Denis Kalinin

Modern Web Development with Kotlin

Modern Web Development with Kotlin

A concise and practical step-by-step guide

Denis Kalinin

This book is for sale at http://leanpub.com/modern-web-development-with-kotlin

This version was published on 2016-09-28

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

Contents

Preface

On 18 March 2014, Java 8 was finally released, and this event marked an important point in the history of programming languages. Java had finally got function literals! There were many other improvements, and all of this basically put to rest myths about Java being outdated. It was obvious to everyone that this release was going to be a tremendous success, and the language landscape was changing for good.

Interestingly, while presenting an upcoming version, Brian Goetz, the Java language architect, noted that even with all these shiny new features, the language still felt like Java. It was a relief for many developers who didn't want to feel at a loss with once a very familiar language. At the same time, it meant that for numerous Web developers, Java was going to stay something relatively heavyweight or even alien. As a result, even now, former Ruby or PHP programmers are much more likely to switch to Scala, another statically-typed JVM language, than Java 8.

So, what about Kotlin? It doesn't seem to be eager to introduce its own build tools or frameworks. Instead, it aims at reusing existing Java libraries and integrating into already established workflows. In particular, throughout this book, we will be using Gradle - a popular build tool initially developed for Java. Generally speaking, Kotlin is quite flexible when it comes to build tools, so you can use Maven or even Ant if you need to.

As a new language developed literally from scratch, Kotlin is also free from legacy problems. As a result, there are quite a few things that it does differently than Java even in supposedly similar situations.

For example, the `if` statement in Java allows to make the control flow depend on a certain condition:

```
1  List<Integer> ints = null;
2  if (checkCondition()) {
3    ints = generateList();
4  } else {
5    ints = Collections.emptyList();
6  }
```

Since `if` statements don't return any values, we have to define a variable and put the assignments in both branches. In Kotlin, `if` is an expression, so `ints` can be defined as a constant and initialized immediately:

```
val ints = if (checkCondition()) {
  generateList()
} else {
  emptyList()
}
```

As an additional bonus, another Kotlin feature called *type inference* liberates us from declaring `ints` as `List<Integer>`.

Similarly, `try/catch` blocks in Java don't return any values, so you have to declare necessary variables before these blocks. Add the requirement to always catch or throw checked exceptions, and you'll get a recipe for crafting complex and difficult to maintain code. As you will see later in this book, with Kotlin you don't actually need to use `try/catch` blocks at all! Instead, it's usually better to wrap dangerous logic and return an `Either` object:

```
val resultT = eitherTry {
  // invoke some dangerous code
}
```

This is only a fraction of features which enable Kotlin developers to write concise and easily readable code. Of course, we will be discussing these features in detail throughout the book.

Since Kotlin tries to fit into the existing Java ecosystem, I decided to take Vert.x - a primarily Java framework - and use it for developing our sample Web application. Basing our narrative on Vert.x[1] allows us to experience first-hand what it is like to use existing libraries in Kotlin.

Vert.x, however, is quite an unusual beast and differs from traditional frameworks in multiple ways.

First, it's completely asynchronous and in this respect similar to NodeJS. Instead of blocking the computation until the result becomes available, Vert.x uses callbacks:

```
router.get("/").handler { routingContext ->
  routingContext.response().end("Hello World")
}
```

As a modern programming language, Kotlin comes with a first-class support for lambdas (function literals) and therefore, allows to work with Vert.x in a pleasant, almost DSL-like way.

Second, Vert.x is not based on Servlets, which makes it a good choice for people without Java background, especially Web developers. The fact that Vert.x comes with an embedded Web server enables simple deployment and easy integration with frontend tools.

Writing a Web application usually involves using many additional tools. While Vert.x provides a lot out of the box, it doesn't provide everything we need. As a result, throughout the book, we will be

[1] http://vertx.io/

relying on popular third-party libraries. For instance, we will be using funKTionale for functional programming, Kovenant for writing asynchronous code, Flyway for migrations and so on.

Since Kotlin is primarily a backend language, most readers are probably more familiar with the server side of things. To make the book even more useful for them, I decided to include an entire chapter devoted to frontend integration. In this chapter, you will see how to build an effective workflow that includes Webpack, EcmaScript 6 and Sass.

I hope that this little introduction gave you several more reasons to learn Kotlin, so turn the page and let's get started!

Build tools

Strictly speaking, you don't need to download the Kotlin compiler to build Kotlin code. However, manually compiling source files could be a good exercise to get a better understanding of how things work, and what more sophisticated tools do behind the scenes. In this section, we're going to look at different approaches to building Kotlin projects, starting with the ubiquitous command line.

Command line

The command line tools provide basic functionality for building Kotlin applications. In order to check that they were installed correctly, simply invoke the following command:

```
$ kotlinc -version
info: Kotlin Compiler version 1.0.3
```

kotlinc

`kotlinc` is the Kotlin Compiler, i.e. a tool that is used to compile source code into bytecode. In order to test the compiler, let's create a simple file called `Hello.kt` with the following contents:

```
fun main(args: Array<String>) {
  println("Hello World!")
}
```

The `main` method, just as in many other languages, is the starting point of the application. Unlike Java or Scala, however, in Kotlin you don't put this method into a class or object, but instead make it standalone. Once we have our file, we can compile it:

```
$ kotlinc Hello.kt
```

There will be no output, but `kotlinc` will create a new file in the current directory:

```
$ ls
Hello.kt  HelloKt.class
```

Note that a new file is named `HelloKt.class`, not `Hello.class`.

kotlin

`kotlin` is the Kotlin interpreter. If it is invoked without arguments, it will complain that it needs at least one file to run. If we want to execute the `main` method of the recently compiled example, we need to type the following:

```
$ kotlin HelloKt
Hello World!
```

Now that we understand how it works on the low level, let's look at something more interesting.

