will use the `filterIsInstance` method; then, we can iterate over the result using the `forEach` method.

```
private fun generateInterfaces(roundEnv: RoundEnvironment) {
    roundEnv
        .getElementsAnnotatedWith(
            GenerateInterface::class.java
        )
        .filterIsInstance<TypeElement>()
        .forEach(::generateInterface)
}

private fun generateInterface(annotatedClass: TypeElement) {
    // ...
}
```

Now, for each annotated element, we should generate an interface in the `generateInterface` function. I will start by finding the expected interface name, which should be specified in the annotation. We can get the annotation reference by finding it in the `annotatedClass` parameter, and then we can use this value to read the annotated class name. All annotation properties must be static, therefore they are exposed in annotation references on annotation processors.

```
val interfaceName = annotatedClass
    .getAnnotation(GenerateInterface::class.java)
    .name
```

We also need to establish the package in which our interface should be located. I decided to just use the same package as the package of the annotated class. To find this package, we can use the `getPackageOf` method from `elementUtils` from `processingEnv` of our `AbstractProcessor`.

```
val interfacePackage = processingEnv
    .elementUtils
    .getPackageOf(annotatedClass)
    .qualifiedName
    .toString()
```

Finally, we need to find the public methods from our annotated class. For that, we will use the `enclosedElements` property to get all the enclosed elements and find those that are methods and have the `public` modifier. All methods should implement the `ExecutableElement` interface; so, to safely cast our elements we can use the `filterIsInstance` again.

```
val publicMethods = annotatedClass.enclosedElements
    .filter { it.kind == ElementKind.METHOD }
    .filter { Modifier.PUBLIC in it.modifiers }
    .filterIsInstance<ExecutableElement>()
```

Based on these values, I will build a file representation for our interface, and I will use the `processingEnv.filer` property to actually write a file. There are a number of libraries that can help us construct a file, but I decided to use JavaPoet (created and open-sourced by Square), which is both popular and simple to use. I extracted the method `buildInterfaceFile` to a Java file and used `writeTo` on its result to write the file.

```
private fun generateInterface(annotatedClass: TypeElement) {
    val interfaceName = annotatedClass
        .getAnnotation(GenerateInterface::class.java)
        .name
    val interfacePackage = processingEnv
        .elementUtils
        .getPackageOf(annotatedClass)
        .qualifiedName
        .toString()

    val publicMethods = annotatedClass.enclosedElements
        .filter { it.kind == ElementKind.METHOD }
        .filter { Modifier.PUBLIC in it.modifiers }
        .filterIsInstance<ExecutableElement>()

    buildInterfaceFile(
        interfacePackage,
        interfaceName,
        publicMethods
    ).writeTo(processingEnv.filer)
}
```

Note that you can also use a library like `KotlinPoet` and generate a Kotlin file instead of a Java file. I decided to generate a Java file for two reasons:

1. If we generate a Kotlin file, such a processor can only be used in projects using Kotlin/JVM⁴⁰. When we generate Java files, such processors can be used on Kotlin/JVM as well as by Java, Scala, Groovy, etc⁴¹.
2. Java element references are not always suitable for Kotlin code generation. For instance, Java `java.lang.String` translates to Kotlin `kotlin.String`. If we rely on Java references, we will use `java.lang.String` for

⁴⁰In this project, the Kotlin compiler must be used in the project build process.

⁴¹In this project, the Java compiler must be used in the project build process.

parameters in generated Kotlin code, which might not work correctly. Such problems can be overcome, but let's keep our example simple.

So, let's start building our elements. JavaPoet is based on the builder pattern that we need to use to construct elements on all levels. We will first build the file with the package and the built interface.

```
private fun buildInterfaceFile(
    interfacePackage: String,
    interfaceName: String,
    publicMethods: List<ExecutableElement>
): JavaFile = JavaFile.builder(
    interfacePackage,
    buildInterface(interfaceName, publicMethods)
).build()
```

To build the interface, we need to specify the name and then the build methods.

```
private fun buildInterface(
    interfaceName: String,
    publicMethods: List<ExecutableElement>
): TypeSpec = TypeSpec
    .interfaceBuilder(interfaceName)
    .addMethods(publicMethods.map(::buildInterfaceMethod))
    .build()
```

To build a method, we need to specify a name based on a method reference, use the same modifiers plus abstract, and add the same parameters (with the same annotations and the same result types). Note that we can find the `annotationMirrors` property in `ExecutableElement`, and it can be transformed to `AnnotationSpec` using the static `get` method.

```
private fun buildInterfaceMethod(  
    method: ExecutableElement  
) : MethodSpec = MethodSpec  
    .methodBuilder(method.simpleName.toString())  
    .addModifiers(method.modifiers)  
    .addModifiers(Modifier.ABSTRACT)  
    .addParameters(  
        method.parameters  
        .map(::buildInterfaceMethodParameter)  
    )  
    .addAnnotations(  
        method.annotationMirrors  
        .map(AnnotationSpec::get)  
    )  
    .returns(method.returnType.toTypeSpec())  
    .build()
```

Inside this method, I used two helpful extension functions, `toTypeSpec` and `getAnnotationSpecs`, which I defined outside our processor class:

```
private fun TypeMirror.toTypeSpec() = TypeName.get(this)  
    .annotated(this.getAnnotationSpecs())  
  
private fun AnnotatedConstruct.getAnnotationSpecs() =  
    annotationMirrors.map(AnnotationSpec::get)
```

To build method parameters, I start from a parameter reference whose type is `VariableElement`. I use it to make type specs and to find out the parameter names. I also use the same annotations as used for this parameter.

```
private fun buildInterfaceMethodParameter(  
    variableElement: VariableElement  
) : ParameterSpec = ParameterSpec  
    .builder()  
        variableElement.asType().toTypeSpec(),  
        variableElement.simpleName.toString()  
    )  
    .addAnnotations(variableElement.getAnnotationSpecs())  
    .build()
```

That is all we need. If you build your main module again, the code using the `GenerateInterface` annotation should compile.

```
@GenerateInterface( name: "UserRepository")  
class MongoUserRepository: UserRepository {  
    override fun findUser(userId: String): User? = TODO()  
    override fun findUsers(): List<User> = TODO()  
    override fun updateUser(user: User) { TODO() }  
    override fun insertUser(user: User) { TODO() }  
}  
  
class FakeUserRepository: UserRepository {  
    private var users = listOf<User>()  
  
    override fun findUser(userId: String): User? =  
        users.find { it.id == userId }
```

You can also jump to the implementation of `UserRepository` and see the Java code that our processor generated. The default location of generated code is “`build/generated/source/kapt/main`”. IntelliJ’s Gradle plugin will mark this location as a source code folder, thus making it navigable in IDEA.

```
interface UserRepository {  
    @Nullable  
    User findUser(@NotNull String userId);  
  
    @NotNull  
    List<User> findUsers();  
  
    void updateUser(@NotNull User user);  
  
    void insertUser(@NotNull User user);  
}
```

Hiding generated classes

Note that for our `UserRepository` to work, the project needs to be built. In a newly opened project, or immediately after adding the `GenerateInterface` annotation, the interface will not yet have been generated and our code will look like it is not correct.

```
@GenerateInterface( name: "UserRepository")  
class MongoUserRepository: UserRepository {  
    override fun findUser(userId: String): User? = TODO()  
    override fun findUsers(): List<User> = TODO()  
    override fun updateUser(user: User) { TODO() }  
    override fun insertUser(user: User) { TODO() }  
}  
  
class FakeUserRepository: UserRepository {  
    private var users = listOf<User>()  
  
    override fun findUser(userId: String): User? =  
        users.find { it.id == userId }
```

This is a significant inconvenience, but many libraries overcome it by hiding generated classes behind reflection. For example, a popular Mocking library, Mockito, uses annotation processing to create and inject mocks. For that, we use annotations like `Mock` and `InjectMocks` in test suites. Based on these annotations, the Mockito annotation processor generates a file that has a method that creates desired mocks and objects with injected mocks. To make it work, we need to call this method before each test by using Mockito's static `initMocks` method, which finds the appropriate generated class that injects mocks and calls its method. We do not even need to know what this class is called, and our project does not show any errors even before it is built.

```
class MockitoInjectMocksExamples {

    @Mock
    lateinit var emailService: EmailService

    @Mock
    lateinit var smsService: SMSService

    @InjectMocks
    lateinit var notificationSender: NotificationSender

    @BeforeEach
    fun setup() {
        MockitoAnnotations.initMocks(this)
    }

    // ...
}
```

Some other frameworks, like Spring, use a simpler approach. Spring generates a complete backend application based on the annotated elements defined by developers using this framework to define how this application should behave. When we use Spring, we don't need to call generated code because it calls the definitions we've made. We only need to specify

our application such that it uses a Spring class to start this application.

```
@RestController
class WelcomeResource {

    @Value("\${welcome.message}")
    private lateinit var welcomeMessage: String

    @Autowired
    private lateinit var configuration: BasicConfiguration

    @GetMapping("/welcome")
    fun retrieveWelcomeMessage(): String = welcomeMessage

    @RequestMapping("/dynamic-configuration")
    fun dynamicConfiguration(): Map<String, Any?> = mapOf(
        "message" to configuration.message,
        "number" to configuration.number,
        "key" to configuration.isValue,
    )
}
```

We can also define our custom entry point. In such cases, we also use reflection to run generated classes.

```
@SpringBootApplication
open class MyApp {
    companion object {
        @JvmStatic
        fun main(args: Array<String>) {
            SpringApplication.run(MyApp::class.java, *args)
        }
    }
}
```

The process of hiding generated classes behind functions that reference them with reflection is very popular and is used in numerous libraries.

Summary

Annotation processing is a really powerful JVM tool that is used by many Java libraries. It generates files based on annotations used by library users. The idea behind annotation processing is relatively simple, but implementing it might be challenging as we need to operate on element references and implement code generation. Generated elements are only accessible once the processed project is built, which is an inconvenience to annotation processor users. This is why many libraries provide an API with functions that use reflection to reference generated classes at runtime.

From Kotlin's perspective, the biggest Annotation processing limitation is that it works only on Kotlin/JVM, therefore we can't use it on other Kotlin flavors or on multiplatform modules. To get around this, Google created an alternative called Kotlin Symbol Processor.

Exercise: Annotation Processing

Your task is to implement a custom annotation processor. It is your decision what do you want it do, but here are a few good choices:

- * Builder Pattern Generator: Write an annotation processor that automatically generates a Builder class for a given class.
- * DSL Builder Pattern Generator: Write an annotation processor that automatically generates a DSL Builder class for a given class.
- * Immutable Objects Checker: Develop an annotation to mark a class as immutable. The annotation processor should generate one class that starts checks for all classes marked with this annotation.
- * Dependency Injection Framework: Create a mini-framework for dependency injection using annotations.
- * Custom Serialization/Deserialization: Implement an annotation processor that generates code for serializing and deserializing objects to and from a specific format (like JSON, XML, etc.) in a way we do not need to use runtime reflection.

- * Code Metrics and Analysis: Build an annotation processor that calculates and reports various code metrics (like number of methods, lines of code, etc.).
- * Database Access: Write an annotation processor that generates code for database access. It should generate code for creating tables, inserting, updating, and deleting rows, and for querying data.

Kotlin Symbol Processing

As a substitute for Java Annotation Processing, Kotlin Foundation introduced the Kotlin Symbol Processing (KSP⁴²) tool⁴³, which works in a similar way but is faster, more modern, and is designed specifically for Kotlin. It makes it possible to define processors that analyze project code and generate files during project compilation. The key advantage of KSP is that it is native to Kotlin and works on all platforms and shared modules.

Kotlin Symbol Processing is similar in many ways to Java Annotation processing, but it offers many advantages and important improvements, including a more modern API. First of all, KSP understands Kotlin code, so we can properly implement support for Kotlin-specific features, like extensions or suspend functions, and it can also be used to process Java code. It can also freely use KotlinPoet instead of JavaPoet to generate Kotlin code. KSP references to code elements are more convenient to use, and their API is more modern and has more consistent naming. We rarely need to do dangerous down-castings; instead, we can rely on methods and properties. Finally, the KSP API is lazy⁴⁴, which makes it generally faster than Annotation Processing. Compared to kapt, annotation processors that use KSP can run up to twice as fast because KSP is not slowed down by creating intermediate stub classes and Gradle javac tasks⁴⁵.

The same as with Annotation Processing, we will explore KSP by implementing an example project.

⁴²Not to be confused with Kerbal Space Program.

⁴³KSP has been developed since 2021 by Google, which is a Kotlin Foundation member.

⁴⁴For KSP, lazy means the API is designed to initially provide references instead of concrete types. These references are resolved to a code element when actually needed.

⁴⁵According to KSP documentation.

