Effectiveness of Kotlin vs. Java in Android App Development Tasks

Effectiveness of Kotlin vs. Java in Android App Development Tasks

Luca Ardito, Riccardo Coppola, Giovanni Malnati, Marco Torchiano

Politecnico di Torino
Department of Control and Computer Engineering
Turin, Italy
name.surname@polito.it

Abstract

Context: Kotlin is a new programming language representing an alternative to Java; they both target the same JVM and can safely coexist in the same application. Kotlin is advertised as capable to solve several known limitations of Java. Recent surveys show that Kotlin achieved a relevant diffusion among Java developers.

Goal: We planned to empirically assess a few typical promises of Kotlin w.r.t. known Java's limitations, in terms of development effectiveness, maintainability, and ease of development.

Method: Our experiment involved 27 teams of 4 people each that completed a set of maintenance tasks (both defect correction and feature addition) on Android apps written in either Java or Kotlin. In addition to the number of fixed defects, effort, and code size, we collected, though a questionnaire, the participants' perceptions about the avoidance of known pitfalls.

Results: We did not observe any significant difference in terms of maintainability between the two languages.We found a significant difference regarding the amount of code written, which constitutes evidence of better conciseness of Kotlin. Concerning ease of development, the frequency of NullPointerExceptions reported by the subjects was significantly lower when developing in Kotlin. On the other hand, no significant difference was found in the occurrence of other common Java pitfalls. Finally, the IDE support was deemed better for Java than Kotlin.

Conclusions: Some of the promises of Kotlin to be a "better Java" have been confirmed by our empirical assessment. Evidence suggests that the

effort in transitioning to Kotlin can provide some advantages to Java developers, especially regarding code conciseness.Our results may serve as the basis for further investigations on the properties of the language.

Keywords: Android, Effectiveness, Kotlin, Java, Maintenance

1. Introduction

Kotlin is a modern programming language, appeared in 2011, which represents an alternative to Java, with which it can seamlessly coexist. Many pieces of evidence are available in the literature underlining that Kotlin is gaining traction among Android software developers. In a previous study, we mined all Android apps hosted on the F-Droid platform and updated after October 2017: we found that nearly one-fifth of them featured Kotlin code, with 2/3 of those projects featuring more Kotlin than Java code [1]. Similar trends have been reported by Oliveira et al. regarding the number of StackOverflow questions about Kotlin programming for Android and GitHub repositories with Kotlin [2].

One of the main design guidelines that led to the development of the Kotlin language is a better handling of null values. In the Java language, without the usage of specific checks, the handling of *null* values can lead to NullPointerExceptions (NPE). Several studies in the literature report the prominent role of NullPointerExceptions among the reasons for Android application to crash. Coelho et al. report that near 30% of all stack traces collected upon the Android app crash contained NPEs as their root causes [3]. The authors also underline the difficulty in protecting the code against those exceptions, especially when the app does not have access to third-party source code. NPEs can also happen – as Payet and Spoto report – in the link between the XML layouts and explicit application code casts [4]. Such a link is obtained utilizing the very commonly used setContentView and findViewById methods. These method calls are very crucial, and frequent operations are executed every time the components of the application screen are instantiated. The effects of those issues are amplified by misuses of the exception handling mechanisms provided by Java, which are documented frequently among Android developers [5].

Readability and conciseness are considered key-features of the Kotlin language, especially for what concerns the declaration of objects and classes with numerous attributes [6].

The novelty of the Kotlin language, and the easiness in adapting existing (and possibly long-running) Java projects to it, suggests the need for an evaluation of the benefits guaranteed to developers from such transition. Many advantages are reported by works in the specialized literature, but to the best of our knowledge, their empirical assessment is still missing. With this work, we aimed at assessing some assumed advantages of Kotlin with respect to Java in the context of Android development and maintenance. To do so, we conducted a controlled study with undergraduate students, a sample that can represent average Kotlin developers due to the low experience possessed – as of today – by developers with such language.

In light of an ever increasing adoption of Kotlin for Android development, this empirical assessment aims to provide practical evidence that could help in a transition from Java to Kotlin. In particular we focused on possible effects on maintainability, conciseness, and avoidance of a few common pitfalls.

The remainder of the paper is organized as follows: Section 2 provides some background for Kotlin programming, its characteristics, and the recent trends of its diffusion, and it provides a brief review of related work in literature; Section 3 describes the goal, procedure, participants and material of the experiment, along with possible threats to the validity of our findings; Section 4 discusses the threats to the validity of this study; Section 5 reports the results of the experiment, that are discussed in section 6; finally, Section 7 concludes the paper.

2. Background

Kotlin first appeared in 2011, but its first stable release was distributed in February 2016. In May 2017, Kotlin became a first-class language on Android, and support was provided by the Android Studio DE since release 3.0 of October 2017. The popularity of Kotlin increased rapidly since then. The State of Developer Ecosystem in 2018 shows that Kotlin is mainly used for mobile and Server applications working mainly in Oreo and Nougat in Android, and JDK 8 in servers. According to statistics provided by JetBrains, only around 40% of Kotlin developers have adopted the language for more than one year[1].

[1] `https://www.jetbrains.com/research/devecosystem-2019/` Last visited January 2020

Kotlin is a statically typed programming language that runs on the Java Virtual Machine (JVM) and fully interoperates with Java: it is possible to mix Kotlin and Java code in the same application, to call Kotlin code from Java code and vice versa [7]. The two languages share several commonalities [2], and the official documentation of Kotlin itself reports its main characteristics by means of comparisons with Java.

Kotlin takes a pragmatic approach, such as not re-implementing the entire Java collections framework making it compatible with the JDK collection interfaces without breaking any existing project implementations. For example, Kotlin still supports Java 6 bytecode because almost half of the Android devices still run on it. It is possible to start using Kotlin for small parts of a large project, including a few UI components and simple business logic. The possible coexistence between Kotlin and Java can be deemed as one of the main factors that are fueling the transition to Kotlin for Android developers.

As a first example of features that are not supported by Java, Kotlin also allows functions in addition to classes to be first level constructs. In Kotlin, everything is an object, even numeric values that in Java are treated as primitive types. Kotlin provides the ability to extend a class with new features without having to inherit from the class or use any design pattern such as Decorator [8] through special declarations called *Extension Functions* and *Extension Properties*.

On the other hand, Kotlin does not feature some characteristics of the Java language, like checked exceptions, static members, non-private fields, and the ternary operator.

A complete description of the features of Kotlin is out of the purpose of this paper[2]. The primary objective of our work has been instead to verify some of the peculiarities of Kotlin, mostly regarding the avoidance of common Java development pitfalls [9]:

- *Nullability*: the typical convention used throughout Java APIs is to let a method return a `null` reference to represent the absence of a result. This approach, when not accompanied by appropriate checks, may lead to NPEs, which reportedly is one of the most common causes for crashes in Android apps [3] [10]. Java 8 introduced the `Optional` class to provide API with a clear way to represent "no result" as an

[2]A large set of open resources about the Kotlin language is available online at `https://kotlinlang.org/docs/reference/`

alternative to returning `null`, but not as a general-purpose solution to the nullability problem [11]. Kotlin provides a way to declare nullable variables explicitly (`?`) and a safe-call operator (`?.`) that can be used in conjunction with the *elvis* operator (`?:`) to avoid most NPEs.

Figure 1 reports side-by-side examples of equivalent Kotlin and Java code. We can observe how Kotlin allows declaring a nullable variable – by default variables are non-nullable – and to use safe call and elvis operators to achieve safer and more compact code.

```
var bob :  Person?  = null;
//...
return bob?.department?.name; // safe call
```

```
Person bob = null;
// ...
if(bob!=null)
  if(bob.department!=null)
      return
bob.department.name;
return null;
```

```
var bob :  Person?  = null;
//...
return bob?.name:"<?>"; // ?:  elvis
```

```
Person bob = null;
// ...
return
bob!=null?bob.name:"<?>";
```

Kotlin | Java

Figure 1: Nullability examples in Kotlin vs. Java

- *Mandatory Casts*: Java often requires several explicit casts to let the compiler cope with type conversions, this makes code longer and hard to read, in addition, a wrong cast could be accepted by the compiler and result into a run-time exception; Kotlin introduced *smart casts* and a *safe* (nullable) cast operator (`as?`).

  Figure 2 report an example of a safe cast, in Kotlin and Java. Safe casts are capable of eliminating the possibility of triggering a `ClassCastException` at run-time. As it is evident from the comparison, the safe cast in Java requires a more verbose syntax – that we reported with the usage of the ternary operator – with respect to that needed by Kotlin. Such a higher verbosity can be deemed as a deterrent for developers to extensively use the practice of safe casting, hence increasing the likelihood of generating ClassCast exceptions.

- *Long argument lists*: the invocation of Java methods uses a strict positional argument mapping. Therefore methods may require passing

Kotlin	Java
`val p: Person? = x as? Person`	`Person p = x instanceof Person?(Person)x:null;`

Figure 2: Mandatory casts examples in Kotlin vs. Java

many arguments even if they assume default or null values; writing overloaded methods might help in such cases, but it may require significant effort without covering all cases. Kotlin adopts a solution to this issue by defining default values for arguments and allowing – in addition to positional arguments – passing arguments by name. Other recent languages have adopted similar solutions, e.g., default values for arguments are provided by Python.

- *Data Classes*: often, a program requires the creation of classes whose primary purpose is to hold data. The amount of boilerplate code required by Java to implement these classes can be relevant. The additional code can often be mechanically derivable from the data: such automatic derivation is done by libraries that are not part of the standard Java library, e.g., in project Lombok[3]. Kotlin introduced the *Data Classes* that the compiler is able to process to generate all the required boilerplate code automatically. In our prior investigations about Kotlin, we found out that the amount of LoCs savings for a data class with few fields can be of up to 90% w.r.t. the Java equivalent. An example of Kotlin class and its Java equivalent is reported in Figure 3.

