Extracted from:

Programming Kotlin

Creating Elegant, Expressive, and Performant JVM and Android Applications

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Programming Kotlin 1.3

Create Elegant, Expressive, and Performant JVM and Android Applications

Venkat Subramaniam

edited by Jacquelyn Carter

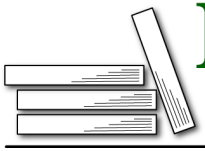
Programming Kotlin

Creating Elegant, Expressive, and Performant JVM and Android Applications

Venkat Subramaniam

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Jacquelyn Carter
Copy Editor: Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-635-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2019

Sensible Warnings

Even if a piece of code is valid syntactically, some potential problems may be lurking. Getting an early warning, during compilation time, can help us to proactively fix such possible issues. The Kotlin compiler looks out for quite a few potential issues in code.

For example, if a parameter that's received in a function or a method isn't used, then the compiler will give a warning. In the following script, the parameter passed to compute() isn't used.

essence/unused.kts

```
fun compute(n: Int) = 0

println(compute(4))
```

When you run this script, in addition to displaying the result, Kotlin will also report any warnings for unused parameters:

```
0
unused.kts:1:13: warning: parameter 'n' is never used
fun compute(n: Int) = 0
            ^
```

It's a good software development practice to *treat warnings as errors*—an agile practice emphasized in *Practices of an Agile Developer [SH06]*. Kotlin makes that easy with the -Werror option. To use this option, place it on the command line when you compile the code or run it as a script, like so:

```
$ kotlinc-jvm -Werror -script unused.kts
```

This option will fail the build or execution. Unlike the previous run without that option, there will be no output when the script is run; instead an error is reported:

```
error: warnings found and -Werror specified
unused.kts:1:13: warning: parameter 'n' is never used
fun compute(n: Int) = 0
            ^
```

The Kotlin compiler is sensible when giving warnings. For example, it's not uncommon for programs to ignore command-line arguments. Forcing us to use parameters given to main() is considered draconian, so Kotlin doesn't complain about unused parameters for main(), as we see in the next example. But if you have an unused parameter in main() within a script (a .kts file instead of a .kt file), then Kotlin will give you a warning—it decides based on the context.

essence/UnusedInMain.kt

```
fun compute(n: Int) = 0

fun main(args: Array<String>) = println(compute(4))
```

When you compile the code using kotlinc-jvm and then run it using either java or kotlin, you'll get the following output. The warning is from kotlinc and the output is from the execution of the generated jar file:

```
UnusedInMain.kt:1:13: warning: parameter 'n' is never used
fun compute(n: Int) = 0
            ^
0
```

Starting from Kotlin 1.3, you may leave out the parameters to main() if you don't need them.

We saw how Kotlin tries to make a preemptive strike against potential errors. Along those lines, the language wants you to be decisive about immutability. Let's explore that choice next.

Prefer val over var

To define an immutable variable–that is, a constant or a *value*—use val, like so:

```
val pi: Double = 3.14
```

Unlike Java, where you'd place the type before the name of the variable, in Kotlin you place the name of the variable first, then a colon, followed by the type. Kotlin considers the sequence Java requires as "placing the cart before the horse" and places a greater emphasis on variable names than variable types.

Since the type of the variable is obvious in this context, we may omit the type specification and ask Kotlin to use type inference:

```
val pi = 3.14
```

Either way, the value of pi can't be modified; val is like final in Java. Any attempt to change or reassign a value to variables defined using val will result in a compilation error. For example, the following code isn't valid:

```
val pi = 3.14
pi = 3.14 //ERROR: val cannot be reassigned
```

What if we want to be able to change the value of a variable? For that, Kotlin has var—also known as "keyword of shame." Variables defined using var may be mutated at will.

Here's a script that creates a mutable variable and then modifies its value:

```
var score = 10
//or var score: Int = 10
```

```
println(score) //10

score = 11
println(score) //11
```

Mutating variables is a way of life in imperative style of programming. But that's a taboo in functional programming. In general, it's better to prefer immutability—that is, val over var. Here's an example to illustrate why that's better:

essence/mutate.kts

```
var factor = 2

fun doubleIt(n: Int) = n * factor

factor = 0

println(doubleIt(2))
```

Don't run the code; instead eyeball it, show it to a few of your colleagues and ask what the output of the code will be. Take a poll. The output will be equal to what most people said it will be—just kidding. Program correctness is not a democratic process; thankfully, I guess.

You probably got three responses to your polling:

- The output is 4.
- The output is 0, I think.
- *WHAT*—the response the code evoked on someone recently.

The output of the above code is 0—maybe you guessed that right, but guessing is not a pleasant activity when coding.