Your first KSP processor

To bring our discussion about KSP into the real world, let's implement the same library as in the previous chapter, but using KSP instead of Java Annotation Processing. So, we will generate an interface for a class that includes all the public methods of this class. This is the code we will use to test our solution:

```
@GenerateInterface("UserRepository")
class MongoUserRepository<T> : UserRepository {

    override suspend fun findUser(userId: String): User? = TODO()

    override suspend fun findUsers(): List<User> = TODO()

    override suspend fun updateUser(user: User) {
        TODO()
    }

    @Throws(DuplicatedUserId::class)
    override suspend fun insertUser(user: User) {
        TODO()
    }
}

class FakeUserRepository : UserRepository {
    private var users = listOf<User>()

    override suspend fun findUser(userId: String): User? = users.find { it.id == userId }

    override suspend fun findUsers(): List<User> = users

    override suspend fun updateUser(user: User) {
        val oldUsers = users.filter { it.id == user.id }
        users = users - oldUsers + user
    }
}
```

```
override suspend fun insertUser(user: User) {
    if (users.any { it.id == user.id }) {
        throw DuplicatedUserId
    }
    users = users + user
}
}
```

We want KSP to generate the following interface:

```
interface UserRepository {
    suspend fun findUser(userId: String): User?

    suspend fun findUsers(): List<User>

    suspend fun updateUser(user: User)

    @Throws(DuplicatedUserId::class)
    suspend fun insertUser(user: User)
}
```

The complete project can be found on GitHub under the name [MarcinMoskala/generateinterface-ksp](#).

To achieve this, in a separate module we need to define the KSP processor, which we'll call `generateinterface-processor`. We also need a module where we will define the annotation, which we'll call `generateinterface-annotations`.

We need to use these modules in the main module configuration⁴⁶. The module with annotation should be attached like any other dependency. To use KSP in Kotlin, we should use the `ksp` plugin. Assuming we use Gradle in our project, this is how we might define our main module dependency in newly created modules.

⁴⁶By “main module”, I mean the module that will use KSP processing.

```
// build.gradle.kts
plugins {
    id("com.google.devtools.ksp")
    // ...
}

dependencies {
    implementation(project(":annotations"))
    ksp(project(":processor"))
    // ...
}
```

If we distribute our solution as a library, we need to publish annotations and processors as separate packages.

Notice that, in the example above, KSP will not be used to test sources. For that, we also need to add the `kspTest` configuration.

```
// build.gradle.kts
plugins {
    id("com.google.devtools.ksp")
}

dependencies {
    implementation(project(":annotations"))
    ksp(project(":processor"))
    kspTest(project(":processor"))
    // ...
}
```

In the `generateinterface-annotations` module, all we need is a file with our annotation definition:

```
package academy.kt

import kotlin.annotation.AnnotationTarget.CLASS

@Target(CLASS)
annotation class GenerateInterface(val name: String)
```

In the `generateinterface-processor` module below, we need to specify two classes. We need to specify the processor provider, a class that implements `SymbolProcessorProvider` and overrides the `create` function, which produces an instance of `SymbolProcessor`. Inside this method, we have access to the environment, which can be used to inject different tools into our processor. Both `SymbolProcessorProvider` and `SymbolProcessor` are represented as interfaces, which makes our processor simpler to unit test.

```
class GenerateInterfaceProcessorProvider
    : SymbolProcessorProvider {

    override fun create(
        environment: SymbolProcessorEnvironment
    ): SymbolProcessor =
        GenerateInterfaceProcessor(
            codeGenerator = environment.codeGenerator,
        )
}
```

The processor provider needs to be specified in a special file called `com.google.devtools.ksp.processing.SymbolProcessorProvider`, under the path `src/main/resources/META-INF/services`. Inside this file, you need to specify the processor provider using its fully qualified name:

```
academy.kt.GenerateInterfaceProcessorProvider
```

Finally, we can implement our processor. The class representing the symbol processor must implement the `SymbolProcessor`

interface and override the single `process` function. The processor does not need to specify annotations or the language versions it supports. The `process` function needs to return a list of annotated elements, which I will explain later in this chapter in the *Multiple round processing* section. In this example, we will return an empty list.

```
class GenerateInterfaceProcessor(
    private val codeGenerator: CodeGenerator,
) : SymbolProcessor {

    override fun process(
        resolver: Resolver
    ): List<KSAnnotated> {
        // ...
        return emptyList()
    }
}
```

The essential part of our processing is generating Kotlin files based on project analysis, for which we'll use the KSP API and `KotlinPoet`. So, to find annotated classes and start interface generation for each of them, we first use a resolver, which is the entrypoint to the KSP API.

```
override fun process(resolver: Resolver): List<KSAnnotated> {
    resolver
        .getSymbolsWithAnnotation(
            GenerateInterface::class.qualifiedName!!
        )
        .filterIsInstance<KSClassDeclaration>()
        .forEach(::generateInterface)

    return emptyList()
}

private fun generateInterface(
    annotatedClass: KSClassDeclaration
```

```
) {  
    // ...  
}
```

In `generateInterface`, we first establish our interface name by finding the interface reference and reading its name.

```
val interfaceName = annotatedClass  
.getAnnotationsByType(GenerateInterface::class)  
.single()  
.name  
  
getAnnotationsByType needs @OptIn(KspExperimental::class)  
annotation to work.
```

We use an annotated class package as our interface package.

```
val interfacePackage = annotatedClass  
.qualifiedName  
?.getQualifier()  
.orEmpty()
```

In this chapter, I won't explain how KSP models Kotlin because it is quite intuitive to those who understand programming nomenclature. It is also quite similar to how Kotlin Reflect models code elements. I could write a long section explaining all the classes, functions, and properties, but this would be useless. Who would want to remember all that? We only need knowledge about an API when we use it, and in this case it is much more convenient to read docs or to look for answers on Google. I believe that books should explain possibilities, potential traps, and ideas that are not easy to deduce.

To find public methods of a class reference, I needed to use the `getDeclaredFunctions` method. The `getAllFunctions` method

is not appropriate because we don't want to include methods from class parents (like `hashCode` and `equals` from `Any`). I also needed to filter out constructors that are considered methods in Kotlinwell.

```
val publicMethods = annotatedClass
    .getDeclaredFunctions()
    .filter { it.isPublic() && !it.isConstructor() }
```

Then I'll use already defined variables to generate the interface and write it to a file using `KotlinPoet`. The `Dependencies` object will be explained later in this chapter, in the *Dependencies and incremental processing* part.

```
val fileSpec = buildInterfaceFile(
    interfacePackage,
    interfaceName,
    publicMethods
)
val dependencies = Dependencies(
    aggregating = false,
    annotatedClass.containingFile!!
)
fileSpec.writeTo(codeGenerator, dependencies)
```

So, now let's implement methods to build our interface. First, we define the `buildInterfaceFile` method for building a file.

```
private fun buildInterfaceFile(
    interfacePackage: String,
    interfaceName: String,
    publicMethods: Sequence<KSFunctionDeclaration>,
): FileSpec = FileSpec
    .builder(interfacePackage, interfaceName)
    .addType(buildInterface(interfaceName, publicMethods))
    .build()
```

When we build an interface, we only need to specify its name and build functions. Note that the parameter representing public methods is a sequence, which is typical of KSP (most elements are lazy for efficiency), so we need to transform it to a list after we transform these methods.

```
private fun buildInterface(
    interfaceName: String,
    publicMethods: Sequence<KSFunctionDeclaration>,
): TypeSpec = TypeSpec
    .interfaceBuilder(interfaceName)
    .addFunctions(
        publicMethods
            .map(::buildInterfaceMethod).toList()
    )
    .build()
```

The method for building methods' definitions is a bit more complicated. To specify the function name, we need to use `getShortName`. We'll separately establish function modifiers, and we also have a separate function to map parameters. We use the same result type and the same annotations, but both need to be mapped to `KotlinPoet` objects.

```
private fun buildInterfaceMethod(
    function: KSFunctionDeclaration,
): FunSpec = FunSpec
    .builder(function.simpleName.getShortName())
    .addModifiers(buildFunctionModifiers(function.modifiers))
    .addParameters(
        function.parameters
            .map(::buildInterfaceMethodParameter)
    )
    .returns(function.returnType!!.toTypeName())
    .addAnnotations(
        function.annotations
            .map { it.toAnnotationSpec() }
            .toList()
```

```
)  
.build()
```

There is no easy way to map the KSP value parameter to the KotlinPoet parameter specification, so we can specify a custom function that builds parameters with the same name, the same type, and the same annotations as the parameter reference.

```
private fun buildInterfaceMethodParameter(  
    variableElement: KSValueParameter,  
) : ParameterSpec = ParameterSpec  
    .builder()  
        variableElement.name!!.getShortName(),  
        variableElement.type.toTypeName(),  
    )  
    .addAnnotations(  
        variableElement.annotations  
            .map { it.toAnnotationSpec() }.toList()  
    )  
.build()
```

Regarding modifiers, mapping them is easy but we need to add the `abstract` modifier, and we should ignore the `override` and `open` parameters as the former is not allowed in interfaces, while the latter is simply redundant in interfaces.

```
private fun buildFunctionModifiers(  
    modifiers: Set<Modifier>  
) = modifiers  
    .filterNot { it in IGNORED_MODIFIERS }  
    .plus(Modifier.ABSTRACT)  
    .mapNotNull { it.toKModifier() }  
  
companion object {  
    val IGNORED_MODIFIERS =  
        listOf(Modifier.OPEN, Modifier.OVERRIDE)  
}
```

The files generated as a result of KSP processing can be found under the path “build/generated/ksp/main/kotlin”. These files will be packed and distributed with the build result (like jar or apk). Since KSP version 1.8.10-1.0.9, generated Kotlin code is treated like a part of our project source set, so generated elements are visible in IntelliJ and can be directly used in our project code.



The screenshot shows the IntelliJ IDEA interface. On the left, the project structure tree displays a folder named 'UserRepository' under 'kotlin [main] sources root'. To the right, a code editor window shows the generated Kotlin code for the UserRepository interface:

```
import ...
public interface UserRepository {
    public fun findUser(userId: String): User?
    public fun findUsers(): List<User>
    public fun updateUser(user: User): Unit
    public fun insertUser(user: User): Unit
}
```

Testing KSP

Thanks to the fact that classes like `CodeGenerator` are represented with interfaces in KSP, we can easily make their fakes and implement unit tests for our processors. However, there are also ways to verify the complete compilation result, together with logged messages and generated files.

Currently, the most popular library for verifying KSP compilation results is `Kotlin Compile Testing`, which is used by many popular libraries, like `Room`, `Dagger` or `Moshi`. We use this library to set up a compilation environment, compile the code, and verify the results of this process. To this compilation environment, we can attach any number of annotation processors, KSP providers, or compiler plugins.

For our example project, I will set up a compilation with our `GenerateInterfaceProcessorProvider` provider and test the source code, compile it, and confirm that it is as expected. This is the function I defined for that:

```
private fun assertGeneratedFile(
    sourceFileName: String,
    @Language("kotlin") source: String,
    generatedResultFileName: String,
    @Language("kotlin") generatedSource: String
) {
    val compilation = KotlinCompilation().apply {
        inheritClassPath = true
        kspWithCompilation = true

        sources = listOf(
            SourceFile.kotlin(sourceFileName, source)
        )
        symbolProcessorProviders = listOf(
            GenerateInterfaceProcessorProvider()
        )
    }
    val result = compilation.compile()
    assertEquals(OK, result.exitCode)

    val generated = File(
        compilation.kspSourcesDir,
        "kotlin/$generatedResultFileName"
    )
    assertEquals(
        generatedSource.trimIndent(),
        generated.readText().trimIndent()
    )
}
```

With such a function, I can easily verify that the code I expect to be generated for a specific annotated class is correct:

```
class GenerateInterfaceProcessorTest {

    @Test
    fun `should generate interface for simple class`() {
        assertGeneratedFile(
            sourceFileName = "RealTestRepository.kt",
            source = """
                import academy.kt.GenerateInterface

                @GenerateInterface("TestRepository")
                class RealTestRepository {
                    fun a(i: Int): String = TODO()
                    private fun b() {}
                }
            """,
            generatedResultFileName = "TestRepository.kt",
            generatedSource = """
                import kotlin.Int
                import kotlin.String

                public interface TestRepository {
                    public fun a(i: Int): String
                }
            """
        )
    }

    // ...
}
```

Instead of reading the result file and comparing its content, we could also load the generated class and analyze it using reflection.

Kotlin's Compile Testing library is also used to verify that code that incorrectly uses annotations fails with a specific message.

```
class GenerateInterfaceProcessorTest {
    // ...

    @Test
    fun `should fail when incorrect name`() {
        assertFailsWithMessage(
            sourceFileName = "RealTestRepository.kt",
            source = """
                import academy.kt.GenerateInterface

                @GenerateInterface("")
                class RealTestRepository {
                    fun a(i: Int): String = TODO()
                    private fun b() {}
                }
                """,
            message = "Interface name cannot be empty"
        )
    }

    // ...
}
```

Dependencies and incremental processing

File generation takes time, so various mechanisms have been introduced to improve the efficiency of this process. A very important one is incremental processing, which is based on a simple idea. When our project is compiled for the first time, it should process elements from all files. If we compile it again, it should only process elements from files that have changed.

So, in our `GenerateInterface` example, consider that you have class `A` in file `A.kt` and class `B` in file `B.kt`, both annotated with `GenerateInterface`.

```
// A.kt
@GenerateInterface("IA")
class A {
    fun a()
}

// B.kt
@GenerateInterface("IB")
class B {
    fun b()
}
```

Incremental processing is enabled by default, so when you compile your project for the first time, `GenerateInterfaceProcessor` will generate both `IA.kt` and `IB.kt`. When you compile your project again without making any changes in `A.kt` or `B.kt`, `GenerateInterfaceProcessor` will not generate any files. If you make a small change in `A.kt`, like adding a space, and you compile your project again, only `IA.kt` will be generated again and will replace the previous `IA.kt`.

Incremental processing is a very powerful mechanism that works nearly effortlessly, but we should understand it if we want to avoid mistakes that could make our libraries work incorrectly.