Gradle

Gradle is one of the most popular build tools for Java, but it can also be used for building Groovy, Scala and even C++ projects. It combines best parts of *Ant* and *Maven* (previous generation build tools) and uses a Groovy-based DSL for describing the build. Not surprisingly, Gradle can also handle Kotlin projects via the "Gradle Script Kotlin" plugin.

You can download a binary distribution from from the official website[2]. The latest version at the time of this writing is 3.0 and it works perfectly well with Kotlin.

After downloading the zip archive, you can install it anywhere (for example, to `~/DevTools/gradle`) and then add the bin directory to your path. A good way to do it is to edit your `.bashrc`:

```
GRADLE_HOME=/home/user/DevTools/gradle
PATH=$JAVA_HOME/bin:$KOTLIN_HOME/bin:$GRADLE_HOME/bin:$PATH
```

To try Gradle, let's create a new directory called `gradle-kotlin` and initialize a new Gradle project there:

```
$ cd gradle-kotlin
$ gradle init --type java-library
Starting a new Gradle Daemon for this build (subsequent builds will be faster).
:wrapper
:init

BUILD SUCCESSFUL

Total time: 5.563 secs
```

The `init` task initializes a skeleton project by creating several files and directories including `build.gradle`. Since we're not interested in learning Java, remove both `src/main/java` and `src/test` directories and copy our `Hello.kt` to `src/main/kotlin`:

[2] https://gradle.org/gradle-download/

```
$ rm -rf src/main/java
$ rm -rf src/test
$ mkdir -p src/main/kotlin
$ cp ../Hello.kt src/main/kotlin/
```

Finally, remove everything from the `gradle.build` file and add the following:

```
buildscript {
  ext.kotlin_version = '1.0.3'

  repositories {
    jcenter()
  }

  dependencies {
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
  }
}

apply plugin: 'kotlin'

repositories {
  jcenter()
}

dependencies {
  compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

The `buildscript` part adds support for Kotlin projects via the `kotlin-gradle-plugin`. Here we're also specifying the language version for later reference. Note that we're also adding the Kotlin standard library to the list of compile dependencies.

Many projects use `mavenCental()` as the main repository for dependencies. Here, however, we're using `jcenter()` as a faster and more secure option.

If we want to build the project, we need to type:

```
$ gradle build
```

Gradle will put the compiled class files into `build/classes/main`. In addition, it will pack these files into a JAR called `gradle-kotlin.jar` (you can find it in `build/libs`). JAR files contain the compiled bytecode of a particular application or library and some metadata. The `-cp` option of the Kotlin interpreter allows you to modify the default classpath, which adds one more way to start the app:

```
$ kotlin -cp build/libs/gradle-kotlin.jar HelloKt
Hello World!
```

Again, the class files that were created during the compilation phase are regular Java bytecode files. Since they are no different than files produced by `javac` (the Java compiler), we should be able to run our Kotlin app using `java` (the Java interpreter). Let's try it:

```
$ java -cp build/libs/gradle-kotlin.jar HelloKt
```

Java will respond with the following error:

```
Exception in thread "main" java.lang.NoClassDefFoundError:
  kotlin/jvm/internal/Intrinsics
          at HelloKt.main(Hello.kt)
```

What happened? Well, Java loaded our JAR file and started running the code. The problem is that all Kotlin projects implicitly depend on the Kotlin standard library. We specified it as a compile-time dependency, but it didn't get to the final JAR file. Java wasn't able to find it, so it terminated with an error. This reasoning actually leads us to the solution to our problem - we need to add the Kotlin library to the classpath:

```
$ java -cp $KOTLIN_HOME/lib/kotlin-runtime.jar:build/libs/gradle-kotlin.jar Hell\
oKt
Hello World!
```

What if we want our JAR file to be self-sufficient? It turns out, this desire is quite common in the Java world, and a typical solution usually involves building a so-called uber JAR[3] file. Building uber JAR files in Gradle is supported via the ShadowJar plugin[4]. In order to add it to our project, we need to make the following changes in our `build.gradle` file:

[3] http://stackoverflow.com/questions/11947037/

[4] https://github.com/johnrengelman/shadow

```
buildscript {
  ext.kotlin_version = '1.0.3'
  ext.shadow_version = '1.2.3'

  repositories {
    jcenter()
  }

  dependencies {
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version",
      "com.github.jengelman.gradle.plugins:shadow:$shadow_version"
  }
}

apply plugin: 'kotlin'
apply plugin: 'com.github.johnrengelman.shadow'

repositories {
  jcenter()
}

dependencies {
  compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

The ShadowJar plugin will analyze the `dependencies` section and copy class files from provided libraries into an uber JAR file. We can build this file by invoking the following command:

```
$ gradle shadowJar
```

After that, we will be able to start our app without adding any other libraries to the classpath:

```
$ java -cp build/libs/gradle-kotlin-all.jar HelloKt
Hello World!
```

All of this can bring us to the conclusion that we actually don't need to install Kotlin tools on production servers to "run Kotlin code". After our application is built, organized into several JAR files and copied to the production server, we will only need the Java runtime to run it.

Editing source files

In the previous section, we learnt how to compile Kotlin files using command line tools and Gradle. Here we'll start using a full-blown IDE, but first let's look at a slightly more lightweight solution - Atom Editor.

Using Atom

When installed from a software repository, Atom adds its executable to the `PATH`, so you can invoke it from anywhere.

