SAMPLE CHAPTER

Kotlin IN ACTION

Dmitry Jemerov
Svetlana Isakova

FOREWORD BY Andrey Breslav

www.itbook.store/books/9781617293290

Kotlin in Action

by Dmitry Jemerov and Svetlana Isakova

Sample Chapter 6

Copyright 2017 Manning Publications

www.itbook.store/books/9781617293290

brief contents

PART 1 INTRODUCING KOTLIN 1

1 ■ Kotlin: what and why 3
2 ■ Kotlin basics 17
3 ■ Defining and calling functions 44
4 ■ Classes, objects, and interfaces 67
5 ■ Programming with lambdas 103
6 ■ The Kotlin type system 133

PART 2 EMBRACING KOTLIN 171

7 ■ Operator overloading and other conventions 173
8 ■ Higher-order functions: lambdas as parameters and return values 200
9 ■ Generics 223
10 ■ Annotations and reflection 254
11 ■ DSL construction 282

www.itbook.store/books/9781617293290

6 The Kotlin type system

This chapter covers

- Nullable types and syntax for dealing with `nulls`
- Primitive types and their correspondence to the Java types
- Kotlin collections and their relationship to Java

By now, you've seen a large part of Kotlin's syntax in action. You've moved beyond creating Java-equivalent code in Kotlin and are ready to enjoy some of Kotlin's productivity features that can make your code more compact and readable.

Let's slow down a bit and take a closer look at one of the most important parts of Kotlin: its type system. Compared to Java, Kotlin's type system introduces several new features that are essential for improving the reliability of your code, such as support for *nullable types* and *read-only collections*. It also removes some of the features of the Java type system that have turned out to be unnecessary or problematic, such as first-class support for arrays. Let's look at the details.

6.1 Nullability

Nullability is a feature of the Kotlin type system that helps you avoid `NullPointerException` errors. As a user of a program, you've probably seen an error message similar to "An error has occurred: java.lang.NullPointerException,"

www.itbook.store/books/9781617293290

with no additional details. Another version is a message like "Unfortunately, the application X has stopped," which often also conceals a NullPointerException as a cause. Such errors can be troublesome for both users and developers.

The approach of modern languages, including Kotlin, is to convert these problems from runtime errors into compile-time errors. By supporting nullability as part of the type system, the compiler can detect many possible errors during compilation and reduce the possibility of having exceptions thrown at runtime.

In this section, we'll discuss nullable types in Kotlin: how Kotlin marks values that are allowed to be null, and the tools Kotlin provides to deal with such values. Moving beyond that, we'll cover the details of mixing Kotlin and Java code with respect to nullable types.

6.1.1 Nullable types

The first and probably most important difference between Kotlin's and Java's type systems is Kotlin's explicit support for *nullable types*. What does this mean? It's a way to indicate which variables or properties in your program are allowed to be null. If a variable can be null, calling a method on it isn't safe, because it can cause a NullPointerException. Kotlin disallows such calls and thereby prevents many possible exceptions. To see how this works in practice, let's look at the following Java function:

```
/* Java */
int strLen(String s) {
    return s.length();
}
```

Is this function safe? If the function is called with a null argument, it will throw a NullPointerException. Do you need to add a check for null to the function? It depends on the function's intended use.

Let's try to rewrite this function in Kotlin. The first question you must answer is, do you expect the function to be called with a null argument? We mean not only the null literal directly, as in strLen(null), but also any variable or other expression that may have the value null at runtime.

If you don't expect it to happen, you declare this function in Kotlin as follows:

```
fun strLen(s: String) = s.length
```

Calling strLen with an argument that may be null isn't allowed and will be flagged as error at compile time:

```
>>> strLen(null)
ERROR: Null can not be a value of a non-null type String
```

The parameter is declared as type String, and in Kotlin this means it must always contain a String instance. The compiler enforces that, so you can't pass an argument containing null. This gives you the guarantee that the strLen function will never throw a NullPointerException at runtime.

www.itbook.store/books/9781617293290

If you want to allow the use of this function with all arguments, including those that can be `null`, you need to mark it explicitly by putting a question mark after the type name:

```
fun strLenSafe(s: String?) = ...
```

You can put a question mark after any type, to indicate that the variables of this type can store `null` references: `String?`, `Int?`, `MyCustomType?`, and so on (see figure 6.1).

Figure 6.1 A variable of a nullable type can store a `null` reference

To reiterate, a type without a question mark denotes that variables of this type can't store `null` references. This means all regular types are non-`null` by default, unless explicitly marked as nullable.

Once you have a value of a nullable type, the set of operations you can perform on it is restricted. For example, you can no longer call methods on it:

```
>>> fun strLenSafe(s: String?) = s.length()
ERROR: only safe (?.) or non-null asserted (!!.) calls are allowed
 on a nullable receiver of type kotlin.String?
```

You can't assign it to a variable of a non-`null` type:

```
>>> val x: String? = null
>>> var y: String = x
ERROR: Type mismatch: inferred type is String? but String was expected
```

You can't pass a value of a nullable type as an argument to a function having a non-`null` parameter:

```
>>> strLen(x)
ERROR: Type mismatch: inferred type is String? but String was expected
```

So what can you do with it? The most important thing is to compare it with `null`. And once you perform the comparison, the compiler remembers that and treats the value as being non-`null` in the scope where the check has been performed. For example, this code is perfectly valid.

Listing 6.1 Handling `null` values using `if` checks

```
fun strLenSafe(s: String?): Int =
    if (s != null) s.length else 0

>>> val x: String? = null
>>> println(strLenSafe(x))
0
>>> println(strLenSafe("abc"))
3
```

By adding the check for null, the code now compiles.

If using `if` checks was the only tool for tackling nullability, your code would become verbose fairly quickly. Fortunately, Kotlin provides a number of other tools to help deal

www.itbook.store/books/9781617293290

with nullable values in a more concise manner. But before we look at those tools, let's spend time discussing the meaning of nullability and what variable types are.

6.1.2 The meaning of types

Let's think about the most general questions: what are types, and why do variables have them? The Wikipedia article on types (http://en.wikipedia.org/wiki/Data_type) gives a pretty good answer to what a type is: "A type is a classification … that determines the possible values for that type, and the operations that can be done on values of that type."

Let's try to apply this definition to some of the Java types, starting with the `double` type. As you know, a `double` is a 64-bit floating-point number. You can perform standard mathematical operations on these values. All of those functions are equally applicable to all values of type `double`. Therefore, if you have a variable of type `double`, then you can be certain that any operation on its value that's allowed by the compiler will execute successfully.

Now let's contrast this with a variable of type `String`. In Java, such a variable can hold one of two kinds of values: an instance of the class `String` or `null`. Those kinds of values are completely unlike each other: even Java's own `instanceof` operator will tell you that `null` isn't a `String`. The operations that can be done on the value of the variable are also completely different: an actual `String` instance allows you to call any methods on the string, whereas a `null` value allows only a limited set of operations.

This means Java's type system isn't doing a good job in this case. Even though the variable has a declared type—`String`—you don't know what you can do with values of this variable unless you perform additional checks. Often, you skip those checks because you know from the general flow of data in your program that a value can't be `null` at a certain point. Sometimes you're wrong, and your program then crashes with a `NullPointerException`.

Other ways to cope with NullPointerException errors

Java has some tools to help solve the problem of `NullPointerException`. For example, some people use annotations (such as `@Nullable` and `@NotNull`) to express the nullability of values. There are tools (for example, IntelliJ IDEA's built-in code inspections) that can use these annotations to detect places where a `NullPointerException` can be thrown. But such tools aren't part of the standard Java compilation process, so it's hard to ensure that they're applied consistently. It's also difficult to annotate the entire codebase, including the libraries used by the project, so that all possible error locations can be detected. Our own experience at JetBrains shows that even widespread use of nullability annotations in Java doesn't completely solve the problem of NPEs.

Another path to solving this problem is to never use `null` values in code and to use a special wrapper type, such as the `Optional` type introduced in Java 8, to represent values that may or may not be defined. This approach has several downsides: the code

www.itbook.store/books/9781617293290

gets more verbose, the extra wrapper instances affect performance at runtime, and it's not used consistently across the entire ecosystem. Even if you do use `Optional` everywhere in your own code, you'll still need to deal with `null` values returned from methods of the JDK, the Android framework, and other third-party libraries.

Nullable types in Kotlin provide a comprehensive solution to this problem. Distinguishing nullable and non-`null` types provides a clear understanding of what operations are allowed on the value and what operations can lead to exceptions at runtime and are therefore forbidden.

> **NOTE** Objects of nullable or non-`null` types at runtime are the same; a nullable type isn't a wrapper for a non-`null` type. All checks are performed at compilation time. That means there's no runtime overhead for working with nullable types in Kotlin.

Now let's see how to work with nullable types in Kotlin and why dealing with them is by no means annoying. We'll start with the special operator for safely accessing a nullable value.

6.1.3 Safe call operator: "?."

One of the most useful tools in Kotlin's arsenal is the *safe-call* operator: `?.`, which allows you to combine a `null` check and a method call into a single operation. For example, the expression `s?.toUpperCase()` is equivalent to the following, more cumbersome one: `if (s != null) s.toUpperCase() else null`.

In other words, if the value on which you're trying to call the method isn't `null`, the method call is executed normally. If it's `null`, the call is skipped, and `null` is used as the value instead. Figure 6.2 illustrates.

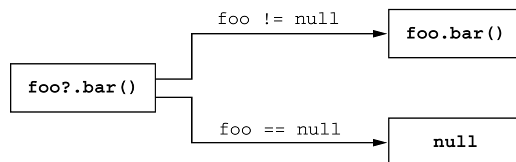

Figure 6.2 The safe-call operator calls methods only on non-`null` values.