To know which files need to be reprocessed, KSP introduced the concept of **dirtiness**. A dirty file is a file that needs to be reprocessed. When a processor is started, the resolver methods `getSymbolsWithAnnotation` and `getAllFiles` will only return files and elements from files that are considered dirty.

- `Resolver::getAllFiles` - returns only dirty file references.
- `Resolver::getSymbolsWithAnnotation` - returns only symbols from dirty files.

Most other resolver methods, like `getDeclarationsFromPackage` or `getClassDeclarationByName`, will still return all files; however, when we determine which symbols to process, we typically base them on `getAllFiles` and `getSymbolsWithAnnotation`.

So now, how does a file become dirty, and how does it become clean? The situation is simple in our project: for each input file, we generate one output file. So, when the input file is changed, it becomes dirty. When the corresponding output file is generated for it, the input file becomes clean. However, things can get much more complex. We can generate multiple output files from one input file, or multiple input files might be used to generate one output file, or one output file might depend on other output files.

Consider a situation where a generated file is based not only on the annotated element but also on its parent. So, if this parent changes, the file should be reprocessed.

```
// A.kt
@GenerateInterface
open class A {
    // ...
}

// B.kt
class B : A() {
    // ...
}
```

To determine which sources need to be reprocessed, KSP needs help from processors to identify which input sources correspond to which generated outputs. More concretely, whenever a new file is created, we must specify dependencies. For this, we use the `Dependencies` class, which lets us specify the `aggregating` parameter and then any number of file dependencies. In our example processor, we specified that the generated file depends only on the file containing the annotated class used to generate this file.

```

val dependencies = Dependencies(
    aggregating = false,
    annotatedClass.containingFile!!
)
val file = codeGenerator.createNewFile(
    dependencies,
    packageName,
    fileName
)

```

If we want to make this file depend on other files as well, such as the files containing the parents of the annotated class, we would need to define them explicitly. The `Dependencies` class allows vararg arguments of type `KsFile`.

```

fun classWithParents(
    classDeclaration: KSClassDeclaration
): List<KSClassDeclaration> =
    classDeclaration.superTypes
        .map { it.resolve().declaration }
        .filterIsInstance<KSClassDeclaration>()
        .flatMap { classWithParents(it) }
        .toListplus(classDeclaration)

val dependencies = Dependencies(
    aggregating = ann.dependsOn.isNotEmpty(),
    *classWithParents(annotatedClass)
        .mapNotNull { it.containingFile }
        .toTypedArray()
)

```

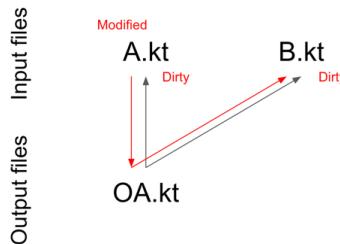
File dependencies are used to:

- Determine that a file not associated with any existing file should be removed.
- Determine which files should be considered dirty.

By rule, if any input file of an output file becomes dirty, all the other dependencies of this output file become dirty too. This relationship is transitive. Consider the following scenario:

If the output file OA.kt depends on A.kt and B.kt, then:

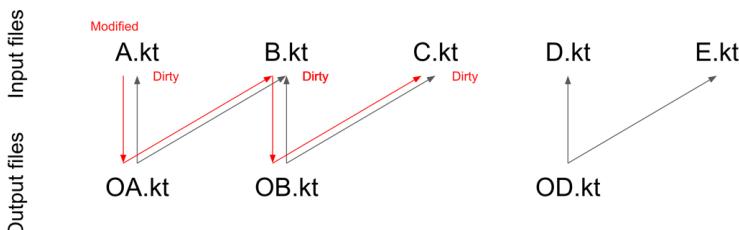
- A change in A.kt makes A.kt and B.kt dirty.
- A change in B.kt makes B.kt and A.kt dirty.



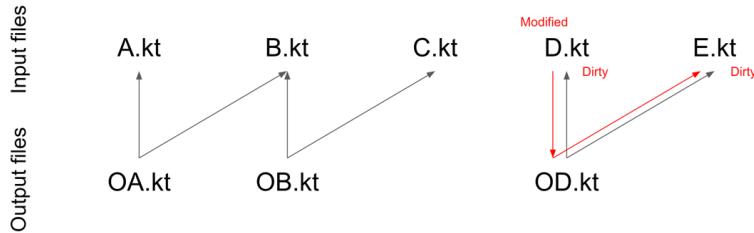
If we change A.kt , then both A.kt and B.kt become dirty.

If OA.kt depends on A.kt and B.kt, and OB.kt depends on B.kt and C.kt, and OD.kt depends on D.kt and E.kt, then:

- A change in A.kt makes A.kt, B.kt, and C.kt dirty.
- A change in B.kt makes A.kt, B.kt, and C.kt dirty.
- A change in C.kt makes A.kt, B.kt, and C.kt dirty.
- A change in D.kt makes D.kt and E.kt dirty.
- A change in E.kt makes D.kt and E.kt dirty.



A change in A.kt makes A.kt, B.kt, and C.kt dirty.

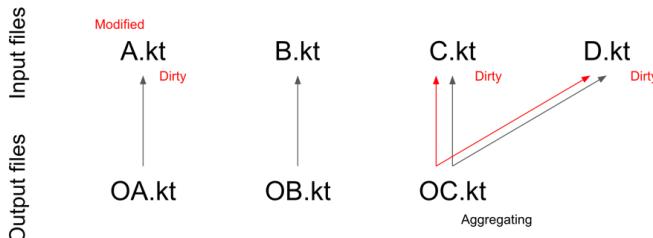


A change in D.kt makes D.kt and E.kt dirty.

As you can see, making dependencies leads to more files being considered dirty; as a result, more files are reprocessed. However, consider files that use all other processing results or collect all annotated elements. Such files are called **aggregating** and set the `aggregating` flag to `true` in their dependency. An aggregating output can potentially be affected by any input changes. All input files that are associated with aggregating outputs will be reprocessed. Consider the following scenario:

If OA.kt depends on A.kt, and OB.kt depends on B.kt, and OC.kt is aggregating and depends on C.kt and D.kt, then:

- A change in A.kt makes A.kt, C.kt, and D.kt dirty.
- A change in B.kt makes B.kt, C.kt, and D.kt dirty.
- A change in C.kt makes C.kt and D.kt dirty.
- A change in D.kt makes C.kt and D.kt dirty.



A change in A.kt makes A.kt, C.kt and D.kt dirty.

It might seem complicated, but using incremental processing is quite simple. In most cases, we set `aggregating` to `false` (so

we set this file as isolating), and we depend on the file used to generate this output file, which is typically the file that includes this annotated element.

```
val dependencies = Dependencies(  
    aggregating = false,  
    annotatedClass.containingFile!!  
)  
val file = codeGenerator.createNewFile(  
    dependencies,  
    packageName,  
    fileName  
)
```

If we base our file generation on other files, we should also list them as dependencies. Files that depend on multiple other files should be set as aggregating, but remember that dependencies of aggregating files become dirty when any file changes.

Multiple rounds processing

KSP supports multiple rounds processing mode, which means that the same process function can be called multiple times if needed. For instance, let's say that you implement a simple Dependency Injection Framework and you use KSP to generate classes that create objects. Consider that you have the following classes:

```
@Single  
class UserRepository {  
    // ...  
}  
  
@Provide  
class UserService(  
    val userRepository: UserRepository  
) {
```

```
// ...
}
```

You want your KSP processor to generate the following providers:

```
class UserRepositoryProvider :  
    SingleProvider<UserRepository>() {  
  
    private val instance = UserRepository()  
  
    override fun single(): UserRepository = instance  
}  
  
class UserServiceProvider : Provider<UserService>() {  
    private val userRepositoryProvider =  
        UserRepositoryProvider()  
  
    override fun provide(): UserService =  
        UserService(userRepositoryProvider.single())  
}
```

The problem is that `UserServiceProvider` depends on `UserRepositoryProvider`, which is also created using our processor. To generate `UserServiceProvider`, we need to reference the class generated for `UserRepositoryProvider`; for that, we need to generate `UserRepositoryProvider` in the first round, and `UserServiceProvider` in the second round. How do we achieve that? From `process`, we need to return a list of annotated elements that could not be generated but that we want to try to generate in the next round. In the next round, `getSymbolsWithAnnotation` from `Resolver` will only return elements that were not generated in the previous round. This way, we can have multiple rounds of processing to resolve deferred symbols.

```
class ProviderGenerator(  
    private val codeGenerator: CodeGenerator,  
) : SymbolProcessor {  
  
    override fun process(  
        resolver: Resolver  
    ): List<KSAnnotated> {  
        val provideSymbols = resolver  
            .getSymbolsWithAnnotation(  
                Provide::class.qualifiedName!!  
            )  
        val singleSymbols = resolver  
            .getSymbolsWithAnnotation(  
                Single::class.qualifiedName!!  
            )  
        val symbols = (singleSymbols + provideSymbols)  
            .filterIsInstance<KSClassDeclaration>()  
  
        val notProcessed = symbols  
            .filterNot(::generateProvider)  
  
        return notProcessed.toList()  
    }  
  
    // ...  
}
```

In our example, we cannot generate both classes during the first round because `UserRepositoryProvider` is not available then. Instead, we should first generate `UserServiceProvider` and then `UserRepositoryProvider` in the second round. Multiple rounds of processing are useful when processor execution depends on elements that might be generated by this or another processor.

A simplified example of this use-case is presented on my GitHub repository [MarcinMoskala/DependencyInjection-KSP](#).

Using KSP on multiplatform projects

We can use KSP on multiplatform projects, but the build configuration is different. Instead of using `ksp` in the `dependencies` block for a specific compilation target, we define a special `dependencies` block at the top-level. Inside it, we specify which targets we should use a specific processor for.

```
plugins {
    kotlin("multiplatform")
    id("com.google.devtools.ksp")
}

kotlin {
    jvm {
        withJava()
    }
    linuxX64() {
        binaries {
            executable()
        }
    }
    sourceSets {
        val commonMain by getting
        val linuxX64Main by getting
        val linuxX64Test by getting
    }
}

dependencies {
    add("kspCommonMainMetadata", project(":test-processor"))
    add("kspJvm", project(":test-processor"))
    add("kspJvmTest", project(":test-processor"))
    // Doing nothing, because there's no such test source set
    add("kspLinuxX64Test", project(":test-processor"))
    // kspLinuxX64 source set will not be processed
}
```

Summary

KSP is a powerful tool that programming libraries can use to generate code based on annotations or our own definitions. This capability is already used by many libraries that make our life easier, like Room or Dagger. KSP is faster than Java Annotation Processing, it understands Kotlin better, and it can be used on multiplatform projects. You will find this knowledge useful to implement some amazing libraries, or it will help you better understand and work with the libraries you use.

The biggest limitation of KSP is that it can only generate files and cannot change how existing code behaves, but in the next chapter, you will learn about Kotlin's Compiler Plugin, which can change how your code behaves.

Exercise: Kotlin Symbol Processing

Your task is to implement a custom KSP processor. It is your decision what do you want it do, but here are a few good choices:

- * Coroutine Wrapper Generator: Write an annotation processor that automatically generates a wrapper class for a given class, that transforms all suspending methods to blocking methods, or to callback methods, or to RxJava methods, or to methods returning `Promise`, or vice versa.
- * Kotlin/JS Wrapper Generator: Write an annotation processor that automatically generates a wrapper class for a given class, that is exported in JS, and that exposes objects that can be used in TypeScript. For instance, that transforms `List<T>` to `Array<T>`, or `Flow` to a special `Flow` class that can be used in TypeScript.
- * Builder Pattern Generator: Write an annotation processor that automatically generates a Builder class for a given class.
- * DSL Builder Pattern Generator: Write an annotation processor that automatically generates a DSL Builder class for a given class.
- * Immutable Objects Checker: Develop an annotation to

mark a class as immutable. The annotation processor should generate one class that starts checks for all classes marked with this annotation.

- * Dependency Injection Framework: Create a mini-framework for dependency injection using annotations.
- * Custom Serialization/Deserialization: Implement an annotation processor that generates code for serializing and deserializing objects to and from a specific format (like JSON, XML, etc.) in a way we do not need to use runtime reflection.
- * Code Metrics and Analysis: Build an annotation processor that calculates and reports various code metrics (like number of methods, lines of code, etc.).
- * Database Access: Write an annotation processor that generates code for database access. It should generate code for creating tables, inserting, updating, and deleting rows, and for querying data.

You can use the following project as a starting point:

<https://github.com/MarcinMoskala/ksp-template>

As a very simple inspiration of how KSP project can be implemented, you can look at the following projects:

- <https://github.com/MarcinMoskala/generateinterface-ksp> - generates interfaces for classes
- <https://github.com/MarcinMoskala/DependencyInjection-KSP> - generates class for simple dependency injection

Kotlin Compiler Plugins

The Kotlin Compiler is a program that compiles Kotlin code but is also used by the IDE to provide analytics for code completion, warnings, and much more. Like many programs, the Kotlin Compiler can use plugins that change its behavior. We define a Kotlin Compiler plugin by extending a special class, called an extension, and then register it using a registrar. Each extension is called by the compiler in a certain phase of its work, thereby potentially changing the result of this phase. For example, you can register a plugin that will be called when the compiler generates supertypes for a class, thus adding additional supertypes to the result. When we write a compiler plugin, we are limited to what the supported extensions allow us to do. We will discuss the currently available extensions soon, but let's start with some essential knowledge about how the compiler works.

