Why Do Developers Remove Lambda Expressions in Java?

Mingwei Zheng[†§], Jun Yang[¶], Ming Wen[†§*], Hengcheng Zhu[‡], Yepang Liu[‖], Hai Jin[§♯]
[†]*Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering*
Huazhong University of Science and Technology, Wuhan, China
[§]*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab*
Huazhong University of Science and Technology, Wuhan, China
[¶]*School of Electronic Information and Communications, Huazhong University of Science and Technology, Wuhan, China*
[♯]*Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, China*
[‡]*Dept. of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China*
[‖]*Dept. of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China*
{zmw12306, claudeyangjun, mwenaa, hjin}@hust.edu.cn, hzhuaq@connect.ust.hk, liuyp1@sustech.edu.cn

***Abstract*— Java 8 has introduced lambda expressions, a core feature of functional programming. Since its introduction, there is an increasing trend of lambda adoptions in Java projects. Developers often adopt lambda expressions to simplify code, avoid code duplication or simulate other functional features. However, we observe that lambda expressions can also incur different types of side effects (i.e., performance issues and memory leakages) or even severe bugs, and developers also frequently remove lambda expressions in their implementations. Consequently, the advantages of utilizing lambda expressions can be significantly compromised by the collateral side effects. In this study, we present the first large-scale, quantitative and qualitative empirical study to characterize and understand inappropriate usages of lambda expressions. Particularly, we summarized seven main reasons for the removal of lambdas as well as seven common migration patterns. For instance, we observe that lambdas using customized functional interfaces are more likely to be removed by developers. Moreover, from a complementary perspective, we performed a user study over 30 developers to seek the underlying reasons why they remove lambda expressions in practice. Finally, based on our empirical results, we made suggestions on scenarios to avoid lambda usages for Java developers and also pointed out future directions for researchers.**

***Index Terms*—Lambda Expression, Empirical Study**

I. Introduction

Lambda expression [1] is an important functional feature, which has been widely applied in functional programming languages, such as Standard ML, Haskell, and so on. Plentiful mainstream *Object-Oriented Languages*, such as Java, C++, and C#, also support lambda expressions to parameterize functionality with time evolves. Since the introduction of lambda expressions in Java 8,[1] there is an increasing trend of lambda adoptions in open-source Java projects as revealed by a recent study [2].

Utilizing the features of lambdas, developers have migrated from anonymous classes to lambda expressions, and from enhanced for loops to `Streams`. Unfortunately, with a wider range of the adoptions of lambda expressions in Java, we also observe an increasing number of misuses of them in practice. In this study, we denote a *misuse of lambda expression* as the case where *a lambda expression is used inappropriately which causes side effects or even induces bugs.* Listing 1 shows an issue (i.e., LUCENE-9478 [3]) from *Apache Lucene*, a large-scale and open-source project. In this example, `DocumentsWriterDeleteQueue` is implemented to advance the queue to the next one on flush, which uses a lambda expression as one of its parameters. However, there are 500 bytes of memory leakage on each full flush due to the inappropriate usage of lambda expression. This is because the lambda expression maintains an implicit reference to the enclosing instance (the current queue object) at runtime in order to access variable `nextSeqNo`, which is not defined in the lambda body. Therefore, on each flush, the new queue will unfortunately keep a reference to the outdated queue which is no longer needed, thus preventing it from being garbage collected. Consequently, memory leaks were observed by developers. Such an issue is reported on JIRA for discussion and marked as the type of *Bug* with the priority of *Blocker*, which reflects the significance of this issue. To fix it, developers have removed such a lambda expression in this context, and replaced it with an invocation to a static method.

```
1 synchronized DocumentsWriterDeleteQueue
2  advanceQueue(int maxNumPendingOps) {
3   ...
4   return new DocumentsWriterDeleteQueue(infoStream,
5     generation + 1, seqNo + 1,
6     () -> nextSeqNo.get() - 1);
7   }
```

Listing 1: Memory Leak Caused by Inappropriate Usage of Lambda Expression (`LUCENE-9478`)

We observe it is pervasive that developers change a lambda expression back into a conventional implementation after introducing it in large-scale open-source projects. For instance,

*Ming Wen is the corresponding author.

[1]We use *lambdas* and *lambda expressions* interchangeably for simplicity.

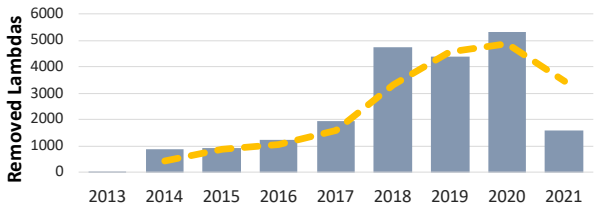


Figure 1: The Trend of Lambda Removals. The number is computed on 103 open-source projects as described in Section III-A.

in project *Apache Camel* [4], we found that 154 lambda expressions have been removed during the last year. Such a number is 784 in project *Apache Geode* [5], which indicates that more than two lambdas have been removed in this project each day on average. Meanwhile, We found that the behavior of developers to delete lambdas is increasing year by year as shown in Figure 1. Worse still, we observe that lambda expressions can often be misused, in which case lambdas will either induce bugs or cause side effects, such as efficiency issues or memory leaks. Therefore, it arouses our great interest to investigate the characteristics of the lambda expressions that are inappropriately used and removed by developers.