Let's start Atom from the directory containing the Gradle project using the following command:

```
$ atom .
```

The `language-kotlin` package adds syntax highlighing, while `linter-kotlin` shows compilation errors. Additionally, a very useful package called `terminal-plus` enhances Atom with a console so that you can compile and run your applications without leaving the editor. However, when experimenting with a simple prototype, constant switching to the console only to check the output may become boring. In this case, it makes sense to adjust the workflow a little bit.

Let's add the `application` plugin to our build.gradle file:

```
1  // buildscript part omitted
2
3  apply plugin: 'kotlin'
4  apply plugin: 'application'
5  apply plugin: 'com.github.johnrengelman.shadow'
6
7  mainClassName = "HelloKt"
8
9  repositories {
10   jcenter()
11 }
12
13 dependencies {
14   compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
15 }
```

The `application` plugin is responsible for running the app. In order to do that, it needs to know the name of the main class, and in our case it is `HelloKt`. Now we have one more way for running our "Hello World" example:

```
$ gradle run
```

Gradle will respond with the following:

```
:compileKotlin UP-TO-DATE
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:run
Hello World!

BUILD SUCCESSFUL


Total time: 0.803 secs
```

Why all this hassle? Well, the important thing about the `:run` task (added by the `application` plugin) is that it becomes part of the Gradle build lifecycle. Gradle *knows* that `:run` depends on `:compileKotlin`, so if there are source code changes, it will recompile first. This is especially handy in the `watch` mode.

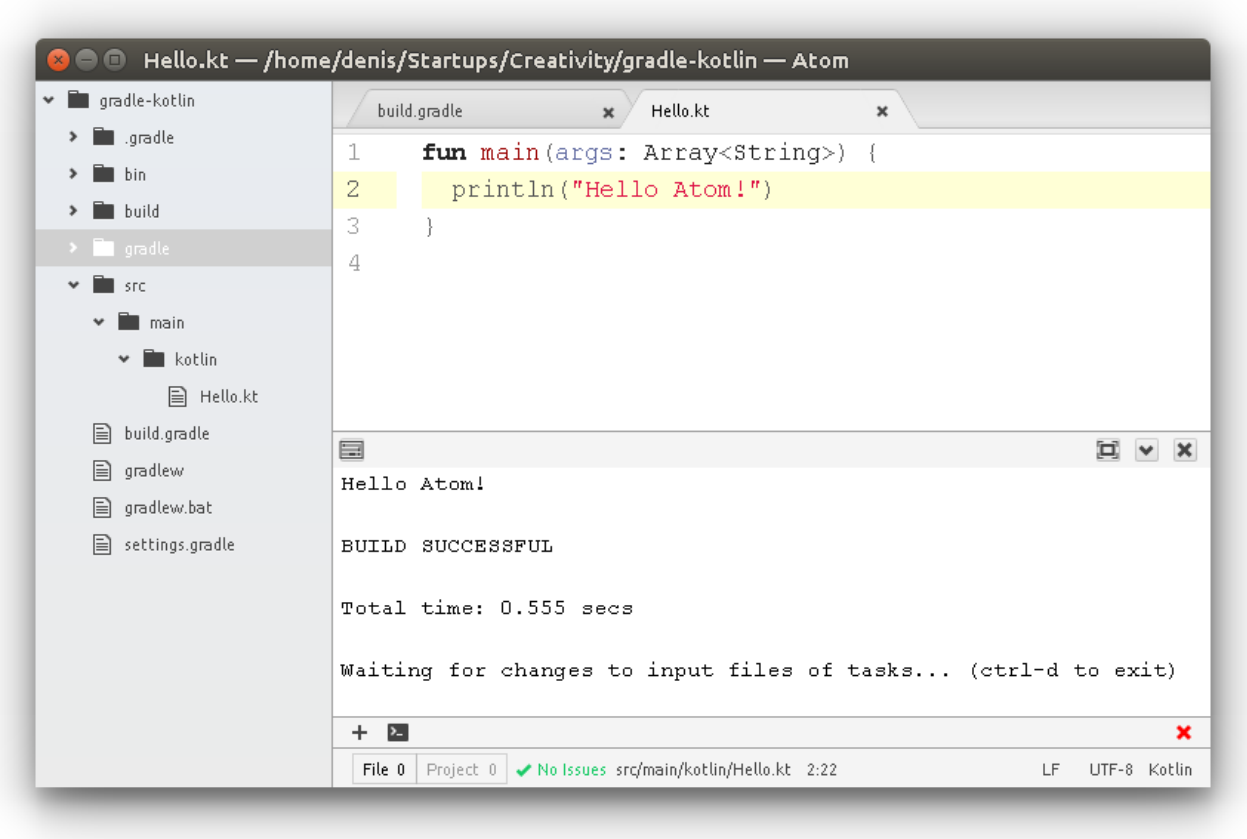
Try typing the following in the Terminal Plus window:

```
$ gradle -t run
```

You will see the already familiar "Hello World" greeting, but in addition to this Gradle will print the following:

```
Waiting for changes to input files of tasks... (ctrl-d to exit)
```

Now if you make any changes to the `Hello.kt` file, Gradle will recompile it and run the app again. And the good news is that subsequent builds will be much faster.