The main contribution of our work is a comparison between Java and Kotlin in the context of Android Mobile Applications, and specifically when performing maintenance tasks on apps written in either language. We perform this comparison with undergraduate students attending the course of Mobile Application Development, inspired by the work done by Kosar et al. [12] for setting up the experiment.

2.1. Related Work

The present manuscript is related to experiments with students in comparing different programming languages or practices, and to literature ded-

[3] `https://projectlombok.org` Last visited March 2019

```
data class User(
val name:  String,
val age:  Int)
```

Kotlin

```
class User
   private String name;
   private int age;
   public User(String name, int age){
     this.name=name; this.age=age;
   }
   public String getName(){ return name; }
   public String getAge(){ return age; }
   public String toString(){
     return "User(name="+name+",age="+age+")");
   }
   public boolean equals(Person){...}
   public int hashCode(){ ...  }
}
```

Java

Figure 3: Data class example in Kotlin vs. Java

icated to the comparison between Kotlin and Java. This section reports a summary of relevant work in these fields.

2.1.1. Studies on Kotlin

Since Kotlin has been released recently, the literature does not include many works related to this topic.

Shah et al. [13] analyzed how to perform code obfuscation with Android applications written in Kotlin. Bryksin et al. [14] discussed a methodology for detecting anomalies – i.e., code fragments are written in ways different from the typical ones for a given language – for Kotlin apps. Skripal et al. proposed an Aspect-Oriented extension for Kotlin [15]. Maeda et al. [16] designed a domain-specific language to specify syntax rules based on Kotlin. Belyakova [17] analyzed the dependencies between different language features. Mateus and Martinez [7] performed an empirical study on the quality of Android apps written in Kotlin, finding that the quality of Android apps is increased by the usage of Kotlin in terms of the presence of code smells.

Flauzino et al. [18] performed a comparative study between Java and Kotlin: they compared 50 Java and 50 Kotlin projects to understand if Kotlin contains fewer smells than Java. Code smells are clusters of negative decisions made by developers in software design that do not directly affect the execution flow of a program but are potential causes of future problems, increasing the complexity, maintainability, and even the cost of software [19].

Their findings support the hypothesis that Kotlin presents fewer code smells than Java. With this paper, however, we did not focus on code smells but on maintenance aspects of code development.

Banerjee et al. [20] performed comparisons between the usage of Java and Kotlin for developing Android applications. They conclude that the usage of Kotlin makes the development of Android applications easier while reducing the number of errors and bugs in the code. The principal limit of the work by Banerjee et al. lies in the fact that their assumptions are based only on coding tasks executed by the authors (thus, significant researcher biases can be introduced), and no empirical evidence is provided to support them. The results of the present manuscript are in line with those authors' findings but – to the best of our knowledge – we provide the first empirical assessment of the claimed advantages of the Kotlin language when compared to Java.

2.1.2. General Programming Language Comparisons

A large number of works in the literature have performed programming language comparisons. For instance, Nanz et al. performed comparisons of 22 different programming languages of 4 different families. They found that functional and scripting languages are more concise than procedural and object-oriented language, and that compiled and strongly-typed languages are less prone to run-time failures than interpreted or weakly-typed languages [21].

Prechelt performed a comparison of seven programming languages, including C++, Java, and several scripting languages, concluding that modern scripting languages offer reasonable alternatives to C and C++ even for tasks that require substantial amounts of computations [22].

Singh performed empirical studies on programming languages used for scientific computing, hinting that in such context, Fortran may be deemed as preferable over Java or C++ [23].

Chen et al. studied the effects of the first programming language that is taught to high-level students, finding that no specific programming language provided better performance in subsequent courses sustained by the students [24].

3. Experimental design

We designed an experiment to be conducted in the context of a Mobile Applications Development course within a Computer Engineering MSc. de-

Table 1: GQM Template for the study

Object of Study :	usage of Java and Kotlin programming languages
Purpose :	comparing
Focus :	effectiveness in avoiding common pitfalls
Context :	maintenance and development tasks performed on Android applications by students
Stakeholders :	developers, researchers

gree, attended by 108 students. During the course, the students usually attend practical labs where they are required to work together in groups to develop code for a course running project. The experiment took place during two such labs and involved working on both a small application and the course running project.

This section follows the reporting guidelines proposed by Jedlitschka et al. [25] and the APA Manual [26] to organize the discussion of the experimental design. More specifically, the following subsections provide details about the high-level goal of the experiment, the participants that were involved, the overall experimental design, and the individual research questions that we formulated. For each research question, we report the materials, the procedure, and the metrics that were used to answer them.

3.1. Experiment Goal

We report the design, goal, research questions, and procedure adopted in this study following the Goal Question Metric (GQM) template [27], as summarized in Table 1. The *goal* of the experiment can be expressed as:

> *Analyze the usage of Java and Kotlin programming languages for the purpose of comparing with respect to their effectiveness in avoiding common pitfalls from the point of view of developers in the context of maintenance and development tasks performed on Android applications.*

3.2. Participants

The experimental units of the experiment were the student groups formed for the Mobile Applications Development course. The groups were formed by

picking students ID randomly, in order to avoid any bias in the composition of the groups that could be introduced by allowing the students to compose the groups as they desired. All the groups were formed by four students enrolled in the course and attending the Computer Engineering MSc degree at Politecnico di Torino.

The sample of the experiment is clearly a convenience sample that might be representative of small teams of novice developers.

Following recommended good practices [28], the subjects were rewarded with points for participating in the experiment. Based on the correctness of their answers, each subject earned up to a 10% bonus on their assignment grade for the course.

3.3. Design and Procedure

The experiment follows a structure that was standard between subjects, with two treatments and two groups. The participating groups worked in two consecutive lab sessions: the first was a warm-up and is not used for the experiment, the latter constituted the actual experiment.

The participating groups have been divided into two sets; each set was administered the tasks with different languages (Java vs. Kotlin) in a different order in the two sessions. After the second lab session, the participating groups were asked to answer a questionnaire about their experience. The questionnaire had the following objectives:

i. Understand the characteristics of the group;
ii. Evaluate the level of understanding of the application that they inspected;
iii. Assess the issues encountered by the group;

The items of the questionnaire were organized into three different sections corresponding to the aims defined above. We report the full questionnaire in Table 2.

The tasks performed by the researchers and the students are summarized in the BPMN diagram reported in Figure 4.

We aimed at answering three different research questions, based on the outcomes of the experiments and on the answers to the questionnaire. In Table 3, we briefly report the research questions and the materials, tasks, and questionnaire sections that are used to respond to each of them. Detailed descriptions of the procedure to answer each RQ are reported in the following subsections.

Table 2: Questionnaire Structure

Group	N	Question	Type	Options
Context	1	What is your group ID?	String	-
	2	How many people are in your group	Numerical	-
	3	How many of you have worked as professional java developers?	Numerical	-
	4	How many of you have worked as professional developers in other languages?	Numerical	-
	5	On average what is your experience in Java programming?	Ordinal	(i) Less than one year (ii) Between one year and three years (iii) More than three years
	6	In this lab what language has been assigned to your group?	Categorical	Java / Kotlin
	7	Have any of you developed programs using Kotlin?	Categorical	Yes / No
	8	What is the Java knowledge of the most experienced member in your group?	Ordinal	(i) Novice: up to 20 classes projects (ii) Intermediate: 20 to 50 classes projects (iii) Advanced: 50+ classes projects
i	1	How easy was to understand the overall structure of the code	Likert	Very Easy - Very difficult
	2	What is the purpose of class RecordingItem?	Categorical	(i)Manage audio registration (ii) Send registration to server (iii) Store registration in database (iv) Wrap registration data (v) Notify the OS when the registration data changes
	3	How many defects did you find in the App?	Numerical	-
	4	How many defects were you able to fix?	Numerical	-
	5	How long did it take to fix the defect(s)? (In minutes)	Numerical	-
	6	Which classes contained defects?	Categorical	(i) Main Activity (ii) FileViewerAdapter (iii) DBHelper (iv) RecordingItem (v) RecordingService
ii	1	Did the IDE (e.g., autocomplete) help in writing code?	Likert	Very Little — Very Much
	2	Did you often experienced `NullPointerExceptions`?	Likert	Never — Very Frequently
	3	Did you often encounter problems with methods having long argument lists?	Likert	Never — Very Frequently
	4	How much the effort required to write classes containing mainly data compare to the added value?	Ordinal	Much Higher — Much Lower
	5	How much the effort required to write code to handle class casts compare to the added value?	Ordinal	Much Higher — Much Lower

