Marcin Moskała

Advanced Kotlin

Advanced Kotlin

Marcin Moskała

This book is available at http://leanpub.com/advanced_kotlin

This version was published on 2024-12-03

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

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*.

The structure of this book

The book is divided into the following parts:

- **Part 1: Advanced Kotlin features** - dedicated to advanced Kotlin features, including generic variance modifiers, delegation, and contracts.
- **Part 2: Kotlin on different platforms** - dedicated to multiplatform programming, including interoperability with Java, JavaScript, and multiplatform libraries.
- **Part 3: Metaprogramming** - dedicated to Kotlin metaprogramming capabilities, including reflection, annotation processing, Kotlin Symbol Processing, and Kotlin Compiler Plugins.

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 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 ("This explains the essential difference between List and Set"), and a list is one of many lists ("The `list` variable holds a list").

Code conventions

Most of the presented snippets are executable, so if you copy-paste them to a Kotlin file, you should be able to execute them. The source code of all the snippets is published in the following repository:

https://github.com/MarcinMoskala/advanced_kotlin_sources

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 other cases, the result will be printed in comments after the code.

```
fun main() {
    (1..3).forEach(::println)
}
// 1
// 2
// 3
```

Sometimes, some parts of code or results are shortened with . . . in a comment. 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() }

    // ...
}
```

In some snippets, you might notice strange formatting. This is because the line length in this book is only 67 characters, so I adjusted the formatting to fit the page width.

Exercises and solutions

At the end of most chapters, you will find exercises. They are designed to help you understand the material better. Starting code and unit tests for most of those exercises can be found in the MarcinMoskala/kotlin-exercises project on GitHub. You can clone this project and solve these exercises locally. Solutions can be found at the end of the book.

Suggestions

If you have any suggestions or corrections regarding this book, send them to contact@kt.academy

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.

Part 1: Advanced Kotlin features

There are Kotlin features that are often not understood even by experienced developers. Many developers use property delegates, like `lazy` or `observable`, without understanding how property delegation works, or benefit from variance modifiers or Kotlin Contracts, without even noticing that they are using them. If you are one of those developers, this chapter is for you, because it will explain in detail how the most advanced Kotlin features work.

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 is[1].

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` 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<B>`. So, in our example, for `class Box<out T>`, the `Box<Puppy>` type is a subtype of `Box<Dog>`.

[1] In 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>`.

```
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<B>` 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>`.

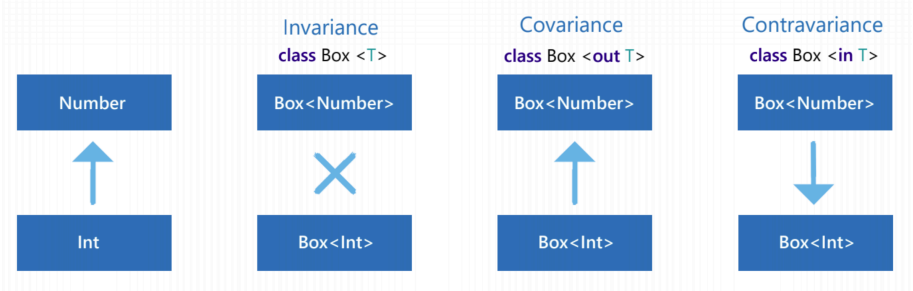
```
class Box<in T>
open class Dog
class Puppy : Dog()

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?

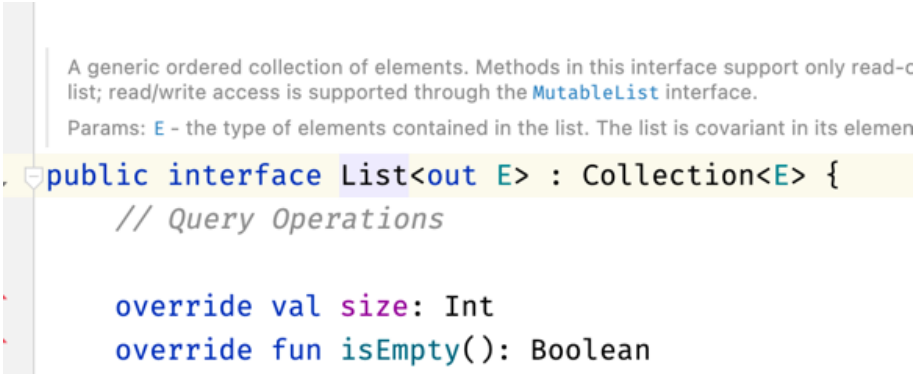
```
interface Animal {
    fun pet()
}

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.