Despite its significance and pervasiveness, little is known about the misuse of lambda expressions in Java. In particular, there is still limited empirical knowledge towards *what* lambdas are often removed by developers, *why* do developers remove lambdas, and the *migration patterns* adopted by developers to remove them. The lack of such knowledge impacts negatively for Java developers to correctly use lambda expressions in practice. This study aims to bridge this gap. Specifically, we performed a *quantitative* and a *qualitative* study to investigate the above-mentioned questions. In the quantitative study, we conducted a large-scale empirical study based on 103 Apache projects to understand the characteristics of those lambda expressions that were removed by developers. Specifically, we collected 3,662 real cases of such lambdas and analyzed their characteristics quantitatively. We then compared the characteristics with those of lambdas that have been kept in a project from beginning to end. Our qualitative analysis aims to answer the following research question:

RQ1: *What lambda expressions are more frequently removed by developers?* We found that lambdas built on top of customized functional interfaces, passed to self-defined method invocations are more likely to be removed. In addition, non-empty argument lambdas with more complex bodies are much more likely to be refactored.

In the qualitative study, we performed two separate experiments with in-depth analysis. First, we collected 92 real issues caused by lambda misuses from large open-source projects, and then analyzed such issues manually to characterize the removal reasons, impacts, and code migration patterns of those inappropriate lambdas. Second, we designed and conducted a survey with experienced Github contributors, from a complementary perspective, to further understand the insights why developers remove lambdas and got 25 additional issues in total. With these two experiments, we aim to answer the following research questions:

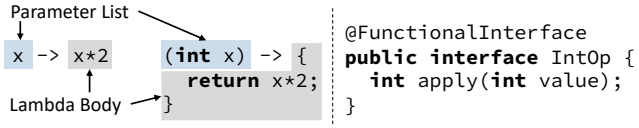


Figure 2: Lambda Expressions and the Corresponding Functional Interface

RQ2: *Why do developers remove lambda expressions in practice? What are the reasons behind and impacts?* We found seven common reasons for lambda removals, including but not limited to performance degradation, poor readability, serialization issues, and lazy evaluation issues.

RQ3: *What are the migration patterns of the inappropriate usages of lambda expressions?* We summarized seven major migration patterns and analyzed the relationship with removal reasons. Our results show that there are common migration patterns for most kinds of lambda misuses, which can facilitate developers to fix those issues caused by inappropriate lambdas. Besides, in order to help developers avoid lambda misuse from the beginning, we summarized five pieces of actionable advice for utilizing lambda expressions appropriately.

The usefulness of this study is well recognized by developers. As mentioned by one developer from *Apache Calcite* [6], "*Awesome work you are doing*". Meanwhile, our empirical results are anticipated by some developers: "*I would be appreciated if you can share your findings of the subject*", "*Thanks for the question and for pointing out the issue*", "*Where will I be able to read about your results*". Hence, we believe this study can shed light on the better utilization of lambda expressions for Java developers in practice. In summary, this study makes the following major contributions:

- ***Originality:*** To our best knowledge, we are the first to comprehensively study the inappropriate usages of lambda expressions in Java.
- ***Quantitative Analysis:*** We collected 3,662 cases of removed lambdas and compared their characteristics with 31,288 kept ones and found that lambda expressions that are large in their sizes, built on top of *customized* functional interfaces, passed to *self-defined* method invocations are more likely to be removed.
- ***Qualitative Analysis:*** We collected 117 real-world issues and conducted a user study to explore the reasons, impacts, and migration patterns of lambda removals. We also generate actionable advice to guide Java programming with lambda expressions.
- ***Dataset:*** We open sourced our collected datasets and the experimental results to facilitate future concerning researches, which is available at GitHub: **https://github.com/CGCL-codes/LambdaMisuse**.

II. Background and Motivation

A. Lambda Expressions

To support functional programming, Java 8 has introduced several functional idioms. For instance, it re-designs the *interface*, introduces *lambda expression*, retrofits the `Collection`

framework, and introduces the *Stream API*. This evolution enables developers to embrace various advantages from both Object-Oriented programming and functional programming. Consequently, there is an increasing trend in the adoption of such functional idioms in open-source Java projects. The usage of certain new Java 8 API methods, especially the Stream API, heavily relies on the usage of lambda expressions. For instance, most `Stream` operations accept parameters that describe user-specified behaviors, which are often in the form of lambda expressions [7]. Therefore, according to an existing study [8], lambda expression is the most accepted function idiom, being accepted by 16% of the investigated projects while the other features, such as the Stream API, are accepted by no greater than 3% of the investigated projects. Due to such a high accept ratio of lambdas and their significance, we put our focus mainly on lambda expressions in this study.

A lambda expression is composed of three parts: *parameter list*, the *arrow token* (i.e., $\rightarrow$), and a *lambda body*. The parameter lists are similar to the formal parameters of a regular method except that the type information can be omitted if it can be inferred by the compiler and the parentheses can be omitted when there is exactly one parameter. The lambda body should be either an expression, or a code block similar to the body of a regular method. In Java, a lambda expression can be assigned to a variable or be passed to a method as an argument if the corresponding type is a *functional interface*, which contains only a single abstract method. Figure 2 shows an example of lambda expressions that doubles the given integer and their corresponding functional interface.

