

Less Typing

We'll cover the following

- Semicolons are optional
- Variable type specification is optional
- Classes and functions are optional
- try-catch is optional

You'll do less typing—that act of hammering keys on the keyboard—to create applications with Kotlin. That's because a lot of things we've come to take for granted as required are optional in Kotlin.

Semicolons are optional

When you start programming in Kotlin, your right pinky immediately gets relief from repetitive stress injury that it may have endured for most of your programming career. Though you could, you don't have to end every single statement or expression with a semicolon. Use the semicolon sparingly—only on occasions when you want to place two or more expressions or statements on a single line.

The following example, is valid syntax in Kotlin, and can be written standalone:

```
6 * 2
```

At first thought it may not appear to be a big deal, but, as you'll see later in the book, not having to place semicolons makes the code fluent, especially when creating internal DSLs.

If you're transitioning into Kotlin from languages like Java and JavaScript, chances are that hitting the semicolon key is an involuntary action by now. That's understandable, but make it a habit to leave out semicolons when writing in Kotlin.

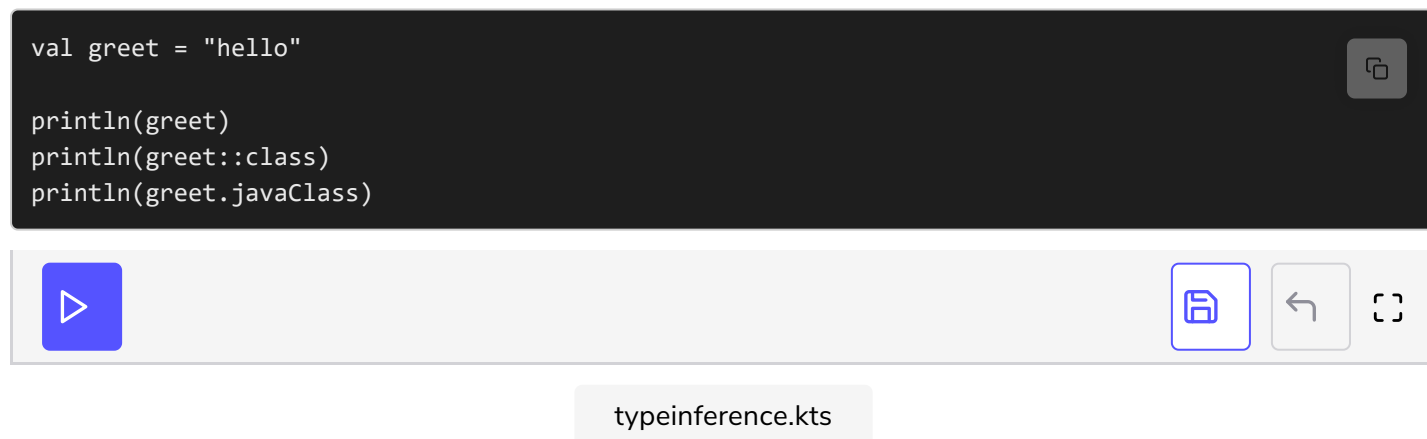
Variable type specification is optional

Kotlin is statically typed, but that doesn't mean you have to specify the details of variables' types. Static typing means that the type of variables is verified and type sanity is guaranteed at compile time.

Kotlin has the smarts—type inference—to determine the type of variables based on the context. Let's define a variable without specifying the type and then ask for the type of that variable.

```
val greet = "hello"

println(greet)
println(greet::class)
println(greet.javaClass)
```



typeinference.kts

The `::class` call is asking for the Kotlin class of the object referenced by the variable. The `.javaClass` call, however, is asking for the underlying Java class. It's rare for Kotlin and Java classes to be different—only classes that are intimately known to the Kotlin compiler will display such differences.

In the previous example, Kotlin's type-inference capability determines that the type of `greet` is `String` based on the value assigned to it. The output shows these details:

```
hello
class kotlin.String
class java.lang.String
```

Some developers fear type inference; they wonder if it's a runtime thing and if it somehow lowers the effectiveness of type checking at compile time. The short answer is no.