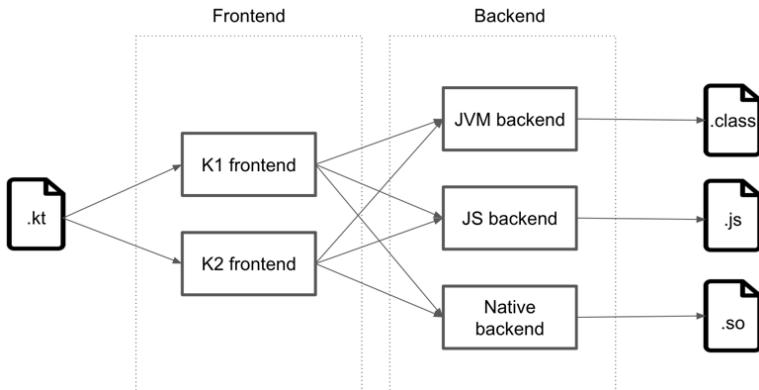
Compiler frontend and backend

Kotlin is a multiplatform language, which means the same code can be used to generate low-level code for different platforms. It is reasonable that Kotlin Compiler is divided into two big parts:

- Frontend, responsible for parsing and transforming Kotlin code into a representation that can be interpreted by the backend and used for Kotlin code analysis.
- Backend, responsible for generating actual low-level code based on the representation received from the frontend.

The compiler frontend is independent of the target, and its results can be reused when we compile a multiplatform module. However, there is a revolution going on at the moment because a new K2 frontend is replacing the older K1 frontend.

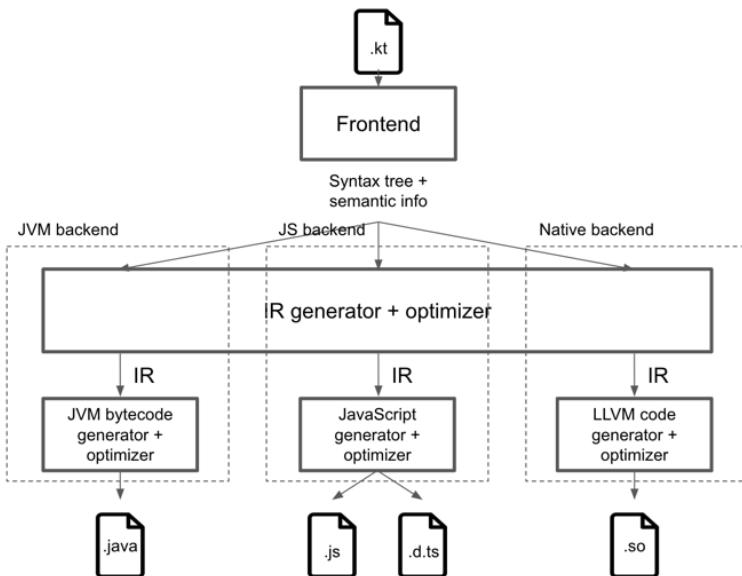
The compiler backend is specific to your compilation target, so there is a separate backend for JVM, JS, and Native, and WASM. They have some shared parts, but they are essentially different.



Compiler frontend is responsible for parsing and analyzing Kotlin code and transforming it into a representation that is sent to the backend, on the basis of which the backend generates platform-specific files. The frontend is target-independent, but there are two frontends: older K1, and newer K2. The backend is target-specific.

When you use Kotlin in an IDE like IntelliJ, the IDE shows you warnings, errors, component usages, code completions, etc., but IntelliJ itself doesn't analyze Kotlin: all these features are based on communication with the Kotlin Compiler, which has a special API for IDEs, and the frontend is responsible for this communication.

Each backend variant shares a part that generates Kotlin intermediate representation from the representation provided by the frontend (in the case of K2, it is FIR, which means frontend intermediate representation). Platform-specific files are generated based on this representation.



Each backend shares a part that transforms the representation provided by the frontend into Kotlin intermediate representation, which is used to generate target-specific files.

You can find detailed descriptions of how the compiler frontend and the compiler backend work in many presentations and articles, like those by Amanda Hinchman-Dominguez or Mikhail Glukhikh. I won't go into detail here because we've already covered everything we need in order to talk about compiler plugins.

Compiler extensions

Kotlin Compiler extensions are also divided into those for the frontend or the backend. All the frontend extensions start with the `Fir` prefix and end with the `Extension` suffix. Here is the complete list of the currently supported K2 extensions⁴⁷:

⁴⁷K1 extensions are deprecated, so I will just skip them.

- `FirStatusTransformerExtension` - called when an element status (visibility, modifiers, etc.) is established and allows it to be changed. The All-open compiler plugin uses it to make all classes with appropriate annotations open by default (e.g., used by Spring Framework).
- `FirDeclarationGenerationExtension` - can specify additional declarations to be generated for a Kotlin file. Its different methods are called at different phases of compilation and allow the generation of different kinds of elements, like classes or methods. Used by many plugins, including the Kotlin Serialization plugin, to generate serialization methods.
- `FirAdditionalCheckersExtension` - allows the specification of additional checkers that will be called when the compiler checks the code; it can also report additional errors or warnings that can be visualized by IntelliJ.
- `FirSupertypeGenerationExtension` - called when the compiler generates supertypes for a class and allows additional supertypes to be added. For instance, if the class `A` inherits from `B` and implements `C`, and the extension decides it should also have supertypes `D` and `F`, then the compiler will consider `A` to have supertypes `B`, `C`, `D` and `F`. Used by many plugins, including the Kotlin Serialization plugin, which uses it to make all classes annotated with the `Serializer` annotation have an implicit `KSerializer` supertype with appropriate type arguments.
- `FirTypeAttributeExtension` - allows an attribute to be added to a type based on an annotation or determines an annotation based on an attribute. Used by the experimental Kotlin Assignment plugin, which allows a number type to be annotated as either positive or negative and then uses this information to throw an error if this contract is broken. Works with the code of libraries used by our project.
- `FirExpressionResolutionExtension` - can be used to add an implicit extension receiver when a function is called. Used by the experimental Kotlin Assignment plugin, which injects `Algebra<T>` as an implicit receiver if `injectAlgebra<T>()` is called.

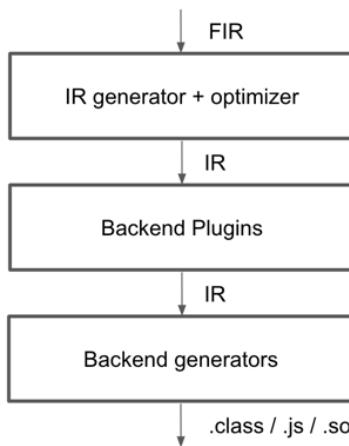
- `FirSamConversionTransformerExtension` - called when the compiler converts a Java SAM interface to a Kotlin function type and allows the result type to be changed. Used by the `SAM-with-receiver` compiler plugin to generate a function type with a receiver instead of a regular function type for SAM interfaces with appropriate annotation.
- `FirAssignExpressionAltererExtension` - allows a variable assignment to be transformed into any kind of statement. Used by the experimental Kotlin Assignment plugin, which allows the assignment operator to be overloaded.
- `FirFunctionTypeKindExtension` - allows additional function types to be registered. Works with the code of libraries used by our project.
- `FirDeclarationsForMetadataProviderExtension` - currently allows additional declarations to be added in Kotlin metadata. Used by the Kotlin Serialization plugin to generate a deserialization constructor or a method to write itself. Its behavior might change in the future.
- `FirScriptConfiguratorExtension` - currently called when the compiler processes a script; it also allows the script configuration to be changed. Its behavior might change in the future.
- `FirExtensionSessionComponent` - currently allows additional extension session components to be added for a session. In other words, it allows a component to be registered so that it can be reused by different extensions. Used by many plugins. For instance, the Kotlin Serialization plugin uses it to register a component that keeps a cache of serializers in a file or `KClass` first from file annotation. Its behavior might change in the future.

Beware! In this chapter we only discuss K2 frontend extensions because the K1 frontend is deprecated and will be removed in the future. However, the K2 compiler frontend is currently not used by default. To use it, you need to have

at least Kotlin version 1.9.0-Beta and add the `-Pkotlin.experimental.tryK2=true` compiler option.

As you can see, these plugins allow us to apply changes to compilation and analysis. They can be used to show a warning or break compilation with an error. They can also be used to change the visibility of specific elements, thus influencing the behavior of the resulting code and suggestions in IDE.

Regarding the backend, there is only one extension: `IrGenerationExtension`. It is used after IR (Kotlin intermediate representation) is generated from the FIR (frontend intermediate representation) but before it is used to generate platform-specific files. `IrGenerationExtension` is used to modify the IR tree. This means that `IrGenerationExtension` can change absolutely anything in the generated code, but using it is hard as we can easily introduce breaking changes, so it must be used with great care. Also, `IrGenerationExtension` cannot influence code analysis, so it cannot impact IDE suggestions, warnings, etc.



Backend plugin extensions are used after IR (Kotlin Intermediate Representation) is generated from the FIR (frontend intermediate representation), but before it is used to generate platform-specific files.

I want to make it clear that the backend cannot influence IDE analysis. If you use `IrGenerationExtension` to add a method to a class, you won't be able to call it directly in IntelliJ because it won't recognize such a method, so you will only be able to call it using reflection. In contrast, a method added to a class using the frontend `FirDeclarationGenerationExtension` can be used directly because the IDE knows about its existence.

The majority of popular Kotlin plugins require multiple extensions, both frontend and backend. For instance, Kotlin Serialization uses backend extensions to generate all the functions for serialization and deserialization; on the other hand, it uses frontend extensions to add implicit supertypes, checks and declarations.

This is the essential knowledge about Kotlin Compiler plugins. To make it a bit more practical, let's take a look at a couple of examples.

Popular compiler plugins

Many compiler plugins and libraries that use compiler plugins are already available. The most popular ones are:

- **Kotlin Serialization** - a plugin that generates serialization methods for Kotlin classes. It's multiplatform and very efficient because it uses a compiler plugin instead of reflection.
- **Jetpack Compose** - a popular UI framework that uses a compiler plugin to support its view element definitions. All the composable functions are transformed into a special representation that is then used by the framework to generate the UI.
- **Arrow Meta** - a powerful plugin introducing support for features known from functional programming languages, like optics or refined types. It also supports Aspect Oriented Programming.
- **Parcelize** - a plugin that generates `Parcelable` implementations for Kotlin classes. It uses a compiler plugin to add appropriate methods to existing classes.

- All-open - a plugin that makes all classes with appropriate annotations open by default. The Spring Framework uses it to make all classes with `@Component` annotation open by default (to be able to create proxies for them).

The majority of plugins use more than one extension, so let's consider the simple `Parcelize` plugin, which uses only the following extensions:

- `IrGenerationExtension` to generate functions and properties that are used under the hood.
- `FirDeclarationGenerationExtension` to generate the functions required for the project to compile.
- `FirAdditionalCheckersExtension` to show errors and warnings.

Kotlin compiler plugins are defined in `build.gradle(.kts)` in the `plugins` section:

```
plugins {  
    id("kotlin-parcelize")  
}
```

Some plugins are distributed as part of individual Gradle plugins.

Making all classes open

We'll start our journey with a simple task: make all classes open. This behavior is inspired by the `AllOpen` plugin, which opens all classes annotated with one of the specified annotations. However, our example will be simpler as we will just open all classes.

As a dependency, we only need `kotlin-compiler-embeddable` that offers us the classes we can use for defining plugins.

Just like in KSP or Annotation Processing, we need to add a file to `resources/META-INF/services` with the

registrar's name. The name of this file should be `org.jetbrains.kotlin.compiler.plugin.CompilerPluginRegistrar`, which is the fully qualified name of the `CompilerPluginRegistrar` class. Inside it, you should place the fully qualified name of your registrar class. In our case, this will be `com.marcinmoskala.AllOpenComponentRegistrar`.

```
// org.jetbrains.kotlin.compiler.plugin.  
// CompilerPluginRegistrar  
com.marcinmoskala.AllOpenComponentRegistrar
```

Our `AllOpenComponentRegistrar` registrar needs to register an extension registrar (we'll call it `FirAllOpenExtensionRegistrar`), which registers our extension. Note that the registrar has access to the configuration so that we can pass parameters to our plugin, but we don't need this configuration now. Our extension is just a class that extends `FirStatusTransformerExtension`; it has two methods: `needTransformStatus` and `transformStatus`. The former determines whether the transformation should be applied; the latter applies it. In our case, we apply our extension to all classes, and we change their status to open, regardless of what this status was before.

```
@file:OptIn(ExperimentalCompilerApi::class)  
  
class AllOpenComponentRegistrar : CompilerPluginRegistrar() {  
    override fun ExtensionStorage.registerExtensions(  
        configuration: CompilerConfiguration  
    ) {  
        FirExtensionRegistrarAdapter  
            .registerExtension(FirAllOpenExtensionRegistrar())  
    }  
  
    override val supportsK2: Boolean  
        get() = true  
}  
  
class FirAllOpenExtensionRegistrar : FirExtensionRegistrar(){
```

```
override fun ExtensionRegistrarContext.configurePlugin() {
    +::FirAllOpenStatusTransformer
}

class FirAllOpenStatusTransformer(
    session: FirSession
) : FirStatusTransformerExtension(session) {
    override fun needTransformStatus(
        declaration: FirDeclaration
    ): Boolean = declaration is FirRegularClass

    override fun transformStatus(
        status: FirDeclarationStatus,
        declaration: FirDeclaration
    ): FirDeclarationStatus =
        status.transform(modality = Modality.OPEN)
}
```

This is just a simplified version, but the actual AllOpen plugin is slightly more complicated as it only opens classes that are annotated with one of the specified annotations. For that, `FirAllOpenExtensionRegistrar` registers a plugin that is used by `FirAllOpenStatusTransformer` to determine if a specific class should be opened or not. If you are interested in the details, see the AllOpen plugin in the `plugins` folder in the Kotlin repository.