There are several studies trying to understand the usages of lambda expressions in Java. Specifically, Mazinanian et al. presented the first large-scale empirical study on how developers use lambda expressions in Java since its introduction [2]. They observed an increasing trend in the adoption of lambdas in Java. Specifically, developers often employ lambda expressions to simplify source code, avoid code duplication and simulate lazy evaluations. Matsumoto et al. conducted a study to explore the current use status of functional idioms in Java [8]. Their statistical results show that lambda expressions are more widely utilized than `Stream` and `Optional` in Java programming. Especially, they discovered that most developers write lambda expressions for good readability and better performance, while refusing them due to the complications when handling exceptions and compatibility issues. Although it is common wisdom that refactoring a legacy code to a lambda expression might simplify code and enhance program comprehension, Lucas et al. made contradictory observations [9]. Specifically, they found no evidence that the introduction of lambda expressions can improve program comprehension via qualitative and quantitative analysis. Gyori et al. implemented *LambdaFicator* to enable automatic refactors: anonymous classes to lambda expressions, for loops that iterate over `Collections` to functional operations that use lambda expressions [10]. Alqaimi et al. designed *LAMBDADOC* to automatically generate documents for lambda expressions to help readers better understand their functionalities [11]. Besides, Tsantalis et al. investigated the applicability of lambda expressions for the refactoring of clones with behavioral differences [12], and found that lambda expressions enable the refactoring of a significant portion of clones that could not be refactored by any other means.

B. Misuse of Lambda Expressions

With an increasing number of lambda expressions adopted in Java, we also observe that the usages of them might cause side effects or even induce bugs. Listing 1 shows an example. Actually, similar issues can be frequently observed among popular open-source projects. For example, in *Apache MyFaces Core*, we observed excessive object allocations (around 1,000,000 object instances) caused by inappropriate usage of lambda expressions that are invoked many times [13]. Besides, we also notice several serialization issues in *Apache FLINK* caused by lambda expressions [14], [15]. Unfortunately, there is no systematic study that has comprehensively investigated and understood the inappropriate usages of lambda expressions in Java. This study aims to bridge this gap.

Be noted that lambda expression is just a type of syntactic structure, which is often used under certain contexts with other code structures, such as the Stream API. Therefore, the side effects of lambda expressions can be collectively affected by their structures and contexts. Actually, we also tried to investigate the misusages of lambda expression itself excluding its contexts. However, most of the incorrect usages caused by itself (more than 80%) are syntactic issues, which are often manifested as compilation errors. Specifically, we collected 186 issues from Stack Overflow related to the inappropriate usages of lambda expressions without considering their contexts, and found that 80.65% of them led to compilation failures.[2] Such syntactic compilation failures are relatively easier for developers to debug and resolve, and thus the research value is limited. Besides, it is rare to observe such issues in real open-source projects since they are often guaranteed to be compiled successfully when released to the market. Therefore, we take the contexts of lambda expressions into consideration, especially the usages of other functional idioms, i.e., `Stream` API, `Collections` API, etc. This enriches our research scope and enables us to understand the side-effects of lambda expressions with respect to their semantics. It also can provide more in-depth guidance for developers to better use lambda expressions under certain scenarios. We also considered conducting a comprehensive study on all the Java functional idioms. However, collecting such data is not an easy task, which often contains too many noises. For instance, while searching the keyword "Stream" on JIRA, most items returned are related to the Java Process Streams instead of the Stream API. Therefore, in this study, we put our focus on the side-effects caused by lambda expressions as well as their contexts, including the functional APIs that rely on them. In the following parts of this paper, we will clearly distinguish

[2]We provide the details of such empirical results over Stack Overflow on https://github.com/CGCL-codes/LambdaMisuse

lambda expressions and the functional APIs while explaining our findings.

III. Characterizing the Removed Lambdas

In this section, we aim to answer RQ1 by characterizing the lambda expressions that have been removed by developers. Specifically, we constructed two datasets containing the lambda expressions that are, respectively, removed and kept by developers. By comparing the characteristics of the lambda expressions in the two datasets, we can answer RQ1 by presenting the potential factors that make the developers prone to remove a lambda expression.

A. Data Collection

In order to characterize the lambda expressions that were removed by developers, we need to first collect a dataset containing the removed ones, which is denoted as follows:

Removed lambda expressions. Includes lambda expressions that were removed by developers in order to address certain issues. Be noted that removal of *lambda expressions due to simply functional deletion is excluded from our scope*. Our aim is to collect a set of lambda expressions that are representatives of those misused by developers.

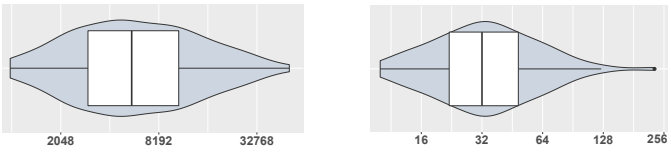
To enable a comprehensive understanding of the characteristics of those removed lambdas, we also collect the following dataset for comparison.

Kept lambda expressions. Includes lambda expressions that have been kept in the project for a long term, which are more likely to be those used correctly by developers.

By comparing the lambda expressions in the above two sets, we can draw a picture of what kind of lambda expressions are likely to be misused and thus should be removed from the projects. Next, we present the data collection process in detail.

1) Project Selection: We select projects from the Apache Software Foundation (ASF) because they are well maintained: they follow a rigorous project management strategy so that we can extract more useful information about the code changes in the commit messages. Specifically, we selected 103 actively maintained Java projects that contain at least 1,000 commits and 10 committers from the ASF project list [16]. On the one hand, a sufficient number of commits guarantees enough code changes for our empirical study. On the other hand, the requirement towards the committers ensures diversified code practices in the projects. Such diversities are important to guarantee the generality of our dataset. As shown in Figure 3, our selected projects have commits ranging from 1.0k to 51.8k (8.71k on average), and have different committers ranging from 10 to 230 (40 on average), which shows that they are large and diversified sufficient.