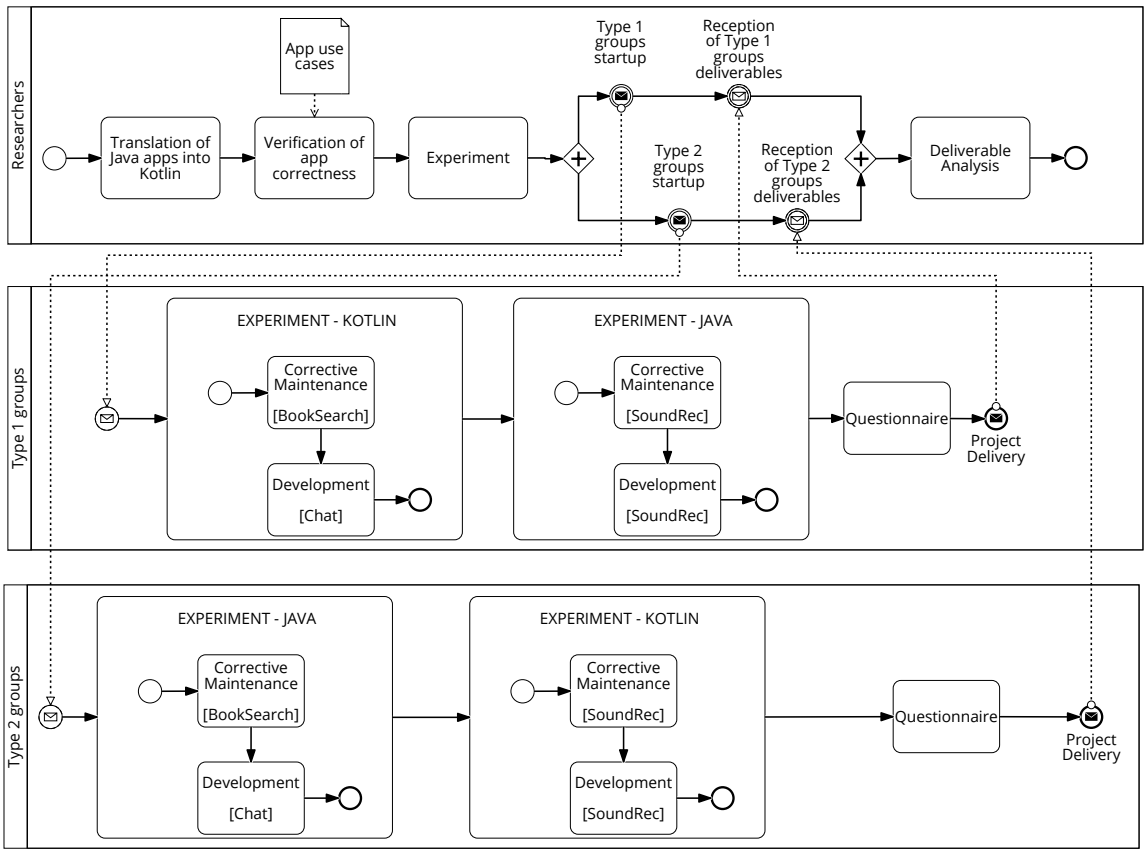


Figure 4: BPMN diagram of the experimental procedure

3.4. RQ1: Maintainability

One of the most cited properties of the Kotlin language is that it makes code easier to understand and thus more maintainable. We hence formulated the following Research Question:

RQ1: *Does the use of Kotlin vs. Java affect the maintainability of Android projects?*

In our study, we focus on corrective maintenance, considering the specific activities of defect location and correction. We aimed, in practice, to understand whether Kotlin's "better syntax" makes it easier to detect the location of a defect and the subsequent code change less difficult.

Table 3: Summary of the research questions

RQ	Materials	Experimental Tasks	Analyzed Variables
RQ1: Maintainability	BookSearch SoundRecorder Questionnaire (sec. 1)	Corrective maintenance	Understanding level Defect location accuracy Fix effort
RQ2: Conciseness	Course-running project Questionnaire (sec. 2)	New feature development	Lines of Code No. of Classes
RQ3: Coding Pitfalls	Questionnaire (sec. 3)	Post-experiment reflection	Experienced pitfalls

Table 4: Characteristics of the applications.

	Java		Kotlin	
App	Classes	LOCs	Classes	LOCs
Booksearch	5	471	5	450
SoundRecorder	13	1525	12	1340

3.4.1. Materials

The maintenance task of the experiment was based on two small applications (both available in Java and in Kotlin). The main characteristics of the two apps are shown in Table 4.

The first application, Booksearch[4], uses the OpenLibrary API[5] to search book-related information and display their cover images. It is quite a small application counting just five classes with 471 LoCs. This application was used in the pre-experiment.

The second application, SoundRecorder[6], is a tool for performing record and playback of sounds. The application package, albeit larger than Booksearch, can be considered still rather small: it counts 1525 LoCs in 13 classes. The second task with this application had the purpose of making the student familiarize with the assigned task.

Both applications are open-source and available on GitHub. Since the application packages are written using Java, the translation to a Kotlin version of the application (needed to allow the participants to perform maintenance tasks in such language) was performed by one of the authors of the paper.

[4]Available at: `https://github.com/shrikant0013/android-booksearch`

[5]Available at: `https://openlibrary.org/dev/docs/api/search`

[6]Available at `https://github.com/dkim0419/SoundRecorder`

While the IntelliJ Idea IDE is capable of automatically translating from Java to Kotlin, we opted for a manual translation performed by one of the authors – having more than ten years of experience in Android development – in order to have the code as close as possible to an application natively developed in Kotlin. For each app, we identified the main use cases that were reproduced on the Kotlin versions by another author of the app. The execution of all the use cases allowed to validate the correctness of the translated apps, and to verify that they provide the same functionalities of their Java counterparts[7].

The first section of the questionnaire administered to the participating groups concerned the maintainability concepts measured to answer RQ1.

3.4.2. Experimental tasks

The participants in the experiment had to perform a task of corrective maintenance on the two apps described above.

The two applications have been injected with two defects each. The defects are equivalent in both applications, one resulting in a NullPointerException, and one to an exception caused by a missing element in the layout hierarchy of the application. Specifically, the NullPointerExceptions were obtained by removing instructions to find a specific view (hence resulting in a null view on which the user can operate) and by removing the call to the initialization of a class. The missing element errors were both obtained by commenting method calls used to populate the GUI of the apps. The bugs were injected in the application code by one author of the paper, and their presence - and their ability to cause crashes - was checked by the other authors independently. Both typologies of bugs are frequently mentioned among the most frequently occurring ones for Android applications [29] [30]. Hence they can be deemed representative of defects happening during typical (either industrial or open-source) Android app development projects.

The goal for the participants was to detect the faults, locate the defects, and fix them.

3.4.3. Hypotheses and Variables

The following null hypotheses were defined to answer RQ1:

[7]We report as a digital appendix the use case narratives of the applications: https://figshare.com/articles/Effectiveness_of_Kotlin_vs_Java_in_Android_App_Development_Tasks_-_Use_Cases_of_experimental_subjects/11808018

Hu_0 There is no difference between the understanding level achieved when using Java or Kotlin.

Hl_0 There is no difference between the capability of locating a defect when using Java or Kotlin.

Ht_0 There is no difference between the reported time required to correct a defect when using Java or Kotlin.

The variables considered in our analysis correspond to the answers collected through the questions in group 1 of the questionnaire.

Besides, we defined three derived measures, that were automatically computed based on the answers to the questionnaire:

Purpose understanding is defined starting from item *ii.2*. The item asks for a specific class in the application. One of the five options is correct; the others are wrong. *Purpose understanding* is a dichotomous variable whose levels can be either *correct* or *wrong*. More specifically, the RecordingItem class is used in the SoundRecorder app to manage data about recordings; hence the Purpose understanding measure was *correct* for all the experimental subjects that selected the fourth answer to question ii.2.

Location accuracy is defined starting from item *ii.6*. The item asks the respondents to identify the classes where the defects are located.

Two out of five classes are expected as a correct answer. We adopt an information retrieval approach and compute the accuracy of the answer.

In particular, *Defect Location Accuracy* (LA) is a ratio measure defined as:

$$LA = \frac{TP + TN}{TP + TN + FP + FN}$$

Where TP are the true positive, TN are the true negatives, FP are the false positives, and FN are the false negatives.

More specifically, the two defects of the SoundRecorder app were injected in the RecordingItem and FileViewerAdapter classes. Hence, the maximum score for the Defect Location Accuracy was obtained if and only if the respondents checked these two classes only in question *ii.6*.

Fix effort is defined starting from item *ii.5*, that in the questionnaire collects the time employed by each group to fix the defects, and item *ii.6*, that reports the defects supposedly identified by the groups.

Fix Effort is defined as the ratio of the number of answers checked for question *ii.6* and the time estimated by the group for fixing the defects; hence, it serves as a self-estimate of the average effort (in minutes) to fix one defect.

3.4.4. Analysis method

Concerning the first research question (RQ1), we analyze three aspects:

- Understanding: we analyze the *Purpose understanding* variable, and we compare the odds of a correct answer when the program is written in Java vs. Kotlin. To this end, in order to assess Hu_0, we apply a Fisher test for 2×2 contingency tables that test the null hypothesis that the odds ratio is equal to one.

- Defect location: we analyze the variable *Location accuracy* to assess Hl_0, in particular, we apply a Mann-Whitney test to check the null hypothesis that there is no difference between the medians.

- Time to fix a defect: we analyze the variable *Fix effort* to assess Ht_0 by using Mann-Whitney test.

3.5. RQ2: Conciseness

Another common claim of Kotlin advocates is that it lends itself to write more compact code. This impacts productivity and indirectly, also maintainability.

We hence formulate the following Research Question:

RQ2: *Does the use of Kotlin vs. Java makes the code more concise?*

We consider conciseness at the macroscopic level, which means less code, both in terms of the number of classes and LoCs.

3.5.1. Materials

The students worked on a larger running project, which is developed throughout the whole course. The running project consists of an app to help people share books. The app had to allow users to sign up easily and set up a basic profile; then users can make books available for sharing, providing all the relevant pieces of information by accessing some shared database. The users can search for shared books and get in contact, via the app, with the book owner in order to arrange the withdrawal and successive return; as a consequence of each sharing, users' reputation must be updated.

The average final size of the projects was around 30 to 40 classes per project, with an average total of 6KLOC, including both Java and Kotlin.

The second section of the questionnaire administered to the participating groups concerned the conciseness concepts measured to answer RQ2.