Mutability makes code hard to reason. Code with mutability also has a higher chance of errors. And code with mutable variables is harder to parallelize. In general, try really hard to use val as much as possible instead of var. You'll see later on that Kotlin defaults toward val and immutability, as well, in different instances.

Whereas val in Kotlin is much like Java's final, Kotlin—unlike Java—insists on marking mutable variables with var. That makes it easier to search for the presence of var in Kotlin than to search for the absence of final in Java. So in Kotlin, it's easier to scrutinize code for potential errors that may arise from mutability.

A word of caution with val, however—it only makes the variable or reference a constant, not the object referenced. So val only guarantees immutability of the reference and doesn't prevent the object from changing. For example, String is immutable but StringBuilder is not. Whether you use val or var, an instance

of String is safe from change, but an instance of StringBuilder isn't. In the following code, the variable message is immutable, but the object it refers to is modified using that variable.

```
val message = StringBuilder("hello ")
//message = StringBuilder("another") //ERROR
message.append("there")
```

In short, val only focuses on the variable or reference at hand, not what it refers to. Nevertheless, prefer val over var where possible.

Improved Equality Check

Just like Java, Kotlin also has two types of equality checks:

- equals() method in Java, or == operator in Kotlin, is a comparison of values, called *structural equality*.
- == operator in Java, or === in Kotlin, is a comparison of references, called *referential equality*. Referential equality compares references and returns true if the two references are identical—that is, they refer to the same exact instance. The operator === in Kotlin is a direct equivalent of the == operator in Java.

But the structural equality operator == in Kotlin is more than the equals() method in Java. If you perform str1.equals(str2); in Java, you may run into a NullPointerException if the reference str1 is null. Not so when you use == in Kotlin.

Kotlin's structural equality operator safely handles null references. Let's examine that with an example:

essence/equality.kts

```
println("hi" == "hi")
println("hi" == "Hi")
println(null == "hi")
println("hi" == null)
println(null == null)
```

If these comparisons were done with equals() in Java, the net result would have been a runtime NullPointerException, but Kotlin handles the nulls safely. If the values held in the two references are equal then the result is true, and false otherwise. If one or the other reference is null, but not both, then the result is false. If both the references are null, then the result of the comparison is true. We can see this in the output, but you'll also see an added bonus in there:

```
true
false
```

```
false
false
true
equality.kts:3:9: warning: condition 'null == "hi"' is always 'false'
println(null == "hi")
        ^
equality.kts:4:9: warning: condition '"hi" == null' is always 'false'
println("hi" == null)
        ^
equality.kts:5:9: warning: condition 'null == null' is always 'true'
println(null == null)
        ^
```

The output confirms the behavior of == operator like mentioned. The output also shows yet another example of Kotlin's sensible warnings—if the result of comparison will always be an expected value, it prompts a warning suggesting we fix the code to remove the redundant conditional check.

When == is used in Kotlin, it performs the null checks and then calls equals() method on the object.

You've learned the difference between using equals() in Java and == in Kotlin. Next let's look at the ease with which we can create strings with embedded expressions.

String Templates

In programs, we often create strings with embedded values of expressions. Concatenating values to create strings using the + operator makes the code verbose and hard to maintain. String templates solve that problem by providing an elegant solution.

Within a double-quoted string, the $ symbol can prefix any variable to turn that into an expression. If the expression is more complex than a simple variable, then wrap the expression with ${}.

A $ symbol that's not followed by a variable name or expression is treated as a literal. You may also escape the $ symbol with a backslash to use it as a literal.

Here's an example with a string template. Also, it contains a plain string with embedded $ symbols that are used as literals.

essence/stringtemplate.kts

```
val price = 12.25
val taxRate = 0.08

val output = "The amount $price after tax comes to $${price * (1 + taxRate)}"
val disclaimer = "The amount is in US$, that's right in \$only"
```

```
println(output)
println(disclaimer)
```

In the string template assigned to output, the first $ symbol is used as a delimiter for the expression, the variable name, that follows it. The second $ symbol is a literal since it's followed by another $, which isn't a variable or expression. The third $ symbol prefixes an expression that's wrapped in {}. The other $ symbols in the code are used as literals. Let's take a peek at the output of the code:

```
The amount 12.25 after tax comes to $13.23
The amount is in US$, that's right in $only
```

The earlier caution to prefer val over var applies here too. Let's take the code with var we saw previously and modify it slightly to use a string template.

essence/mutateconfusion.kts

```
var factor = 2

fun doubleIt(n: Int) = n * factor
var message = "The factor is $factor"

factor = 0

println(doubleIt(2))
println(message)
```

Once again, don't run the code, but eyeball it and figure out the output of this code. Does it correspond with the following output?

```
0
The factor is 2
```

The variable factor within the function doubleIt() binds to the variable outside its immediate scope—that is, in its lexical scope. The value of factor at the time of the function call is used. The string template, on the other hand, is evaluated when the variable message is created, not when its value is printed out. These kinds of differences increase cognitive load and makes the code hard to maintain and also error prone. No need to torture fellow programmers with code like this. It's inhumane. Again, as much as possible prefer val over var.