Atom as a mini-IDE

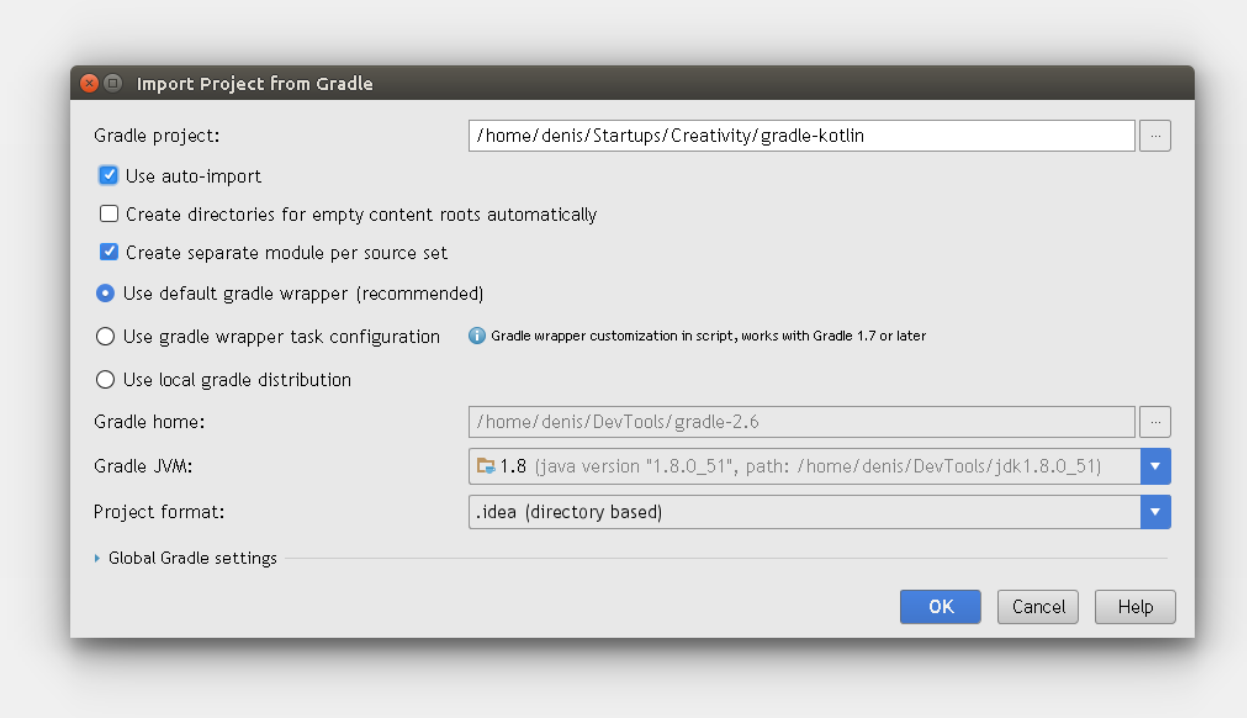
Using IntelliJ IDEA

If you installed IntelliJ IDEA by simply extracting the archive, you can start it by navigating to the `bin` subdirectory and invoking the `idea.sh` script from the command line:

```
$ cd ~/DevTools/idea/bin
$ ./idea.sh
```

When started for the first time, it will ask about importing old settings and show the Welcome window. I recommend using this window to create desktop entries. Simply select "Configure/Create Desktop Entry" and follow the instructions.

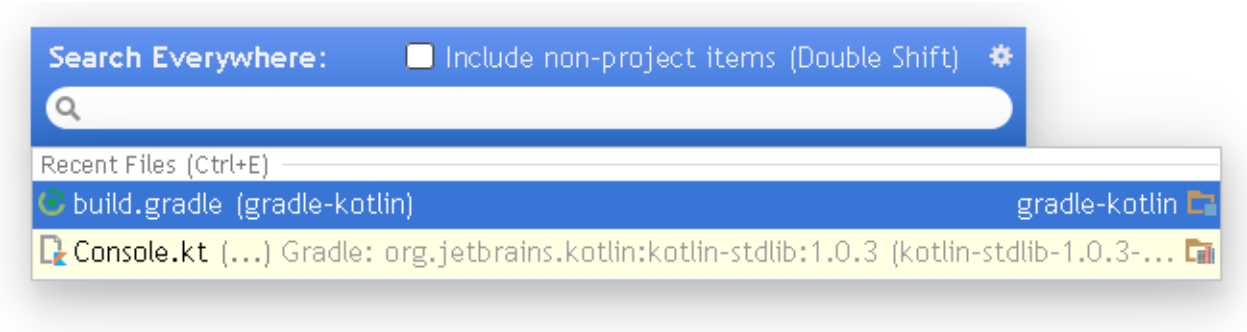
Now, since both Gradle and Kotlin are already integrated in IntelliJ IDEA, importing our project will be an extremely easy task. Simply click "Open" and then choose the directory of the project. The Gradle importing wizard will appear, but you can leave all settings as is. The only thing I recommend doing here is to check "Use auto-import". With auto-importing enabled, IDEA will download and add new libraries automatically once it detects a change in the `build.gradle` file.



Importing Gradle projects

I suspect that most developers are already familiar with at least one IDE from JetBrains, so I'm not going to explain how to use it in detail. However, here are a couple of tricks that you may find useful.

If you type <Shift> twice, IDEA will open the "Search Everywhere" window. This will allow you to quickly open any file, type or method.



Search Everywhere

To see the type of any variable or value, simply move the caret to the part of code you are interested in and press <Alt> + <Equals>. This is especially useful when you rely on the type inferrer but want to check that you're getting the type you want.

Language fundamentals

In this section, I will demonstrate the basics of the Kotlin programming language. This includes things that are common to most programming languages as well as several features which are specific to Kotlin.

Using the REPL

If we run the Kotlin compiler without parameters, it will start an interactive shell (also known as REPL or Read-Eval-Print-Loop). In this shell, every entered command will be interpreted, and its result will be printed to the console:

```
$ kotlinc
Welcome to Kotlin version 1.0.3 (JRE 1.8.0_51-b16)
Type :help for help, :quit for quit
>>> "Hello World!"
Hello World!
```

In order to quit from the REPL, type `:quit`.

While playing with REPL, it may also be desirable to enable reflection capabilities. Simply start the REPL with the corresponding library:

```
$ kotlinc -cp ~/DevTools/kotlinc/lib/kotlin-reflect.jar
```

The interpreter accompanied with reflection is especially useful while learning the language, and while I'm going to show all output here, feel free to start your own REPL session and follow along.

