EPiC Series in Computing
Volume 58, 2019, Pages 191–202
Proceedings of 34th International Conference on Computers and Their Applications

Vetting Anti-patterns in Java to Kotlin Translation

Chaney Courtney, Mitchell Neilsen

Department of Computer Science
Kansas State University
Manhattan, KS, USA

Abstract

With Kotlin becoming a viable language replacement for Java, there is a need for translators and data flow analysis libraries to create maintainable and readable source code. Instagram, Uber, and Gradle are only a few of the large corporations that have either switched from Java to Kotlin completely or started to use it in internal tools in order to reduce code base size. Developers have claimed that Kotlin is fun to use in comparison to Java and much of the boilerplate code is reduced. With Java being the main language for the open source organization, PhenoApps, there is a need to support both Java and Kotlin to increase the maintainability of the code. Fortunately, JetBrains has an open-source IDE plugin for translating Java to Kotlin; however, the translation has some fundamental issues which shall be discussed further in this paper. Introducing, j2k, a CLI translation tool which includes various anti-pattern detection for syntactical formatting, performance, and other Android requirements. The new tool introduced within this paper, j2kCLI allows users to directly translate strings of Java code to Kotlin, or entire directories. This facilitates the maintainability of a large open source code base.

Keywords: Android, Kotlin, abstract syntax tree, command line interface, null pointer exception

1 Introduction

PhenoApps is an open source organization aimed at developing fundamental phenotypical workflow Android applications to replace and improve physical counterparts [1]. One example being FieldBook, an application that guides users to track various phenotypes for any given plot of land [2]. Much of PhenoApps efforts has been contributing extensible code and deploying state-of-the-art technology to third-world countries. Efforts to extend the functionality of these various applications along with the maintainability of the source code have surfaced with the first Hackathon in 2018 [3]. With the combined effort of Cornell University's Plant Science Department, the Boyce Thompson Institute, and Kansas State University's Cyber Physical Systems Lab, PhenoApps has grown and developed a repository with over seven Android applications [2,6,7]. The combination of various commits and influx of code by new contributors and the need to maintain this code exemplifies the drive to support both Java and Kotlin. Kotlin, being a fun alternative to Java with more readability, has decreased the overall lines of code (LOC) and number of null pointer exceptions (NPE). NPEs are a bane to Android developers (and users who encounter them). These errors typically leave the user confused, as the application will crash with no feedback to the user. Kotlin's null-safety syntax, when used correctly, is a panacea for most of these NPE's. This paper will dive into the various anti-patterns that should be caught, including NPE usage when using Kotlin. Along with NPE catching, the anti-patterns will cover optimization, formatting, and basic Android guidelines for application development. This paper introduces a two-step process of translation followed by data-flow analysis for Java to Kotlin code. The open source project ktlint is used for linting and creating visitor implementations (to walk the abstract syntax tree (AST) and find anti-patterns) [8]. Work has been done to extend ktlint's base functionality to make writing data flow analysis easier. The final product translation tool uses a novel command-line interface Kotlin program that is backed by JetBrain's j2k module of the Kotlin plugin for the IntelliJ IDE. Various improvements and modifications have been made from the j2k module, but much of the translations for Java to Kotlin constructs have remained the same. The reason for creating a command line tool is obvious as some developers may prefer this in comparison to an integrated development environment (IDE). Along with giving developers this option, the j2k command line tool has shown run time improvements from the typical IDE translation. The goal for this paper is to describe not only the functionality of this tool but also the differences between it and JetBrain's default converter. The rest of this paper will start by describing the current state of translation with JetBrains' tools and then how j2kCLI differs from the current implementation, which are the main contributions of this paper. Finally, there is a brief run-time and compiler error analysis that describes the current state of post-translated code.

2 Background

The open source JetBrains' Kotlin plugin contains methods for translating Java 6-8 to valid Kotlin code [4]. The command line tool developed for this work uses a headless IntelliJ IDEA in order to use the plugin libraries. The translation is not exactly the same as the non-headless version which will be covered in this section.