3.6. Experimental Tasks

The students were asked on a weekly basis to perform development tasks on the course-running Android project.

For the purpose of the evaluation of Kotlin vs. Java usage for the maintenance tasks of Android applications, we designed two features to be implemented by the students.

The specific features were defined by one of the authors of the paper, and were designed to be related to the category of the application under development, compatible with the subjects' expertise with Android, and feasible in the time frame allocated to the experiment.

The features that were defined for the participant were: (1) implement a user chat with notifications, using Firebase (referred as CHAT); (2) implement a way to express user ratings for the exchanged books, using a five-star scheme (referred as RATINGS).

3.6.1. Hypotheses and Variables

The following null hypotheses were defined to answer RQ2:

Hc_0 There is no difference between the measured amount of classes written to implement a new feature when using Java or Kotlin.

Hl_0 There is no difference between the measured lines of code written to implement a feature when using Java or Kotlin.

3.6.2. Analysis method

Regarding code conciseness, we focused on two measures collected through static analysis of the submitted experimental assignments:

- Classes: number of classes developed for the required feature;
- Lines of Code: LoCs written to implement the required feature;

Both above measures were used to asses Hc_0 by applying a non-parametric Mann-Whitney test.

3.7. RQ3: Coding Pitfalls

One important principle in the design of Kotlin was to avoid several common pitfalls of the Java programming language.

In this work, we decided to investigate four of the common pitfalls, i.e., *Nullability*, *Mandatory Casts*, *Long argument list*, and *Data Classes*, and are described in section 2.

To evaluate the occurrence of known issues in Kotlin vs. Java programming in the context of Android development, we defined our third and final research question:

RQ3: *Does the use of Kotlin vs. Java effectively avoids the occurrence of common pitfalls?*

3.7.1. Materials

The section (ii) of the questionnaire administered to the participating groups (see table 2) concerned their experience with the occurrence of the investigated common pitfalls. The reported occurrence of the pitfalls was used to answer RQ3.

We decided to use the answer to those questions as proxies of the actual occurrence of the pitfalls. This choice is due to the limited observability of the teams while performing their development task. In fact, the participants wrote the code on their own machines; therefore it was not feasible to install a monitoring plug-in as it would be possible had they worked on lab devices.

3.7.2. Hypotheses and variables

The following null hypotheses were formulated to answer RQ3:

$Hp1_0$ There is no perceived difference in terms of the number of NPEs occurrences with Java or Kotlin.

$Hp2_0$ There is no perceived difference in terms of the number of casts with Java or Kotlin.

$Hp3_0$ There is no perceived difference in terms of issues with long argument lists with Java or Kotlin.

$Hp4_0$ There is no perceived difference in terms of tool support to Java or Kotlin.

$Hp5_0$ There is no perceived difference in terms of effort required to write data classes.

3.7.3. Analysis Method

To analyze the perceived pitfalls (RQ3), we resort to the responses to the items *ii.1* to *ii.5* of the questionnaire. For each variable, we compute the effect size using the Cliff Delta statistic, and we used the relative confidence interval for deciding about hypothesis rejection.

4. Threats to validity

External validity threats concern the generalization of the results. We have considered two real-world (even if small) open-source applications, thus selecting a realistic context for Kotlin or Java maintenance of Android applications. The course running project also has functional requirements that are common among real-world Android apps, so the used software artifacts can be considered as representative of typical Android apps. Hamedani et al. provided a classification of Android apps under twelve different categories [31]. The considered applications in this experiment can be categorized under *Office & Business* and *Music & Video*, both belonging to the top three categories that were identified in the study.

It is also possible that the results obtained regarding bug-fixing efforts are not generalizable to other categories of defects that are proper for Android applications. However, classifications of Android defects are provided by Hu

et al. [29] [30], and NPEs and layout issues are among the most popular categories of bugs.

A final threat to the generalizability of results may be linked to how undergraduate students may be considered representative of Android developers in general. The use of students as participants in experiments is, however, widely recognized: Sjoeberg et al. [28] report that 50% of the 2,969 experiments in 12 leading software engineering journals and conferences between 1993 and 2002 used undergraduate students as participants [28]. Carver et al. define a model for conducting a valid empirical study with students (ESWS). They identify research and pedagogical requirements that need to be managed while preparing and executing an experiment in a university course. In short, researchers have to make sure that the study is well-integrated with the course goals and materials, give realistic time estimates for experimental tasks, properly motivate the participants without revealing the goals, measures and analysis prior to the study, allow students to give feedback, convince the participants of the relevance of what they are learning, avoid conflicts with students' other commitments, and give students feedback on the results of the experiment [32]. All these guidelines were followed in the conduction of the work documented in this paper. Besides, Carver et al. [32] also provide a checklist to explain when the various activities should occur (i.e., before starting the study, as soon as the study begins, during the study, or after the study is completed). The requirements and checklist provide a useful guide for judging how well a study is integrated into the university course and for judging the reliability of the results. We used that checklist to verify the research and pedagogical goals in this study.

Construct validity threats concern the relationship between theory and observation. It is not assured that the *Purpose Understanding*, *Location Accuracy*, and *Fix Effort* metrics defined in this paper are the best possible proxies for providing answers to the Research Questions identified for this study. We measure conciseness in terms of code size, though shorter code could – at least in principle – bear a higher cognitive load, thus reversing the benefits stemming from more concise code. Considering the specific features introduced in Kotlin, we do not believe this is the case though there is no empirical evidence supporting such belief. As explained in section 3.7.1, we decided to measure the occurrence of pitfalls by means of proxies. In practice, we inferred the actual occurrence on the basis of the reported pitfall manifestation, recorded through the questionnaire. While this choice was dictated by practical feasibility reasons, we have no reason to believe any

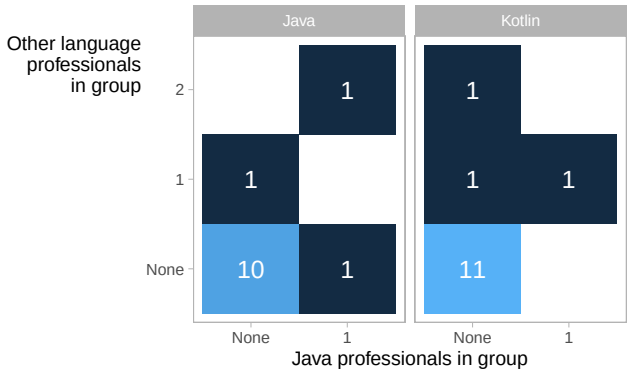


Figure 5: Groups with members having professional experience with Java or other languages

misreporting took place. Moreover, we argue the pitfalls do not represent a problem *per se* but rather in as much they affect the development activities of the developer, thus in this specific case the reported experience of such pitfalls is probably closer to the original construct than a mechanical count of pitfall frequency.

As far as the IDE support, we have to notice that in the presence of participants familiar with Java but not at all with Kotlin, the perceived support could be more related to familiarity than to actual IDE support.

Finally, *Researcher bias* is another possible threat to the validity of this study, since the authors were involved in the creation of the starting Kotlin versions of the two considered apps and the bug injection phase. However, the authors have no reason to favor any language; neither are they inclined to demonstrate any specific result.

5. Results

In this section, we report the results measured for the three Research Questions of the experiment. We also provide details about the population that participated in the experiment.

5.1. Population

The population of experimental units that performed all the required activities consisted of 27 groups, all made up of four students. Thirteen groups performed the development tasks in Kotlin, the others in Java. Overall the

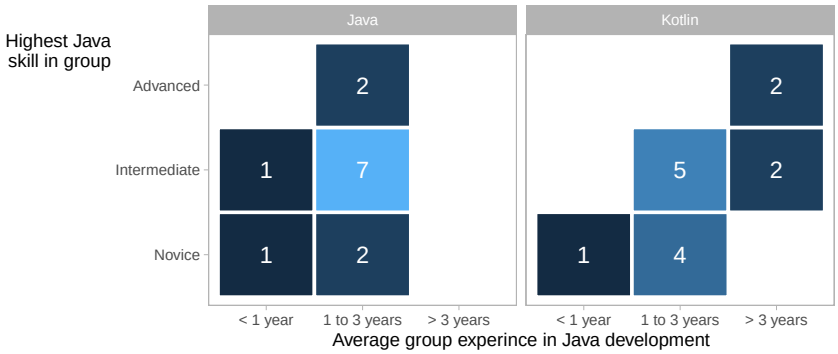


Figure 6: Java skill level and experience of the respondents

experiment involved 108 students aged between 25 years old and 40 years old of different gender and ethnicity.

The professional experience of group members was measured through the answers to question 3 and 4 of the questionnaire. Figure 5 summarizes the answers to those questions for both typologies of groups. Three groups included participants that had professional Java development experience, and overall, 6 out of 27 groups included components with experience as professional software developers in any language. No participant had any previous experience in Kotlin.

The skill level of the population was measured through the answers to question 5 and 8 of the Context section of the questionnaire. Figure 6 summarizes the answers to those questions for both typologies of groups. The experience with Java development was mostly between one and three years, although three groups had no member with more than one year of experience in Java, and four groups included a member with more than three years of experience.