Defining values

There are three main keywords for defining everything in Kotlin:

keyword	description
`val`	defines a constant (or value)
`var`	defines a variable, very rarely used
`fun`	defines a method or function

Defining a constant is usually as simple as typing

```
>>> val num = 42
>>>
```

Since the assignment operator in Kotlin doesn't return any value, there was no output in the console. However, if you type `num`, you will see its value:

```
>>> num
42
```

By default, the REPL doesn't print type information. If you want to know the actual Kotlin type, type the following:

```
>>> num.javaClass.kotlin
class kotlin.Int
```

OK, now we know that it's an integer (or `kotlin.Int` in the Kotlin world). Notice that Kotlin guessed the type of the constant by analyzing its value. This feature is called *type inference*.

Sometimes you want to specify the type explicitly. If in the example above you want to end up with a value of type `Short`, simply type

```
>>> val num: Short = 42
>>> num.javaClass.kotlin
class kotlin.Short
```

Variables are not as necessary in Kotlin as they were in Java, but they are supported by the language and could be useful sometimes:

```
>>> var hello = "Hello"
>>> hello.javaClass.kotlin
class kotlin.String
```

Just as with types you may explicitly specify the type or allow the type inferrer to do its work.

As we will see later, most Java statements are expressions in Kotlin, and therefore they evaluate to some value. This feature essentially allows you to write code using `val`s and resort to `var`s only occasionally.

There are several types of functions in Kotlin, and they are defined using slightly different syntax. When defining so-called *single-expression* functions, it is required to specify argument types, but specifying the return type is optional:

```
>>> fun greet(name: String) = "Hello " + name
>>> greet("Joe")
Hello Joe
```

Notice that for single-expression functions the body is specified after the = symbol and curly braces are completely unnecessary. The regular syntax for defining functions in Kotlin is the following:

```
fun greet(name: String): String {
  return "Hello " + name
}
```

Here we're specifying `String` as the return type (after parentheses), defining the body inside curly braces and using the `return` keyword.

If you want to create a procedure-like method that doesn't return anything and is only invoked for side effects, you should specify the return type as `Unit`. This is a Kotlin replacement for `void` (which, by the way, is not even a keyword in Kotlin):

```
fun printGreeting(name: String): Unit {
  println("Hello " + name)
}
```

Declaring `Unit` for procedure-like functions is optional. If you don't specify any type, the `Unit` type is assumed.

A parameter of a function can have a default value. If this is the case, users can call the function without providing a value for the argument:

```
>>> fun greet(name: String = "User") = "Hello " + name
>>> greet()
Hello User
```

Compiler *sees* that the argument is absent, but it also *knows* that there is a default value, so it takes this value and then invokes the function as usual.

Lambdas

Sometimes it is useful to declare a function literal (or *lambda*/*anonymous function*) and pass it as a parameter or assign it to a variable (or constant). For example, the `greet` function from the example above could also be defined the following way:

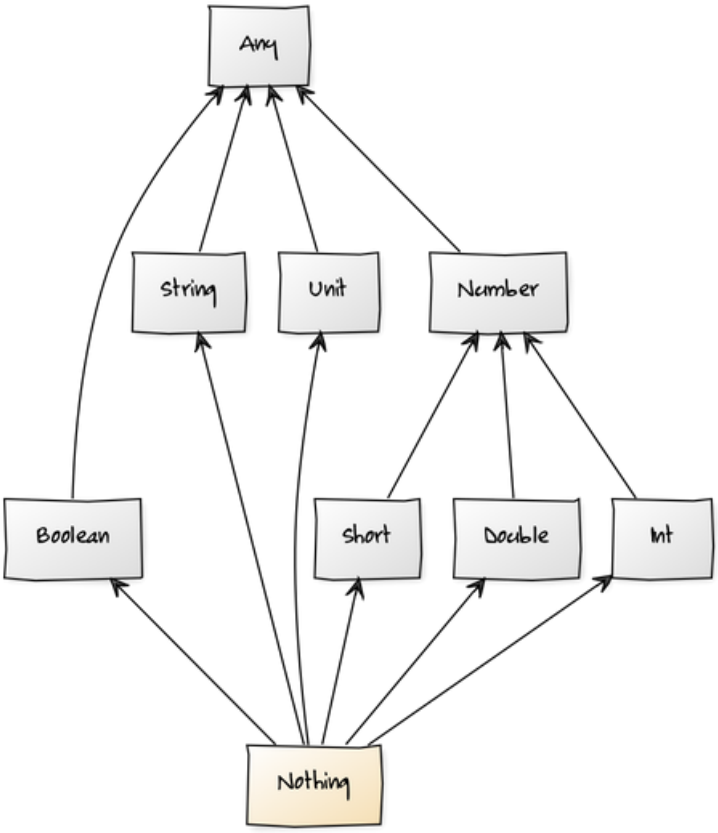
```
>>> var greetVar: (String) -> String = { name -> "Hello " + name }
```

You don't have to type the type of the variable, but if you want the type inferrer to do its work, you'll have to specify the type of parameters:

```
>>> var greetVar = { name: String -> "Hello " + name }
```

Type hierarchy

Since we are on the topic of types, let's take a look at a very simplified type hierarchy (with nullable types omitted, more on them later):



Type hierarchy

`Any` serves as the supertype of all types and resides at the top of the hierarchy. If you inspect its Java class, you will see that it translates to `java.lang.Object` in the Java world:

```
>>> Any::class.java
class java.lang.Object
```

You can use `is` to check the type of a value:

```
>>> val str = "Hello"
>>> str is String
true
>>> str is Any
true
```

Kotlin, just like Scala, has a concept of so-called *bottom types* which implicitly inherit from all other types. In the diagram above, `Nothing` implicitly inherits from all types (including user-defined ones, of course). You are unlikely to use bottom types directly in your programs, but they are useful to understand type inference. More on this later.