2.1 Global Variables

The current implementation of JetBrains' Java to Kotlin translation handles global variables by creating a mutable version of the object with a null-able type. For example, consider the code that declares this global variable in Java:

```
private A a;
```

The converted Kotlin counterpart is translated as:

```
var a: A? = null
```

The translation has modified the type to be possibly null (with the '?' token), and initialized this variable to null. Because Java allows objects to be null references, this is a valid translation. Although, if the user wants null-pointer-free code then they may require a slightly different translation. There are a few alternatives to this conversion that ultimately should be decided by the user depending on how this global variable is used. For example to keep an immutable conversion of a global variable the developer may consider this translation:

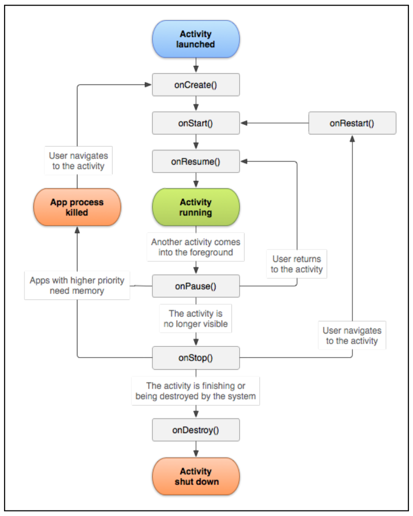
Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 1: Android Life Cycles

```
val a: A by lazy {
  A()
}
```

Along with the above, the user may want to keep a mutable type but ensure that this global variable never has a null value, using:

```
lateinit var a: A
```

This Java to Kotlin translator aims at reducing null pointer exceptions, therefore it chooses to give the user the option to either lazily evaluate a global variable or use the lateinit modifier. After the initial conversion of Java to Kotlin, this command line tool will run various visitor patterns to find these patterns.

2.2 Imports

Import statements in Java are not translated to Kotlin with the current translator implementation. Because Kotlin and Java have the same import statement syntax, the translation of this was trivial. A later development may add the optimization of imports, as it is recommended in Kotlin to not use wild-card import statements.

2.3 Android Life Cycle Functions

Life cycle functions are the very representation of Android applications [5]. All application contexts in an application require the implementation of certain functions, including the life cycle method onCreate. These life cycle functions are callbacks that can be interpreted as a state machine as seen in Figure 1, the functions include: onCreate, onStart, onResume, onPause, onStop, onRestart, onDestroy. As stated in the Android Developers documentation, not all activities necessarily need to implement each life cycle method, but using them incorrectly or ignoring them completely may lead to unexpected functionality [5]. For example, changing the screen orientation while an application is running will automatically call the onCreate method. If variables are constructed during this method such as for database management then some data may not persist after changing orientations. For this reason, various visitor patterns were added to reason about the usage and implementation of each function. It is recommended that most of these functions have a low complexity in terms of their inner functionality. In addition, other visitors have specific patterns for certain life cycle methods. For example the onStart method should contain all code that handles UI elements, therefore if any view callbacks are implemented in other functions, it is noted that they should be placed in the onStart method.

2.4 Overridden Methods

When certain abstract classes are implemented and functions are overridden, it's possible to have a function that only calls its' parent implementation. This is a known Kotlin anti-pattern. A visitor was created to look for functions with only one statement, that being a super method call to the functions original implementation. Users are recommended to either delete this method stub or fill the method with code pertaining to that specific life cycle.

2.5 Inter-translated Methods

Some methods exist in Java but have a different name in Kotlin. Because method call syntax is the same in Kotlin and Java, translating primitive Java functions to the new Kotlin equivalents is fairly easy. Kotlin also has some operator that replace typical Java code, for example the 'in' operator in Kotlin replaces the functionality for the 'contains' method in Java. Along with these various operators, Kotlin has concise getter and setter syntax for accessing members where Java would normally call a function. For example, in Java you would typically call 'size()', a function to calculate the size of an array; however, in Kotlin the equivalent call would be 'size' which is accessed by a getter. Another common function in Java is the string formatting method, 'format'. Kotlin has replaced this function with a unique string formatting where variables can be templates inside of string literals.

Java	Kotlin
getBytes	toByteArray
replaceAll	replace
contains	in (operator)
equalsToIgnoreCase	==
format	(built-in string formatting)