Note that the result type of such an invocation is nullable. Although `String.toUpperCase` returns a value of type `String`, the result type of an expression `s?.toUpperCase()` when `s` is nullable will be `String?`:

```
fun printAllCaps(s: String?) {
    val allCaps: String? = s?.toUpperCase()     <-- allCaps may be null.
    println(allCaps)
}
```

www.itbook.store/books/9781617293290

```
>>> printAllCaps("abc")
ABC
>>> printAllCaps(null)
null
```

Safe calls can be used for accessing properties as well, not just for method calls. The following example shows a simple Kotlin class with a nullable property and demonstrates the use of a safe-call operator for accessing that property.

Listing 6.2 Using safe calls to deal with nullable properties

```
class Employee(val name: String, val manager: Employee?)

fun managerName(employee: Employee): String? = employee.manager?.name

>>> val ceo = Employee("Da Boss", null)
>>> val developer = Employee("Bob Smith", ceo)
>>> println(managerName(developer))
Da Boss
>>> println(managerName(ceo))
null
```

If you have an object graph in which multiple properties have nullable types, it's often convenient to use multiple safe calls in the same expression. Say you store information about a person, their company, and the address of the company using different classes. Both the company and its address may be omitted. With the `?.` operator, you can access the `country` property for a `Person` in one line, without any additional checks.

Listing 6.3 Chaining multiple safe-call operators

```
class Address(val streetAddress: String, val zipCode: Int,
              val city: String, val country: String)

class Company(val name: String, val address: Address?)

class Person(val name: String, val company: Company?)

fun Person.countryName(): String {
   val country = this.company?.address?.country      <-- Several safe-call
   return if (country != null) country else "Unknown"     operators can be in a chain.
}
>>> val person = Person("Dmitry", null)
>>> println(person.countryName())
Unknown
```

Sequences of calls with `null` checks are a common sight in Java code, and you've now seen how Kotlin makes them more concise. But listing 6.3 contains unnecessary repetition: you're comparing a value to `null` and returning either that value or something else if it's `null`. Let's see if Kotlin can help get rid of that repetition.

www.itbook.store/books/9781617293290

6.1.4 Elvis operator: "?:"

Kotlin has a handy operator to provide default values instead of `null`. It's called the *Elvis operator* (or the *null-coalescing operator*, if you prefer more serious-sounding names for things). It looks like this: `?:` (you can visualize it being Elvis if you turn your head sideways). Here's how it's used:

```
fun foo(s: String?) {
    val t: String = s ?: ""
}
```

If "s" is null, the result is an empty string.

The operator takes two values, and its result is the first value if it isn't `null` or the second value if the first one is `null`. Figure 6.3 shows how it works.

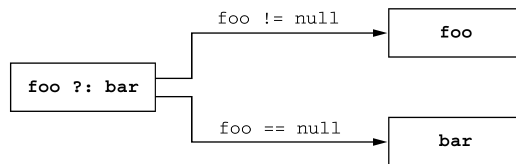

Figure 6.3 The Elvis operator substitutes a specified value for `null`.

The Elvis operator is often used together with the safe-call operator to substitute a value other than `null` when the object on which the method is called is `null`. Here's how you can use this pattern to simplify listing 6.1.

Listing 6.4 Using the Elvis operator to deal with `null` values

```
fun strLenSafe(s: String?): Int = s?.length ?: 0

>>> println(strLenSafe("abc"))
3
>>> println(strLenSafe(null))
0
```

The `countryName` function from listing 6.3 also fits on one line now.

```
fun Person.countryName() =
    company?.address?.country ?: "Unknown"
```

What makes the Elvis operator particularly handy in Kotlin is that operations such as `return` and `throw` work as expressions and therefore can be used on the operator's right side. In that case, if the value on the left side is `null`, the function will immediately return a value or throw an exception. This is helpful for checking preconditions in a function.

www.itbook.store/books/9781617293290

Let's see how you can use this operator to implement a function to print a shipping label with the person's company address. The following listing repeats the declarations of all the classes—in Kotlin, they're so concise that it's not a problem.

Listing 6.5 Using `throw` together with Elvis operator

```
class Address(val streetAddress: String, val zipCode: Int,
              val city: String, val country: String)

class Company(val name: String, val address: Address?)

class Person(val name: String, val company: Company?)

fun printShippingLabel(person: Person) {
    val address = person.company?.address
      ?: throw IllegalArgumentException("No address")      // Throws an exception if the address is absent
    with (address) {                                       // "address" is non-null.
        println(streetAddress)
        println("$zipCode $city, $country")
    }
}

>>> val address = Address("Elsestr. 47", 80687, "Munich", "Germany")
>>> val jetbrains = Company("JetBrains", address)
>>> val person = Person("Dmitry", jetbrains)

>>> printShippingLabel(person)
Elsestr. 47
80687 Munich, Germany

>>> printShippingLabel(Person("Alexey", null))
java.lang.IllegalArgumentException: No address
```

The function `printShippingLabel` prints a label if everything is correct. If there's no address, it doesn't just throw a `NullPointerException` with a line number, but instead reports a meaningful error. If an address is present, the label consists of the street address, the ZIP code, the city, and the country. Note how the `with` function, which you saw in the previous chapter, is used to avoid repeating `address` four times in a row.

Now that you've seen the Kotlin way to perform "if not-`null`" checks, let's talk about the Kotlin safe version of `instanceof` checks: the *safe-cast operator* that often appears together with safe calls and Elvis operators.

6.1.5 Safe casts: "as?"

In chapter 2, you saw the regular Kotlin operator for type casts: the `as` operator. Just like a regular Java type cast, `as` throws a `ClassCastException` if the value doesn't have the type you're trying to cast it to. Of course, you can combine it with an `is` check to ensure that it does have the proper type. But as a safe and concise language, doesn't Kotlin provide a better solution? Indeed it does.

www.itbook.store/books/9781617293290

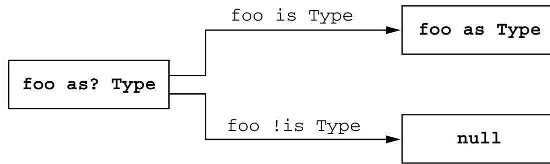

Figure 6.4 The safe-cast operator tries to cast a value to the given type and returns `null` if the type differs.

The `as?` operator tries to cast a value to the specified type and returns `null` if the value doesn't have the proper type. Figure 6.4 illustrates this.

One common pattern of using a safe cast is combining it with the Elvis operator. For example, this comes in handy for implementing the `equals` method.

Listing 6.6 Using a safe cast to implement `equals`

```
class Person(val firstName: String, val lastName: String) {
   override fun equals(o: Any?): Boolean {
      val otherPerson = o as? Person ?: return false      // Checks the type and returns false if no match

      return otherPerson.firstName == firstName &&        // After the safe cast, the variable otherPerson is smart-cast to the Person type.
             otherPerson.lastName == lastName
   }

   override fun hashCode(): Int =
      firstName.hashCode() * 37 + lastName.hashCode()
}

>>> val p1 = Person("Dmitry", "Jemerov")
>>> val p2 = Person("Dmitry", "Jemerov")
>>> println(p1 == p2)                                     // The == operator calls the "equals" method.
true
>>> println(p1.equals(42))
false
```

With this pattern, you can easily check whether the parameter has a proper type, cast it, and return `false` if the type isn't right—all in the same expression. Of course, smart casts also apply in this context: after you've checked the type and rejected `null` values, the compiler knows that the type of the `otherPerson` variable's value is `Person` and lets you use it accordingly.

The safe-call, safe-cast, and Elvis operators are useful and appear often in Kotlin code. But sometimes you don't need Kotlin's support in handling `null`s; you just need to tell the compiler that the value is in fact not `null`. Let's see how you can achieve that.

6.1.6 Not-null assertions: "!!"

The *not-null assertion* is the simplest and bluntest tool Kotlin gives you for dealing with a value of a nullable type. It's represented by a double exclamation mark and converts

www.itbook.store/books/9781617293290

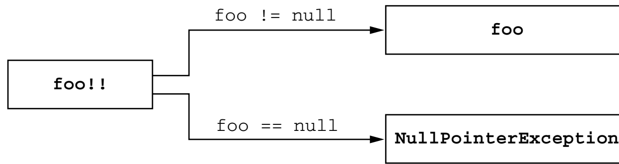

Figure 6.5 By using a not-`null` assertion, you can explicitly throw an exception if the value is `null`.

any value to a non-`null` type. For `null` values, an exception is thrown. The logic is illustrated in figure 6.5.

Here's a trivial example of a function that uses the assertion to convert a nullable argument to a non-`null` one.

Listing 6.7 Using a not-`null` assertion

```
fun ignoreNulls(s: String?) {
    val sNotNull: String = s!!          <- The exception points to this line.
    println(sNotNull.length)
}

>>> ignoreNulls(null)
Exception in thread "main" kotlin.KotlinNullPointerException
    at <...>.ignoreNulls(07_NotnullAssertions.kt:2)
```

What happens if `s` is `null` in this function? Kotlin doesn't have much choice: it will throw an exception (a special kind of `NullPointerException`) at runtime. But note that the place where the exception is thrown is the assertion itself, not a subsequent line where you're trying to use the value. Essentially, you're telling the compiler, "I know the value isn't `null`, and I'm ready for an exception if it turns out I'm wrong."

> **NOTE** You may notice that the double exclamation mark looks a bit rude: it's almost like you're yelling at the compiler. This is intentional. The designers of Kotlin are trying to nudge you toward a better solution that doesn't involve making assertions that can't be verified by the compiler.

But there are situations when not-`null` assertions are the appropriate solution for a problem. When you check for `null` in one function and use the value in another function, the compiler can't recognize that the use is safe. If you're certain the check is always performed in another function, you may not want to duplicate it before using the value; then you can use a not-`null` assertion instead.