Nullable types

If you look at the diagram above, you will see that numeric types in Kotlin have a common supertype called `Number`. Unlike Java, though, `Numbers` disallow `nulls`:

```
>>> var num: Int = null
error: null can not be a value of a non-null type Int
```

If you need a type that allows `nulls`, you should append `?` to the type name:

```
>>> var num: Int? = null
>>> num
null
```

After you defined your variable as nullable, Kotlin enforces certain restrictions on its use. Unconditional access to its member fields and methods will fail at compile time:

```
>>> var str: String? = "Hello"
>>> str.length
error: only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable\
 receiver of type String?
str.length
```

As the compiler suggests, you can either use a safe call with `?.`, or ignore the danger with `!!.`:

```
>>> str?.length
null
str!!.length
5
```

Obviously, with the latter there is a risk of getting `NullPointerException`:

```
>>> str = null
>>> str?.length
null
>>> str!!.length
kotlin.KotlinNullPointerException
```

Another useful thing when working with nullable types is the *elvis* operator written as `?:`:

```
<expression_1> ?: <expression_2>
```

As almost everything in Kotlin, the elvis operator returns a value. This result is determined by the `<expression_1>`. If it's not `null`, this non-null value is returned, otherwise the `<expression_2>` is returned. Consider:

```
>>> var str: String? = "Hello"
>>> str ?: "Hi"
Hello
>>> var str: String? = null
>>> str ?: "Hi"
Hi
```

The elvis operator is often used when it is necessary to convert a value of a nullable type into a value of a non-nullable type.

Collections

If you need to create a collection, there are several helper functions available:

```
>>> val lst = listOf(1, 2, 3)
>>> lst
[1, 2, 3]
>>> val arr = arrayOf(1, 2, 3)
```

In Kotlin, square brackets translate to `get` and `set` calls, so you can use either `[]` or `get` to access both array and list elements:

```
>>> arr[0]
1
>>> arr.get(0)
1
>>> lst[0]
1
>>> lst.get(0)
1
```

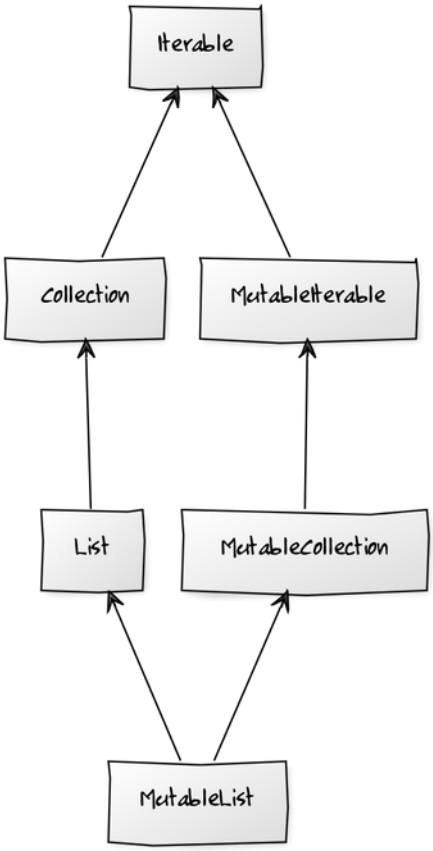
Kotlin distinguishes between mutable and immutable collections, and when you use `listOf`, you actually get an immutable list:

```
>>> lst.add(4)
error: unresolved reference: add
```

If you need a mutable one, use a corresponding function with the `mutable` prefix:

```
>>> val lst = mutableListOf(1,2,3)
>>> lst.add(4)
true
>>> lst
[1, 2, 3, 4]
```

A greatly simplified List hierarchy is shown below:



The List type hierarchy

As you see here, lists come in two flavours - mutable and immutable. Kotlin uses immutable collections by default, but mutable ones are also available.

Another important point is that Kotlin collections always have a type. If the type wasn't specified, it will be inferred by analyzing provided elements. In the example above we ended up with the list and array of `Int`s because the elements of these collections looked like integers. If we wanted to have, say, a list of `Short`s, we would have to set the type explicitly like so:

```
>>> val lst = listOf<Short>(1, 2, 3)
>>> lst[0].javaClass.kotlin
class kotlin.Short
```

Defining classes

Users can define their own types using the omnipresent `class` keyword:

```
>>> class Person {}
>>>
```

Needless to say, the `Person` class defined above is completely useless. You can create an instance of this class, but that's about it (note the absence of the `new` keyword):

```
>>> val person = Person()
>>> person
Line1$Person@4e04a765
```

Note that the REPL showed `Line1$Person@4e04a765` as a value of this instance. We {line-numbers=off}`kotlin // simplified signature fun toString(): String`

By convention, the `toString` method is used to create a String representation of the object, so it makes sense that REPL uses this method here. The base implementation of the `toString` method simply returns the name of the class followed by an object hashcode.

By the way, if you don't have anything inside the body of your class, you don't need curly braces.