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.

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

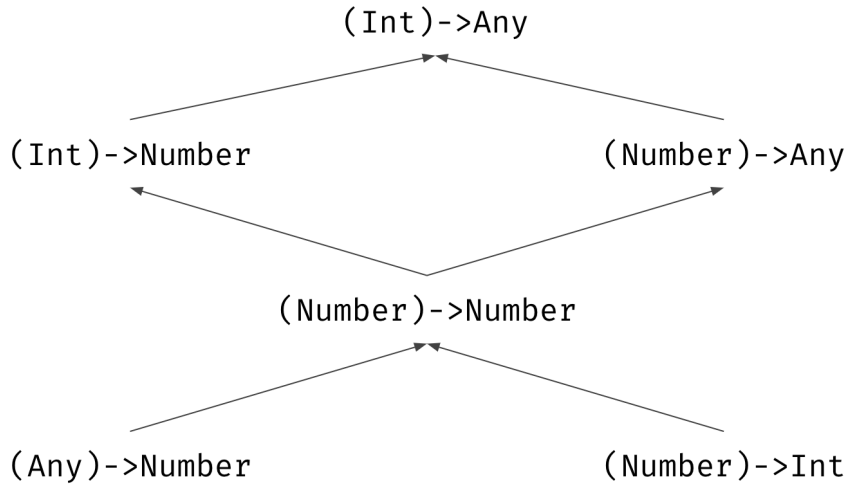
In function types, there are relations between function types with different parameters and result types. To see this in practice, think of a function that as an argument expects a function that accepts an `Int` and returns an `Any`:

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

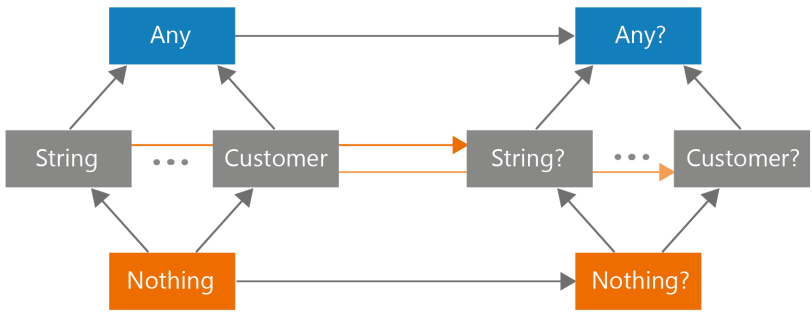
Based on its definition, such a function can accept a function of type `(Int)->Any`, but it would work with `(Int)->Number`, `(Number)->Any`, `(Number)->Number`, `(Any)->Number`, `(Number)->Int`, etc.

```
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:

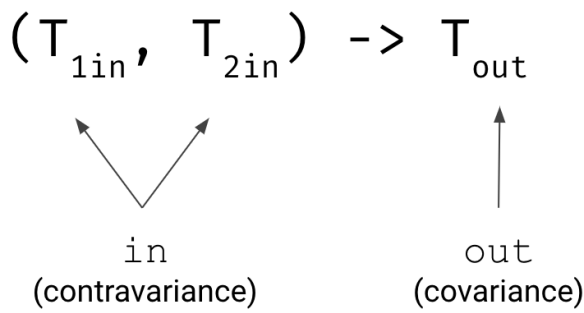


Notice that when we go down in this hierarchy, the parameter type moves toward types that are higher in the typing system hierarchy, and the return type moves toward lower types.



Kotlin type hierarchy

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.

Exercise: Usage of generic types

The code below will not compile due to a 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>())
```

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.