2) Collecting Removed Lambda Expressions: To identify the removed lambda expressions by developers, we utilized GumTree [17], an AST-based code differencing tool, to analyze those commits that modify Java source files. Since there can be multiple changes to a file in a single commit, when analyzing a commit, we further looked into the edit hunks [18] of the code diff and extracted the removed lambda expressions



(a) # of Commits (b) # of Committers

Figure 3: Commits and Committers of the Selected Projects

at the hunk level, which enables us to apply the following rules to improve the soundness of our dataset.

- We excluded those commits modifying more than 20 source files since they are likely to be tangled with multiple intentions [19], thus introducing noises of irrelevant changes.
- We keep only the commits that either contain an issue ID, or have keywords that may describe an issue (e.g., *bug*, *fix*) in its commit message. Such commits are likely to be made to fix an issue. The complete list is available on our page.[3]
- Merge commits are excluded as they aggregate multiple changes from other branches and are considered redundant.
- We ignored the hunks in which more than 50% of the modified lines are irrelevant to the removed lambda expression since they may not focus on lambda expressions. Instead, the lambda might be deleted in collateral with other changes.
- We ignored the hunks that only delete source code since the intention of such changes is more likely to be functionality detection instead of removing misused lambda expressions.

Be noted that ***rigorous rules*** have been adopted here to collect removed lambdas since our goal is to collect a dataset of high-quality which contains lambda expressions removed by inappropriate usages. Noises can still be inevitable, while we have made our best efforts to collect such a dataset. Finally, we collected 3,662 cases of removed lambda expressions.

3) Collecting Kept Lambda Expressions: In addition, we collected lambda expressions in our selected projects that are kept by developers, which are considered as representatives of the correctly used ones. Specifically, we applied the following rules to filter such lambda expressions.

- It must still exist in the latest version of the code repository.
- It must have stayed in the project for a sufficiently long time since its introduction. In this paper, we consider 24 months to be sufficiently long since it has been revealed that bugs can be usually exposed and repaired within 24 months [20], [21]. Besides, most of the removed lambda expressions (97.37%) collected by us have stayed in the project for less than 24 months as shown in Figure 4a.
- Lambda expressions introduced in commits that modify more than 100 files are excluded since they are usually introduced by large-scale refactoring.

By adopting the above process, we obtained a dataset containing 31,228 kept lambda expressions.

B. Methodology

To answer RQ1, we first inspect the lifetime of the removed lambda expressions. Specifically, for each case, we calculated

[3]https://github.com/CGCL-codes/LambdaMisuse

the time period it has stayed in the project. Additionally, we extracted several features of lambdas that may relate to the removal of it from the following three different perspectives:

The usages of functional interfaces. There are 43 functional interfaces introduced into JDK 8, providing plentiful input/output parameter combinations and covering a wide range of functionalities. Besides, developers can also implement their own customized functional interfaces to enable throwing exceptions or extending other interfaces. We investigated whether those lambda removals are related to the usages of different functional interfaces.

The complexity of lambda expressions. From our dataset, we observed that some lambda expressions are removed due to the poor readability caused by their large sizes. Therefore, we are motivated to explore whether complexity can be a factor related to the removal of a lambda expression. Specifically, we measure the complexity of a lambda expression with respect to four different aspects: number of parameters, lines of code, the height of the AST of the lambda body, and the number of variables accessed in the lambda body.

The contexts of lambda expressions. We investigated whether a lambda expression is likely to be removed under certain contexts, such as the locations and the type of code structures where lambda expressions are used. We found that both kept and removed lambda expressions are mostly used in method invocations. Specifically, the ratio of method invocation is 85.64% for removed lambdas and 85.13% for kept ones. Therefore, we are motivated to analyze the method APIs which invoke the lambda expressions, and see whether the utilized APIs are different over removed and kept lambdas.

Note that when extracting features such as the fully qualified name of functional interfaces, we need to compile the whole project to resolve the type information and method binding. However, some of the revisions may fail to compile. Therefore, for certain features, the analysis is performed on the revisions that can be compiled (including 1,847 removed lambdas and 17,823 kept ones), which is our best effort.

C. Empirical Results

1) Lifetime of removed lambdas: We investigated the lifetime of the removed lambdas, 38.26% (1,401 out of 3,662) of the lambdas have survived no longer than one month. However, 20.23% (741 out of 3,662) of the lambdas are removed after more than one year. Especially, we found one lambda has lived as long as 61 months before its removal. Such results reveal that inappropriate usages of lambdas can be removed as quickly as certain side effects are observed. However, some lambdas can be long-lived in the project, continuously affecting the project until it is discovered by developers, for example, performance degradation.

2) The usage of functional interface: As shown in Figure 5, 35.68% of the removed lambdas are implemented on top of customized functional interfaces (i.e., functional interfaces that are defined by developers) while that ratio is only 28.03% among kept ones. Such a high ratio difference indicates that customized functional interfaces are more likely to be misused. A Chi-Square test [22] of independence was performed to examine the relation between the removal of lambda expressions and usage of customized functional interface, which reveals that the relation was significant at the significance level of 0.05, $\chi^2(1, n = 19,670) = 47.49$, $p = 5.52 \times 10^{-12}$. The effect size φ was 0.05, indicating that the magnitude of the effect is small. This is because customized functional interfaces mostly define more complex functionalities that are not implemented by the current built-in functional interfaces, and thus are much easier to introduce bugs. Besides, built-in functional interfaces are defined with specific functionalities and usually used together with built-in APIs, e.g., `Stream` API with lambdas. Therefore, using such well-defined interfaces is less likely to introduce issues.