Changing a type

Our following example will be the SAM-with-receiver compiler plugin, which changes the type of function types generated from SAM interfaces with appropriate annotations to function types with a receiver. It uses the `FirSamConversionTransformerExtension`, which is quite specific to this plugin because it is only called when a SAM interface is converted to a function type, and it allows the type that will be generated to be changed. This example is

interesting because it adds a type that will be recognized by the IDE and can be used directly in code. The complete implementation can be found in the Kotlin repository in the `plugins/sam-with-receiver` folder, but here I only want to show a simplified implementation of this extension:

```
class FirScriptSamWithReceiverConventionTransformer(
    session: FirSession
) : FirSamConversionTransformerExtension(session) {
    override fun getCustomFunctionTypeForSamConversion(
        function: FirSimpleFunction
    ): ConeLookupTagBasedType? {
        val containingClassSymbol = function
            .containingClassLookupTag()
            ?.toFirRegularClassSymbol(session)
        ?: return null

        return if (shouldTransform(it)) {
            val parameterTypes = function.valueParameters
                .map { it.returnTypeRef.coneType }
            if (parameterTypes.isEmpty()) return null
            createFunctionType(
                getFunctionType(it),
                parameters = parameterTypes.drop(1),
                receiverType = parameterTypes[0],
                rawReturnType = function.returnTypeRef
                    .coneType
            )
        } else null
    }

    // ...
}
```

If the `getCustomFunctionTypeForSamConversion` function doesn't return `null`, it overrides the type that will be generated for a SAM interface. In our case, we determine whether the function should be transformed; if so, we create a function type with a receiver by using the `createFunctionType` function.

There are builder functions that help us to create many elements that are represented in FIR. Examples include `buildSimpleFunction` or `buildRegularClass`, and most of them offer a simple DSL. Here, the `createFunctionType` function creates a function type with a receiver representation of type `ConeLookupTagBasedType`, which will replace automatically generated types from a SAM interface. In essence, this is how this plugin works.

Generate function wrappers

Let's consider the following problem: Kotlin suspend functions can only be called in Kotlin code. This means that if you want to call a suspend function from Java, you can use, for example, `runBlocking` to wrap it in a regular function that calls the suspend function in a coroutine.

```
suspend fun suspendFunction() = ...  
  
fun blockingFunction() = runBlocking { suspendFunction() }
```

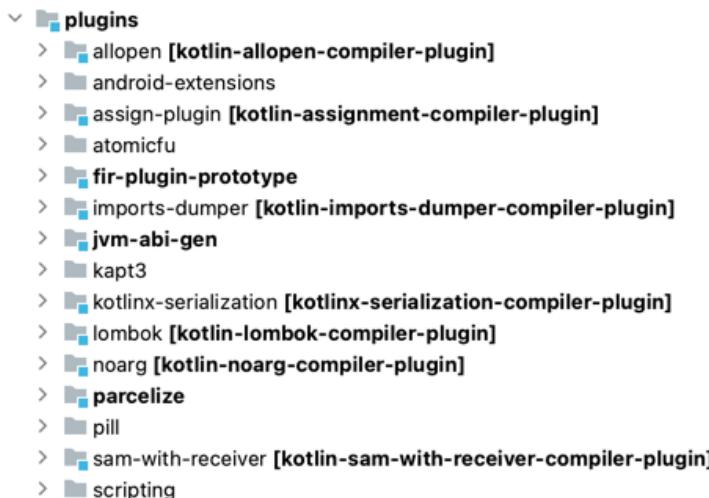
We might use a plugin to generate such wrappers over suspend functions automatically using either a backend or a frontend plugin.

A backend plugin would require an extension for `IrGenerationExtension` that generates an additional wrapper function in IR for the appropriate function. These wrapper functions will be present in the generated platform-specific code and are therefore available for Java, Groovy, and other languages. The problem is that these wrapper classes will not be visible in Kotlin code. This is fine if our wrapper functions are meant to be used from other languages anyway, but we need to know about this serious limitation. There is an open-source plugin called `kotlin-jvm-blocking-bridge` that generates blocking wrappers for suspend functions using a backend plugin; you can find its source code under the link github.com/Him188/kotlin-jvm-blocking-bridge.

A frontend plugin would require an extension for `FirDeclarationGenerationExtension` to generate wrapper functions for the appropriate suspend functions in FIR. These additional functions would then be used to generate IR and finally platform-specific code. Those functions would also be visible in IntelliJ, so we would be able to use them in both Kotlin and Java. However, such a plugin would only work with the K2 compiler, so since Kotlin 2.0. To support the previous language version, we need to define an additional extension that supports K1.

Example plugin implementations

Kotlin Compiler Plugins are currently not documented, and generated elements must respect many restrictions for our code to not break, so defining custom plugins is quite hard. If you want to define your own plugin, my recommendation is to first get the Kotlin Compiler sources and then analyze the existing plugins in the `plugins` folder.



This folder includes not only K2 plugins but also K1 and KSP-based plugins. We are only interested in K2 plugins, so you can

ignore the rest.

A list of all the supported extensions can be found in the `FirExtensionRegistrar` class. To analyze how the compiler uses an extension, you can search for the usage of its open methods. To do this, hit command/Ctrl and click on a method name to jump to its usage. This should show you where the Kotlin Compiler uses this extension. Beware, though, that all the knowledge that is not documented is more likely to change in the future.

Summary

As you can see, the capabilities of Kotlin Compiler Plugins are determined by the extensions supported by the Kotlin Compiler. On the compiler's frontend, these extension capabilities are limited, so there is currently only a specific set of things that can be done on the frontend with Kotlin Compiler Plugins. On the compiler backend, you can change generated IR representation in any way; this offers many possibilities but can also easily cause breaking changes in your code.

Kotlin Compiler Plugins technology is still young, undocumented, and changing. It should be used with great care as it can easily break your code, but it is also extremely powerful and offers possibilities beyond comprehension. Jetpack Compose is a great example. I have only been able to share with you the general idea of how Kotlin Compiler Plugins work and what they can do, but I hope it is enough for you to understand the key concept and possibilities.

In the next chapter, we will talk about another tool that helps with code development: static code analyzers. On the one hand, it is more limited than KSP or Compiler Plugins because it cannot generate any code; on the other hand, static code analyzers are also extremely powerful as they can seriously influence our development process and help us improve our actual code.

Static Code Analysers

This chapter was written by Nicola Corti, who is the best person I could ask to write about Static Code Analysers. He is one of the maintainers of detekt and the author of many of its rules; he is also a well-known speaker and a true programming practitioner.

One could argue that what distinguishes a novice Kotlin developer from a senior one is their mastery of recognizing and appropriately using idiomatic patterns.

Learning how to write a “Hello, world!” or a simple `for` loop takes no longer than a couple of hours, but learning how to use advanced techniques such as generics, delegation, and reflection can take much longer. Some of these patterns require years of experience to be fully mastered. After all, you don’t expect a junior Kotlin engineer to write an annotation processor.

Discovering recurring patterns and applying them has always been a problem in the field of computer science.

That’s why engineers who work on programming language tools, such as compilers, have also developed tools to help engineers identify recurrent patterns: these are called **static analysers**. While you’re already taking a step toward mastering these patterns by reading this book, having a tool that automatically discovers and applies them will take you to the next level.

For example, several of the patterns that are described in the “Effective Kotlin” book can be automated with the help of static analysers.

However, the real power of static analysers lies in the fact that they can be customized, therefore you can discover and enforce your **own** patterns. When working on bigger codebases with hundreds of contributors, you can’t really rely only on code reviews. It’s crucial to have tools that automate

and help you bring consensus and arbitrate conflict between developers. That's why static analysers are tools that should never be missing in the advanced Kotlin developer's toolbox.

In this chapter, we'll walk through the basics of static analysis, and we'll learn how to use one of the most popular static analysers for Kotlin: **detekt**.

What are Static Analysers?

As the name suggests, static analysers are tools that spot bugs and recurrent patterns by analysing your code statically, i.e., without running it⁴⁸.

Running your code is generally considered an expensive operation. Think about Kotlin/JVM, where you need to start a Java Virtual Machine, or Android, where you need an emulator or a physical device to run your code. One technique that can be used to protect your code from bugs is writing tests! They're a great tool to prevent bugs from reaching production, but they require a runtime and are an expensive operation that becomes more and more expensive as your codebase grows.

On the other hand, static analysers look at your code without executing it at all, so there is no need to start a runtime environment at all.

For example, there is no need to execute your code to warn you that you've declared a variable that you're not using. A static analyser can keep track of all the variables you've declared and those you access. The ones you declare but never access

⁴⁸A common example of static analysers is spellcheckers. You probably use one daily when you write documents or emails. Similarly, a compiler also is a form of static analyser as it exposes warnings while it compiles your code.

are most likely⁴⁹ unused and can be removed.

Static analysers can leverage language-specific properties to determine the correctness of your code. For instance, the Kotlin type system can be used to infer types and use type information to do further analysis, similarly to how the Kotlin compiler does. Let's take a look at this simple example:

```
fun getTheAnswer(answer: String = "24"): String {  
    return answer?.reversed()!!  
}
```

This code behaves correctly in the sense that it correctly returns “42” when `getTheAnswer` is invoked. You could write a unit test for it and confirm that the result is as expected, but we could immediately spot a couple of potential problems just by looking at it: as the type of `answer` is a non-nulliable `String`, there is no need to use either the `?.` (safe call) or the `!!` (not-null assert) operator. Instead, the return expression can be simplified to just `return answer.reversed()`⁵⁰.

A static analyser can help you with this and other types of inspection, which could be helpful to enforce a style guide across your codebase (e.g., all unnecessary usages of `import` should be removed). Other types could help prevent potential bugs ending up in your codebase (e.g., when an expression’s return value is ignored and never used).

Static analysers are often referred to as ‘linters’, a term which comes from ‘lint’, the first static analyser for C, developed

⁴⁹Here we oversimplify the unused variable inspection. There are a variety of cases where you need to keep a variable even if it’s not accessed; for instance, `public` definitions in a library module can’t be removed if never accessed. If you’re keen to learn more about this inspection, check the `UnusedPrivateMember` inspection [page](#).

⁵⁰Please consider this simplified snippet as an example. In reality, even if you don’t simplify this snippet, the compiler will do so for you during compilation.

in the 70s⁵¹. Lint itself takes its name from the tiny bits of fabric that appear on your clothes as you wear them. A linter should inspect your codebase and capture all these small imperfections and potential problems.



A real-world linter. Similarly, a static analyser inspects your codebase and captures potential bugs and problems before they hit production. Photo is licensed under Creative Commons - Source <https://www.pexels.com/photo/woman-in-black-long-sleeve-shirt-holding-a-lint-roller-6865186/>

Now, you might think that a static analyser's goal is really similar to another popular process in IT: code reviews.

When you review another engineer's code, you look at it and try to spot potential bugs. I like to think of static analysers and code reviews as complementary, but there is a difference: the former help you spot and automate a lot of recurring coding issues, while the latter are crucial to ensure high code quality and share responsibility for shipped changes. Unfortunately, static analysers can't currently fully understand a developer's intent. You could potentially ship malicious code or "beautifully broken code" that is fully compliant with static analysers as it passes all the checks.

Perhaps, in a not too distant future, advanced AI models could power static analysers to fully automate the code review process. At the time of writing, this is not the case, so code reviews are still considered a fundamental process and are

⁵¹You can read more about the history of the Lint tool on the [Lint \(software\) Wikipedia page](#)

widely adopted in the IT sector.

In academia, static analysers are a vast subject of study and research. Researchers have developed numerous techniques to formally verify that code satisfies requirements and behaves as expected. Formally verifying code behavior can be business-critical in various sectors (think about software in the medical or aerospace industries).

It's outside the scope of this chapter to present all the formal methods of inspecting and verifying your code, but you can find plenty of literature on this. However, to better understand the tools available in the Kotlin ecosystem, we'll walk through some of the various types of static analysers for Kotlin.

Types of analysers

It's hard to make a neat division in the ecosystem of static analysers, as you'll find a plethora of tools doing all sorts of inspection and automation. Here we present some of the most popular families of analysers. Beware that this categorization is not exhaustive, and there are tools that fit in multiple categories.

Formatters

If you've written any code, you've probably already interacted with formatters, which are responsible for making sure your code follows a coding convention⁵². They can help you with simple tasks such as removing unnecessary white spaces and making sure your files have the correct copyright headers.

Most of the time, formatters are embedded inside your IDEs and auto-format your code when you save your files. Formatters are typically referred to as pretty-printers or beautifiers.

Each popular programming language has a set of formatters in its ecosystem. Popular formatters in the industry are tools

⁵²Kotlin has two popular guidelines, [JetBrain's Coding Convention](#) and [Google's Kotlin style guide](#).

like [Prettier](#) for Web, the various `*fmt` tools like [gofmt](#) for Go, and [rustfmt](#) for Rust.