This happens in practice with action classes, which appear in many UI frameworks such as Swing. In an action class, there are separate methods for updating the state of an action (to enable or disable it) and for executing it. The checks performed in the `update` method ensure that the `execute` method won't be called if the conditions aren't met, but there's no way for the compiler to recognize that.

www.itbook.store/books/9781617293290

Let's look at an example of a Swing action that uses a not-null assertion in this situation. The CopyRowAction action is supposed to copy the value of the selected row in a list to the clipboard. We've omitted all the unnecessary details, keeping only the code responsible for checking whether any row was selected (meaning therefore the action can be performed) and obtaining the value for the selected row. The Action API implies that actionPerformed is called only when isEnabled is true.

Listing 6.8 Using a not-null assertion in a Swing action

```
class CopyRowAction(val list: JList<String>) : AbstractAction() {
    override fun isEnabled(): Boolean =
        list.selectedValue != null

    override fun actionPerformed(e: ActionEvent) {
        val value = list.selectedValue!!
        // copy value to clipboard
    }
}
```

actionPerformed is called only if isEnabled returns "true".

Note that if you don't want to use !! in this case, you can write val value = list.selectedValue ?: return to obtain a value of a non-null type. If you use that pattern, a nullable value of list.selectedValue will cause an early return from the function, so value will always be non-null. Although the not-null check using the Elvis operator is redundant here, it may be a good protection against isEnabled becoming more complicated later.

There's one more caveat to keep in mind: when you use !! and it results in an exception, the stack trace identifies the line number in which the exception was thrown but not a specific expression. To make it clear exactly which value was null, it's best to avoid using multiple !! assertions on the same line:

```
person.company!!.address!!.country
```

Don't write code like this!

If you get an exception in this line, you won't be able to tell whether it was company or address that held a null value.

So far, we've discussed mostly how to *access* the values of nullable types. But what should you do if you need to pass a nullable value as an argument to a function that expects a non-null value? The compiler doesn't allow you to do that without a check, because doing so is unsafe. The Kotlin language doesn't have any special support for this case, but there's a standard library function that can help you: it's called let.

6.1.7 The "let" function

The let function makes it easier to deal with nullable expressions. Together with a safe-call operator, it allows you to evaluate an expression, check the result for null, and store the result in a variable, all in a single, concise expression.

www.itbook.store/books/9781617293290

One of its most common uses is handling a nullable argument that should be passed to a function that expects a non-`null` parameter. Let's say the function `sendEmailTo` takes one parameter of type `String` and sends an email to that address. This function is written in Kotlin and requires a non-`null` parameter:

```
fun sendEmailTo(email: String) { /*...*/ }
```

You can't pass a value of a nullable type to this function:

```
>>> val email: String? = ...
>>> sendEmailTo(email)
ERROR: Type mismatch: inferred type is String? but String was expected
```

You have to check explicitly whether this value isn't `null`:

```
if (email != null) sendEmailTo(email)
```

But you can go another way: use the `let` function, and call it via a safe call. All the `let` function does is turn the object on which it's called into a parameter of the lambda. If you combine it with the safe call syntax, it effectively converts an object of a nullable type on which you call `let` into a non-`null` type (see figure 6.6).

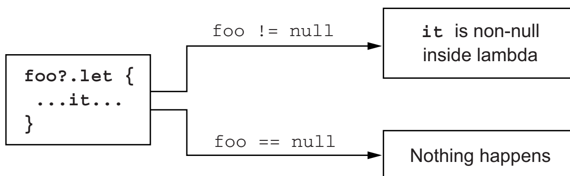

Figure 6.6 Safe-calling "let" executes a lambda only if an expression isn't `null`.

The `let` function will be called only if the email value is non-`null`, so you use the email as a non-`null` argument of the lambda:

```
email?.let { email -> sendEmailTo(email) }
```

After using the short syntax, the autogenerated name `it`, the result is much shorter: `email?.let { sendEmailTo(it) }`. Here's a more complete example that shows this pattern.

Listing 6.9 Using `let` to call a function with a non-`null` parameter

```
fun sendEmailTo(email: String) {
    println("Sending email to $email")
}

>>> var email: String? = "yole@example.com"
>>> email?.let { sendEmailTo(it) }
Sending email to yole@example.com
>>> email = null
>>> email?.let { sendEmailTo(it) }
```

www.itbook.store/books/9781617293290

Note that the let notation is especially convenient when you have to use the value of a longer expression if it's not null. You don't have to create a separate variable in this case. Compare this explicit if check

```
val person: Person? = getTheBestPersonInTheWorld()
if (person != null) sendEmailTo(person.email)
```

to the same code without an extra variable:

```
getTheBestPersonInTheWorld()?.let { sendEmailTo(it.email) }
```

This function returns null, so the code in the lambda will never be executed:

```
fun getTheBestPersonInTheWorld(): Person? = null
```

When you need to check multiple values for null, you can use nested let calls to handle them. But in most cases, such code ends up fairly verbose and hard to follow. It's generally easier to use a regular if expression to check all the values together.

One other common situation is properties that are effectively non-null but can't be initialized with a non-null value in the constructor. Let's see how Kotlin allows you to deal with that situation.

6.1.8 Late-initialized properties

Many frameworks initialize objects in dedicated methods called after the object instance has been created. For example, in Android, the activity initialization happens in the onCreate method. JUnit requires you to put initialization logic in methods annotated with @Before.

But you can't leave a non-null property without an initializer in the constructor and only initialize it in a special method. Kotlin normally requires you to initialize all properties in the constructor, and if a property has a non-null type, you have to provide a non-null initializer value. If you can't provide that value, you have to use a nullable type instead. If you do that, every access to the property requires either a null check or the !! operator.

Listing 6.10 Using non-null assertions to access a nullable property

```
class MyService {
    fun performAction(): String = "foo"
}

class MyTest {
    private var myService: MyService? = null      <-- Declares a property of a nullable type to initialize it with null

    @Before fun setUp() {
        myService = MyService()                   <-- Provides a real initializer in the setUp method
    }

    @Test fun testAction() {
        Assert.assertEquals("foo",
```

www.itbook.store/books/9781617293290

```
            myService!!.performAction())        <-- You have to take care of
    }                                                nullability: use !! or ?.
}
```

This looks ugly, especially if you access the property many times. To solve this, you can declare the `myService` property as *late-initialized*. This is done by applying the `lateinit` modifier.

Listing 6.11 Using a late-initialized property

```
class MyService {
    fun performAction(): String = "foo"
}

class MyTest {
    private lateinit var myService: MyService     <-- Declares a property of a non-null
                                                       type without an initializer
    @Before fun setUp() {
        myService = MyService()                   <-- Initializes the property in the
    }                                                  setUp method as before

    @Test fun testAction() {
        Assert.assertEquals("foo",
            myService.performAction())            <-- Accesses the property
    }                                                  without extra null checks
}
```

Note that a late-initialized property is always a `var`, because you need to be able to change its value outside of the constructor, and `val` properties are compiled into final fields that must be initialized in the constructor. But you no longer need to initialize it in a constructor, even though the property has a non-`null` type. If you access the property before it's been initialized, you get an exception "`lateinit` property `myService` has not been initialized". It clearly identifies what has happened and is much easier to understand than a generic `NullPointerException`.

> **NOTE** A common use case for `lateinit` properties is dependency injection. In that scenario, the values of `lateinit` properties are set externally by a dependency-injection framework. To ensure compatibility with a broad range of Java frameworks, Kotlin generates a field with the same visibility as the `lateinit` property. If the property is declared as `public`, the field will be `public` as well.

Now let's look at how you can extend Kotlin's set of tools for dealing with `null` values by defining extension functions for nullable types.

6.1.9 Extensions for nullable types

Defining extension functions for nullable types is one more powerful way to deal with `null` values. Rather than ensuring that a variable can't be `null` before a method call, you can allow the calls with `null` as a receiver, and deal with `null` in the function.

www.itbook.store/books/9781617293290

This is only possible for extension functions; regular member calls are dispatched through the object instance and therefore can never be performed when the instance is null.

As an example, consider the functions isEmpty and isBlank, defined as extensions of String in the Kotlin standard library. The first one checks whether the string is an empty string "", and the second one checks whether it's empty or if it consists solely of whitespace characters. You'll generally use these functions to check that the string is non-trivial in order to do something meaningful with it. You may think it would be useful to handle null in the same way as trivial empty or blank strings. And, indeed, you can do so: the functions isEmptyOrNull and isBlankOrNull can be called with a receiver of type String?.

Listing 6.12 Calling an extension function with a nullable receiver

```
fun verifyUserInput(input: String?) {
    if (input.isNullOrBlank()) {                       <- No safe call is needed.
        println("Please fill in the required fields")
    }
}

>>> verifyUserInput(" ")
Please fill in the required fields
>>> verifyUserInput(null)       <- No exception happens when you call isNullOrBlank with "null" as a receiver.
Please fill in the required fields
```

You can call an extension function that was declared for a nullable receiver without safe access (see figure 6.7). The function handles possible null values.

Figure 6.7 Extensions for nullable types can be accessed without a safe call.

The function isNullOrBlank checks explicitly for null, returning true in this case, and then calls isBlank, which can be called on a non-null String only:

```
fun String?.isNullOrBlank(): Boolean =      <- Extension for a nullable String
        this == null || this.isBlank()     <- A smart cast is applied to the second "this".
```

When you declare an extension function for a nullable type (ending with ?), that means you can call this function on nullable values; and this in a function body can be null, so you have to check for that explicitly. In Java, this is always not-null, because it references the instance of a class you're in. In Kotlin, that's no longer the case: in an extension function for a nullable type, this can be null.

www.itbook.store/books/9781617293290

Note that the `let` function we discussed earlier can be called on a nullable receiver as well, but it doesn't check the value for `null`. If you invoke it on a nullable type without using the safe-call operator, the lambda argument will also be nullable:

```
>>> val person: Person? = ...
>>> person.let { sendEmailTo(it) }
ERROR: Type mismatch: inferred type is Person? but Person was expected
```

No safe call, so "it" has a nullable type.