In the whole population, four groups had members having advanced Java skills (i.e., they developed at least one project of over than 50 classes); eight groups had a most experienced member that considered him/herself a novice (i.e., they developed a few projects featuring up to 20 classes); in the remaining fifteen groups the most experienced member considered him/herself an intermediate (i.e., at least one medium-sized project of 20 to 50 classes). Regarding the years of experience with Java, four groups had an average experience of more than three years, and three groups had an average experience of less than one year; the remaining groups had an average experience

Table 5: Null hypotheses for RQ1

Name	Description	p-value	Decision
Hu_0	There is no difference between the understanding level achieved when using Java or Kotlin	0.68	Don't Reject
Hl_0	There is no difference between the capability of locating a defect when using Java or Kotlin	0.81	Don't Reject
Ht_0	There is no difference between the reported time required to correct a defect when using Java or Kotlin	0.43	Don't Reject

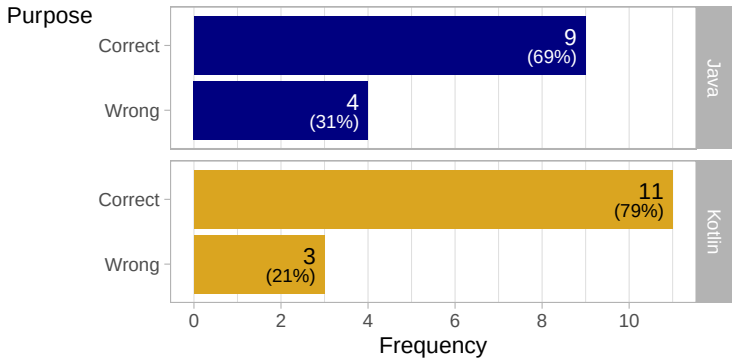


Figure 7: Frequency of correct answer to the purpose understanding question

with Java between one and three years.

5.2. Maintainability (RQ1)

To address the research question concerning maintainability, we focused on three different aspects: the understanding of the code, the defect detection ability, and the time required to fix the defects.

Table 5 reports the null hypotheses formulated to answer RQ1 and the related decisions.

5.2.1. Understanding

Figure 7 reports the measured purposed understanding, based on question i.2 of the questionnaire. We can observe that four out of thirteen Java groups failed in the purpose of properly understanding the purpose of a class of the experimental object. On the other hand, only three out of fourteen Kotlin groups failed in the same task.

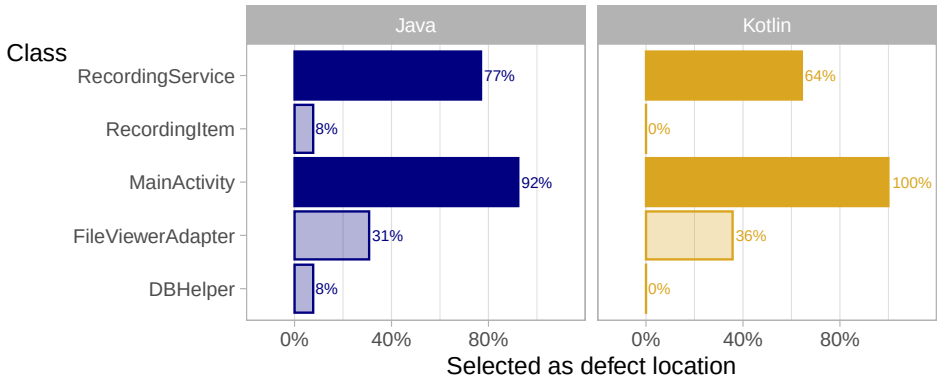


Figure 8: Frequency of the answers to the defect location question selected by the respondents

To test the hypothesis Hu_0, we performed a Fisher-test that provided a p-value=0.68. Therefore we cannot reject the null hypothesis.

Even though the estimated odds ratio is around 2, such difference is not statistically significant.

5.2.2. Defect location accuracy

Concerning the accuracy in defect location tasks, we report in Figure 8 the frequency with which the respondents selected each of the possible answers to question 6 of the questionnaire. As it can be deduced from the graph, all groups working with Kotlin and 92% of groups working with Java were able to find the bug in the MainActivity class of the app. Fewer groups (respectively 77% and 64% of those working with Java and Kotlin) were able to spot the other defect injected in the RecordingService class.

Figure 9 reports the distributions of the defect location accuracy for the two languages. We can observe a substantial similarity that is confirmed by the applied Mann-Whitney test (p-value=0.81). Therefore we cannot reject Hl_0 either.

5.2.3. Defect correction time

To test the hypothesis H_{t0} concerning the time employed in defect fixing tasks, we analyzed the average time reported by the groups, normalized by the number of defects they had found. The distribution of the average time per found defect is reported in Figure 10.

The hypothesis was tested using a Mann-Whitney test that returned a p-value of 0.43. Therefore we cannot reject the null hypothesis.

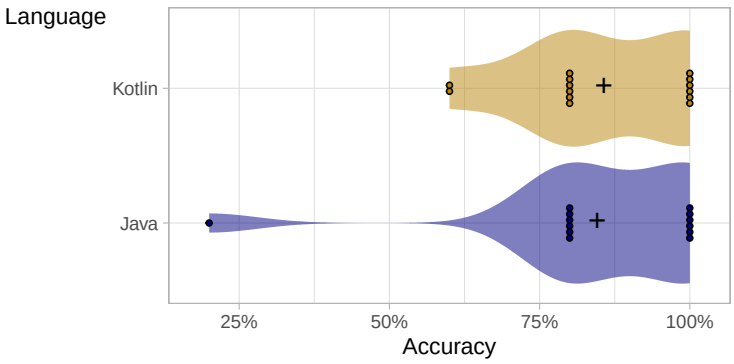


Figure 9: Distributions of the defect location accuracy metric

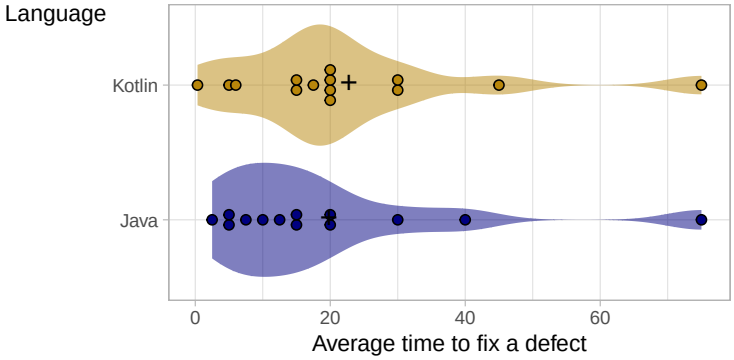


Figure 10: Distributions of the average time to fix a defect

The average time to fix defects is similar between Kotlin and Java, with no statistically significant difference.

5.3. Conciseness (RQ2)

After the development tasks, we measured both the number of new classes added to the application design and the lines of new code written overall.

Table 6 reports the null hypotheses formulated to answer RQ2, and the related decisions.

In Table 7, we report the measured amount of classes and code that were added by the respondents to implement the required features. The table reports the raw count of classes and code and the percentage over the total amount of code of the application. We also report in the last column the number of data classes that were developed. The code was automatically

Table 6: Null hypotheses for RQ2

Name	Description	p-value	Decision
Hc_0	There is no difference between the measured amount of classes written to implement a new feature when using Java or Kotlin	0.2138	Don't Reject
Hl_0	There is no difference between the measured lines of code written to implement a feature when using Java or Kotlin	0.033	**Reject**

measured by a script that leveraged the open-source cloc tool[8]. Blank and comment lines were not included in the computation.

From the table, it can be seen that the number of classes and the amount of code development had a very high variability between groups. Groups that worked with Kotlin to implement the new features produced a number of classes that ranged from 3 to 12 (246 to 1568 lines of code). Four different groups developed data classes. Groups that worked with Java produced a number of classes that ranged from 0 to 26 (583 to 4745 lines of code).

The distribution of the number of new classes is reported in Figure 11. We observe a lower number of classes developed for the participating groups using Kotlin. The mean number of classes developed with Kotlin is 7 while it is 12 for Java; the medians are respectively 8 and 13.

The hypothesis Hc_0 was tested using a Mann-Whitney test that returned a p-value=0.2138. Therefore we cannot reject the null hypothesis.

The effect size can be considered small, as Cliff's Delta is 0.29; the 95% CI for the effect size is (-0.24; 0.68): it includes the 0. Therefore, it cannot be considered as statistically significant.

The same kind of analysis can be applied to the amount of Lines-Of-Code (LOCs) written in order to implement the new feature. The distribution of the LOCs by language is reported in Figure 12. The median LOCs reported for Java is between 1526, while for Kotlin, it is Less than 589.5.

The hypothesis Hl_0 was tested using a Mann-Whitney test that returned a p-value=0.003. Therefore we can reject the null hypothesis.