Next, let's look at using raw strings to remove some clutter and to create multiple lines of strings.

Raw Strings

Dealing with escape characters makes the code messy. Instead of using escaped strings, in Kotlin we may use raw strings which start and end with

three double quotes. We may use raw strings to place any character, without the need to use escapes, and may also use them to create multiline strings.

No Escape

In an escaped string which starts and ends with a single double quote, we can't place a variety of characters, like new line or a double quote, for example, without using the escape character \. Even a simple case can be unpleasant to read, like this one:

```
val escaped = "The kid asked, \"How's it going, $name?\""
```

We had to escape the double quotes that were needed within the string. The more we use escaped strings, the messier it becomes. Instead of using escaped strings, in Kotlin we use raw strings. Just like escaped strings, raw strings can also be used as string templates, but without the mess of escaping characters. Here's the above escaped string changed to raw string—less clutter, more readable:

```
val raw = """The kid asked, "How's it going, $name?""""
```

Use escaped string, ironically, when you don't need to escape anything—for small, simple, plain vanilla strings. If you need anything more complex or multiple lines of string, then reach over to raw strings.

Multiline Strings

The infamous + operator is often used to create multiple lines of strings, and that leads to nasty code that's hard to maintain. Kotlin removes that ceremony with a multiline string, which is a raw string that contains line breaks. Multiline strings can also act as string templates.

Let's create a string that runs across several lines, but without the + operator.

essence/memo.kts

```
val name = "Eve"

val memo = """Dear $name, a quick reminder about the
party we have scheduled next Tuesday at
the 'Low Ceremony Cafe' at Noon. | Please plan to..."""

println(memo)
```

The multiline string starts with three double quotes, contains the string template expression to evaluate the variable name, and ends with three double quotes. The output of this code is multiple lines of string with the embedded expression evaluated.

```
Dear Eve, a quick reminder about the
party we have scheduled next Tuesday at
the 'Low Ceremony Cafe' at Noon. | Please plan to...
```

That worked beautifully, but—there always is a *but*—what if the multiline string were within a function, maybe within an if? Would the nesting mess things up? Let's find out.

essence/nestedmemo.kts

```
fun createMemoFor(name: String): String {
  if (name == "Eve") {
    val memo = """Dear $name, a quick reminder about the
        party we have scheduled next Tuesday at
        the 'Low Ceremony Cafe' at Noon. | Please plan to..."""

    return memo
  }

  return ""
}

println(createMemoFor("Eve"))
```

The createMemoFor() function returns a multiline string if the parameter passed is equal to Eve. Let's see what the output beholds:

```
Dear Eve, a quick reminder about the
        party we have scheduled next Tuesday at
        the 'Low Ceremony Cafe' at Noon. | Please plan to...
```

The resulting string has preserved the indentation—yikes. Thankfully, it's not too hard to get rid of. Let's rework the example:

```
fun createMemoFor(name: String): String {
  if (name == "Eve") {
    val memo = """Dear $name, a quick reminder about the
        |party we have scheduled next Tuesday at
        |the 'Low Ceremony Cafe' at Noon. | Please plan to..."""

    return memo.trimMargin()
  }

  return ""
}

println(createMemoFor("Eve"))
```

We made two changes. First, we placed a | on each line of the multiline string, starting with the second line. Second, we used the trimMargin() method, an extension function (we discuss these in Chapter 12, Fluency in Kotlin, on page ?), to strip the margin out of the string. With no arguments, the trimMargin() method removes the spaces until the leading | character. The | character

that's not in the leading position doesn't have any impact. Here's the output that shows the fix worked.

```
Dear Eve, a quick reminder about the
party we have scheduled next Tuesday at
the 'Low Ceremony Cafe' at Noon. | Please plan to...
```

If you do not want to use | as the leading delimiter, because maybe your text contains that character in arbitrary places, including the first character of a new line, then you may choose some other character—for example, let's go ahead and choose ~:

```
val memo = """Dear $name, a quick reminder about the
    ~party we have scheduled next Tuesday at
    ~the 'Low Ceremony Cafe' at Noon. | Please plan to..."""

return memo.trimMargin("~")
```

In the multiline string we use ~ as the delimiter instead of the default |, and in the call to trimMargin() we pass that specially chosen delimiter as argument. The output of this version is the same as the one where we used the default delimiter.

So far in this chapter, we've looked at the improvements to expressions and statements in Kotlin when compared to languages like Java. But Kotlin prefers expressions over statements. Let's discuss that next.