Therefore, if you want to check the arguments for being non-`null` with `let`, you have to use the safe-call operator `?.`, as you saw earlier: `person?.let { sendEmailTo(it) }`.

NOTE When you define your own extension function, you need to consider whether you should define it as an extension for a nullable type. By default, define it as an extension for a non-`null` type. You can safely change it later (no code will be broken) if it turns out it's used mostly on nullable values, and the `null` value can be reasonably handled.

This section showed you something unexpected. If you dereference a variable without an extra check, as in `s.isNullOrBlank()`, it doesn't immediately mean the variable is non-`null`: the function can be an extension for a nullable type. Next, let's discuss another case that may surprise you: a type parameter can be nullable even without a question mark at the end.

6.1.10 Nullability of type parameters

By default, all type parameters of functions and classes in Kotlin are nullable. Any type, including a nullable type, can be substituted for a type parameter; in this case, declarations using the type parameter as a type are allowed to be `null`, even though the type parameter `T` doesn't end with a question mark. Consider the following example.

Listing 6.13 Dealing with a nullable type parameter

```
fun <T> printHashCode(t: T) {
    println(t?.hashCode())
}
>>> printHashCode(null)
null
```

You have to use a safe call because "t" might be null.

"T" is inferred as "Any?".

In the `printHashCode` call, the inferred type for the type parameter `T` is a nullable type, `Any?`. Therefore, the parameter `t` is allowed to hold `null`, even without a question mark after `T`.

To make the type parameter non-`null`, you need to specify a non-`null` upper bound for it. That will reject a nullable value as an argument.

www.itbook.store/books/9781617293290

Listing 6.14 Declaring a non-null upper bound for a type parameter

```
fun <T: Any> printHashCode(t: T) {          <-- Now "T" can't be nullable.
    println(t.hashCode())
}
>>> printHashCode(null)                     <-- This code doesn't compile: you can't pass null because a non-null value is expected.
Error: Type parameter bound for `T` is not satisfied
>>> printHashCode(42)
42
```

Chapter 9 will cover generics in Kotlin, and section 9.1.4 will cover this topic in more detail.

Note that type parameters are the only exception to the rule that a question mark at the end is required to mark a type as nullable, and types without a question mark are non-`null`. The next section shows another special case of nullability: types that come from the Java code.

6.1.11 Nullability and Java

The previous discussion covered the tools for working with `null`s in the Kotlin world. But Kotlin prides itself on its Java interoperability, and you know that Java doesn't support nullability in its type system. So what happens when you combine Kotlin and Java? Do you lose all safety, or do you have to check every value for `null`? Or is there a better solution? Let's find out.

First, as we mentioned, sometimes Java code contains information about nullability, expressed using annotations. When this information is present in the code, Kotlin uses it. Thus `@Nullable String` in Java is seen as `String?` by Kotlin, and `@NotNull String` is just `String` (see figure 6.8)

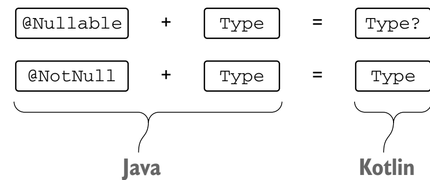

Figure 6.8 Annotated Java types are represented as nullable and non-`null` types in Kotlin, according to the annotations.

Kotlin recognizes many different flavors of nullability annotations, including those from the JSR-305 standard (in the `javax.annotation` package), the Android ones (`android.support.annotation`), and those supported by JetBrains tools (`org.jetbrains.annotations`). The interesting question is what happens when the annotations aren't present. In that case, the Java type becomes a *platform type* in Kotlin.

Platform types

A platform type is essentially a type for which Kotlin doesn't have nullability information; you can work with it as either a nullable or a non-`null` type (see figure 6.9). This means, just as in Java, you have full responsibility for the operations you perform with that type. The compiler will allow all operations. It also won't highlight

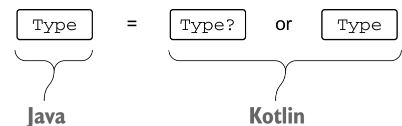

Figure 6.9 Java types are represented in Kotlin as platform types, which you can use either as a nullable type or as a non-`null` type.

www.itbook.store/books/9781617293290

as redundant any null-safe operations on such values, which it normally does when you perform a null-safe operation on a value of a non-null type. If you know the value can be null, you can compare it with null before use. If you know it's not null, you can use it directly. Just as in Java, you'll get a NullPointerException at the usage site if you get this wrong.

Let's say the class Person is declared in Java.

Listing 6.15 A Java class without nullability annotations

```
/* Java */
public class Person {
    private final String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Can getName return null or not? The Kotlin compiler knows nothing about nullability of the String type in this case, so you have to deal with it yourself. If you're sure the name isn't null, you can dereference it in a usual way, as in Java, without additional checks. But be ready to get an exception in this case.

Listing 6.16 Accessing a Java class without null checks

```
fun yellAt(person: Person) {
    println(person.name.toUpperCase() + "!!!")
}

>>> yellAt(Person(null))
java.lang.IllegalArgumentException: Parameter specified as non-null
 is null: method toUpperCase, parameter $receiver
```

The receiver person.name of the toUpperCase() call is null, so an exception is thrown.

Note that instead of a plain NullPointerException, you get a more detailed error message that the method toUpperCase can't be called on a null receiver.

In fact, for public Kotlin functions, the compiler generates checks for every parameter (and a receiver as well) that has a non-null type, so that attempts to call such a function with incorrect arguments are immediately reported as exceptions. Note that the value-checking is performed right away when the function is called, not when the parameter is used. This ensures that incorrect calls are detected early and won't cause hard-to-understand exceptions if the null value is accessed after being passed around between multiple functions in different layers of the codebase.

Your other option is to interpret the return type of getName() as nullable and access it safely.

www.itbook.store/books/9781617293290

Listing 6.17 Accessing a Java class with `null` checks

```
fun yellAtSafe(person: Person) {
    println((person.name ?: "Anyone").toUpperCase() + "!!!")
}

>>> yellAtSafe(Person(null))
ANYONE!!!
```

In this example, `null` values are handled properly, and no runtime exception is thrown.

Be careful while working with Java APIs. Most of the libraries aren't annotated, so you may interpret all the types as non-`null`, but that can lead to errors. To avoid errors, you should check the documentation (and, if needed, the implementation) of the Java methods you're using to find out when they can return `null`, and add checks for those methods.

Why platform types?

Wouldn't it be safer for Kotlin to treat all values coming from Java as nullable? Such a design would be possible, but it would require a large number of redundant `null` checks for values that can never be `null`, because the Kotlin compiler wouldn't be able to see that information.

The situation would be especially bad with generics—for example, every `ArrayList<String>` coming from Java would be an `ArrayList<String?>?` in Kotlin, and you'd need to check values for `null` on every access or use a cast, which would defeat the safety benefits. Writing such checks is extremely annoying, so the designers of Kotlin went with the pragmatic option and allowed the developers to take responsibility for correctly handling values coming from Java.

You can't declare a variable of a platform type in Kotlin; these types can only come from Java code. But you may see them in error messages and in the IDE:

```
>>> val i: Int = person.name
ERROR: Type mismatch: inferred type is String! but Int was expected
```

The `String!` notation is how the Kotlin compiler denotes platform types coming from Java code. You can't use this syntax in your own code, and usually this exclamation mark isn't connected with the source of a problem, so you can usually ignore it. It just emphasizes that the nullability of the type is unknown.

As we said already, you may interpret platform types any way you like—as nullable or as non-`null`—so both of the following declarations are valid:

```
>>> val s: String? = person.name      <-- Java's property can be seen as nullable ...
>>> val s1: String = person.name      <-- ... or non-null.
```

www.itbook.store/books/9781617293290

In this case, just as with the method calls, you need to make sure you get the nullability right. If you try to assign a `null` value coming from Java to a non-`null` Kotlin variable, you'll get an exception at the point of assignment.

We've discussed how Java types are seen from Kotlin. Let's now talk about some pitfalls of creating mixed Kotlin and Java hierarchies.

INHERITANCE

When overriding a Java method in Kotlin, you have a choice whether to declare the parameters and the return type as nullable or non-`null`. For example, let's look at a `StringProcessor` interface in Java.

Listing 6.18 A Java interface with a `String` parameter

```
/* Java */
interface StringProcessor {
    void process(String value);
}
```

In Kotlin, both of the following implementations will be accepted by the compiler.

Listing 6.19 Implementing the Java interface with different parameter nullability