The effect size can be considered large, Cliff's Delta is 0.65, with the relative 95% CI being (0.24; 0.86); since the CI does not include the 0, the

[8]https://github.com/AlDanial/cloc

Table 7: Absolute and relative added classes and LOCs for the development of the required features

		Added		
Language	Group	Classes (%)	LOCs (%)	Data classes
Kotlin	1	7 (17.1%)	483 (8.3%)	0
	3	8 (12.3%)	337 (5.2%)	0
	5	12 (24.0%)	1568 (18.9%)	4
	7	8 (12.7%)	745 (8.1%)	0
	9	5 (4.4%)	515 (4.0%)	0
	11	8 (13.6%)	978 (12.9%)	1
	13	3 (8.8%)	322 (3.2%)	1
	15	9 (20.4%)	664 (10.2%)	0
	17	9 (17.0&)	972 (13.0%)	0
	19	8 (7.1%)	460 (4.0%)	1
	21	5 (9.1%)	380 (5.0%)	0
	23	6 (12.8%)	835 (14.4%)	0
	25	3 (5.1%)	246 (4.3%)	0
	27	9 (14.1%)	744 (7.5%)	0
Java	2	17 (24.3%)	4099 (37.3%)	0
	4	4 (11.8%)	679 (15.6%)	0
	6	17 (37.8%)	1526 (31.6%)	0
	8	22 (31.4%)	2208 (24.9%)	0
	10	26 (32.9%)	4745 (45.0%)	0
	12	17 (30.0%)	2688 (32.0%)	0
	14	0 (0.0%)	583 (7.3%)	0
	16	19 (16.4%)	3810 (31.9%)	0
	18	3 (8.6%)	775 (18.1%)	0
	20	13 (17.6%)	742 (14.9%)	0
	22	6 (15.4%)	1755 (29.6%)	0
	24	9 (15.0%)	1424 (14.7%)	0
	26	2 (6.1%)	647 (15.4%)	0

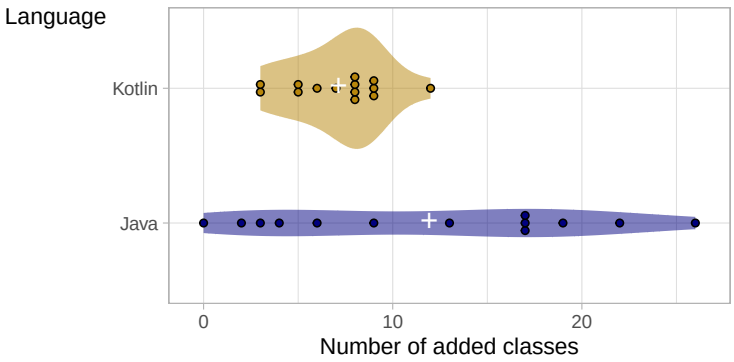


Figure 11: Distribution of number of classes developed.

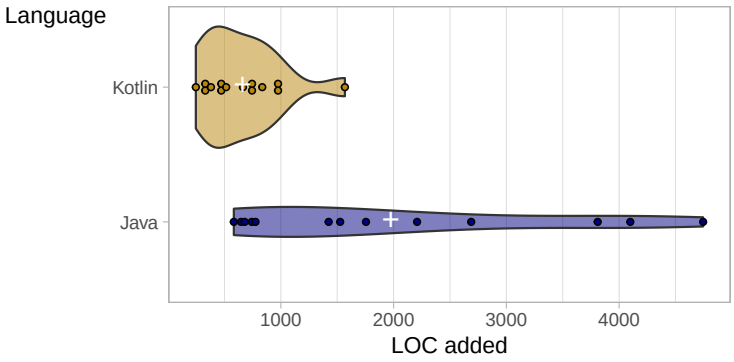


Figure 12: Distribution of LOC written

difference can be considered significant.

As far as the LOCs are concerned, we can, therefore, reject the null hypothesis Hl_0 and conclude that there is a significant difference in terms of the number of LOCs written when developing the same feature with Kotlin or Java. It is also worth underlining that only four groups used data classes in Kotlin. Also, one of those groups was the one that wrote most code to implement the new feature (1568 LOCs). The low number of developed data classes suggests that the Kotlin language is more concise than Java, even without taking into consideration the data class construct.

5.4. Pitfalls (RQ3)

To answer our research questions about the experienced pitfalls by the developers, we analyzed the self-reported perceptions on the questionnaire. The null hypotheses for RQ3 are reported in Table 8.

Table 8: Null hypotheses for RQ3

Name	Description	Decision
$Hp1_0$	There is no perceived difference in terms of number of NPEs occurrences with Java or Kotlin	**Reject**
$Hp2_0$	There is no perceived difference in terms of the number of casts with Java or Kotlin	Don't Reject
$Hp3_0$	There is no perceived difference in terms of issues with long argument lists with Java or Kotlin	Don't Reject
$Hp4_0$	There is no perceived difference in terms of tool support to Java or Kotlin	Don't Reject
$Hp5_0$	There is no perceived difference in terms of effort required to write data classes	Don't Reject

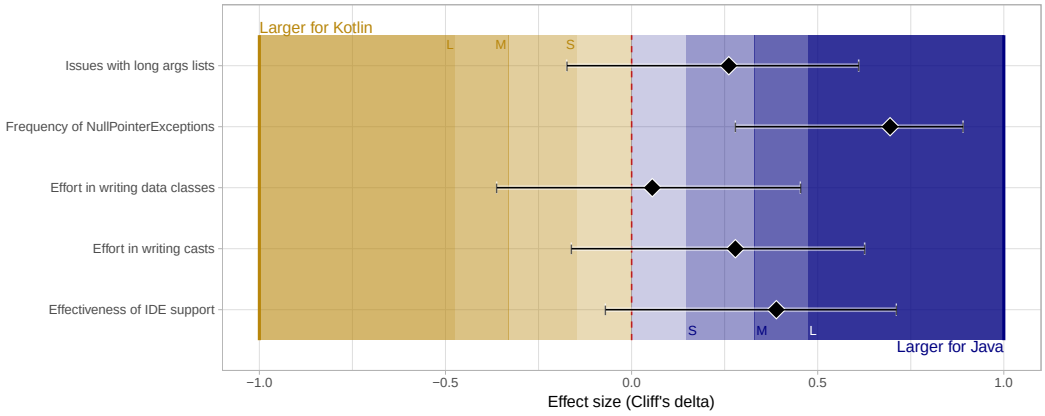


Figure 13: Effect size with confidence interval of language for different aspects.

The distributions of the answers to the perception questions are reported in Figure 13. For each aspect, the figure reports Cliff's Delta estimate effect size as a diamond shape, and the relative 95% CI is as a whiskered segment. The shades of the background represent the standard quantification ranges for the practical magnitude of the effect size.

With reference to the 95% confidence interval, the following assumptions can be based on the respondents' perceptions:

- More NPEs and null-pointers related issues are likely to happen when coding with Java; the difference in the respondents' answers is statistically significant, and the size of the effect is of large magnitude;

- The usage of Kotlin for writing code casts is perceived as less effort consuming than Java; this difference is not statistically significant though the effect size magnitude is medium;
- The respondents considered the definition of long argument lists with Kotlin easier than with Java; this difference is not statistically significant though the effect size magnitude is medium;
- Java is perceived as better supported by tooling than Kotlin; this difference is not statistically significant though the effect size magnitude is medium;
- Negligible differences are observed in terms of the effort required to write data-intensive classes by the developers, in Java or Kotlin.

6. Discussion

The overall goal of our comparative investigation on Java vs. Kotlin was focused on assessing the consequences of a possible transition from one programming language to the other. The switch could occur at different levels: from a single project to a unit, up to a whole company.

We know that, by design, Kotlin is fully compatible at the bytecode-level with Java; therefore, a smooth, progressive transition between the two languages is technically possible. The focus of our research questions addressed the development part of the transition that entails:

- Understandability of the code written in the new language;
- Defect location effectiveness;
- Defect correction efficiency;
- Code conciseness;
- Error proneness.

6.1. Maintenance (RQ1)

The first three former items can be comprised under the broader area of maintenance and were addressed by our first research question (**RQ1**). The results from our experiment show that no specific difference was detected in

terms of the ability to detect defects, to fix them, and the effort required to perform the change.

*There is no evidence that the use of Kotlin, as a substitute for Java, either enhances or lessens software maintainability (**RQ1**).*

It is important to asses how this finding can be generalized. For this purpose, we have to consider the background of the participants in our experiment: they are students in a computer engineering master's degree, only three teams in each language group reported some professional experience. Overall we may consider them close to a junior developer profile. Moreover, we wish to stress that none of our participants had any previous experience in Kotlin. Nonetheless, they were able to detect and fix the defects seamlessly. This may represent evidence in favor of limited risks deriving from the switch to Kotlin in a company, even with little previous knowledge of Kotlin.

From our understanding, it remains an open question whether the lack of evidence in terms of maintainability was due to the confounding factors represented by the characteristics of the participants and the small size of the applications.

6.2. Conciseness (RQ2)

One of the design goals of Kotlin, likewise other recent generation languages, is an increased expressive power that enables writing code in a more concise way. This feature is extremely important because it can improve both the productivity – by reducing the sheer number of keystrokes required – and the understandability of the code. The second research question (**RQ2**) in out study addressed this aspect. In this respect, we observed a significant effect of using Kotlin on the amount of code written to develop new features. More specifically, we found a large and statistically significant difference in terms of lines of code developed (see Figure 12): Java development required writing three times more lines than Kotlin. Concerning the number of classes created in the development of new features, while the average is 67% higher for Java, the difference is not statistically significant.

*We found evidence that the usage of Kotlin led to writing more concise code than Java (**RQ2**).*

While we believe that, in general, the adoption of Kotlin can lead to a more concise code, the extent of code reduction depends on the specific type of application and environment. As we mentioned in section 4, our

experiment was able to provide evidence limited to the Android environment and specifically concerning two small applications.

An important aspect that deserves further investigation is the capability of modern construct that is present in Kotlin – but we argue in many modern programming languages – to actually enable more concise code in many different settings.

Also related to the previous research question, we wonder it conciseness stemming from more expressive constructs also translates into a higher understandability of the code.

6.3. Error reduction (RQ3)

An important selling point of Kotlin, as opposed to Java, is the capability to reduce programming errors through a set of new syntax constructs. Kotlin offers several new constructs, though, in our experiment, we focused our attention on four of them: Nullability, Mandatory Casts, Long argument lists, and Data Classes (see details in Section 2).

Our study provides evidence suggesting a reduction in the occurrence of `NullPointerException` during development. For teams using Java, *NPE* occasionally occurred too frequently, while for teams using Kotlin, they happened mostly rarely. Although not statistically significant, we also observed a lesser reduction of issues with long arguments lists and with the effort in writing casts. Finally, no evidence was found of any reduction in the effort devoted to writing data classes.