> **Finding #1:** *Lambda expressions built on top of customized functional interfaces are more likely to be removed.*

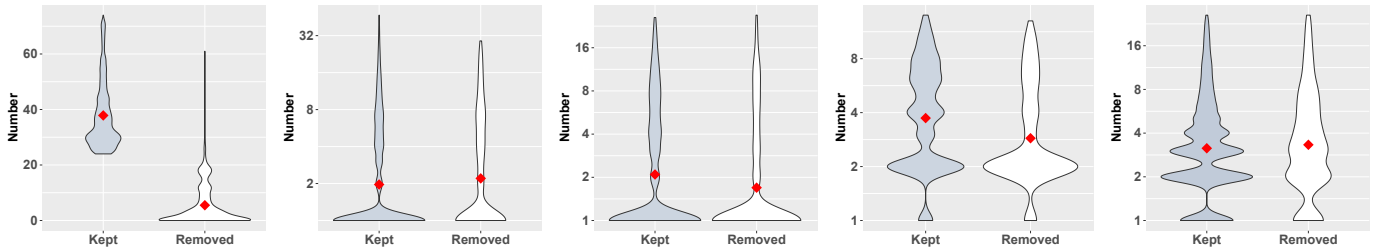
3) The complexity of lambda expressions: We measure the complexity of lambda expressions from four perspectives as mentioned above. The measure of parameter number is shown in Figure 6 while the statistics of the other three are depicted in Figure 4 (clipped down outliers exceeding $\mu+3\sigma$). It shows that removed lambdas (38.34%) have a higher percentage of *empty argument lambda* (lambda expressions that do not receive any argument) than kept ones (30.75%). Similarly, we applied the Chi-Square test [22], and observe the relation is significant at the significance of 0.05 with a small effect size of 0.05. Besides, for empty argument lambdas, we observed that removed lambdas have significantly fewer lines and smaller body depth than kept ones as shown in Figure 4c and Figure 4d with a p-value of 2.44×10^{-16} and 8.91×10^{-28} respectively by the Mann-Whitney U Test [23]. One possible reason is that empty argument lambdas with terse bodies can always be easily refactored to *method reference* for code simplification.

As for *non-empty argument lambdas*, removed lambdas occupy more lines of code than kept ones as shown in Figure 4b (p-value=2.47×10^{-12} at the significance level of 5% by the Mann-Whitney U Test [23]). In other words, for such lambdas, occupying more lines of code or having a more complex body tend to exhibit a higher probability to be removed. That is not a surprise because lambda expressions aim at providing a lightweight mechanism to deliver functionalities [24], while lambdas with complicated structures violate such a philosophy.

We also investigated the number of variables inside lambda bodies. As shown in Figure 4e, kept lambdas contain 4.89 variables on average while removed lambdas contain 5.49 variables. The Mann-Whitney U test [23] reveals that kept lambdas have significantly fewer variables in their body than removed lambdas at the significance of 5% (p-value=0.0036).

> **Finding #2:** *Empty argument lambdas has a larger possibility to be removed, while non-empty argument lambdas with more complex bodies are more likely to be refactored.*

4) The contexts of lambda expressions: We inspect the contexts of lambda expressions as follows. First, we investigate



(a) Life Time of Kept and Removed Lambda Expressions (b) Line Number of Removed and Kept Lambdas with Parameters (c) Line Number of Lambdas without Parameters (d) Lambda Body Depth of Lambdas without Parameters (e) Variable Number of Kept and Removed Lambdas

Figure 4: Comparison between Removed and Kept Lambdas

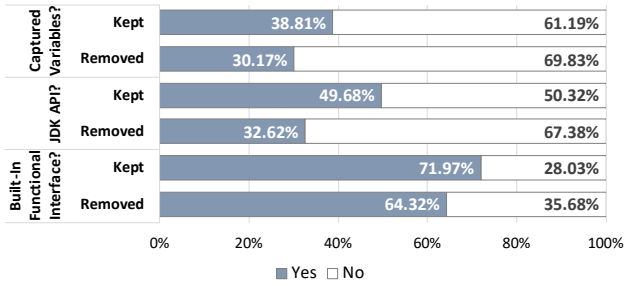


Figure 5: Comparison between Removed and Kept Lambdas

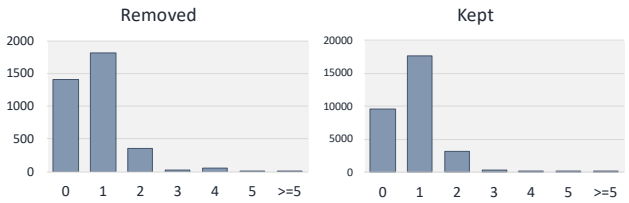


Figure 6: Parameter Numbers of Kept and Removed Lambdas

what specific APIs lambdas are often passed to. Figure 5 shows that lambda expressions passed to self-defined methods are more likely to be removed by developers (i.e., 67.38% vs 50.32%). A Chi-Square test reveals that the removal of lambda expressions is significantly correlated with whether the invoked method is self-defined or not with a p-value of $p = 8.50 \times 10^{-46}$ and a small effect size of 0.10. One possible reason is that APIs defined in JDK are a safer context for lambdas. For those lambdas passed to built-in APIs, we further dissect the distributions of specific APIs and show our result in Table I. We found that lambdas passed to APIs of `Map#computeIfAbsent` and `Optional#ifPresent` are more likely to be removed than those passed to other APIs. On the contrary, lambdas passed to `Stream#map` and `Stream#filter` are less likely to be removed. It is because that the `Stream` API was designed to mainly work with lambdas. The manner to write lambdas in `Stream` API methods is also well defined. On the contrary, `Optional` was not designed with the functional programming style instead although it was also introduced in Java 8. Meanwhile, implementing with `Optional` is sometimes more complex than that with conventional `if else` for branch logic processing [25]. For the same reason, `Map`, which was introduced since JDK1.2, is not as compatible with lambda expressions as `Stream` API. Therefore, if developers