Kotlin has various formatters to pick from:

IntelliJ's built-in formatter

[ktlint](#)

[ktfmt](#)

[diktat](#)

Enforcing coding conventions and reducing *bikeshedding* is a common challenge for large engineering teams.

You probably don't want to manually read every line of code change to evaluate whether it conforms to the common patterns. You may also not want to waste a significant amount of time with your colleagues arguing over whether curly braces should be put on a newline or not.

Formatters increase productivity by allowing you to have such discussions once during the initial configuration and objectively apply your preferred style to every code change.

Code Quality Analysers

Code quality analysers perform more advanced inspections than formatters. Generally, they operate on a tree-like representation of your code called an Abstract Syntax Tree (AST).

With an Abstract Syntax Tree, analysers can inspect specific nodes of the tree to perform inspections. Static analysers generally use the visitor pattern to register inspection on specific nodes of an AST.

Say, for example, you want to prevent usage of the `println` function in your codebase as you have a dedicated logger and want to ensure all logging goes through it.

Your inspector can ask to visit all the nodes in your AST that are of type "function call". On these nodes, the inspector will report a finding whenever the caller function is `println`⁵³.

⁵³If you're curious about this type of inspection, check the `ForbiddenMethodCalls` inspection [source code](#).

An example of an Abstract Syntax Tree: on the left, a simple Kotlin file; on the right, PSI-Viewer shows the PSI representation of your code ([Program Structure Interface](<https://plugins.jetbrains.com/docs/intellij/psi.html>)). The caret is on the `println` function invocation, which corresponds to the `CALL_EXPRESSION` node in the tree.

The amount of information available in an AST might vary depending on different tools and the type of inspection that you're interested in doing. Sometimes, it is sufficient to have only syntactic information in the AST (e.g., the name of the parameters), but for more advanced inspections you might need an AST which has type information linked to it (e.g., the return type of a function invoked from a third-party library).

Popular code quality analysers in the industry are tools like [Checkstyle](#) for Java or [ESlint](#) for JavaScript.

Data-Flow Analysers

Data-flow analysers operate by building a model that can determine at a particular point where values are coming from and what possible values can be produced.

As mentioned before, running your code is an expensive operation.

By looking at the expressions inside your code, a data-flow analyser can infer conditions and execution state without running your code.

Let's take a look at the following code:

```
val answer : String? = maybeGetTheAnswer()
if (answer != null) {
    println("The answer is $answer")
}
```

Reading this code, we know that the type of `answer` inside the if block is smart-casted to `String` and is not a `String?` anymore. The `!= null` check is effectively a type restriction on the set

of possible values of `answer` and removes `null` from that set. Similarly, in a potential `else` branch or after the `if`, the set of possible values would be restricted to just `null`.

Data-flow analysers use the rules of the relevant programming language to compute the set of possible values of your identifier at each statement. With this information, you can perform more advanced analysis. For instance, you can inform the users of types that are too broad and can be restricted (e.g., an analyser could tell you to use `List` instead of `Collection` if the set of values is restricted to `List` only).

Kotlin uses data-flow analysis inside the compiler⁵⁴ to power some of its popular features, such as smart-casting.

Most code quality analysers in the industry also do data-flow analysis but are just marketed as “static analysers”. Typically, they use AST analysis or data-flow analysis based on the type of inspection they need to run.

Code Manipulation

All the previously mentioned tools inspect your code and raise warnings whenever they discover something untoward.

However, some tools can go one step further and **manipulate** your code if they find a violation.

Formatters manipulate your code as they usually provide a mechanism to “reformat” your codebase so that you don’t have to edit the code manually.

Code quality analysers can also manipulate your code, but this isn’t always the case with formatters. For example, if an analyser discovers an unused variable, the manipulation would be to entirely remove the line where this variable is declared.

However, this is often considered an invasive operation as analysers generally need to perform substantial modification

⁵⁴An interesting read is the [Control- and data-flow analysis chapter](#) of the Kotlin language spec.

to make code conform to their rules. Sometimes, removing a line or an annotation is sufficient; other times, there are multiple solutions to a violation but the analyser can't pick one without a human decision.

On top of this, newly created code could raise other violations or might be improperly formatted.

For this reason, static analysers tend to not manipulate code automatically; instead, they raise a violation warning and suggest one or more alternatives to resolve it.

Embedded vs Standalone

Finally, it is worth noting that some static analysers are embedded in tools we use daily, while others are distributed as standalone software.

For instance, the Kotlin Compiler has a static analyser built-in that raises warnings while it compiles code. IntelliJ IDEA also has a built-in static analyser which offers inspections as you type, with code suggestions in the contextual menu.

On the other hand, tools like detekt, ktlint, and others are standalone. You can typically execute them from the command line by specifying your source files. These standalone static analysers offer plugins and integrations for the most popular developer environments (IntelliJ IDEA, Gradle, Visual Studio Code and so on). On top of this, standalone static analysers are generally better suited to be integrated with continuous integration servers or pre-commit hooks.

Kotlin Code Analysers

Let's now walk through the various static analysers available for Kotlin in order to discover what they offer and understand how you can integrate them in your projects.

Kotlin Compiler

As mentioned above, the Kotlin Compiler ships an embedded static analyser, which is already a great starting point for

analyzing your Kotlin code.

Sadly, the Kotlin Compiler offers no easy way to write a custom inspection, so you'll have to resort to writing a custom Kotlin Compiler plugin. Moreover, the compiler doesn't offer an easy way to selectively toggle its inspections, so you'll always get all the warnings the compiler finds, and you won't be able to enable/disable some⁵⁵.

The Kotlin Compiler also doesn't ship with an embedded formatter, so you'll need to resort to other tools for code formatting.

IntelliJ IDEA

Alongside the Kotlin Compiler, JetBrains also ships a static analyser as part of IntelliJ IDEA.

This static analyser is user-friendly as it has a UI to configure it and offers suggestions inline. On the other hand, these inspections require an IntelliJ installation to run, therefore they're less portable and less suitable for continuous integration servers.

IntelliJ IDEA also offers an embedded formatter, making it a great companion for the Kotlin Compiler.

ktlint

[ktlint](#) is a standalone formatter for Kotlin which offers great configurability and allows you to easily reformat your code. Ktlint is well established in the Kotlin community and has been maintained in open-source. It can be extended with custom rules and integrated with Gradle, Maven, IntelliJ, or even Vim via third-party plugins.

ktfmt

[ktfmt](#) is another standalone formatter for Kotlin, but it has a more opinionated take on code-formatting. ktfmt is non-

⁵⁵There is an [open issue](#) on JetBrains Issue Tracker about this specific problem.

configurable and less extensible than ktlint, with the intention to reformat code in a stricter manner. Thanks to third-party plugins, ktfmt can be easily integrated with IntelliJ, Gradle or Maven.

Android Lint

[Android Lint](#) is a static analyser for Android. It offers inspections which are Android specific, so it can inspect not only Java and Kotlin files, but also Gradle files and XML files (and other Android-related files). Android Lint is integrated inside Android Studio and it's probably the best pick for developers looking to integrate static analysis in their Android projects.

The Android Studio integration allows Android Lint to also act as a code manipulator by offering [quick fixes](#): small tooltips on top of the inspection that allow the user to selectively apply the suggested changes.

Android Lint is also extensible with custom detectors and can be executed via Gradle or as a standalone tool, without needing to have Android Studio installed.

detekt

[detekt](#) is a standalone static-analyser for Kotlin. Unlike Android Lint, it's general purpose and focuses only on Kotlin code. It can't inspect Java code, but it's easier to integrate in any Kotlin project.

detekt is community developed and, at the time of writing, offers more than 300 inspections. It primarily offers a Gradle plugin and a Kotlin Compiler plugin to integrate with existing projects. Moreover, detekt can be integrated with IntelliJ IDEA, Maven, Bazel and much more.

One key feature of detekt is its extensibility, which makes it easier to write custom rules to achieve your desired inspections. Let's take a closer look at how you can configure and customize it.

Setting up detekt

Over the rest of this chapter, we'll look at practical examples of how to use detekt inside your projects. We chose detekt as the preferred tool because of its great flexibility, and we will learn how you can write a custom rule for it.

Let's start by configuring detekt as part of your Gradle project. For the sake of brevity, we'll assume you already have a single-module Gradle setup which builds properly.

To set up detekt, you just need to edit your `build.gradle.kts` file as follows:

```
plugins {  
    kotlin("jvm") version "..."  
    // Add this line  
    id("io.gitlab.arturbosch.detekt") version "..."  
}
```

This line adds the Detekt Gradle Plugin to your project and is sufficient to set up detekt in your project with the default configuration.

You can see it in action by invoking the `build` Gradle task from the command-line:

```
$ ./gradlew build  
  
BUILD SUCCESSFUL in 549ms
```

As an alternative, you can also invoke the `detekt` task directly with `./gradlew detekt`

Now, if we try to run detekt on a simple file that contains the following:

```
fun main() {  
    println(42)  
}
```

we will receive the following output:

```
$ ./gradlew build  
  
> Task :example:detekt FAILED  
/tmp/example/src/main/java/Main.kt:2:11:  
    This expression contains a magic number. Consider  
    defining it to a well named constant. [MagicNumber]  
  
[...]  
BUILD FAILED in 470ms
```

Here, detekt is flagging the number 42 as it's considered a "Magic Number", a number without a clear semantic which should be extracted as a well-named constant.

The ID of this violation is inside the square brackets (`MagicNumber`) and is important for several reasons:

- It is the name of the rule that generated this inspection. You can use it to search for documentation about this inspection on the detekt website.
- It can be used to suppress the inspection locally with a `@Suppress("MagicNumber")` annotation just above the offending statement.
- It can be used in the configuration file (see following sections) to configure the rule to adapt detekt to the style of your codebase.

This is just one very simple example of the various inspections that detekt offers, so let's take a closer look at some more of them.

detekt Rules and Rulesets

The 200 inspections that detekt offers out of the box are called **rules**. Remembering all of them is hard, which is why they're organized in **rulesets**, collections of rules that serve the same purposes.

Let's take a brief look at them:

- **comments**: rules that help enforce good documentation of your functions and classes.
- **complexity**: rules that report unnecessary statements or complex code, like methods and interfaces that are larger than usual.
- **coroutines**: rules that report potential problems with Coroutines, such as usage of `GlobalScope` or other antipatterns.
- **empty-blocks**: rules that flag empty blocks and function bodies.
- **exceptions**: rules that inspect how your code throws and catches exceptions.
- **libraries**: rules that help library authors write good APIs.
- **naming**: rules that help you enforce naming conventions across your codebase.
- **performance**: rules that flag potential performance regressions in your code.
- **potential-bugs**: rules that flag code that could lead to potential bugs or crashes.
- **ruleauthors**: rules that help you write good external detekt rules.
- **style**: rules that check the style and formatting of your code and flag unnecessary code.

`potential-bugs` is one of detekt's biggest rulesets and contains some of the most popular rules. An honorable mention goes to `DontDowncastCollectionTypes`, which was inspired by [Effective Kotlin: Item 1 - Limit Mutability](#)⁵⁶

⁵⁶You can find more information on the `DontDowncastCollectionTypes` rule on its dedicated [page](#)

On top of these rulesets, the community has developed a variety of third-party rules that serve specific purposes. For instance, a collection of rules that offer inspections specific to JetPack Compose is available. These and other rules are shared in the [detekt Marketplace](#).

Configuring detekt

By default, detekt comes with a sensible default configuration. For example, not all the rules are enabled, while some rules are configured to suit a wide range of users.

However, you might want to enable/disable or customize some rules. To do so, you need to provide a YAML configuration file that lets you customize detekt as you prefer.

The easiest way to do this is to ask detekt to create one for you by invoking the `detektGenerateConfig` task as follows:

```
$ ./gradlew detektGenerateConfig

> Task :example:detektGenerateConfig
Successfully copied default config to
/tmp/example/config/detekt/detekt.yml

BUILD SUCCESSFUL in 473ms
1 actionable task: 1 executed
```

This will create a config file at the path shown in the console by copying the default detekt config file. The configuration file looks as follows:

```
...
comments:
  active: true
AbsentOrWrongFileLicense:
  active: false
  licenseTemplateFile: 'license.template'
  licenseTemplateIsRegex: false
style:
  active: true
  MagicNumber:
    ignorePropertyDeclaration: true
    ignoreAnnotation: true
    ignoreEnums: true
    ignoreNumbers:
      - '-1'
      - '0'
      - '1'
...
...
```

Rules are grouped by ruleset, and you can toggle each rule via the `active` key. For instance, the `AbsentOrWrongFileLicense` rule is disabled by default as you need to provide a `licenseTemplateFile` to enable it.

Incremental Adoption

When running detekt on a big codebase for the first time, you could be overwhelmed by the number of findings that detekt reports. Fixing them all at once could be unfeasible, so you should probably take an incremental approach to adopting detekt.

You could use the config file to disable some rules. This has the side effect of also turning off the inspection entirely for newer code added to your codebase.

A smarter approach is to use a **baseline**, which is a snapshot of a detekt run that can be used to suppress a group of inspections for future runs of detekt. Using a baseline is a two-step process:

Create the baseline with the `detektBaseline` task. detekt will run and then store all its findings in a baseline file in XML format, with the filename and line of each finding. You can then subsequently run detekt as usual with the `detekt` command. detekt will not report any issues which are listed in the baseline.