Our respondents also reported slightly better language support for Java than for Kotlin. Even though the difference is not statistically significant, it is surprising considering that the producer of the IDE is the same company that designed the Kotlin language. Probably the much longer experience available for Java development allowed for better support.

*We found evidence that Kotlin was able to reduce the frequency of NullPointerExceptions. While no evidence was found concerning effects on other investigated pitfalls. (**RQ3**).*

The extent to which these findings can be generalized to experienced programmers is unknown. We can speculate that NPE would represent a lesser problem for experienced developers, though both writings cast and dealing with long arguments lists can be expected to be issued less dependent on the developers' experience. The essentially inconclusive result concerning data classes might be due to the architecture of the application and the

characteristic of the required new features, which did not require the use of any data class (see Table 7).

Concerning the IDE support, we have to keep in mind that no student had any experience with Kotlin's development before the experiment. This bias could have affected the participants' perception. Therefore we could have measured the familiarity with the language rather than the actual support provided by the IDE.

The limited evidence and partially counter-intuitive results concerning the coding pitfalls deserve further investigation. Research should be aimed at understanding whether and under which circumstances the adoption of Kotlin allows avoiding the pitfalls.

6.4. Practical implications

The above findings, though preliminary and subject to some generalizability limitations, can bear significant implications. We can summarize such implications for three categories of stakeholders:

- **Developers** willing the transition from Java to Kotlin: we provide evidence that no negative effect on maintainability can be expected, a more concise code is likely to be written, and that the Kotlin language is able to reduce the occurrence of one of the four pitfalls we investigated.

- **Researchers** interested in Kotlin: we highlighted a few interesting aspects, regarding some of them we could not get any conclusive result, e.g., the effect on writing casts, long arguments lists, and data classes. Such aspects are good candidates for further studies, as well as confirmative replications of our findings.

- **Tool builders**: we found some hint of support that is perceived to be better for Java than for Kotlin, further studies could confirm this and provide directions for improving the IDE support.

7. Conclusion

Kotlin is a modern programming language that represents a relevant alternative to Java in several development domains. In particular, it has been adopted as an official development language for the Android OS. In this work, we focused on the main promises of this new language. In particular, we investigated how Kotlin can improve the maintainability of code, make code

more compact, and avoid common pitfalls. For this purpose, we carried on an experiment in the context of a Mobile Application Development course in an MSc. degree. The experiment compared the Kotlin programming language to its ancestor, Java.

With our experiment, we found that the usage of Kotlin apparently does not affect the maintainability with respect to Java, when working on two small applications. At the same time, we found evidence that the adoption of Kotlin leads to more compact code when the subjects of the experiments were asked to develop new features for an ongoing software project.

The adoption of Kotlin makes a few common Java annoyances less frequent, thus making the development safer. We registered evidence of a reduction in the frequency of Null Pointer Exceptions. We also observed fewer issues with long argument lists and reduced effort when dealing with casts, although no definitive evidence could be found with this respect.

Those findings represent a first empirical assessment of the advantages of Kotlin with respect to Java, as reported by many works in the related literature. The findings showed that most of the promises of the development of the Kotlin language are reflected by the code produced and by the developers' perception.

The study has few limitations, mainly due to the academic settings: the software artifacts were small, the developers were students with limited experience; therefore, the number of bugs and tasks that were studied was limited. The study may not be representative of bigger, real-world projects that require many development tasks and may expose many typologies of defects and issues. It is important to collect more evidence for different and possibly larger applications and outside the Android ecosystem.

As future work, we hence plan to investigate the advantages brought by Kotlin in other domains, e.g., server-side development. Also, we aim at finding whether other expected Kotlin benefits hold.

References

[1] R. Coppola, L. Ardito, M. Torchiano, Characterizing the transition to kotlin of android apps: a study on f-droid, play store, and github, in: Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics, pp. 8–14.

[2] L. M. T. Victor L. de Oliveira, Felipe Ebert, On the adoption of kotlin

on android development: a triangulation study, in: 27th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2020), IEEE, pp. 1–6.

[3] R. Coelho, L. Almeida, G. Gousios, A. v. Deursen, C. Treude, Exception handling bug hazards in android, Empirical Software Engineering 22 (2017) 1264–1304.

[4] É. Payet, F. Spoto, Static analysis of android programs, Information and Software Technology 54 (2012) 1192–1201.

[5] J. Oliveira, D. Borges, T. Silva, N. Cacho, F. Castor, Do android developers neglect error handling? a maintenance-centric study on the relationship between android abstractions and uncaught exceptions, Journal of Systems and Software 136 (2018) 1–18.

[6] S. Hellbrück, A Data Mining Approach to Compare Java with Kotlin, Metropolia Ammattikorkeakoulu, 2019.

[7] B. Góis Mateus, M. Martinez, An empirical study on quality of android applications written in kotlin language, Empirical Software Engineering 24 (2019) 3356–3393.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series, Pearson Education, 1994.

[9] VV.AA., Kotlin Language Documentation, v 1.3, Technical Report, Kotlin Foundation, 2018.

[10] A. Levy, Top 5 crashes on android, `https://www.apteligent.com/technical-resource/top-5-crashes-on-android/`, 2016. Accessed: 2018-02-23.

[11] B. Goetz, Response to "should java 8 getters return optional type?", `https://stackoverflow.com/a/26328555/3687824`, 2014. Accessed: 2018-02-23.

[12] T. Kosar, M. Mernik, J. C. Carver, Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments, Empirical Software Engineering 17 (2012) 276–304.

[13] Y. Shah, J. Shah, K. Kansara, Code obfuscating a kotlin-based app with proguard, in: 2018 Second International Conference on Advances in Electronics, Computers and Communications (ICAECC), pp. 1–5.

[14] T. Bryksin, V. Petukhov, K. Smirenko, N. Povarov, Detecting anomalies in kotlin code, in: Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ACM, pp. 10–12.

[15] B. Skripal, V. Itsykson, Aspect-oriented extension for the kotlin programming language, in: CEUR Workshop Proceedings, volume 1864, pp. 1–6.

[16] K. Maeda, Statically typed domain-specific language to define syntax rules, in: WMSCI 2017 - 21st World Multi-Conference on Systemics, Cybernetics and Informatics, Proceedings, volume 1, pp. 132–135.

[17] J. Belyakova, Language support for generic programming in object-oriented languages: Peculiarities, drawbacks, ways of improvement, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 9889 LNCS (2016) 1–15.

[18] M. Flauzino, J. Veríssimo, R. Terra, E. Cirilo, V. H. S. Durelli, R. S. Durelli, Are you still smelling it?: A comparative study between java and kotlin language, in: Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse, SBCARS '18, ACM, New York, NY, USA, 2018, pp. 23–32.

[19] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Boston, MA, USA, 1999.

[20] M. Banerjee, S. Bose, A. Kundu, M. Mukherjee, A comparative study: Java vs kotlin programming in android application development, International Journal of Advanced Research in Computer Science 9 (2018) 41.

[21] S. Nanz, C. A. Furia, A comparative study of programming languages in rosetta code, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, volume 1, IEEE, pp. 778–788.

[22] L. Prechelt, An empirical comparison of seven programming languages, Computer 33 (2000) 23–29.

[23] D. Singh, An empirical study of programming languages from the point of view of scientific computing, Int. J. Innov. Sci. Eng. Technol 4 (2017) 367–371.

[24] C. Chen, P. Haduong, K. Brennan, G. Sonnert, P. Sadler, The effects of first programming language on college students' computing attitude and achievement: a comparison of graphical and textual languages, Computer Science Education 29 (2019) 23–48.

[25] A. Jedlitschka, M. Ciolkowski, D. Pfahl, Reporting Experiments in Software Engineering, Springer London, London, pp. 201–228.

[26] G. R. VandenBos, Publication manual of the american psychological association (6th ed.), 2010. American Psychological Association, Washington, DC.

[27] R. Van Solingen, V. Basili, G. Caldiera, H. D. Rombach, Goal question metric (GQM) approach, Encyclopedia of software engineering (2002).

[28] D. I. K. Sjoberg, J. E. Hannay, O. Hansen, V. By Kampenes, A. Karahasanovic, N.-K. Liborg, A. C. Rekdal, A survey of controlled experiments in software engineering, IEEE Trans. Softw. Eng. 31 (2005) 733–753.

[29] G. Hu, X. Yuan, Y. Tang, J. Yang, Efficiently, effectively detecting mobile app bugs with appdoctor, in: Proceedings of the Ninth European Conference on Computer Systems, pp. 1–15.

[30] C. Hu, I. Neamtiu, Automating gui testing for android applications, in: Proceedings of the 6th International Workshop on Automation of Software Test, pp. 77–83.

[31] M. Reyhani Hamedani, D. Shin, M. Lee, S.-J. Cho, C. Hwang, Androclass: An effective method to classify android applications by applying deep neural networks to comprehensive features, Wireless Communications and Mobile Computing 2018 (2018).

[32] J. C. Carver, L. Jaccheri, S. Morasca, F. Shull, A checklist for integrating student empirical studies with research and teaching goals, Empirical Softw. Engg. 15 (2010) 35–59.

Author Biography

Luca Ardito Luca Ardito is an Assistant Professor at Dept. of Control and Computer Engineering at Politecnico di Torino where he works in the Software Engineering research group. He received BSc, MSc, and PhD in Computer Engineering from Politecnico di Torino. His current research interests are: mobile development and testing, green software and empirical software engineering methodologies.

Riccardo Coppola Riccardo Coppola is a Post-Doctoral Research Fellow at Dept. of Control and Computer Engineering at Politecnico di Torino, where he received his MSc and PhD degree in Computer Engineering. He is currently a member of the Software Engineering research group, and his research interests include automated GUI testing for web and mobile applications, and the evaluation of non-functional properties of testware.