Table I: APIs that Lambda Expressions Are Passed To

API	Removed	Kept	Difference
`Iterable.forEach`	13.78%	12.65%	1.13%
`Stream.map`	8.49%	13.93%	-5.44%
`Map.computeIfAbsent`	7.69%	4.57%	3.12%
`Optional.ifPresent`	7.05%	2.07%	4.98%
`Stream.forEach`	5.29%	4.96%	0.33%
`Stream.filter`	5.29%	14.20%	-8.91%
`Collectors.toMap`	4.49%	4.70%	-0.21%
`Map.forEach`	4.01%	3.50%	0.51%
`ExecutorService.submit`	3.69%	2.59%	1.10%
`Optional.map`	3.53%	1.51%	2.02%
`IntStream.forEach`	2.40%	1.48%	0.92%

intend to call lambda expressions within JDK built-in methods in the functional programming style, `Stream` API is a better choice, and thus lambda expressions used in `Stream` API are less likely to be removed in practice.

> **Finding #3:** *Lambda expressions that are built on top of customized functional interfaces, passed to self-defined method invocations are more likely to be removed.*

IV. Concerns and Actions of Developers

In this section, we investigated the concerns of the developers when removing a lambda expression and the migration patterns afterwards. Specifically, we analyzed the issue descriptions from the issue trackers and commit messages to figure out the reason why developers remove a lambda expression. On the other hand, we also looked into the code changes of fixes to explore how developers fix the problem induced by the improper usages of lambda expressions. To have a deeper understanding of the reasons behind the removals of lambda expressions, we also communicated with some developers who removed lambda expressions in our subjects. Finally, we found seven common reasons for removing lambda expressions and seven major migration patterns. We present the details of the empirical results as follows.

A. Data Collection

We collect a dataset of issues from the following sources to understand the reason behind the removal of a lambda expression and the action taken by developers after the removal.

Apache JIRA. Most Apache open-source projects track their issues on the JIRA issue tracker. Therefore, we search the issues on JIRA using the following query to collect the issue related to lambda expressions.

```
status in (Resolved, Closed) AND text ~ "lambda"
```

The query returned 6,647 items. we further filter out issues in which the keyword *lambda* only appears in stack traces, and those associated with Python lambda. We obtained 1,175 issues afterwards and manually validate them to see whether they contain sufficient information for us to understand the behind reasons. Finally, 25 issues are kept.

Code Commits. Not all the fixes for improper usage of lambda expressions are tracked in Apache JIRA. Meanwhile, the associated issues may not use the keyword "lambda" in the issue summary or description, thus escaping from our queries. In this study, we also search the commit messages of the commits that remove a lambda expression and contains certain keywords (i.e., *lambda*, *bug*, *fix*, *issue*, *fixup*, *problem*, *abuse*, *error*, *optimize*, *etc*). Such commits are likely to fix a problem by removing a lambda expression. We finally kept 60 of such collected commits after manual validation.

GitHub Issues. Apart from JIRA, part of the issues in our subjects are tracked on GitHub for discussion. Therefore, we searched the GitHub issues of our subjects with keyword *lambda* and then manually selected the issues containing sufficient information. We obtained seven issues in this step.

User Study. We further communicated with developers asking for the details about the removal of lambda expressions. Specifically, we discussed with developers, who have recently removed lambdas in open-source projects (we automatically mined such information via monitoring the commit histories of the 103 projects), on the reason for removing the lambda expression, the impacts caused by the lambda expression, and scenarios in which they would avoid using lambda expressions. Specifically, we sent 364 emails to the developers which have recently removed lambda expressions and got in touch with 38 of them, with a response rate of 10.4%. We retained those responses with clear descriptions and got 25 issues included.

In total, we collected 117 issues for exploring the concern and actions of the developers while removing lambda expressions, including 25 from Apache JIRA, 60 from code commits, seven from GitHub issues, and 25 from User Study.

B. Analysis Approach

For the collected data collection, we performed a process similar to open coding [26] to identify common reasons for removing lambda expressions as well as the migration patterns. For data collected from *Apache JIRA*, *Code Commits*, and *GitHub Issues*, we read the descriptions of the issues or commit messages to understand the reason for the removal. For data collected in the *user study*, we extract the removal reasons from email replies. Besides, we read the code diff for all the data to find common migration patterns. In the classification process, two authors inspected and labeled the dataset independently and they discussed with each other to reach a consensus when there is a conflict.

After the analysis, we summarized the results of our qualitative study in this Section from two perspectives: the *reasons* why such lambdas are deemed as misused and the associated impacts, and the *migration patterns* adopted by developers to replace such removed lambdas. We present our results in the following two subsections.