```
class StringPrinter : StringProcessor {
    override fun process(value: String) {
        println(value)
    }
}

class NullableStringPrinter : StringProcessor {
    override fun process(value: String?) {
        if (value != null) {
            println(value)
        }
    }
}
```

Note that it's important to get nullability right when implementing methods from Java classes or interfaces. Because the implementation methods can be called from non-Kotlin code, the Kotlin compiler will generate non-`null` assertions for every parameter that you declare with a non-`null` type. If the Java code does pass a `null` value to the method, the assertion will trigger, and you'll get an exception, even if you never access the parameter value in your implementation.

Let's summarize our discussion of nullability. We've discussed nullable and non-`null` types and the means of working with them: operators for safe operations (safe call `?.`, Elvis operator `?:`, and safe cast `as?`), as well as the operator for unsafe dereference (the not-`null` assertion `!!`). You've seen how the library function `let` can help you accomplish concise non-`null` checks and how extensions for nullable types can help move a not-`null` check into a function. We've also discussed platform types that represent Java types in Kotlin.

www.itbook.store/books/9781617293290

Now that we've covered the topic of nullability, let's talk about how the primitive types are represented in Kotlin. This knowledge of nullability will be important for understanding how Kotlin handles Java's boxed types.

6.2 Primitive and other basic types

This section describes the basic types used in programs, such as `Int`, `Boolean`, and `Any`. Unlike Java, Kotlin doesn't differentiate primitive types and wrappers. You'll shortly learn why, and how it works under the hood. You'll see the correspondence between Kotlin types and such Java types as `Object` and `Void`, as well.

6.2.1 Primitive types: Int, Boolean, and more

As you know, Java makes a distinction between primitive types and reference types. A variable of a *primitive type* (such as `int`) holds its value directly. A variable of a *reference type* (such as `String`) holds a reference to the memory location containing the object.

Values of primitive types can be stored and passed around more efficiently, but you can't call methods on such values or store them in collections. Java provides special wrapper types (such as `java.lang.Integer`) that encapsulate primitive types in situations when an object is needed. Thus, to define a collection of integers, you can't say `Collection<int>`; you have to use `Collection<Integer>` instead.

Kotlin doesn't distinguish between primitive types and wrapper types. You always use the same type (for example, `Int`):

```
val i: Int = 1
val list: List<Int> = listOf(1, 2, 3)
```

That's convenient. What's more, you can call methods on values of a number type. For example, consider this snippet, which uses the `coerceIn` standard library function to restrict the value to the specified range:

```
fun showProgress(progress: Int) {
    val percent = progress.coerceIn(0, 100)
    println("We're ${percent}% done!")
}

>>> showProgress(146)
We're 100% done!
```

If primitive and reference types are the same, does that mean Kotlin represents all numbers as objects? Wouldn't that be terribly inefficient? Indeed it would, so Kotlin doesn't do that.

At runtime, the number types are represented in the most efficient way possible. In most cases—for variables, properties, parameters, and return types—Kotlin's `Int` type is compiled to the Java primitive type `int`. The only case in which this isn't possible is generic classes, such as collections. A primitive type used as a type argument of a generic class is compiled to the corresponding Java wrapper type. For example, if the `Int` type is used as a type argument of the collection, then the collection will store instances of `java.lang.Integer`, the corresponding wrapper type.

www.itbook.store/books/9781617293290

The full list of types that correspond to Java primitive types is:

- *Integer types*—`Byte`, `Short`, `Int`, `Long`
- *Floating-point number types*—`Float`, `Double`
- *Character type*—`Char`
- *Boolean type*—`Boolean`

A Kotlin type such as `Int` can be easily compiled under the hood to the corresponding Java primitive type, because the values of both types can't store the `null` reference. The other direction works in a similar way: When you use Java declarations from Kotlin, Java primitive types become non-`null` types (not platform types), because they can't hold `null` values. Now let's discuss the nullable versions of the same types.

6.2.2 Nullable primitive types: Int?, Boolean?, and more

Nullable types in Kotlin can't be represented by Java primitive types, because `null` can only be stored in a variable of a Java reference type. That means whenever you use a nullable version of a primitive type in Kotlin, it's compiled to the corresponding wrapper type.

To see the nullable types in use, let's go back to the opening example of the book and recall the `Person` class declared there. The class represents a person whose name is always known and whose age can be either known or unspecified. Let's add a function that checks whether one person is older than another.

Listing 6.20 Using nullable primitive types

```
data class Person(val name: String,
                  val age: Int? = null) {

    fun isOlderThan(other: Person): Boolean? {
        if (age == null || other.age == null)
            return null
        return age > other.age
    }
}

>>> println(Person("Sam", 35).isOlderThan(Person("Amy", 42)))
false
>>> println(Person("Sam", 35).isOlderThan(Person("Jane")))
null
```

Note how the regular nullability rules apply here. You can't just compare two values of type `Int?`, because one of them may be `null`. Instead, you have to check that both values aren't `null`. After that, the compiler allows you to work with them normally.

The value of the `age` property declared in the class `Person` is stored as a `java.lang.Integer`. But this detail only matters if you're working with the class from Java. To choose the right type in Kotlin, you only need to consider whether `null` is a possible value for the variable or property.

As mentioned earlier, generic classes are another case when wrapper types come into play. If you use a primitive type as a type argument of a class, Kotlin uses the

www.itbook.store/books/9781617293290

boxed representation of the type. For example, this creates a list of boxed `Integer` values, even though you've never specified a nullable type or used a `null` value:

```
val listOfInts = listOf(1, 2, 3)
```

This happens because of the way generics are implemented on the Java virtual machine. The JVM doesn't support using a primitive type as a type argument, so a generic class (both in Java and in Kotlin) must always use a boxed representation of the type. As a consequence, if you need to efficiently store large collections of primitive types, you need to either use a third-party library (such as Trove4J, http://trove.starlight-systems.com) that provides support for such collections or store them in arrays. We'll discuss arrays in detail at the end of this chapter.

Now let's look at how you can convert values between different primitive types.

6.2.3 Number conversions

One important difference between Kotlin and Java is the way they handle numeric conversions. Kotlin doesn't automatically convert numbers from one type to the other, even when the other type is larger. For example, the following code won't compile in Kotlin:

```
val i = 1
val l: Long = i                 <- Error: type mismatch
```

Instead, you need to apply the conversion explicitly:

```
val i = 1
val l: Long = i.toLong()
```

Conversion functions are defined for every primitive type (except `Boolean`): `toByte()`, `toShort()`, `toChar()` and so on. The functions support converting in both directions: extending a smaller type to a larger one, like `Int.toLong()`, and truncating a larger type to a smaller one, like `Long.toInt()`.

Kotlin makes the conversion explicit in order to avoid surprises, especially when comparing boxed values. The `equals` method for two boxed values checks the box type, not just the value stored in it. Thus, in Java, `new Integer(42).equals(new Long(42))` returns `false`. If Kotlin supported implicit conversions, you could write something like this:

```
val x = 1                          <- Int variable
val list = listOf(1L, 2L, 3L)      <- List of Long values
x in list                          <- False if Kotlin supported implicit conversions
```

This would evaluate to `false`, contrary to everyone's expectations. Thus the line `x in list` from this example doesn't compile. Kotlin requires you to convert the types explicitly so that only values of the same type are compared:

```
>>> val x = 1
>>> println(x.toLong() in listOf(1L, 2L, 3L))
true
```

www.itbook.store/books/9781617293290

If you use different number types in your code at the same time, you have to convert variables explicitly to avoid unexpected behavior.

Primitive type literals

Kotlin supports the following ways to write number literals in source code, in addition to simple decimal numbers:

- Literals of type `Long` use the `L` suffix: `123L`.
- Literals of type `Double` use the standard representation of floating-point numbers: `0.12`, `2.0`, `1.2e10`, `1.2e-10`.
- Literals of type `Float` use the `f` or `F` suffix: `123.4f`, `.456F`, `1e3f`.
- Hexadecimal literals use the `0x` or `0X` prefix (such as `0xCAFEBABE` or `0xbcdL`).
- Binary literals use the `0b` or `0B` prefix (such as `0b000000101`).

Note that underscores in number literals are only supported starting with Kotlin 1.1.

For character literals, you use mostly the same syntax as in Java. You write the character in single quotes, and you can also use escape sequences if you need to. The following are examples of valid Kotlin character literals: `'1'`, `'\t'` (the tab character), `'\u0009'` (the tab character represented using a Unicode escape sequence).

Note that when you're writing a number literal, you usually don't need to use conversion functions. One possibility is to use the special syntax to mark the type of the constant explicitly, such as `42L` or `42.0f`. And even if you don't use it, the necessary conversion is applied automatically if you use a number literal to initialize a variable of a known type or pass it as an argument to a function. In addition, arithmetic operators are overloaded to accept all appropriate numeric types. For example, the following code works correctly without any explicit conversions:

```
fun foo(l: Long) = println(l)

>>> val b: Byte = 1
>>> val l = b + 1L
>>> foo(42)
42
```

Constant value gets the correct type

+ works with Byte and Long arguments.

The compiler interprets 42 as a Long value.

Note that the behavior of Kotlin arithmetic operators with regard to number-range overflow is exactly the same in Java; Kotlin doesn't introduce any overhead for overflow checks.

Conversion from String

The Kotlin standard library provides a similar set of extension functions to convert a string into a primitive type (`toInt`, `toByte`, `toBoolean`, and so on):

```
>>> println("42".toInt())
42
```

www.itbook.store/books/9781617293290

Each of these functions tries to parse the contents of the string as the corresponding type and throws a NumberFormatException if the parsing fails.

Before we move on to other types, there are three more special types we need to mention: Any, Unit, and Nothing.

6.2.4 "Any" and "Any?": the root types

Similar to how Object is the root of the class hierarchy in Java, the Any type is the supertype of all non-nullable types in Kotlin. But in Java, Object is a supertype of all reference types only, and primitive types aren't part of the hierarchy. That means you have to use wrapper types such as java.lang.Integer to represent a primitive type value when Object is required. In Kotlin, Any is a supertype of all types, including the primitive types such as Int.

Just as in Java, assigning a value of a primitive type to a variable of type Any performs automatic boxing:

```
val answer: Any = 42
```

The value 42 is boxed, because Any is a reference type.

Note that Any is a non-nullable type, so a variable of the type Any can't hold the value null. If you need a variable that can hold any possible value in Kotlin, including null, you must use the Any? type.

Under the hood, the Any type corresponds to java.lang.Object. The Object type used in parameters and return types of Java methods is seen as Any in Kotlin. (More specifically, it's viewed as a platform type, because its nullability is unknown.) When a Kotlin function uses Any, it's compiled to Object in the Java bytecode.

As you saw in chapter 4, all Kotlin classes have the following three methods: toString, equals, and hashCode. These methods are inherited from Any. Other methods defined on java.lang.Object (such as wait and notify) aren't available on Any, but you can call them if you manually cast the value to java.lang.Object.

6.2.5 The Unit type: Kotlin's "void"

The Unit type in Kotlin fulfills the same function as void in Java. It can be used as a return type of a function that has nothing interesting to return:

```
fun f(): Unit { ... }
```

Syntactically, it's the same as writing a function with a block body without a type declaration:

```
fun f() { ... }
```

Explicit Unit declaration is omitted

In most cases, you won't notice the difference between void and Unit. If your Kotlin function has the Unit return type and doesn't override a generic function, it's compiled to a good-old void function under the hood. If you override it from Java, the Java function just needs to return void.

www.itbook.store/books/9781617293290

What distinguishes Kotlin's Unit from Java's void, then? Unit is a full-fledged type, and, unlike void, it can be used as a type argument. Only one value of this type exists; it's also called Unit and is returned *implicitly*. This is useful when you override a function that returns a generic parameter and make it return a value of the Unit type:

```
interface Processor<T> {
    fun process(): T
}

class NoResultProcessor : Processor<Unit> {
    override fun process() {      // Returns Unit, but you omit the type specification
        // do stuff
    }                             // You don't need an explicit return here.
}
```

The signature of the interface requires the process function to return a value; and, because the Unit type does have a value, it's no problem to return it from the method. But you don't need to write an explicit return statement in NoResultProcessor.process, because return Unit is added implicitly by the compiler.

Contrast this with Java, where neither of the possibilities for solving the problem of using "no value" as a type argument is as nice as the Kotlin solution. One option is to use separate interfaces (such as Callable and Runnable) to represent interfaces that don't and do return a value. The other is to use the special java.lang.Void type as the type parameter. If you use the second option, you still need to put in an explicit return null; to return the only possible value matching that type, because if the return type isn't void, you must always have an explicit return statement.

You may wonder why we chose a different name for Unit and didn't call it Void. The name Unit is used traditionally in functional languages to mean "only one instance," and that's exactly what distinguishes Kotlin's Unit from Java's void. We could have used the customary Void name, but Kotlin has a type called Nothing that performs an entirely different function. Having two types called Void and Nothing would be confusing because the meanings are so close. So what's this Nothing type about? Let's find out.

6.2.6 The Nothing type: "This function never returns"

For some functions in Kotlin, the concept of a "return value" doesn't make sense because they never complete successfully. For example, many testing libraries have a function called fail that fails the current test by throwing an exception with a specified message. A function that has an infinite loop in it will also never complete successfully.

When analyzing code that calls such a function, it's useful to know that the function will never terminate normally. To express that, Kotlin uses a special return type called Nothing:

```
fun fail(message: String): Nothing {
    throw IllegalStateException(message)
}
```

www.itbook.store/books/9781617293290

```
>>> fail("Error occurred")
java.lang.IllegalStateException: Error occurred
```

The `Nothing` type doesn't have any values, so it only makes sense to use it as a function return type or as a type argument for a type parameter that's used as a generic function return type. In all other cases, declaring a variable where you can't store any value doesn't make sense.

Note that functions returning `Nothing` can be used on the right side of the Elvis operator to perform precondition checking:

```
val address = company.address ?: fail("No address")
println(address.city)
```

This example shows why having `Nothing` in the type system is extremely useful. The compiler knows that a function with this return type never terminates normally and uses that information when analyzing the code calling the function. In the previous example, the compiler infers that the type of `address` is non-`null`, because the branch handling the case when it's `null` always throws an exception.

We've finished our discussion of the basic types in Kotlin: primitive types, `Any`, `Unit`, and `Nothing`. Now let's look at the collection types and how they differ from their Java counterparts.

6.3 Collections and arrays

You've already seen many examples of code that uses various collection APIs, and you know that Kotlin builds on the Java collections library and augments it with features added through extension functions. There's more to the story of the collection support in Kotlin and the correspondence between Java and Kotlin collections, and now is a good time to look at the details.

6.3.1 Nullability and collections

Earlier in this chapter, we discussed the concept of nullable types, but we only briefly touched on nullability of type arguments. But this is essential for a consistent type system: it's no less important to know whether a collection can hold `null` values than to know whether the value of a variable can be `null`. The good news is that Kotlin fully supports nullability for type arguments. Just as the type of a variable can have a `?` character appended to indicate that the variable can hold `null`, a type used as a type argument can be marked in the same way. To see how this works, let's look at an example of a function that reads a list of lines from a file and tries to parse each line as a number.

Listing 6.21 Building a collection of nullable values

```
fun readNumbers(reader: BufferedReader): List<Int?> {
    val result = ArrayList<Int?>()                    // Creates a list of nullable Int values
    for (line in reader.lineSequence()) {
        try {
            val number = line.toInt()
```

www.itbook.store/books/9781617293290

```
            result.add(number)          <- Adds an integer (a non-null value) to the list
        }
        catch(e: NumberFormatException) {
            result.add(null)            <- Adds null to the list, because the current line can't be parsed to an integer
        }
    }
    return result
}
```

List<Int?> is a list that can hold values of type Int?: in other words, Int or null. You add an integer to the result list if the line can be parsed, or null otherwise. Note that since Kotlin 1.1, you can shrink this example by using the function String.toIntOrNull, which returns null if the string value can't be parsed.

Note how the nullability of the type of the variable itself is distinct from the nullability of the type used as a type argument. The difference between a list of nullable Ints and a nullable list of Ints is illustrated in figure 6.10.

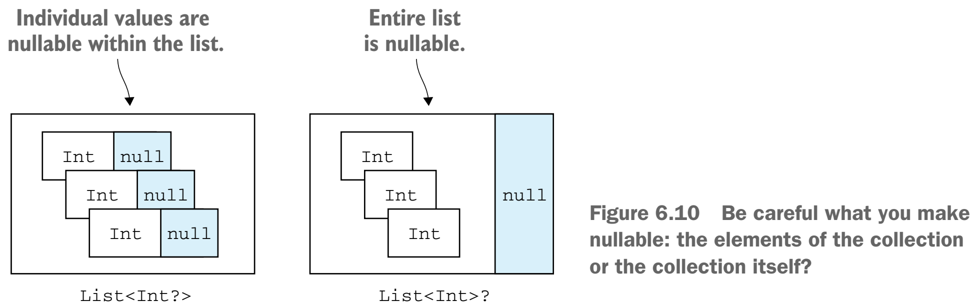

Figure 6.10 Be careful what you make nullable: the elements of the collection or the collection itself?

In the first case, the list itself is always not null, but each value in the list can be null. A variable of the second type may contain a null reference instead of a list instance, but the elements in the list are guaranteed to be non-null.

In another context, you may want to declare a variable that holds a nullable list of nullable numbers. The Kotlin way to write this is List<Int?>?, with two question marks. You need to apply null checks both when using the value of the variable and when using the value of every element in the list.

To see how you can work with a list of nullable values, let's write a function to add all the valid numbers together and count the invalid numbers separately.

Listing 6.22 Working with a collection of nullable values

```
fun addValidNumbers(numbers: List<Int?>) {
    var sumOfValidNumbers = 0
    var invalidNumbers = 0
    for (number in numbers) {           <- Reads a nullable value from the list
```

www.itbook.store/books/9781617293290

```
        if (number != null) {                        <-- Checks the
            sumOfValidNumbers += number                  value for null
        } else {
            invalidNumbers++
        }
    }
    println("Sum of valid numbers: $sumOfValidNumbers")
    println("Invalid numbers: $invalidNumbers")
}

>>> val reader = BufferedReader(StringReader("1\nabc\n42"))
>>> val numbers = readNumbers(reader)
>>> addValidNumbers(numbers)
Sum of valid numbers: 43
Invalid numbers: 1
```

There isn't much special going on here. When you access an element of the list, you get back a value of type `Int?`, and you need to check it for `null` before you can use it in arithmetical operations.

Taking a collection of nullable values and filtering out `null` is such a common operation that Kotlin provides a standard library function `filterNotNull` to perform it. Here's how you can use it to greatly simplify the previous example.

Listing 6.23 Using `filterNotNull` with a collection of nullable values

```
fun addValidNumbers(numbers: List<Int?>) {
    val validNumbers = numbers.filterNotNull()
    println("Sum of valid numbers: ${validNumbers.sum()}")
    println("Invalid numbers: ${numbers.size - validNumbers.size}")
}
```

Of course, the filtering also affects the type of the collection. The type of `validNumbers` is `List<Int>`, because the filtering ensures that the collection doesn't contain any `null` elements.

Now that you understand how Kotlin distinguishes between collections that hold nullable and non-`null` elements, let's look at another major distinction introduced by Kotlin: read-only versus mutable collections.

6.3.2 Read-only and mutable collections

An important trait that sets apart Kotlin's collection design from Java's is that it separates interfaces for accessing the data in a collection and for modifying the data. This distinction exists starting with the most basic interface for working with collections, `kotlin.collections.Collection`. Using this interface, you can iterate over the elements in a collection, obtain its size, check whether it contains a certain element, and perform other operations that read data from the collection. But this interface doesn't have any methods for adding or removing elements.

To modify the data in the collection, use the `kotlin.collections.MutableCollection` interface. It extends the regular `kotlin.collections.Collection` and

www.itbook.store/books/9781617293290

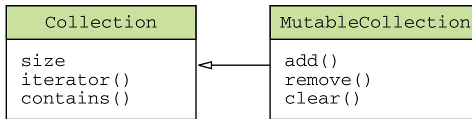

Figure 6.11 `MutableCollection` **extends** `Collection` **and adds methods to modify a collection's contents.**

provides methods for adding and removing the elements, clearing the collection, and so on. Figure 6.11 shows the key methods defined in the two interfaces.

As a general rule, you should use read-only interfaces everywhere in your code. Use the mutable variants only if the code will modify the collection.

Just like the separation between `val` and `var`, the separation between read-only and mutable interfaces for collections makes it much easier to understand what's happening with data in your program. If a function takes a parameter that is a `Collection` but not a `MutableCollection`, you know it's not going to modify the collection, but only read data from it. And if a function requires you to pass a `MutableCollection`, you can assume that it's going to modify the data. If you have a collection that's part of the internal state of your component, you may need to make a copy of that collection before passing it to such a function. (This pattern is usually called a *defensive copy*.)

For example, you can clearly see that the following `copyElements` function will modify the target collection but not the source collection.

Listing 6.24 Using read-only and mutable collection interfaces