Giovanni Malnati Giovanni Malnati is an Assistant Professor at Politecnico di Torino. He has participated to several European and national research projects and to many technology transfer activities with private companies, addressing different topics in the areas of embedded, mobile, and multimedia programming. His research activities covers software and network technologies for mobile and pervasive systems, vehicular network applications, indoor positioning systems and multimedia technologies supporting e-learning en-

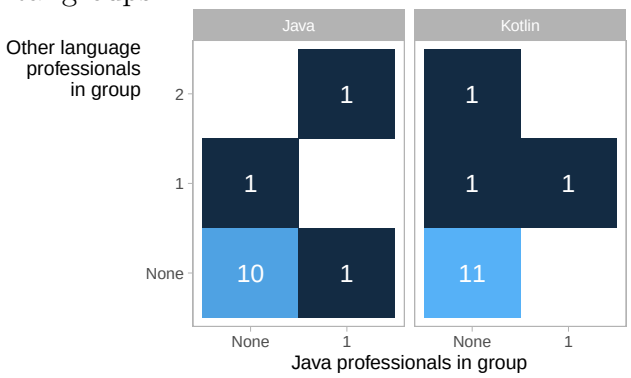
vironments. He is a co-author of seven patents. Since 1999, he cooperates with Istituto Superiore "Mario Boella", participating to a shared laboratory for the development of mobile services and applications. He supervised the research activities of several graduate and PhD students at Politecnico di Torino. He has been the advisor of four PhD students in Computer and Control Engineering and more than 40 master students.

Marco Torchiano is an associate professor at the Control and Computer Engineering Dept. of Politecnico di Torino, Italy; he has been post-doctoral research fellow at Norwegian University of Science and Technology (NTNU), Norway. He received an MSc and a PhD in Computer Engineering from Politecnico di Torino. He is Senior Member of the IEEE and member of the software engineering committee of UNINFO (part of ISO/IEC JTC 1). He is author or co-author of over 140 research papers published in international journals and conferences, of the book "Software Development—Case studies in Java" from Addison-Wesley, and co-editor of the book "Developing Services for the Wireless Internet" from Springer. He recently was a visiting professor at Polytechnique Montréal studying software energy consumption. His current research interests are: green software, UI testing methods, open-data quality, and software modeling notations. The methodological approach he adopts is that of empirical software engineering.

Appendix

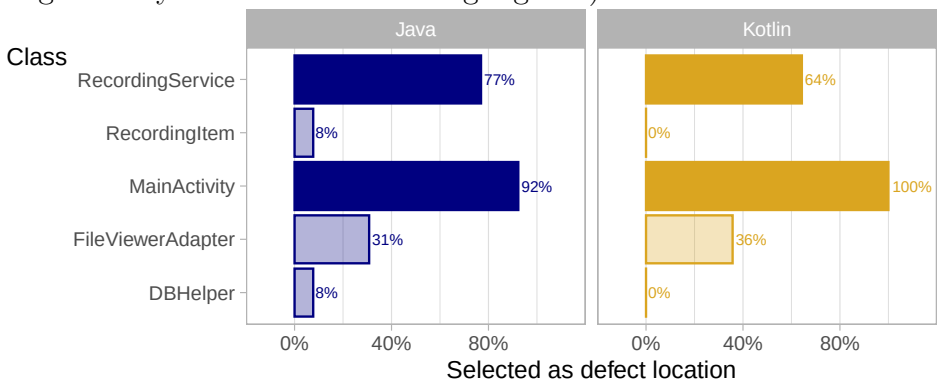
7.1. Population details

Professional experience in Java and other languages in the two experimental groups.



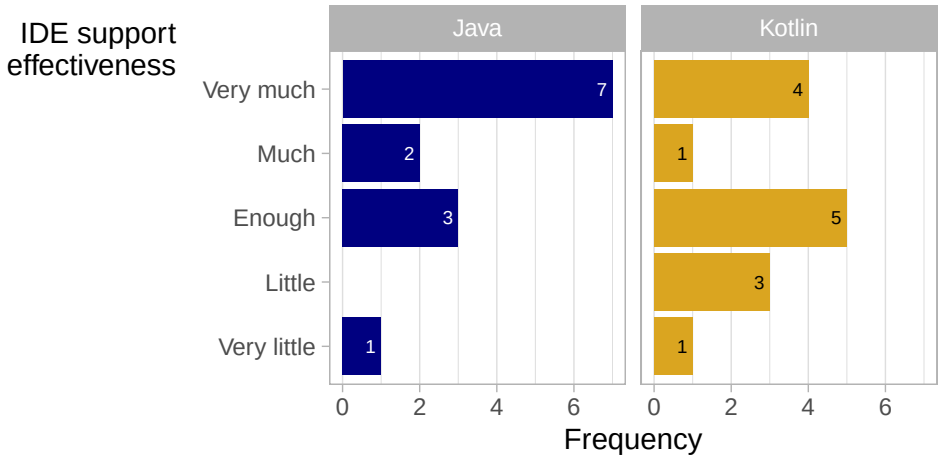
7.2. RQ1

Classes selected as location for defects (correct answers, i.e. classes containing actually seeded defects are highlighted)

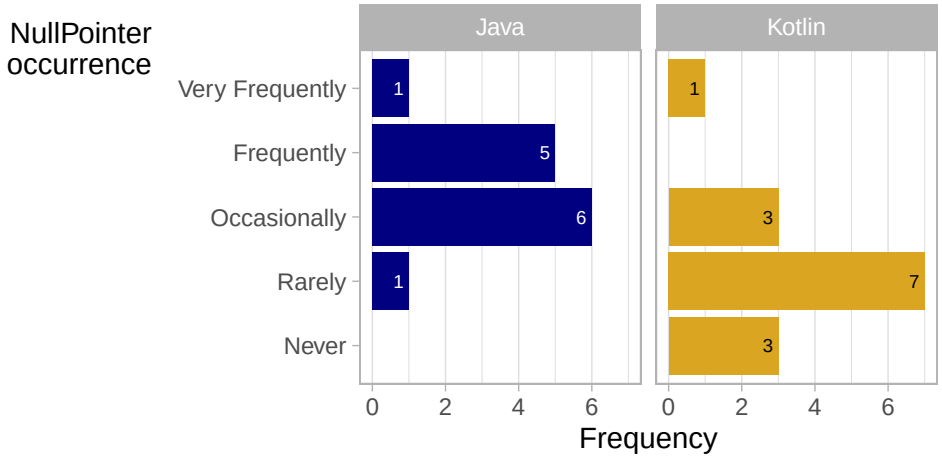


7.3. Detailed answer for perceptions

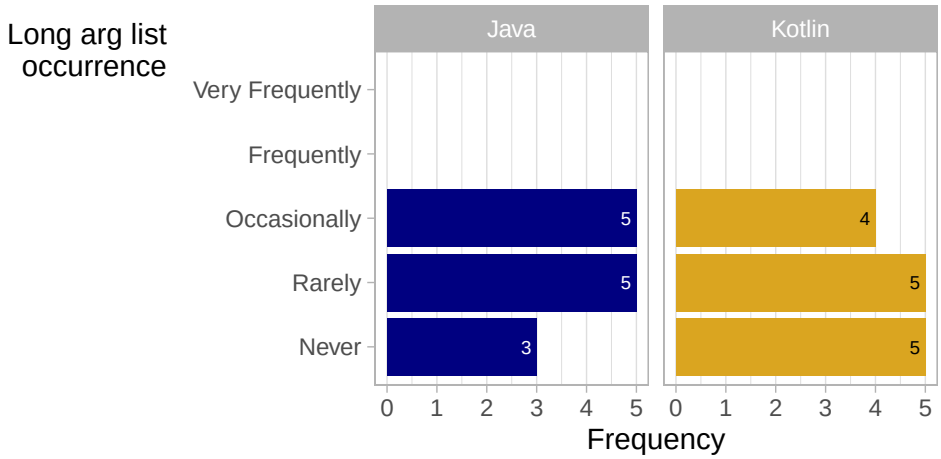
7.3.1. IDE support effectiveness



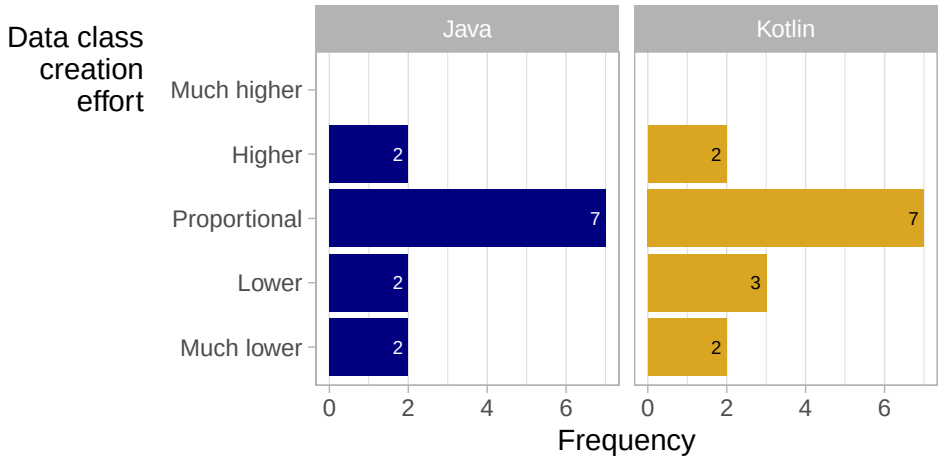
7.3.2. NullPointerException issues frequency



7.3.3. Frequency of Long arguments list issues



7.3.4. Efficacy of data class creation



7.3.5. Effort to write casts