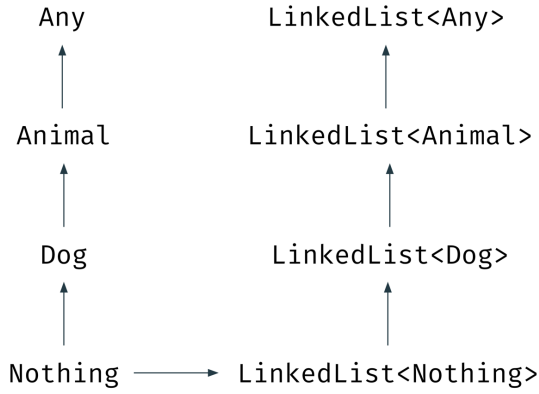
```
sealed class LinkedList<T>
data class Node<T>(
    val head: T,
    val tail: LinkedList<T>
) : LinkedList<T>()
class Empty<T> : LinkedList<T>()

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>
data class Change<T>(val newValue: T) : ChangesTrackerMessage<T>
data object Reset : ChangesTrackerMessage<Nothing>
data object UndoChange : ChangesTrackerMessage<Nothing>

sealed interface SchedulerMessage<out T>
data class Schedule<T>(val task: Task<T>) : SchedulerMessage<T>
data class Delete(val taskId: String) : SchedulerMessage<Nothing>
data object StartScheduled : SchedulerMessage<Nothing>
data object Reset : SchedulerMessage<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:

```
data 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.

```
data 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.

```
data 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>
data object Keep : TaskPropertyUpdate<Nothing>
data 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), // COMPILATION ERROR
    // type mismatch, expecting String
    priority = ChangeTo(null), // COMPILATION 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.

```
data 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>
data object Keep : TaskPropertyUpdate<Nothing>
data class ChangeTo<T>(val newValue: T) : TaskPropertyUpdate<T>
data object RestorePrevious : TaskPropertyUpdate<Nothing>
data 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>
data class Left<out L>(val value: L) : Either<L, Nothing>()
data 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[] letters= {"B", "C", "A"};
Arrays.sort(letters); // 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 upcasted; 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>)
}
```

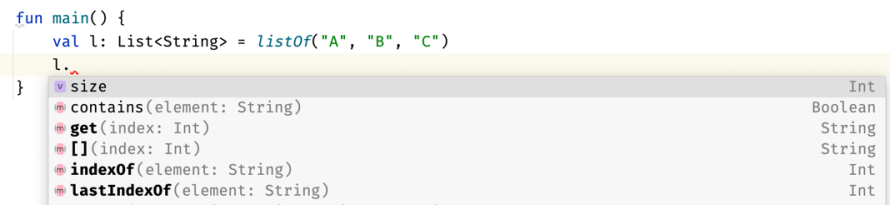
Read-write property types are invariant, so public read-write properties support neither covariant nor contravariant types.

```
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 positions[2]. 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.

[2]This is also called mixed-site variance.

```
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:

```
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
`T`	`T`	`T`
`out T`	`Nothing`	`T`
`in T`	`T`	`Any?`
`*`	`Nothing`	`Any?`

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<B>`.
- 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<B>`. 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<B>` 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 Response

You needed to model a response from a server that can be represented as a success or a failure, both of which contain data of a generic type. This is how you modeled 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` and `Failure` in a way that each can be created with only one generic type argument. You want to be able to use `Success` and `Failure` without specifying generic types, so just use `Success(1)` or `Failure("Error")`.

You also want to allow `Success<Int>` to be upcast to `Success<Number>` or to `Success<Any>`, and `Failure<Error>` to be upcast to `Failure<Throwable>` or to `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 example usage 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 `T` should also accept `S` if it is a subtype of `T`. Update the `Consumer`, `Printer` and `Sender` classes to achieve this.