```
fun <T> copyElements(source: Collection<T>,
                     target: MutableCollection<T>) {
    for (item in source) {              // Loops over all items in the source collection
        target.add(item)                // Adds items to the mutable target collection
    }
}

>>> val source: Collection<Int> = arrayListOf(3, 5, 7)
>>> val target: MutableCollection<Int> = arrayListOf(1)
>>> copyElements(source, target)
>>> println(target)
[1, 3, 5, 7]
```

You can't pass a variable of a read-only collection type as the `target` argument, even if its value is a mutable collection:

```
>>> val source: Collection<Int> = arrayListOf(3, 5, 7)
>>> val target: Collection<Int> = arrayListOf(1)
>>> copyElements(source, target)        // Error on the "target" argument
Error: Type mismatch: inferred type is Collection<Int>
  but MutableCollection<Int> was expected
```

A key thing to keep in mind when working with collection interfaces is that *read-only collections aren't necessarily immutable.*[1] If you're working with a variable that has a

[1] Immutable collections are planned to be added to the Kotlin standard library later.

www.itbook.store/books/9781617293290

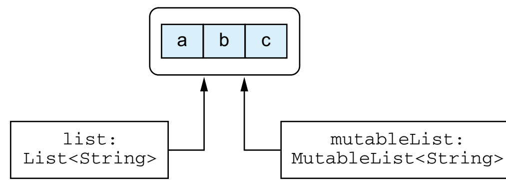

Figure 6.12 Two different references, one read-only and one mutable, pointing to the same collection object

read-only interface type, this can be just one of the many references to the same collection. Other references can have a mutable interface type, as illustrated in figure 6.12.

If you call the code holding the other reference to your collection or run it in parallel, you can still come across situations where the collection is modified by other code while you're working with it, which leads to `ConcurrentModificationException` errors and other problems. Therefore, it's essential to understand that *read-only collections aren't always thread-safe*. If you're working with data in a multithreaded environment, you need to ensure that your code properly synchronizes access to the data or uses data structures that support concurrent access.

How does the separation between read-only and mutable collections work? Didn't we say earlier that Kotlin collections are the same as Java collections? Isn't there a contradiction? Let's see what really happens here.

6.3.3 Kotlin collections and Java

It's true that every Kotlin collection is an instance of the corresponding Java collection interface. No conversion is involved when moving between Kotlin and Java; there's no need for wrappers or copying data. But every Java collection interface has two *representations* in Kotlin: a read-only one and a mutable one, as you can see in figure 6.13.

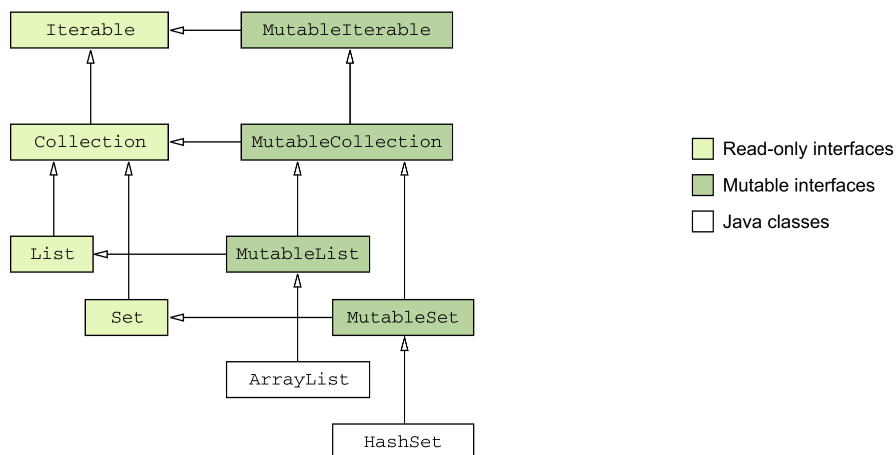

Figure 6.13 The hierarchy of the Kotlin collection interfaces. The Java classes `ArrayList` and `HashSet` extend Kotlin mutable interfaces.

www.itbook.store/books/9781617293290

All collection interfaces shown in figure 6.13 are declared in Kotlin. The basic structure of the Kotlin read-only and mutable interfaces is parallel to the structure of the Java collection interfaces in the `java.util` package. In addition, each mutable interface extends the corresponding read-only interface. Mutable interfaces correspond directly to the interfaces in the `java.util` package, whereas the read-only versions lack all the mutating methods.

Figure 6.13 also contains the Java classes `java.util.ArrayList` and `java.util.HashSet` to show how Java standard classes are treated in Kotlin. Kotlin sees them as if they inherited from the Kotlin's `MutableList` and `MutableSet` interfaces, respectively. Other implementations from the Java collection library (`LinkedList`, `SortedSet`, and so on) aren't presented here, but from the Kotlin perspective they have similar supertypes. This way, you get both compatibility and clear separation of mutable and read-only interfaces.

In addition to the collections, the `Map` class (which doesn't extend `Collection` or `Iterable`) is also represented in Kotlin as two distinct versions: `Map` and `MutableMap`. Table 6.1 shows the functions you can use to create collections of different types.

Table 6.1 Collection-creation functions

Collection type	Read-only	Mutable
List	`listOf`	`mutableListOf`, `arrayListOf`
Set	`setOf`	`mutableSetOf`, `hashSetOf`, `linkedSetOf`, `sortedSetOf`
Map	`mapOf`	`mutableMapOf`, `hashMapOf`, `linkedMapOf`, `sortedMapOf`

Note that `setOf()` and `mapOf()` return instances of classes from the Java standard library (at least in Kotlin 1.0), which are all mutable under the hood.[2] But you shouldn't rely on that: it's possible that a future version of Kotlin will use truly immutable implementation classes as return values of `setOf` and `mapOf`.

When you need to call a Java method and pass a collection as an argument, you can do so directly without any extra steps. For example, if you have a Java method that takes a `java.util.Collection` as a parameter, you can pass any `Collection` or `MutableCollection` value as an argument to that parameter.

This has important consequences with regard to mutability of collections. Because Java doesn't distinguish between read-only and mutable collections, Java code *can modify the collection* even if it's declared as a read-only `Collection` on the Kotlin side. The Kotlin compiler can't fully analyze what's being done to the collection in the Java code, and therefore there's no way for Kotlin to reject a call passing a read-only `Collection` to Java code that modifies it. For example, the following two snippets of code form a compilable cross-language Kotlin/Java program:

[2] Wrapping things into `Collection.unmodifiable` introduces indirection overhead, so it's not done.

www.itbook.store/books/9781617293290

```
/* Java */
// CollectionUtils.java
public class CollectionUtils {
    public static List<String> uppercaseAll(List<String> items) {
        for (int i = 0; i < items.size(); i++) {
            items.set(i, items.get(i).toUpperCase());
        }
        return items;
    }
}

// Kotlin
// collections.kt
fun printInUppercase(list: List<String>) {        <- Declares a read-only parameter
    println(CollectionUtils.uppercaseAll(list))     <- Calls a Java function that modifies the collection
    println(list.first())                           <- Shows that the collection has been modified
}

>>> val list = listOf("a", "b", "c")
>>> printInUppercase(list)
[A, B, C]
A
```

Therefore, if you're writing a Kotlin function that takes a collection and passes it to Java, *it's your responsibility to use the correct type for the parameter*, depending on whether the Java code you're calling will modify the collection.

Note that this caveat also applies to collections with non-`null` element types. If you pass such a collection to a Java method, the method can put a `null` value into it; there's no way for Kotlin to forbid that or even to detect that it has happened without compromising performance. Because of that, you need to take special precautions when you pass collections to Java code that can modify them, to make sure the Kotlin types correctly reflect all the possible modifications to the collection.

Now, let's take a closer look at how Kotlin deals with collections declared in Java code.

6.3.4 Collections as platform types

If you recall the discussion of nullability earlier in this chapter, you'll remember that types defined in Java code are seen as *platform types* in Kotlin. For platform types, Kotlin doesn't have the nullability information, so the compiler allows Kotlin code to treat them as either nullable or non-`null`. In the same way, variables of collection types declared in Java are also seen as platform types. A collection with a platform type is essentially a collection of unknown mutability—the Kotlin code can treat it as either read-only or mutable. Usually this doesn't matter, because, in effect, all the operations you may want to perform just work.

The difference becomes important when you're overriding or implementing a Java method that has a collection type in its signature. Here, as with platform types for nullability, you need to decide which Kotlin type you're going to use to represent a Java type coming from the method you're overriding or implementing.

www.itbook.store/books/9781617293290

You need to make multiple choices in this situation, all of which will be reflected in the resulting parameter type in Kotlin:

- Is the collection nullable?
- Are the elements in the collection nullable?
- Will your method modify the collection?

To see the difference, consider the following cases. In the first example, a Java interface represents an object that processes text in a file.

Listing 6.25 A Java interface with a collection parameter

