Table of Contents

Introduction

What is Kotlin?

Kotlin is a programming language developed by JetBrains, the creators of the IntelliJ Java IDE.

It is built on the foundations of the JVM and is highly interoperable with Java code. Many aspects of Kotlin (generics, arrays, etc.) therefore function almost identically to how they do in Java.

Why should I use Kotlin?

Kotlin is many things:

It is cross platform. Kotlin's current compilation targets are:

- JVM bytecode
- JavaScript
- Native binaries via LLVM:
  - iOS (arm32, arm64)
  - macOS (x86_64)
  - watchOS (x86_64, arm32, arm64)
  - tvOS (x86_64, arm64)
  - Android (x86, x86_64, arm32, arm64)
  - Windows (MinGW x86_64, x86)
  - Linux (x86_64, arm32, MIPS, RPi)
  - WebAssembly (wasm32)

It is multi-paradigm. Kotlin supports a mix of functional and object-oriented programming to achieve maximum productivity.

Its expansive standard library provides hundreds of utilities, ranging from basic

helpers to entire suites of functions that, in order to work properly in other, less-flexible languages, would have had to be fully-fledged language features; the previous sentence may or may not be a not-so-subtle jab at C# and LINQ.

It is flexible. Kotlin is a great fit for domain-specific languages (DSLs) and has great support for being used as an embedded scripting language — in fact, the Gradle buildscript used to create this document was itself written in Kotlin.

How do I write Kotlin?

Language idioms

Kotlin was originally developed as a JVM language, and has thus been considerably influenced by the Java world. While this means that some fragments of may be translated line-by-line into Kotlin, this often does not result in optimal Kotlin code.

Take the following example:

```
final var person = ...;

if (person != null) {
    final var addresses = person.getAddresses();
    if (addresses != null && !addresses.isEmpty()) {
        final var firstAddress = addresses.get(0);
        if (firstAddress != null) {
            final var street = firstAddress.getStreet();
            if (street != null) {
                final var streetName = street.getName();
                if (streetName != null) {
                    ...
                }
            }
        }
    }
}
```

One might argue that this Java antipattern is obsolete, due to the advent of Java 8 and its Optional utility class. Let us give this sample a more modern twist:

```java
final var person = ...;

Optional.ofNullable(person)
        .flatMap(Person::getAddresses)
        .flatMap(as -> as.stream().findFirst())
        .flatMap(Address::getStreet)
        .flatMap(Street::getName)
        .ifPresent(streetName -> {
            ...
        });
```

While this is much more concise, consider the overhead of wrapping every value that *may or may not* have a value with an extra object that:

- must be allocated on the stack (or heap)
- incurs a runtime performance penalty for each of the chained method calls
- results in code that is non-trivial to read

Not only is this be an unnecessarily complex approach to handling Tony Hoare's billion-dollar mistake, there exist *core language features* in Kotlin designed to avoid this problem entirely:

```kotlin
val person = ...

person?.addresses?.firstOrNull()?.street?.name?.let {
    ...
}
```

In short: to write proper, idiomatic Kotlin, you will *need* to ignore your instincts and find new approaches to solving problems. Pretending to write Kotlin when you are doing nothing more than translating the Java you *would've* written will not result in good code.

IntelliJ's built-in Java-to-Kotlin converter (*Ctrl+Alt+Shift+K*) is often a good start; its conversions are relatively decent due to the IDE's advanced static analysis capabilities. However, one should not rely solely on this tool — it will often produce suboptimal (or even invalid) code.