```
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) {
        // ...
    }
}
```

Example usage:

```
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 example usage 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

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

The delegation pattern

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

Delegation and inheritance

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

Kotlin interface delegation support

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

Wrapper classes

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

The decorator pattern

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

Intersection types

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

Limitations

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

Conflicting elements from parents

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

Summary

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

Exercise: ApplicationScope

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

Property delegation

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

How property delegation works

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

Other `getValue` and `setValue` parameters

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

Implementing a custom property delegate

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

Provide a delegate

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

Property delegates in Kotlin stdlib

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

The `notNull` delegate

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

Exercise: Lateinit delegate

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

The `lazy` delegate

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

Exercise: Blog Post Properties

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

The `observable` delegate

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

The `vetoable` delegate

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

A map as a delegate

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

Review of how variables work

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

Summary

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

Exercise: Mutable lazy delegate

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

Kotlin Contracts

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

The meaning of a contract

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

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

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

Implications of the fact that a function has returned a value

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

Using contracts in practice

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

Summary

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

Exercise: Coroutine time measurement

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

Part 2: Kotlin on different platforms

Kotlin is a multiplatform language. It means that you can write code once and run it on different platforms. This is a powerful feature, especially when you want to share the business logic between different platforms, like Android, iOS, and the web. But with great power comes great responsibility. Interoperability with each platform has its own challenges. In this part, we will explore how to write multiplatform code and how to deal with interoperability between Kotlin and other languages.

Java interoperability

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

Nullable types

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

Kotlin type mapping

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

JVM primitives

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

Collection types

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

Annotation targets

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

Static elements

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

JvmField

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

Using Java accessors in Kotlin

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

JvmName

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

JvmMultifileClass

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

JvmOverloads

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

Unit

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

Function types and function interfaces

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

Tricky names

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

Throws

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

JvmRecord

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

Summary

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

Exercise: Adjust Kotlin for Java usage

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

Using Kotlin Multiplatform

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

Multiplatform module configuration

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

Expect and actual elements

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

Possibilities

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

Multiplatform libraries

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

A multiplatform mobile application

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

ViewModel class

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

Platform-specific classes

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

Observing properties

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

Summary

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

Exercise: Multiplatform LocalDateTime

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

JavaScript interoperability

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

Setting up a project

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

Using libraries available for Kotlin/JS

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

Using Kotlin/JS

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

Building and linking a package

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

Distributing a package to npm

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

Exposing objects

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

Exposing Flow and StateFlow

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

Adding npm dependencies

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

Frameworks and libraries for Kotlin/JS

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

JavaScript and Kotlin/JS limitations

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

Summary

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

Exercise: Migrating a Kotlin/JVM project to KMP

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

Part 3: Metaprogramming

Some tools and libraries analyze your code to help with programming. They can generate code, analyze it, or even modify it. This is called metaprogramming. In this part, we will explore different ways of how to use metaprogramming in Kotlin. Those techniques can help you understand how popular libraries work, since they are used by many of them. This can help you better understand how to use them and how to write your own libraries.

Reflection

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

Hierarchy of classes

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

Function references

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

Parameter references

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

Property references

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

Class reference

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

Serialization example

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

Referencing types

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

Type reflection example: Random value

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

Kotlin and Java reflection

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

Breaking encapsulation

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

Summary

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

Exercise: Function caller

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

Exercise: Object serialization to JSON

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

Exercise: Object serialization to XML

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

Exercise: DSL-based dependency injection library

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

Annotation processing

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

Your first annotation processor

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

Hiding generated classes

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

Summary

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

Exercise: Annotation Processing execution measurement wrapper

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

Kotlin Symbol Processing

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

Your first KSP processor

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

Testing KSP

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

Dependencies and incremental processing

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

Multiple rounds processing

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

Using KSP on multiplatform projects

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

Summary

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

Exercise: KSP execution measurement wrapper

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

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.