A slightly better version of this class can be defined as follows:

```
>>> class Person(name: String)
>>> val person = Person("Joe")
```

By typing `name: String` inside the parentheses we defined a constructor with a parameter of type `String`. Although it is possible to use `name` inside the class body, the class still doesn't have a field to store it. Fortunately, it's easily fixable:

```
>>> class Person(val name: String)
>>> val person = Person("Joe")
>>> person.name
Joe
```

Much better! Adding `val` in front of the parameter name defines a read-only property (similar to an immutable field in other languages). Similarly `var` defines a mutable property (field). In any case, this field will be initialized with the value passed to the constructor when the object is instantiated. What's interesting, the class defined above is roughly equivalent of the following Java code:

```
class Person {
  private final String mName;
  public Person(String name) {
    mName = name;
  }
  public String name() {
    return mName;
  }
}
```

To define a method, simply use the `fun` keyword inside a class:

```
class Person(val name: String) {
  override fun toString() = "Person(" + name + ")"
  operator fun invoke(): String = name
  fun sayHello(): String = "Hi! I'm " + name
}
```

When working with classes and objects in REPL, it is often necessary to type several lines of code at once. Fortunately, you can easily paste multi-line snippets of code to the REPL, no special commands are necessary.

In the example above, we defined two methods - `invoke` and `sayHello` and overrode (thus keyword `override`) the existing method `toString`. Of these three, the simplest one is `sayHello` as it only prints a phrase to `stdout`:

```
>>> val person = Person("Joe")
>>> person
Person(Joe)
>>> person.sayHello()
Hi! I'm Joe
```

Notice that REPL used our `toString` implementation to print information about the instance to `stdout`. As for the `invoke` method, there are two ways to call it.

```
>>> person.invoke()
Joe
>>> person()
Joe
```

Yep, using parentheses on the instance of a class actually calls the `invoke` method defined on this class. Note that in order to achieve this, we had to mark the method definition with the `operator` keyword.

Defining objects

Kotlin doesn't have the `static` keyword, but it does have syntax for defining *singletons*[5]. If you need to define methods or values that can be accessed on a type rather than an instance, use the `object` keyword:

```
object RandomUtils {
  fun random100() = Math.round(Math.random() * 100)
}
```

After `RandomUtils` is defined this way, you will be able to use the `random100` method without creating any instances of the class:

```
>>> RandomUtils.random100()
35
```

You can put an object declaration inside of a class and mark it with the `companion` keyword:

```
class Person(val name: String) {
  fun score() = Person.random100()
  companion object Person {
    fun random100() = Math.round(Math.random() * 100)
  }
}
```

With `companion object`, you can even omit its name:

```
class Person(val name: String) {
  fun score() = random100()
  companion object {
    fun random100() = Math.round(Math.random() * 100)
  }
}
```

Finally, you can always define a standalone method just as we did with the `main` function in our "Hello World" example.

Type parametrization

Classes can be parametrized, which makes them more generic and possibly more useful:

[5] https://en.wikipedia.org/wiki/Singleton_pattern

```
class Cell<T>(val contents: T) {
  fun get(): T = contents
}
```

We defined a class called `Cell` and specified one read-only property `contents`. The type of this property is not yet known. Kotlin doesn't allow *raw types*, so you cannot simply create a new `Cell`, you must create a `Cell` *of something*:

```
>>> val cell = Cell<Int>(2)
>>>
```

Here we're defining a `Cell` passing the `Int` as a type parameter. Fortunately, in many cases it's not necessary as the type can be inferred (for example, from the values passed as constructor arguments):

```
>>> val cell = Cell(2)
>>>
```

Here we passed `2` as an argument, and it was enough for the type inferrer to decide on the type of this instance.

In addition to classes, we can also parametrize functions and methods. When we were discussing collections, we used the `listOf` method, which is parametrized. Of course, you can create your own parametrized methods as well:

```
>>> fun <T> concat(a: T, b: T): String = a.toString() + b.toString()
>>> concat(1, 2)
12
```

And again, usually it's not necessary to specify the type explicitly, but you can always do this to override type inference. This is exactly what we did to define a `List` of `Shorts`:

```
>>> val lst = listOf<Short>(1, 2, 3)
```

Extension functions

Sometimes it is necessary to add a new method to a type, which source code is not in your control. Kotlin (like C#) allows you to do this with *extension functions*.

For example, let's enhance the `String` type by adding a new function that checks whether the string contains only spaces or not:

```
>>> fun String.onlySpaces(): Boolean = this.trim().length == 0
```

Here we're prepending the function name with the name of a *receiver* type. Inside the function, we can refer to the receiver object using the `this` keyword. After our extension function is defined, it becomes available on all `String` instances:

```
>>> "   ".onlySpaces()
true
>>> " 3 ".onlySpaces()
false
```

Interestingly, we can define an extension method on `Any?`:

```
fun Any?.toStringAlt(): String {
  if (this == null)
    return "null"
  else
    return this.toString()
}
```

This, in turn, allows us to invoke the method even if the value is `null`:

```
>>> null.toStringAlt()
null
>>> "22".toStringAlt()
22
```

Packages and imports

Kotlin classes are organized into packages similarly to Java or C#. For example, we could move our `main` function from the "Hello World" example into a package called `hello`. In order to do this, we would put a package declaration on the top of the file:

```
package hello

fun main(args: Array<String>) {
  println("Hello World!")
}
```

Since packages follow the directory structure, we would need to create a new folder called `hello` and move our file there.

After this, the main function becomes a package-level function, and this must be also reflected in the `gradle.build` file:

```
mainClassName = "hello.HelloKt"
```

The `hello.HelloKt` is known as fully qualified name[6] (FQN for short).

More generally, when we need to use something from another package, we need to either reference it by its fully-qualified name or use *imports*.

For example, if we want to use the `File` class from the Java standard library (to which, by the way, all Kotlin programs have access), we need to import it first:

```
>>> import java.io.File
>>> val file = File("readme.txt")
```

Just like in Java or C#, we can import all classes from a particular package using a wildcard `*`:

```
>>> import java.io.*
```

Unlike Java or C#, though, Kotlin can also import standalone functions defined on the package level.