```
/* Java */
interface FileContentProcessor {
    void processContents(File path,
        byte[] binaryContents,
        List<String> textContents);
}
```

A Kotlin implementation of this interface needs to make the following choices:

- The list will be nullable, because some files are binary and their contents can't be represented as text.
- The elements in the list will be non-`null`, because lines in a file are never `null`.
- The list will be read-only, because it represents the contents of a file, and those contents aren't going to be modified.

Here's how this implementation looks.

Listing 6.26 Kotlin implementation of `FileContentProcessor`

```
class FileIndexer : FileContentProcessor {
    override fun processContents(path: File,
        binaryContents: ByteArray?,
        textContents: List<String>?) {

        // ...
    }
}
```

Contrast this with another interface. Here the implementations of the interface parse some data from a text form into a list of objects, append those objects to the output list, and report errors detected when parsing by adding the messages to a separate list.

Listing 6.27 Another Java interface with a collection parameter

```
/* Java */
interface DataParser<T> {
    void parseData(String input,
        List<T> output,
        List<String> errors);
}
```

www.itbook.store/books/9781617293290

The choices in this case are different:

- `List<String>` will be non-`null`, because the callers always need to receive error messages.
- The elements in the list will be nullable, because not every item in the output list will have an associated error message.
- `List<String>` will be mutable, because the implementing code needs to add elements to it.

Here's how you can implement that interface in Kotlin.

Listing 6.28 Kotlin implementation of `DataParser`

```
class PersonParser : DataParser<Person> {
    override fun parseData(input: String,
        output: MutableList<Person>,
        errors: MutableList<String?>) {

        // ...
    }
}
```

Note how the same Java type—`List<String>`—is represented by two different Kotlin types: a `List<String>?` (nullable list of strings) in one case and a `MutableList<String?>` (mutable list of nullable strings) in the other. To make these choices correctly, you must know the exact contract the Java interface or class needs to follow. This is usually easy to understand based on what your implementation needs to do.

Now that we've discussed collections, it's time to look at arrays. As we've mentioned before, you should prefer using collections to arrays by default. But because many Java APIs still use arrays, we'll cover how to work with them in Kotlin.

6.3.5 Arrays of objects and primitive types

The syntax of Kotlin arrays appears in every example, because an array is part of the standard signature of the Java `main` function. Here's a reminder of how it looks:

Listing 6.29 Using arrays

```
fun main(args: Array<String>) {
    for (i in args.indices) {          // Uses the array.indices extension property to iterate over the range of indices
        println("Argument $i is: ${args[i]}")   // Accesses elements by index with array[index]
    }
}
```

An array in Kotlin is a class with a type parameter, and the element type is specified as the corresponding type argument.

www.itbook.store/books/9781617293290

To create an array in Kotlin, you have the following possibilities:

- The `arrayOf` function creates an array containing the elements specified as arguments to this function.
- The `arrayOfNulls` function creates an array of a given size containing `null` elements. Of course, it can only be used to create arrays where the element type is nullable.
- The `Array` constructor takes the size of the array and a lambda, and initializes each array element by calling the lambda. This is how you can initialize an array with a non-`null` element type without passing each element explicitly.

As a simple example, here's how you can use the `Array` function to create an array of strings from `"a"` to `"z"`.

Listing 6.30 Creating an array of characters

```
>>> val letters = Array<String>(26) { i -> ('a' + i).toString() }
>>> println(letters.joinToString(""))
abcdefghijklmnopqrstuvwxyz
```

The lambda takes the index of the array element and returns the value to be placed in the array at that index. Here you calculate the value by adding the index to the `'a'` character and converting the result to a string. The array element type is shown for clarity; you can omit it in real code because the compiler can infer it.

Having said that, one of the most common cases for creating an array in Kotlin code is when you need to call a Java method that takes an array, or a Kotlin function with a `vararg` parameter. In those situations, you often have the data already stored in a collection, and you just need to convert it into an array. You can do this using the `toTypedArray` method.

Listing 6.31 Passing a collection to a `vararg` method

```
>>> val strings = listOf("a", "b", "c")
>>> println("%s/%s/%s".format(*strings.toTypedArray()))
a/b/c
```

The spread operator (*) is used to pass an array when vararg parameter is expected.

As with other types, *type arguments of array types always become object types*. Therefore, if you declare something like an `Array<Int>`, it will become an array of boxed integers (its Java type will be `java.lang.Integer[]`). If you need to create an array of values of a primitive type without boxing, you must use one of the specialized classes for arrays of primitive types.

To represent arrays of primitive types, Kotlin provides a number of separate classes, one for each primitive type. For example, an array of values of type `Int` is called `IntArray`. For other types, Kotlin provides `ByteArray`, `CharArray`,

www.itbook.store/books/9781617293290

`BooleanArray`, and so on. All of these types are compiled to regular Java primitive type arrays, such as `int[]`, `byte[]`, `char[]`, and so on. Therefore, values in such an array are stored without boxing, in the most efficient manner possible.

To create an array of a primitive type, you have the following options:

- The constructor of the type takes a `size` parameter and returns an array initialized with default values for the corresponding primitive type (usually zeros).
- The factory function (`intArrayOf` for `IntArray`, and so on for other array types) takes a variable number of values as arguments and creates an array holding those values.
- Another constructor takes a size and a lambda used to initialize each element.

Here's how the first two options work for creating an integer array holding five zeros:

```
val fiveZeros = IntArray(5)
val fiveZerosToo = intArrayOf(0, 0, 0, 0, 0)
```

Here's how you can use the constructor accepting a lambda:

```
>>> val squares = IntArray(5) { i -> (i+1) * (i+1) }
>>> println(squares.joinToString())
1, 4, 9, 16, 25
```

Alternatively, if you have an array or a collection holding boxed values of a primitive type, you can convert them to an array of that primitive type using the corresponding conversion function, such as `toIntArray`.

Next, let's look at some of the things you can do with arrays. In addition to the basic operations (getting the array's length and getting and setting elements), the Kotlin standard library supports the same set of extension functions for arrays as for collections. All the functions you saw in chapter 5 (`filter`, `map`, and so on) work for arrays as well, including the arrays of primitive types. (Note that the return values of these functions are lists, not arrays.)

Let's see how to rewrite listing 6.30 using the `forEachIndexed` function and a lambda. The lambda passed to that function is called for each element of the array and receives two arguments, the index of the element and the element itself.

Listing 6.32 Using `forEachIndexed` with an array

```
fun main(args: Array<String>) {
    args.forEachIndexed { index, element ->
        println("Argument $index is: $element")
    }
}
```

Now you know how to use arrays in your code. Working with them is as simple as working with collections in Kotlin.

www.itbook.store/books/9781617293290

6.4 Summary

- Kotlin's support of nullable types detects possible `NullPointerException` errors at compile time.
- Kotlin provides tools such as safe calls (`?.`), the Elvis operator (`?:`), not-`null` assertions (`!!`), and the `let` function for dealing with nullable types concisely.
- The `as?` operator provides an easy way to cast a value to a type and to handle the case when it has a different type.
- Types coming from Java are interpreted as platform types in Kotlin, allowing the developer to treat them as either nullable or non-`null`.
- Types representing basic numbers (such as `Int`) look and function like regular classes but are usually compiled to Java primitive types.
- Nullable primitive types (such as `Int?`) correspond to boxed primitive types in Java (such as `java.lang.Integer`).
- The `Any` type is a supertype of all other types and is analogous to Java's `Object`. `Unit` is an analogue of `void`.
- The `Nothing` type is used as a return type of functions that don't terminate normally.
- Kotlin uses the standard Java classes for collections and enhances them with a distinction between read-only and mutable collections.
- You need to carefully consider nullability and mutability of parameters when you extend Java classes or implement Java interfaces in Kotlin.
- Kotlin's `Array` class looks like a regular generic class, but it's compiled to a Java array.
- Arrays of primitive types are represented by special classes such as `IntArray`.

www.itbook.store/books/9781617293290

JAVA/PROGRAMMING LANGUAGES

Kotlin IN ACTION

Jemerov • Isakova

Developers want to get work done—and the less hassle, the better. Coding with Kotlin means less hassle. The Kotlin programming language offers an expressive syntax, a strong intuitive type system, and great tooling support along with seamless interoperability with existing Java code, libraries, and frameworks. Kotlin can be compiled to Java bytecode, so you can use it everywhere Java is used, including Android. And with an efficient compiler and a small standard library, Kotlin imposes virtually no runtime overhead.

Kotlin in Action teaches you to use the Kotlin language for production-quality applications. Written for experienced Java developers, this example-rich book goes further than most language books, covering interesting topics like building DSLs with natural language syntax. The authors are core Kotlin developers, so you can trust that even the gnarly details are dead accurate.

What's Inside

- Functional programming on the JVM
- Writing clean and idiomatic code
- Combining Kotlin and Java
- Domain-specific languages

This book is for experienced Java developers.

Dmitry Jemerov and **Svetlana Isakova** are core Kotlin developers at JetBrains.

“Explains high-level concepts and provides all the necessary details as well.”

—From the Foreword by Andrey Breslav Lead Designer of Kotlin

“Like all the other great *in Action* titles from Manning, this book gives you everything you need to become productive quickly.”

—Kevin Orr, Sumus Solutions

“Kotlin is fun and easy to learn when you have this book to guide you!”

—Filip Pravica, Info.nl

“Thorough, well written, and easily accessible.”

—Jason Lee, NetSuite

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/kotlin-in-action

ISBN-13: 978-1-61729-329-0
ISBN-10: 1-61729-329-6

$44.99 / Can $51.99 [INCLUDING eBOOK]

www.itbook.store/books/9781617293290