Here is an example of how a baseline file looks for a simple project with a couple of findings:

```
<SmellBaseline>
  <ManuallySuppressedIssues/>
  <CurrentIssues>
    <ID>ImplicitUnitReturnType:HelloWorld.kt$Hello$fun
      aFunctionWithImplicitUnitReturnType()</ID>
    </CurrentIssues>
</SmellBaseline>
```

Baselines are also common in other static analysers and are a great tool to incrementally introduce and enable new rules to your codebase. They can also be useful when dealing with legacy code or massive codebases contributed to by multiple developers.

Writing your first detekt Rule

Now that you know the basics of how to use detekt, it's time to learn how to write a custom rule to run your own inspection.

Setting up your rule project

To get you up to speed with new detekt rules, you can use the official template for custom rules: <https://github.com/detekt/detekt-custom-rule-template>

This will scaffold a new project for you with all the files needed to create a new rule. The crucial files to look into are:

`src/main/kotlin/org/example/detekt/MyRule.kt` - the code of your rule. This is where your inspection logic will live.

`src/main/kotlin/org/example/detekt/MyRuleSetProvider.kt` - the code of your ruleset. In order to be used, your rule needs to live inside a ruleset, which allows you to add multiple custom rules and distribute all of them together.

`src/main/resources/config/config.yml` - the default config file for your rule. This is used to offer the default configuration for your rules.

Please note that detekt uses the Java Service Provider API, so the file inside `src/main/resources/META-INF/services` is also needed to properly discover your ruleset. The template also comes with two tests that can help you write your rule.

Coding your rule

When writing custom rules, the best approach to follow is Test-driven Development (TDD).

Let's write a rule that flags usages of `System.out.println()` and suggests replacing them with Kotlin's `println()`. So, the following code will flag the first statement but not the second:

```
fun main() {
    // Non compliant
    System.out.print("Hello")

    // Compliant
    println("World!")
}
```

Let's open the `src/test/kotlin/org/example/detekt/MyRuleTest.kt` and code our intention. A rule test looks as follows:

```
@KotlinCoreEnvironmentTest
internal class MyRuleTest(
    private val env: KotlinCoreEnvironment
) {
    @Test
    fun `reports usages of System.out.println`() {
        val code = """
            fun main() {
                System.out.println("Hello")
            }
        """.trimIndent()

        val findings = MyRule(Config.empty())
            .compileAndLintWithContext(env, code)
        findings shouldHaveSize 1
    }

    @Test
    fun `does not report usages Kotlin's println`() {
        val code = """
            fun main() {
                println("Hello")
            }
        """.trimIndent()

        val findings = MyRule(Config.empty())
            .compileAndLintWithContext(env, code)
        findings shouldHaveSize 0
    }
}
```

This test allows us to follow a declarative approach for our rule and define all of its requirements. Moreover, it is a great source of documentation for our rule, as other developers can immediately see which code triggers this inspection and which should not.

If we try to run these tests, they will both fail. Let's open the `MyRule.kt` and code our rule.

The `MyRule.kt` file created by the template looks like this:

```
class MyRule(config: Config) : Rule(config) {
    override val issue = Issue(
        javaClass.simpleName,
        Severity.CodeSmell,
        "Custom Rule",
        Debt.FIVE_MINS,
    )

    override fun visitClass(klass: KtClass) {
        // ...
    }
}
```

Let's walk through the code. We start by extending an external class provided by detekt: our rule is a class called `MyRule`, which extends detekt's `Rule` class. The `Config` parameter we pass to the constructor is what we can use to access the configuration file.

`Rule` is actually an abstract class and requires us to implement the `issue` property, which is a representation of the issue this rule is currently reporting; it contains the error message and information about the severity of the issue. Each rule can report one or more of these issues.

detekt performs code quality analysis and allows the Abstract Syntax Tree to be visited by overriding one of the various `visit*` methods. In the template, we implement the `visitClass` method that will allow each `class` declaration to be inspected.

As our rule is designed to inspect expressions, we need to

implement the `visitDotQualifiedExpression` instead⁵⁷. To implement our rule, we'll have to implement the method as follows:

```
class MyRule(config: Config) : Rule(config) {  
    override val issue = //...  
  
    override fun visitDotQualifiedExpression(  
        expression: KtDotQualifiedExpression  
    ) {  
        super.visitDotQualifiedExpression(expression)  
        if (expression.text.startsWith("System.out.println")) {  
            report(CodeSmell(  
                issue,  
                Entity.from(expression),  
                "Use Kotlin stdlib's println instead.",  
            ))  
        }  
    }  
}
```

This implementation of `visitDotQualifiedExpression` checks if the expression starts with `System.out.println`; if it does, it reports an issue for it and invokes the `report` function from `detekt`. This also allows you to attach an error message and information on the relevant line and column.

It's worth noting that the first statement of `visit...` is a call to the superclass implementation of this method. This is

⁵⁷For the sake of brevity, we won't explain every API that PSI and `detekt` expose for running inspections, but let's clarify the `visitDotQualifiedExpression` method. A dot-qualified expression is, as the name suggests, an expression which has a dot in it (e.g., a fully qualified method call). We're interested in finding `System.out.println` calls, which are not fully qualified method calls (as they would be `java.lang.System.out.println`) but they have a dot in them, so they are treated as dot-qualified expressions in the AST.

needed to make sure we don't break the visitor pattern and is generally good practice when using inheritance.

If you try to run all tests again, you will see that they pass the tests, thus verifying that our rule implementation is correct.

Once you get more experienced with writing rules, you'll probably notice how using TDD (Test-driven Development) is a great approach. You can declare the snippets of code you wish to be flagged and then code your rule so that all your tests are green.

Using your rule

Now that you've written and tested your rule, the last part is distributing it and letting others use it.

Your rule should be published to a Maven Repository⁵⁸ and consumed like any other dependency in a Gradle project, but you'll be using a `detektPlugin` dependency instead of an `implementation` dependency.

```
plugins {  
    kotlin("jvm") version "..."  
    id("io.gitlab.arturbosch.detekt") version "..."  
}  
  
dependencies {  
    detektPlugin("org.example:detekt-custom-rule:...")  
}
```

Users will then have to activate your rule in their config file:

⁵⁸You don't necessarily need to publish to a remote repository like Maven Central. While testing, you can use Maven Local, which publishes the rule on your local computer, or you can use an internal repository for your organization.

```
...
MyRuleSet:
  MyRule:
    active: true
...
...
```

And they'll start seeing findings when they run detekt normally:

```
$ ./gradlew build

> Task :example:detekt FAILED
/tmp/example/src/main/java/Main.kt:2:11:
  Use Kotlin stdlib's println instead. [MyRule]

[...]
BUILD FAILED in 470ms
```

Rules with type resolution

This is a really simple rule that relies only on syntactic information as we treat code as just text. It's a good start and will cover most use cases, but a more precise way to perform these inspections would be to have a richer Abstract Syntax Tree which has type information. The current implementation does not ensure that the `System.out.println` you're invoking comes from the `java.lang` package and not from another `System` class the user created in their project.

To have access to richer Abstract Syntax trees, we would need to use a more advanced type of rule. These rules rely not on a single file but on the compile-time information for the entire project so that they can retrieve type information computed by the Kotlin compiler for every declaration and identifier. It's outside the scope of this book to deep dive into compiler topics or the PSI API as these are quite extensive and would probably require their own book to cover in detail.

You can find plenty of examples of more advanced rules in the detekt codebase, which can be used as a source of inspiration for your custom rules.

Conclusion

In this chapter, we've discovered what static analysers are and which types of analyses they can perform. We've learned what the differences are between formatters, code quality analysers, data-flow analysers, and other types of analysers.

The Kotlin ecosystem is bursting with tools which offer all sorts of capabilities and can be integrated in the vast majority of projects. Some of these are first-party tools, like the Kotlin Compiler or IntelliJ IDEA, while others are community contributions, like detekt.

We've also learned how you can easily integrate detekt as part of your project, and how you can use the detekt config file and the baseline feature to incrementally adopt it in your projects.

But the real power of static analysers comes from their extensibility, which is why this chapter has shown you how to write your own custom detekt rules.

Ending

Congratulations, you've reached the end of this book. I must say I'm impressed! I was worried that only reviewers and I would reach this point, but, well, you give me hope for the future of the programming community. It's time, though, to do something with the knowledge you've acquired. Maybe you'll write a useful library or an amazing application, or maybe you'll share this knowledge with others. Whatever you do, I wish you good luck and a lot of fun.

If you're thinking about the next book to read, consider *Effective Kotlin*, which explains the best practices in Kotlin. Or you might just take a break from books, you deserve it.

Exercise solutions

Solution: Generic types usage

The failing lines are marked with x:

```
fun takeIntList(list: List<Int>) {}
takeIntList(listOf<Any>())           X
takeIntList(listOf<Nothing>())

fun takeIntMutableList(list: MutableList<Int>) {}
takeIntMutableList(mutableListOf<Any>())      X
takeIntMutableList(mutableListOf<Nothing>())  X

fun takeAnyList(list: List<Any>) {}
takeAnyList(listOf<Int>())
takeAnyList(listOf<Nothing>())

class BoxOut<out T>
fun takeBoxOutInt(box: BoxOut<Int>) {}
takeBoxOutInt(BoxOut<Int>())
takeBoxOutInt(BoxOut<Number>())   X
takeBoxOutInt(BoxOut<Nothing>())

fun takeBoxOutNumber(box: BoxOut<Number>) {}
takeBoxOutNumber(BoxOut<Int>())
takeBoxOutNumber(BoxOut<Number>())
takeBoxOutNumber(BoxOut<Nothing>())

fun takeBoxOutNothing(box: BoxOut<Nothing>) {}
takeBoxOutNothing(BoxOut<Int>())        X
takeBoxOutNothing(BoxOut<Number>())  X
takeBoxOutNothing(BoxOut<Nothing>())

fun takeBoxOutStar(box: BoxOut<*>) {}
takeBoxOutStar(BoxOut<Int>())
takeBoxOutStar(BoxOut<Number>())
takeBoxOutStar(BoxOut<Nothing>())
```

```
class BoxIn<in T>
fun takeBoxInInt(box: BoxIn<Int>) {}
takeBoxInInt(BoxIn<Int>())
takeBoxInInt(BoxIn<Number>())
takeBoxInInt(BoxIn<Nothing>())  X
takeBoxInInt(BoxIn<Any>())
```

Solution: Generic Response

```
sealed class Response<out R, out E>
class Success<out R>(val value: R) : Response<R, Nothing>()
class Failure<out E>(val error: E) : Response<Nothing, E>()
```

Solution: Generic Consumer

```
abstract class Consumer<in T> {
    abstract fun consume(elem: T)
}

class Printer<in T> : Consumer<T>() {
    override fun consume(elem: T) {
        // ...
    }
}

class Sender<in T> : Consumer<T>() {
    override fun consume(elem: T) {
        // ...
    }
}
```

Solution: ApplicationScope

```
class ApplicationScope(
    private val scope: CoroutineScope,
    private val applicationScope: ApplicationControlScope,
    private val loggingScope: LoggingScope,
) : CoroutineScope by scope,
    ApplicationControlScope by applicationScope,
    LoggingScope by loggingScope
```

Solution: Lateinit delegate

We need to define a class `Lateinit` with functions `getValue` and `setValue` that will be called when we get or set the property. We can also make it implement `ReadWriteProperty<Any?, T>` interface. Inside of it, we need to store the value. The naive solution to this problem is using `null` as a marker for value not being initialized. This solution will not work correctly for nullable types.

```
// Solution that does not work correctly for nullable types
class Lateinit<T>: ReadWriteProperty<Any?, T> {
    var value: T? = null

    override fun getValue(
        thisRef: Any?,
        property: KProperty<*>
    ): T =
        value ?: error("... property ${property.name}")

    override fun setValue(
        thisRef: Any?,
        property: KProperty<*>,
        value: T
    ) {
        this.value = value
    }
}
```

You should replace “...” with “Uninitialized lateinit”.