Table 1: Table of Java to Kotlin functions

3 Kotlin Anti-patterns

The translation of Java to Kotlin using the IDEA converter leaves the user with code that: does not always compile or has certain anti-patterns for Android. It is not always that case that post-translated code will work perfectly. Java and Kotlin are similar languages, but the translation is not trivial and some methods that exist in Java don't necessarily exist in Kotlin. Along with certain method names, a direct translation of Java to Kotlin may not generate Kotlin-esque code; however, this should partly be the developers job to use common Kotlin

constructs to simplify the Java equivalent. To make this process easier on the developer, the second part of this new Java to Kotlin translator uses various data flow visitors to capture certain 'bad-kotlin' or 'bad-android' coding and either warn the user or automatically adapt them to another translation. The results section later describes the difference between this translation/linting and IDEA's default translation. The linting portion utilizes ktlint, an open-source project for linting with automated formatting for Kotlin [8]. Because the headless IDEA translator implementation only gives a raw string as output, ktlint can format this string to Kotlin's given style guidelines to create prettified, Kotlin-standard code [8].
Three main categories for these visitors were created to aim at: Android guidelines, format, and performance. The format category includes most default ktlint rule sets. The performance category focuses on optimization null-safeties, primitives, and other computational tasks. Finally, the category Android is used to ensure the code complies with the tips and guidelines of the Android developers documentation. The visitors rely on both the JetBrain's class representation of the language constructs, and the tokenization of syntactical terms depicted by an abstract syntax tree, a small section shown below.

Listing 1: AST Example

```
KtFile (KtFileElementType.kotlin.FILE)
  KtPackageDirective
  LeafPsiElement "package"
  PsiWhiteSpaceImpl
  KtDotQualifiedExpression
    KtDotQualifiedExpression
      KtNameReferenceExpression
        LeafPsiElement "org"
      LeafPsiElement "."
      KtNameReferenceExpression
        LeafPsiElement "phenoapps"
        ...
```

3.1 Global Variable Liveness

This visitor uses three different data structures to analyze the usages of global variables. The visitor finds all available classes in the source code and creates a stack of classes by pushing each class name it sees. Once a class is pushed, it visits the declaration section of the class and finds all variables, which are then mapped to a variable hash map with class names as keys. Next, binary expressions are searched within that class to find assignment statements to those variables. Once a new class is found, the old class is popped so a new class can be recorded in the set and variable maps. Once the analysis completes a full liveness report is given to the user. This shows the user whether these global variables could possibly be finalized with a given value or possibly stay as a var declaration if they have multiple reassignments. A simple example below shows the usage of such an analysis; however, the scope of this segment may not reflect its entire utility as production code may have many more variables and assignments.

Listing 2: Kotlin Liveness Example

```
class A {
  var count = 0
  val name = "Grunky"
  lateinit var sum: Int
  fun f() {
    sum = 0
    count++
  }
  fun g() {
    count--
  }
}
```

Variable ID	Line Occurrences
count	1,6,9
sum	3,5
name	2

Table 2: Table of variable liveness

3.2 Life Cycle Functions

As described previously in the introduction, an Android application's life cycle is defined by a given state machine. This state machine has various methods that the user should normally implement when trying to develop an activity. This visitor simply uses an array data structure to keep a static list of the names of these life cycle functions. The visitor will then visit any named function within the code and attempt to find the name in the life cycle method list. Once all classes are visited, the user is notified whether or not they are missing some life cycle methods. This visitor message should be considered more of a warning as some life cycle methods are not required to implement.

3.3 Life Cycle Bogging

Similar to the above visitor, this rule set uses the following data structures to capture behaviour of each given life cycle function: a method-to-block-count map, and a function-name-to-AST-node map. The visitor visits all possible functions and saves their name and AST node to the hash map. The visitor then visits all block expressions as well and increments the count of block segments that occurs within a given method. This is based (but not an exact implementation) of cyclomatic complexity, which gives a value to a function given the number of statements within it. Intuitively, each function should not have too many statement blocks; although, when the method being analyzed is a life cycle function this is even more pertinent, as these methods should not contain highly complex code.

3.4 For Optimization

This simple visitor finds all for expressions and analyzes the loop range construct. The visitor will find any usages of methods in the range construct and notify the user that a new variable should be created with the method calculating its range as its value. This is a simple optimization that patches the times where users are calling a method too many times within a for expression (specifically the range construct).
This example shows the improper usage of methods within the range of a for expression:

```
for (i in 1..arrayOf(1,8,9,2).size) {
  println(i)
}
```

Instead, the 'arrayOf' call should be hoisted above the for expression and saved within a variable.

```
val size = arrayOf(1,8,9,2).size
for (i in 1..size) {
  println(i)
}
```

3.5 Not Null Assertions

The not null assertion rule set is one of the simplest within this project. The visitor simply finds all leaf nodes of the AST which are of the form '!!'. This will find all null-assertion operator usages within Kotlin. The user should consider leaning away from this unsafe operator as it will throw null pointer exceptions when possible [9].
Here is an implementation that uses the null assertion operator.