In addition to the imports defined by a developer, Kotlin automatically imports `java.lang.*`, `kotlin.io.*`, `kotlin.collections.*`, `kotlin.*` and so on. This is exactly why we could use methods like `println` or `listOf` in previous examples without importing anything.

When importing a type or function, you can choose how to refer to it in your code:

```
>>> import kotlin.collections.listOf as immutableListOf
>>> immutableListOf<Short>(1, 2, 3)
[1, 2, 3]
```

Loops and conditionals

Unlike Java with *if statements*, in Kotlin *if expressions* always result in a value. In this respect they are similar to Java's ternary operator `?:`.

Let's define a lambda that uses the `if` expression to determine whether the argument is an even number:

```
>>> val isEven = { num: Int -> if (num % 2 == 0) true else false }
>>> isEven
(kotlin.Int) -> kotlin.Boolean
```

When `isEven` is inspected in REPL, the interpreter responds with `(kotlin.Int) -> kotlin.Boolean`. Even though we haven't specified the return type, the type inferrer determined it as the type of the last (and only) expression, which is the *if expression*. How did it do that? Well, by analyzing the types of both branches:

[6] https://en.wikipedia.org/wiki/Fully_qualified_name

expression	type
if branch	`Boolean`
else branch	`Boolean`
whole expression	`Boolean`

If an argument is an even number, the *if branch* is chosen and the result type has type `Boolean`. If an argument is an odd number, the *else branch* is chosen but result still has type `Boolean`. So, the whole expression has type `Boolean` regardless of the "winning" branch.

But what if branch types are different, for example `Int` and `Double`? In this case the nearest common supertype will be chosen. For `Int` and `Double` this supertype will be `Any`, so `Any` will be the type of the whole expression:

```
>>> val isEven = { num: Int -> if (num % 2 == 0) 1 else 1.2 }
>>> isEven
(kotlin.Int) -> kotlin.Any
```

If you give it some thought, it makes sense because the result type must be able to hold values of both branches. After all, we don't know which one will be chosen until runtime.

What if the *else branch* never returns and instead throws an exception? Well, it's time to recall `Nothing`, which sits at the bottom of the Kotlin type hierarchy. In this case, the type of the *else branch* is considered to be `Nothing`. Since `Nothing` is the bottom type, any other type is considered a supertype for it and therefore the whole expression will have the type of the *if branch*.

expression	type
if branch	`Any`
else branch	`Nothing`
whole expression	`Any`

There's not much you can do with `Nothing`, but it is included in the type hierarchy, and it makes the rules that the type inferrer uses more clear.

while and for loops

The `while` loop is almost a carbon copy of its Java counterpart. Unlike *if*, *while* is not an expression, so it's called for a side effect. Moreover, it almost always utilizes a `var` for iterations:

```
var it = 5
while (it > 0) {
  println(it)
  it -= 1
}
```

In a modern language it looks rather *old school* and in fact, you will rarely use it in Kotlin. There are many alternative ways to achieve the same thing, including `for` loops:

```
>>> for (i in 5 downTo 1) println(i)
5
4
3
2
1
```

This syntax looks odd and certainly requires some explanation, so here you go. The `for` loop in Kotlin works with everything that has the `iterator` function (extension functions will work as well). The type of the value that `iterator` returns doesn't matter as long as it has two functions: `next()` and `hasNext()`. The `hasNext()` function is used by the `for` loop to determine whether there are more items or it's time to stop.

OK, what about `5 downTo 1`. It turns out that `downTo` is an extension function defined on all integer types (`Int`, `Long`, `Short` and so on). It accepts an argument and builds an `IntProgression`. Not surprisingly, the `IntProgression` has the `iterator` method (required by `for`).

One thing that is left to explain is the fact that we used the `downTo` function without dots or parentheses. Why is that? If you look at Kotlin source code, you will see that its signature looks like this:

```
// simplified signature
infix fun Int.downTo(to: Int): IntProgression { /* implementation code */ }
```

The magic word here is `infix` because it marks the function as suited for infix notation.

By the way, nothing prevents you from defining your own functions in this way!

String templates

Just as most modern languages, Kotlin allows to execute arbitrary code inside string literals. In order to use this feature, simply put `$` in front of any variable or value you want to interpolate:

```
>>> val name = "Joe"
>>> val greeting = "Hello $name"
>>> greeting
Hello Joe
```

If an interpolated expression is more complex, e.g contains dots or operators, it needs to be taken in curly braces:

```
>>> "Random number is ${Math.random()}"
Random number is 0.4884723200806935
```

If you need to spread your string literal across multiple lines or include a lot of special characters without escaping you can use triple-quote to create *raw strings*. Note that the usual backslash escaping doesn't work there:

```
>>> """First line\nStill first line"""
First line\nStill first line
```

Interfaces

Interfaces in Kotlin are quite similar to Java 8. They can contain both abstract (without implementation) methods as well as concrete ones. You can declare your class as implementing multiple interfaces, but if they contain abstract members, then you must either implement them or mark your class as `abstract`, so the Java rule still holds.

In Java, a class can implement many interfaces, but if you want to make your class concrete (i.e. allow to instantiate it), you need to provide implementations for all methods defined by all interfaces.

Let's look at a rather simplistic example:

```
1  interface A { fun a(): Unit = println("a") }
2
3  interface B { fun b(): Unit }
```

We defined two interfaces so that interface `A` has one concrete method, and interface `B` has one abstract method (the absence of a body that usually comes after the equals sign means exactly that). If we want to create a new class `C` that inherits functionality from both interfaces, we will have to implement the `b` method:

```
class C: A, B { override fun b(): Unit = println("b") }
```

If we don't implement `b`, we will have to make the `C` class `abstract` or get an error.