To be fair, the above code revealed the type of the referenced object at runtime, but what's the type of the variable `greet` at compile time? We can find that by making a mistake in code, like so:

```
val greet = "hello"
```

```
println(greet)
```

```
greet = 0
```



typechecking.kts

Kotlin determined that the type of `greet` is `String` at compile time and, as a result, knew that assigning an integer to it isn't valid. In addition, reassigning a `val` isn't permitted. So the code didn't execute and resulted in compilation errors, even though it was run as script. The resulting compilation errors are:

```
typechecking.kts:5:1: error: val cannot be reassigned
greet = 0
^
typechecking.kts:5:9: error: the integer literal does not conform
to the expected type String
greet = 0
      ^
```

Kotlin doesn't take type inference to the extreme—it permits leaving out the type details only in places where the type is obvious. When defining functions and methods, you're required to specify the type of the parameters, although you may leave out the type of the return. In general, specify the return type for APIs that aren't internal to your libraries but are visible to the outside user. We'll discuss this further when we explore creating functions.

For your part, encourage your colleagues to give meaningful names for variables so it becomes easier to identify the type and the intent of variables. For example,

`val taxRate = 0.08` is better than `val t = 0.08`.

Also, when using type inference, resist the urge to embed the type information into variable names—such efforts are the programmers' desire to overly compensate for not specifying the type that they're so used to providing. For example, avoid the following:

```
val taxRateDouble = 0.08 //Don't do this
//or
val dTaxRate = 0.08 //Also, don't do this
```

Local variables are internal and aren't seen by the users of your code. So the use of type inference doesn't take away any details from the users of your functions. Leave out the type details where possible and instead use type inference with descriptive, but not necessarily long, names for variables.

Classes and functions are optional

Unlike languages like Java, Kotlin doesn't require a statement or expression to belong to a method and a method to belong to a class, at least not in the source code we write. When the code is compiled, or executed as script, Kotlin will create wrapper classes and methods as necessary to satisfy the JVM expectations.

In the following source code, the function doesn't belong to a class, and the code below the function is standalone and isn't part of any function. Nevertheless, Kotlin takes care of wrapping these, when necessary, into classes to satisfy the JVM.

Let's create a script with a function that doesn't belong to any class and some standalone code that's not part of any function.

```
fun nofluff() {
    println("nofluff called...")

    throw RuntimeException("oops")
}

println("not in a function, calling nofluff()")

try {
    nofluff()
} catch (ex: Exception) {
    val stackTrace = ex.getStackTrace()
    println(stackTrace[0])
    println(stackTrace[1])
}
```



standalone.kts

The standalone body of code below the function calls the function `nofluff()`, which doesn't belong to any class. The function blows up with an exception, and the calling code prints the top two frames of the stack from the exception. The output from this code shows that, first, Kotlin doesn't force us to write classes and

methods, and, second, it wraps the code into a class automatically.

```
not in a function, calling nofluff()
nofluff called...
Standalone.nofluff(standalone.kts:4)
Standalone.<init>(standalone.kts:10)
```

Kotlin quietly turned the function `nofluff()` into a method of a synthesized class named `Standalone`—a name inferred from the file name—and the standalone code into the constructor of the class, as indicated by `<init>` in the output.

When writing small pieces of code, place the code directly in a file and run it as a script—no need for the ceremony to create classes and methods. But when working on larger applications, you may create classes and methods. Simple code can be simple, and more complex code can have better rigor and structure.

try-catch is optional

The Java compiler forces us to either explicitly catch or propagate checked exceptions. The debate about whether checked exceptions are good or bad may never be settled—some developers love it while others hate it. We don't need to get dragged into that brawl; instead let's focus on what Kotlin provides.

Kotlin doesn't force you to catch any exceptions—checked or unchecked. If you don't place a `try-catch` around a function call and if that function blows up, the exception is automatically propagated to the caller of your function or code. If an exception is unhandled, it'll result in a fateful termination of your program.

In Java, for example, the `sleep()` method of the `Thread` class throws a checked exception, and the compiler forces us to deal with it. As a result, any call to `sleep()` has to be surrounded by a try and followed by a sleepless night wondering what to do with that stinking `InterruptedException` that may potentially be thrown from that call. No need to lose sleep over such issues in Kotlin:

```
println("Lemme take a nap")
Thread.sleep(1000)
println("ah that feels good")
```



The code doesn't have any `try` and `catch`, but when executed it will print the two lines of output with a one-second delay after the first line is printed.

It's a good practice to program defensively to handle exceptions. At the same time, since Kotlin doesn't force a try-catch on us, we're not tempted to place those gnarly empty catch blocks that many Java programmers seem to write simply to quiet the Java compiler. Remember, what you don't handle is automatically propagated to the caller.

You've seen how the Kotlin compiler gives you a lot of flexibility by placing fewer demands. At the same time, the compiler looks out for potential errors in code, to make the code safer as we'll see in the next lesson.