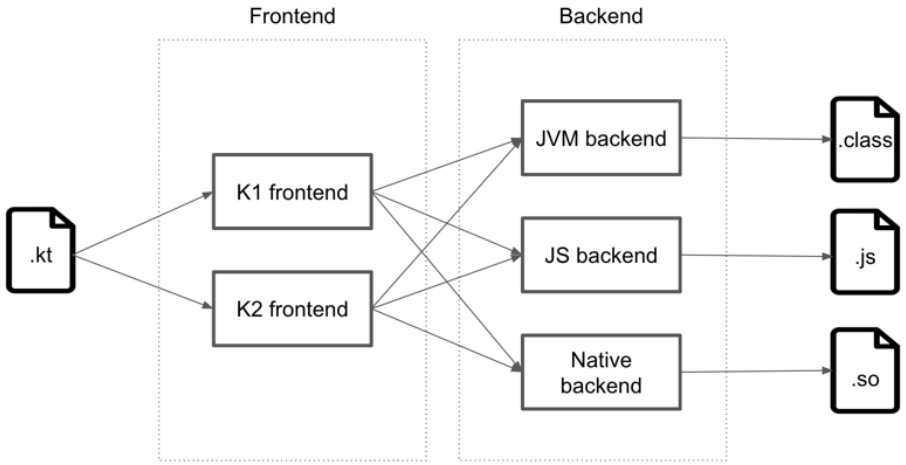
Compiler frontend and backend

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.

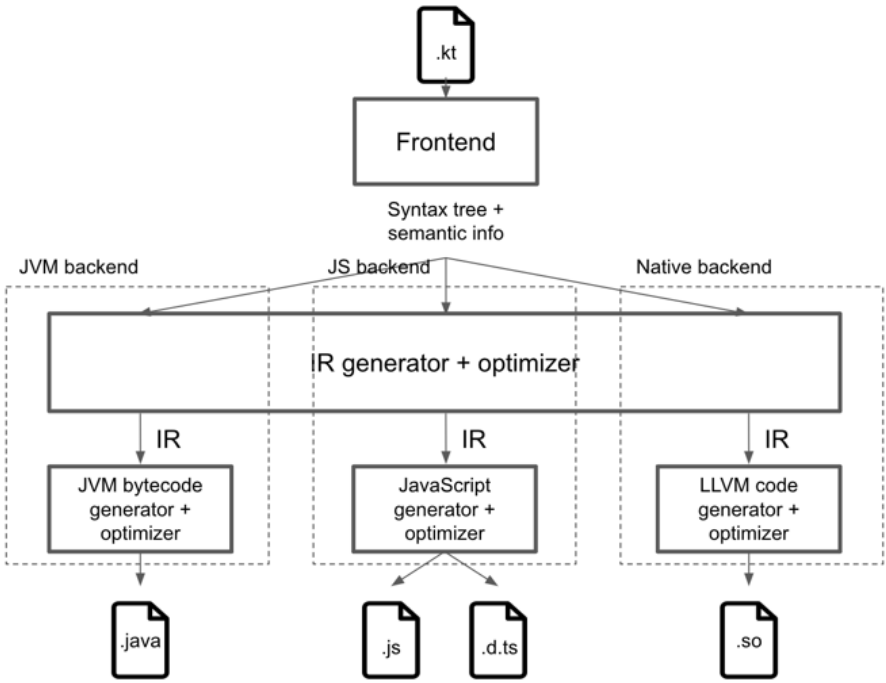
The compiler backend is specific to your compilation target, so there is a separate backend for JVM, JS, Native, and WASM. They have some shared parts, but they are essentially different.



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 extensions[3]:

- `FirStatusTransformerExtension` - called when an element status (visibility, modifiers, etc.) is established and allows it to be changed. The All-open

[3] K1 extensions are deprecated, so I will just skip them.

compiler plugin uses it to make all classes with appropriate annotations open by default (e.g., used by Spring Framework).