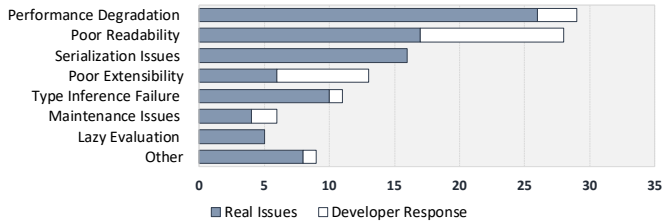


Figure 7: Reasons of Removing Lambda Expressions

C. Reasons for Misusing Lambda Expressions

We summarized seven major reasons (with over five cases for each reason) that why lambdas are deemed as misused by developers and the associated impacts. Figure 7 shows the statistics and we present the details as follows:

Performance Degradation (29/117). Performance degradation is one of the most crucial reasons that why developers remove lambda expressions. Specifically, nearly 25% of the lambdas in our study are removed since their usages introduced performance issues. Such lambdas will be removed if such side effects are significant and perceived by developers. The most common inappropriate lambda usages that cause performance issues is invoking lambdas in computation-critical code, especially for captured lambdas which access variables outside the lambda body. For captured lambdas with multiple invocations, although the lambda class is only created once on the first invocation, each invocation will create a new lambda instance. This may introduce excessive object allocations and bring significant pressure to the garbage collector. Therefore, a developer in Apache Lucene removed such lambdas to avoid memory allocation for each invocation of `collectValFirstPhase` [27]. Furthermore, as shown in Listing 1, the lambda expression will hold a reference to their enclosure instance when it accesses the enclosing instance's non-static fields and methods, thus causing memory leaks.

Actually, the new Java 8 API methods (i.e., `Collections`, `Stream`, and `Optionals`) can also introduce performance differences when they are used with lambdas. Although it is still controversial whether the performance of using lambdas with such Java 8 API methods is worse than their counterpart implementations in diverse scenarios [2]. There are many real issues which refactor these usages into conventional ways (i.e., `for loop`, `do while`, `enhanced for loops`, `if else`, etc) due to the excessive CPU cycles and non-trivial memory allocation overhead based on performance test results [28].

Poor Readability (28/117). Poor readability is another important reason for lambda removals in our dataset. Such lambda expressions usually have a long body or contain complex logic, setting obstacles for developers to understand them. Worse still, lambda expressions are anonymous, and thus the intention of a lambda expression is far more difficult to understand than a method. Besides, the syntax of lambda expressions looks less compact than that of method reference.

```
- .collect(Collectors.groupingBy(url -> { ... }));
+ .collect(Collectors.groupingBy(this::judgeCategory));
+ private String judgeCategory(URL url) {
+    //  The same as the lambda body
+ }
```

Listing 2: Replacing a Lambda with a Method to Improve Readability (Commit#9e9517d, Project Dubbo)

```
-      .apply(o -> Collections.emptyList());
+      .apply(
+        new MultimapView() {
+          @Override public Iterable get() { ... }
+          @Override public Iterable get(Object o) { ... }
+      });

   public interface MultimapView<K, V> {
+     Iterable<K> get();
      Iterable<V> get(@Nullable K k);
   }
```

Listing 3: Switching from Lambda to Anonymous Class for Better Extensibility (PR#10147, Project Beam)

```
- HasDisplayData subComponent = builder ->
-       builder.include("b", builder1 -> builder1.add(...));
+ HasDisplayData subComponent = new HasDisplayData(){
+   @Override
+   public void populateDisplayData(Builder builder){
+     builder.include("b", new HasDisplayData(){
+       @Override
+       public void populateDisplayData(Builder builder){
+         ...
```

Listing 4: Replacing Lambda Expressions with Anonymous Classes (Commit#aedb4c8, Project Beam)

```
@groovy.transform.CompileStatic
java.lang.ClassCastException:
    ThisTest$_m_lambda1 cannot be cast to ThisTest
    at ThisTest$_m_lambda1$_lambda2.doCall(TS4.groovy:9)
    at ThisTest$_m_lambda1.doCall(TS4.groovy:11)
    at ThisTest.m(TS4.groovy:13)
    at ThisTest$m.call(Unknown Source)
```

Listing 5: Stack Trace Related to Lambdas in GROOVY-9341

Therefore, developers sometimes tend to remove lambda expressions to improve readability as well as make the code more compact. For example, in Listing 2, developers in project *Dubbo* replaced a lambda containing more than 10 lines of code with a method to make it more readable as shown in the description: *simplify "collect" body to make it more readable*. Our finding is also confirmed by developers in the user study as a developer in project *Apache Calcite* [6] mentioned that: "*I generally dislike long lambdas due to bad readability.*"

Serialization Issues (16/117). There are 16 lambda expressions in our dataset that are removed due to serialization issues. Since Java 8, one can make a lambda expression serializable by extending `Serializable` in its corresponding functional interface. Serializing lambdas can be used for *persisting configuration, or as a visitor pattern to remote resources* [29]. However, this is strongly discouraged according to the Java Language Specification [30] because the serialization behavior of synthetic class (what will be created when compiling a lambda expression) can vary among JVM implementations, which can cause compatibility issues when the bytes serialized on one JVM are shipped to another JVM to deserialize, and vise versa. For instance, a developer successfully serialized a lambda and shipped the byte code via RemoteGraph to a remote server, but can not deserialize it on the new server [31].

Poor Extensibility (13/126). During code evolution, functional interfaces can be evolved to accept no lambda any more. In this case, a lambda expression is no longer accepted when an object implementing such interfaces is required. Therefore, developers need to switch from lambda expressions to anonymous classes for better extensibility. Listing 3 shows an example in project *Beam*. As shown in the code change, a new method `get()` is added to inference `MultimapView` and thus it is no longer a functional interface. To this end, developers replaced the lambda expression with an anonymous class to make the code syntactic correct. Similarly, 13 such cases are observed in our dataset.