Correct solutions require a different (than `null` value) way to keep information about value not being initialized. It can be another property, a special flag, or an object.

```
class Lateinit<T>: ReadWriteProperty<Any?, T> {
    var value: T? = null
    var isInitialized = false

    override fun getValue(
        thisRef: Any?,
        property: KProperty<*>
    ): T {
        if (!isInitialized) {
            error("... property ${property.name}" )
        }
        return value as T
    }

    override fun setValue(
        thisRef: Any?,
        property: KProperty<*>,
        value: T
    ) {
        this.value = value
        this.isInitialized = true
    }
}
```

This solution is not thread safe. If two threads try to set the value at the same time, it might happen that one thread will overwrite the value set by the other thread. To make it thread safe, we can use `synchronized` block.

```
class Lateinit<T>: ReadWriteProperty<Any?, T> {
    var value: Any? = NOT_INITIALIZED

    override fun getValue(
        thisRef: Any?,
        property: KProperty<*>
    ): T {
        if (value == NOT_INITIALIZED) {
            error("... property ${property.name}" )
        }
        return value as T
    }

    override fun setValue(
        thisRef: Any?,
        property: KProperty<*>,
        value: T
    ) {
        this.value = value
    }

    companion object {
        val NOT_INITIALIZED = Any()
    }
}
```

```
class Lateinit<T> : ReadWriteProperty<Any?, T> {
    var value: ValueHolder<T> = NotInitialized

    override fun getValue(
        thisRef: Any?,
        property: KProperty<*>
    ): T = when (val v = value) {
        NotInitialized ->
            error("... property ${property.name}" )
        is Value -> v.value
    }
```

```
override fun setValue(
    thisRef: Any?,
    property: KProperty<*>,
    value: T
) {
    this.value = Value(value)
}

sealed interface ValueHolder<out T>
class Value<T>(val value: T) : ValueHolder<T>
object NotInitialized : ValueHolder<Nothing>
}
```

Solution: Blog Post Properties

```
data class BlogPost(
    val title: String,
    val content: String,
    val author: Author,
) {
    // Since this property is needed on average more than
    // once per blog post, and it is not expensive to
    // calculate, it is best to define it as a value.
    val authorName: String =
        "${author.name} ${author.surname}"

    // Since this property is needed on average more than
    // once per blog post, and it is expensive to
    // calculate, it is best to define it as a lazy
    val wordCount: Int by lazy {
        content.split("\\s+").size
    }

    // Since this property is needed on average less than
    // once per blog post, and it is not expensive to
    // calculate, it is best to define it as a getter.
    val isLongRead: Boolean
        get() = content.length > 1000
}
```

```
// Since this property is very expensive to calculate,
// it is best to define it as a lazy
val summary: String by lazy {
    generateSummary(content)
}

private fun generateSummary(content: String): String =
    content.take(100) + "..."

data class Author(
    val key: String,
    val name: String,
    val surname: String,
)
```

Solution: Mutable lazy delegate

```
fun <T> mutableLazy(
    initializer: () -> T
): ReadWriteProperty<Any?, T> = MutableLazy(initializer)

private class MutableLazy<T>(
    private var initializer: ((() -> Any?)? = null,
): ReadWriteProperty<Any?, T> {
    private var value: Any? = null

    override fun getValue(
        thisRef: Any?,
        property: KProperty<*>
    ): T {
        if (initializer != null) {
            value = initializer?.invoke()
            initializer = null
        }
        return value as T
    }

    override fun setValue(
```

```
        thisRef: Any?,
        property: KProperty<*>,
        value: T
    ) {
    this.value = value
    this.initializer = null
}
}
```

This solution lacks thread safety. The simplest way to make it thread safe is to use synchronized block. A more efficient solution is to use AtomicReference and compareAndSet function.

Solution: Coroutine time measurement

```
@OptIn(ExperimentalContracts::class)
suspend fun measureCoroutine(
    body: suspend () -> Unit
): Duration {
    contract {
        callsInPlace(body, InvocationKind.EXACTLY_ONCE)
    }
    val dispatcher = coroutineContext[ContinuationInterceptor]
    return if (dispatcher is TestDispatcher) {
        val before = dispatcher.scheduler.currentTimeMillis
        body()
        val after = dispatcher.scheduler.currentTimeMillis
        after - before
    } else {
        measureTimeMillis {
            body()
        }
    }.milliseconds
}
```

Solution: Adjust Kotlin for Java usage

```
@file:JvmName("MoneyUtils")
package advanced.java

import java.math.BigDecimal

data class Money @JvmOverloads constructor(
    val amount: BigDecimal = BigDecimal.ZERO,
    val currency: Currency = Currency.EUR,
) {
    companion object {
        @JvmStatic
        fun eur(amount: String) =
            Money(BigDecimal(amount), Currency.EUR)

        @JvmStatic
        fun usd(amount: String) =
            Money(BigDecimal(amount), Currency.USD)

        @JvmField
        val ZERO_EUR = eur("0.00")
    }
}

@JvmName("sumMoney")
fun List<Money>.sum(): Money? {
    if (isEmpty()) return null
    val currency = this.map { it.currency }.toSet().single()
    return Money(
        amount = sumOf { it.amount },
        currency = currency
    )
}

operator fun Money.plus(other: Money): Money {
    require(currency == other.currency)
    return Money(amount + other.amount, currency)
}
```

```
enum class Currency {  
    EUR, USD  
}
```

Solution: Multiplatform LocalDateTime

Kotlin/JVM code:

```
import java.time.LocalDateTime as JavaLocalDateTime  
  
actual typealias LocalDateTime = java.time.LocalDateTime  
  
actual fun now(): LocalDateTime = JavaLocalDateTime.now()  
  
actual fun parseLocalDateTime(str: String): LocalDateTime =  
    JavaLocalDateTime.parse(str)
```

Kotlin/JS code:

```
import kotlin.js.Date  
  
actual class LocalDateTime(  
    val date: Date = Date(),  
) {  
    actual fun getSecond(): Int = date.getSeconds()  
  
    actual fun getMinute(): Int = date.getMinutes()  
  
    actual fun getHour(): Int = date.getHours()  
  
    actual fun plusSeconds(seconds: Long): LocalDateTime =  
        LocalDateTime(Date(date.getTime() + seconds * 1000))  
    }  
  
actual fun now(): LocalDateTime = LocalDateTime()  
  
actual fun parseLocalDateTime(str: String): LocalDateTime =  
    LocalDateTime(Date(Date.parse(str)))
```

Solution: Migrating a Kotlin/JVM project to KMP

First, you need to transform `build.gralde.kts` configuration to Kotlin Multiplatform. You need to change the plugin “`kotlin(“jvm”)`” to “`kotlin(“multiplatform”)`”. Then you need to configure targets. This is what this configuration might look like:

```
plugins {
    kotlin("multiplatform") version "1.8.0"
    application
}

group = "org.example"
version = "1.0-SNAPSHOT"

repositories {
    mavenCentral()
}

kotlin {
    jvm {
        withJava()
    }
    js(IR) {
        moduleName = "sudoku-generator"
        browser()
        binaries.library()
    }
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation(
                    "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.4")
            }
        }
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test"))
            }
        }
    }
}
```

```
        }
    }
    val jvmMain by getting
    val jvmTest by getting
    val jsMain by getting
    val jsTest by getting
}
}
```

Now move all your code from `src/main/kotlin` to `src/commonMain/kotlin`. Then move all your tests from `src/test/kotlin` to `src/commonTest/kotlin`.

In tests, you need to replace test annotations with those from `kotlin-test` library.

You need to add `src/jsMain/kotlin` folder, and create a file `SudokuGeneratorJs.kt` in it. It could contain the following code:

```
@file:OptIn(ExperimentalJsExport::class)

import generator.SudokuGenerator
import solver.SudokuSolver

@JsExport
@JsName("SudokuSolver")
class SudokuSolverJs {
    private val generator = SudokuGenerator()
    private val solver = SudokuSolver()

    fun generateSudoku() = generator
        .generate(solver)
        .let {
            Sudoku(it.solved.toJs(), it.sudoku.toJs())
        }
}

@JsExport
class Sudoku(
    val solved: Array<Array<Int?>>,
```

```

    val sudoku: Array<Array<Int?>>
)

fun SudokuState.toJs(): Array<Array<Int?>> = List(9) { row ->
    List(9) { col ->
        val cell = this.cells[SudokuState.Position(row, col)]
        when (cell) {
            is SudokuState.CellState.Filled -> cell.value
            is SudokuState.CellState.Empty, null -> null
        }
    }.toTypedArray()
}.toTypedArray()

```

Solution: Function caller

```

class FunctionCaller {
    private var values: MutableMap<KType, Any?> =
        mutableMapOf()

    inline fun <reified T> setConstant(value: T) {
        setConstant(typeOf<T>(), value)
    }

    fun setConstant(type: KType, value: Any?) {
        values[type] = value
    }

    fun <T> call(function: KFunction<T>): T {
        val args = function.parameters
        .filter { param ->
            values.containsKey(param.type)
        }
        .associateWith { param ->
            val type = param.type
            val value = values[type]
            require(param.isOptional || value != null) {
                "No value for $type"
            }
            value
        }
    }
}

```

```
        }
    return function.callBy(args)
}
}
```

Solution: Object serialization to JSON

```
fun serializeToJson(value: Any): String = valueToJson(value)

private fun objectToJson(any: Any): String {
    val reference = any::class
    val classNameMapper = reference
        .findAnnotation<SerializationNameMapper>()
        ?.let(::createMapper)
    val ignoreNulls = reference
        .hasAnnotation<SerializationIgnoreNulls>()

    return reference
        .memberProperties
        .filterNot { it.hasAnnotation<SerializationIgnore>() }
        .mapNotNull { prop ->
            val annotationName = prop
                .findAnnotation<SerializationName>()
            val mapper = prop
                .findAnnotation<SerializationNameMapper>()
                ?.let(::createMapper)
            val name = annotationName?.name
                ?: mapper?.map(prop.name)
                ?: classNameMapper?.map(prop.name)
                ?: prop.name
            val value = prop.call(any)
            if (ignoreNulls && value == null) {
                return@mapNotNull null
            }
            "\"${name}\": ${valueToJson(value)}"
        }
        .joinToString(
            prefix = "{",
            postfix = "}"
        )
}
```

```
        )  
    }  
  
private fun valueToJson(value: Any?): String = when (value) {  
    null, is Number, is Boolean -> "$value"  
    is String, is Char, is Enum<*> -> "\"$value\""  
    is Iterable<*> -> iterableToJson(value)  
    is Map<*, *> -> mapToJson(value)  
    else -> objectToJson(value)  
}  
  
private fun iterableToJson(any: Iterable<*>): String = any  
    .joinToString(  
        prefix = "[",  
        postfix = "]",  
        transform = ::valueToJson  
    )  
  
private fun mapToJson(any: Map<*, *>) = any.toList()  
    .joinToString(  
        prefix = "{",  
        postfix = "}",  
        transform = {  
            "\"${it.first}\": ${valueToJson(it.second)}"  
        }  
    )  
  
private fun createMapper(  
    annotation: SerializationNameMapper  
) : NameMapper =  
    annotation.mapper.newInstance  
        ?: createWithNoargConstructor(annotation)  
        ?: error("Cannot create mapper")  
  
private fun createWithNoargConstructor(  
    annotation: SerializationNameMapper  
) : NameMapper? =  
    annotation.mapper  
        .constructors
```

```
.find { it.parameters.isEmpty() }
?.call()
```

Solution: Object serialization to XML

```
fun serializeToXml(value: Any): String = valueToXml(value)

private fun objectToXml(any: Any): String {
    val reference = any::class
    val classNameMapper = reference
        .findAnnotation<SerializationNameMapper>()
        ?.let(::createMapper)
    val simpleName = reference.simpleName.orEmpty()
    val className = classNameMapper?.map(simpleName)
        ?: simpleName
    val ignoreNulls = reference
        .hasAnnotation<SerializationIgnoreNulls>()

    return reference
        .memberProperties
        .filterNot { it.hasAnnotation<SerializationIgnore>() }
        .mapNotNull { prop ->
            val annotationName = prop
                .findAnnotation<SerializationName>()
            val mapper = prop
                .findAnnotation<SerializationNameMapper>()
                ?.let(::createMapper)
            val name = annotationName?.name
                ?: mapper?.map(prop.name)
                ?: classNameMapper?.map(prop.name)
                ?: prop.name
            val value = prop.call(any)
            if (ignoreNulls && value == null) {
                return@mapNotNull null
            }
            "<$name>${valueToXml(value)}</$name>"
        }
        .joinToString(
            separator = ""),
```

```
        prefix = "<$className>",
        postfix = "</${className}>",
    )
}

private fun valueToXml(value: Any?): String = when (value) {
    null, is Number, is Boolean, is String,
    is Char, is Enum<*> -> "$value"
    is Iterable<*> -> iterableToJson(value)
    is Map<*, *> -> mapToJson(value)
    else -> objectToXml(value)
}

private fun iterableToJson(any: Iterable<*>): String = any
    .joinToString(
        separator = "",

        transform = ::valueToXml
    )

private fun mapToJson(any: Map<*, *>) = any.toList()
    .joinToString(
        separator = "",

        transform = { (name, value) ->
            "<${name}>${valueToXml(value)}</${name}>"
        }
    )

private fun createMapper(
    annotation: SerializationNameMapper
): NameMapper =
    annotation.mapper.newInstance
        ?: createWithNoargConstructor(annotation)
        ?: error("Cannot create mapper")

private fun createWithNoargConstructor(
    annotation: SerializationNameMapper
): NameMapper? =
    annotation.mapper
        .constructors
```

```
.find { it.parameters.isEmpty() }
?.call()
```

Solution: DSL-based dependency injection library

```
class Registry {
    private val creatorsRegistry =
        mutableMapOf<KType, () -> Any?>()
    private val instances =
        mutableMapOf<KType, Any?>()

    inline fun <reified T> singleton(
        noinline creator: Registry.() -> T
    ) {
        singleton(typeOf<T>(), creator)
    }

    fun singleton(type: KType, creator: Registry.() -> Any?) {
        creatorsRegistry[type] = {
            instances.getOrPut(type) { creator.invoke(this) }
        }
    }

    inline fun <reified T> register(
        noinline creator: Registry.() -> T
    ) {
        register(typeOf<T>(), creator)
    }

    fun register(type: KType, creator: Registry.() -> Any?) {
        creatorsRegistry[type] = { creator(this) }
    }

    inline fun <reified T> get(): T = get(typeOf<T>()) as T

    fun get(key: KType): Any? {
        require(exists(key)) { "The $key not in registry." }
        return creatorsRegistry[key]?.invoke()
    }
}
```

```
fun exists(key: KType) =  
    creatorsRegistry.containsKey(key)  
  
inline fun <reified T> exists(): Boolean =  
    exists(typeOf<T>())  
}  
  
fun registry(init: Registry.() -> Unit) = Registry()  
    .apply(init)
```