Listing 3: Not null assertion example

```
val file = File(path)
val ktFile = FileUtil
  .getParentFile(file)!!.path
```

A simple solution to avoiding the use of null assertions is to use the null safety operator along with the Elvis operator to supply a default value if a null is actually encountered [9].

Listing 4: Not null assertion replacement example

```
val file = File(path)
val parent = FileUtil
  .getParentFile(file)?.path ?: "-1"
if (parent == "-1") {
  println("error")
}
```

3.6 Null In Binary Expression

This visitor looks for binary expressions that are comparing anything to null. Because it is such common practice in Java to compare an object to null to ensure that it has a value, this is an abundant pattern found after translation. Kotlin's null-safe syntax allows the user to replace this code with a few alternatives [9].

Listing 5: Null in a binary expression example

```
fun main(args: Array<String>) {
  if (args == null) {
    println(args.size)
  }
}
```

In Kotlin, the null safety operator '?' is used within a statement that is calling a method or retrieving a value member from a possibly null object s.a: "obj?.call()" [9]. Another solution to replacing the null equality check is to use the 'let' function in Kotlin which supplies a code block to a given object method call which is only run when the caller is not null.
This is an example of the code above replaced with the null safety operator.

Listing 6: Null safety operator usage example

```
fun main(args: Array<String>) {
  println(args?.size)
}
```

Here is an example of the code above replaced with the let function. Within the let block, a special variable 'it' is created that represents the object that calls let.

Listing 7: Let function example

```
fun main(args: Array<String>) {
  args?.let {
    println(it.size)
  }
}
```

It is also possible to change 'it' into any other name when expanding the function within let.

Listing 8: Expanded let function example

```
fun main(args: Array<String>) {
  args.let { obj ->
    println(obj.size)
  }
}
```

3.7 Override Methods

As mentioned previously, Android developers recommend that methods that are overridden should not solely call the parent object's method (and do nothing else) [5]. In the example below, class B should simply not implement 'f'. The open keyword used below simply means that class A can have child classes, and the function 'f' can be overridden by its children.

Listing 9: Open modifier example

```
open class A {
  open fun f() {}
}
class B : A() {
  override fun f() {
    super.f()
  }
}
```

3.8 Primitive Arrays

Another simple and straightforward visitor pattern is finding usages of primitive arrays. Kotlin provides specialized array instances that should replace any usage of primitive arrays. Primitive arrays lead to unnecessary auto-boxing where more objects are being created than necessary [10]. This visitor simply searches for all type elements within the AST and if any primitive types are found they are replaced with their specialized counterpart.

3.9 Unreachable Code

This visitor searches for all return statements within the Kotlin code. Once a return statement is found, the visitor backtracks to the block it is called within. If the return statement found is not the last statement then there is unreachable code within this block. It is a good rule to generally not include statements that will never run in code. Whether this is a mistake or commented section it is against the Kotlin guidelines.

3.10 Results

As expected in language translations, the result is not exactly a guaranteed working piece. Here are a few examples where both this project and IDEA translation create results that could be fixed with new visitors.

3.11 Incorrect Translations

Listing 10: Constructor used on interface

```
list.setOnItemClickListener(object:
  AdapterView.OnItemClickListener() {
    // item click code
})
```

The error is very subtle and comes from the "()" constructor on the OnItemClickListener. This is actually an interface, and no constructor should be used for interfaces. This is a result of the current IDEA translator.
Another common error found across these translations that is more specific to each language is the usages of member methods. Methods pertaining to an object for Java are not necessarily named the same for Kotlin. One such example below shows how the usage of "getBytes" from Java has translated over to Kotlin. Developers need to have knowledge on both the Java and Kotlin side to know which methods should replace these Java functions. In this case, replacing "getBytes" with "toByteArray" is a functional replacement.

Listing 11: Example where getBytes is used but not accepted by the compiler.

```
for (i in 1..size) {
  if (i != 0) {
    fstream.write(",".getBytes())
  }
  fstream.write(headers[i].getBytes())
}
```

Finally, methods are not correctly overridden after translation. For example, the below method is overriding the onOptionsItemSeleted method for an Activity class. It is apparent that the Java to Kotlin translation does not add an 'override' keyword modifier to respective functions that are annotated with '@Override'. Instead, like the example below, the '@Override' annotation is kept. This could be another possible visitor pattern when wherever a '@Override' annotation is found, the corresponding function must be appended with the override keyword.