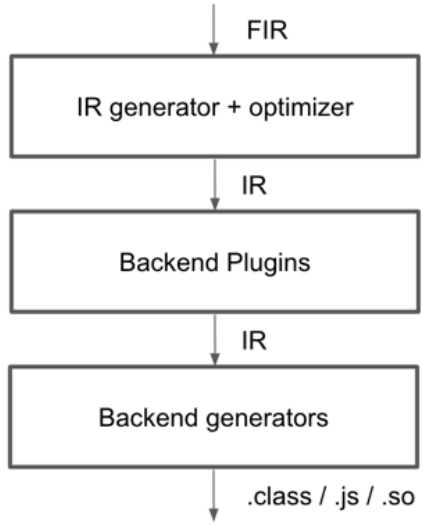
- `FirDeclarationGenerationExtension` - can specify additional declarations to be generated for a Kotlin file. Its different methods are called at different phases of compilation and allow the generation of different kinds of elements, like classes or methods. Used by many plugins, including the Kotlin 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.

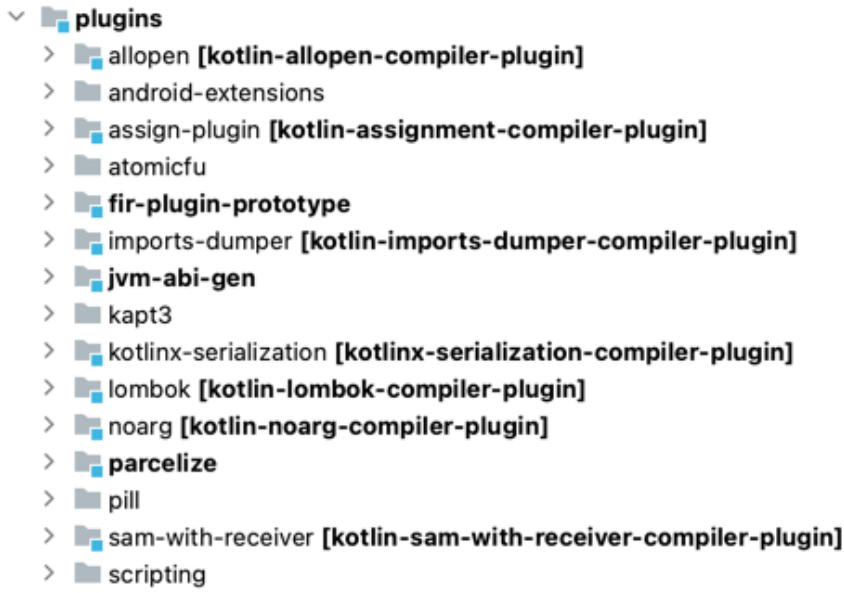
A backend plugin would require an extension for `IrGenerationExtension` that generates an additional wrapper function in IR for the appropriate function.

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 the class `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 content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/advanced_kotlin.

What are Static Analysers?

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

Types of analysers

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

Formatters

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

Code Quality Analysers

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

Data-Flow Analysers

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

Code Manipulation

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

Embedded vs Standalone

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

Kotlin Code Analysers

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

Kotlin Compiler

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

IntelliJ IDEA

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

ktlint

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

ktfmt

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

Android Lint

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

detekt

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

Setting up detekt

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

detekt Rules and Rulesets

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

Configuring detekt

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

Incremental Adoption

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

Writing your first detekt Rule

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

Setting up your rule project

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

Coding your rule

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

Using your rule

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

Rules with type resolution

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

Conclusion

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

Ending

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

Exercise solutions

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

Solution: Usage of generic types

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

Solution: Generic Response

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

Solution: Generic Consumer

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

Solution: ApplicationScope

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

Solution: Lateinit delegate

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

Solution: Blog Post Properties

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

Solution: Mutable lazy delegate

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

Solution: Coroutine time measurement

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

Solution: Adjust Kotlin for Java usage

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

Solution: Multiplatform LocalDateTime

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

Solution: Migrating a Kotlin/JVM project to KMP

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

Solution: Function caller

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

Solution: Object serialization to JSON

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

Solution: Object serialization to XML

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

Solution: DSL-based dependency injection library

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

Solution: Annotation Processing execution measurement wrapper

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

Solution: KSP execution measurement wrapper

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