Listing 12: Example where a method is overridden but only an annotation is given.

```
@Override
fun onOptionsItemSelected(i:MenuItem) {
    when (item.getItemId()) {
        android.R.id.home -> {
            setResult(RESULT_OK)
            finish()
            return true
        }
    }
}
```

3.12 Error analysis

The unexpected results and incorrect translations described in the section above relate to the current state of Android Studio and j2kCLI's translations and linting. Table3 shows the number of errors found after translation and linting in Android Studio 3.1. Because Kotlin is a fairly new language, it is possible that these values will change as Android Studio develops their translator and linting tools. An extra column to Table3 was added to compare total errors when j2kCLI fixes all global variable declarations. This table shows that almost half of the errors were due to other cascaded errors from incorrectly declared global variables. A future addition to the global variable liveness pattern will be to automatically deduce how each global should be declared.

AS	j2kCLI	j2kCLI-globals-fixed
86	160	84

Table 3: Errors after translation.

3.13 Run time Results

Typically, calculating run time is trivial in programming languages. The developer can simply track the system clock time before and after a given statement and calculate the run time in between this interval. Unfortunately, there is no equivalent way to calculate the run time of a user interaction with a GUI followed by the given function. Therefore, the following results are a non-exact formulation of run time and do not at all use the system clock to determine the speed. The run times given were simply generated from the timer on a mobile device. For the

GUI environment, the user will click on the 'Convert Java to Kotlin' option while simultaneously starting the timer, and stopping the timer once the conversion is complete.

The runtime for j2kCLI is substantially faster than AndroidStudio's 'Convert Java to Kotlin' function. Because both j2kCLI's and AndroidStudio's translation functions are based on JetBrain's j2k module in their Kotlin plugin, they must have similar translation times but differ in post processing. Post processing could range from inter-procedural analysis to using similar linting techniques described in the visitor patterns.

Source	Number of Files	AS	j2kCLI
FieldBook	42	3m33s	17s
Verify	9	9s	6s
GeoNav	4	8s	4s
Inventory	20	12s	6s
OneKK	35	1m42s	10s
Watershed	11	24s	6s

Table 4: j2kCLI translation and linting runtime vs AndroidStudio

4 Conclusion

It is apparent that j2kCLI has developed a faster run-time than its GUI equivalent, and Android Studio detects about the same amount of general errors for both translators. Stepping forward, j2kCLI should maintain the development of visitors and seek to automate certain decision tasks for the developer. Consider the case where Android Studio's translation and linting will generate global variables with a nullable version of the original type, this is an approach that opens the source code to null pointer values but also reduces the compile time errors. j2kCLI seeks to not only reduce the number of compile time errors but also improve the maintenance time for developers without introducing nullable types. Further work may look at how variable liveness data flow results can be leveraged to enhance this area of translation, along with other various patterns such as detecting pure functions. Therefore, further developments to this project include augmenting the visitor rule sets to capture more types of optimization and guidelines. j2kCLI is taking a first step, with a novel approach, at translating Java to Kotlin over a CLI using various JetBrains open source libraries.

5 Acknowledgements

This material is based upon work supported by the National Science Foundation (NSF) under NSF-BREAD Grant No. 1543958. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] PhenoApps Organization Webpage http://phenoapps.org/

[2] T.W. Rife and J. A. Poland, *"Field Book: An open-source application for field data collection on Android"*. In Crop Science 54, 1624-1627, 2014. DOI: 10.2135/cropsci2013.08.0579.

[3] PhenoApps Github Hackathon Repository https://github.com/trife/Hackathon

[4] JetBrains' Github Repository for the Kotlin plugin https://github.com/JetBrains/kotlin

[5] Android Developers Documentation *"Documentation website"* https://developer.android.com/

[6] Chaney Courtney (2017) *Open source application development for phenotypical data acquisition,* Kansas State University, Manhattan, KS, USA

[7] M.L. Neilsen, S.D. Gangadhara, T. Rife, *"Extending watershed segmentation algorithms for high throughput phenotyping"*. in Proceedings of the 29th International Conference on Computer Applications in Industry and Engineering, Denver, CO, Sept. 26-28, 2016.

[8] Ktlint Github Repository, downloaded from https://ktlint.github.io/

[9] Kotlin Webpage, downloaded from https://kotlinlang.org/

[10] Java documentation, downloaded from docs.oracle.com/javase/tutorial/java/data/autoboxing