Type Inference Failure (11/117). We observed 11 cases in our dataset where lambda expressions are removed to resolve type inference failures. As mentioned in Section II, the type information in a lambda expression can be omitted and inferred by the compiler. However, there are cases where there is no sufficient contextual information for the compiler to infer the type, thus causing a compilation error. Listing 4 shows an example in project *Beam* where developers replaced two nested lambda expressions with anonymous classes to *"fix of some call sites where lambdas mess up coder inference"*, as mentioned in the commit message.

Maintenance Issues (6/117). Apart from technical issues, lambda expressions can also be removed due to human factors (i.e., causing maintainability issues or inconsistent code styles). Listing 5 shows a stack trace when debugging a code snippet with lambda expressions in project *Groovy*. The names of the stack frame related to lambda expressions are hard for developers to recognize, which makes debugging code related to lambda expressions difficult. Besides, the stack traces for those lambdas used together with the new Java API 8 methods are even more complex. For instance, as mentioned by a developer in our user study, "*They are often annoying when debugging step by step inside an IDE compared to imperative programming. Especially in cases like this: collection.stream().flatMap(...).filter(...).map(...).collect(...).*".

Lazy Evaluation (5/117). One key feature of lambda expressions is that they are not evaluated when they are defined, and are actually evaluated when they are called instead. Therefore, lambda expressions are usually used for implementing lazy evaluation, which is delaying the evaluation of an expression until its value is needed, thus avoiding needless calculations and reducing memory footprints. However, some developers ignored the nature of lazy evaluation and used lambda expressions in the cases where lazy evaluation is not desired, and thus introduced unexpected behaviors into the program. For instance, the code change in Listing 6 shows an inappropriate implementation of lazy evaluation with lambda expressions. Since `Supplier` does not have memorization (cache the result of a function call and return the cached value for the

```
- final Supplier<Blackboard> bb = () -> {
-   RexNode sourceRef = rexBuilder.makeRangeRef(scan);
-       return createInsertBlackboard(table, sourceRef,
-           table.getRowType().getFieldNames());
- };
+ final RexNode sourceRef = rexBuilder.makeRangeRef(scan);
+ final Blackboard bb = createInsertBlackboard(table,
+   sourceRef, table.getRowType().getFieldNames());
...
- list.add(ief.newColumnDefaultValue(table,
-    f.getIndex(), bb.get()));
+ list.add(ief.newColumnDefaultValue(table,
+    f.getIndex(), bb));
```

Listing 6: Remove the Lazy Initialization Logic for the Blackboard Instance (`Commit#22c76fb`, Project Calcite)

following calls to save computation costs), each access to the generated `Blackboard` instance needs to call method `get()` of `Supplier` again, thus increasing repeated computations. Besides, the evaluations of intermediate operations in `Stream`, such as `map`, `filter`, etc, can also be lazy. Applying these `Stream` APIs with lambda expressions performs no operations until the terminal operation is executed. Consequently, ignoring the lazy nature of intermediate operations in `Stream` may also introduce problems (i.e., CASSANDRA-13905 [32]).

Others (9/117). Other issues are too specific to be summarized into a single category. For instance, lambda expression misuse leads to an ambiguous method call, unhandled exceptions, and so on. These problems are rare because most of them can be found at compilation time and easily solved before commit.

To summarize, we make the following finding with respect to why developers removed lambdas.

> **Finding #4:** *Performance degradation and poor readability are the most common reasons why developers remove lambdas. Lambdas can also be removed due to the nature of lazy evaluation. Besides, they might cause extensibility and maintenance issues, thus being removed by developers.*

D. Common Migration Patterns

We further identify how developers migrate those inappropriate usages of lambda expressions. For the collected 117 cases, we drop those cases whose corresponding commits are unavailable, and finally obtained 104 unique cases for further classification. Via investigating the available associated commits, we observed seven major code migration patterns (with over five cases for each pattern), which are summarized in Table II. Be noted that there may be more than one pattern for each issue. We also analyzed the relationship between the reasons why lambdas got removed and the corresponding migration patterns, which is shown in Figure 8. Our major observations are described as follows:

We observed that the most common migration pattern (**24/104**) is to change new Java 8 API methods with lambda expressions back to conventional methods. According to our statistical results, 14 of such 24 cases are for performance improvement. It is because the compiler has well explored how to optimize old-fashioned code while the support of

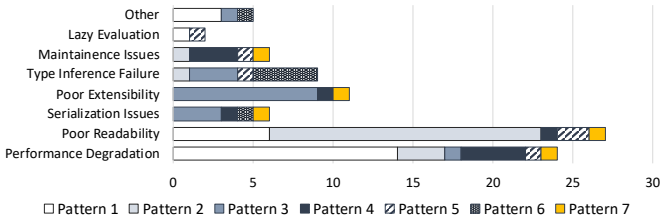


Figure 8: Correlation between Lambda Removal Reasons and Migration Patterns. The Number Denotes the Migration Pattern ID as Shown in Table II

performance optimization for new language features such as `Stream` API is insufficient. Therefore, when performance is of significant concern, developers should be careful when using these new Java 8 API methods with lambda expressions.

We observed that it is also common (**22/104**) to replace lambda expressions with method references. Specifically, 17 of them aim to improve readability, which shows that it is a common strategy to simplify the code with method references. We have elaborated above that how method references can make the code more succinct, thus facilitating comprehension.

Besides, in 17 cases, lambda expressions are replaced with anonymous objects (**17/104**). Among them, nine are associated with poor extensibility of lambda expressions. This confirms the intuition that anonymous objects have better extensibility compared with lambdas, which is especially important for interface evolution. Besides, six of the 17 cases are solving the serialization and type inference problems since anonymous objects are more friendly to serialization and type inference compared with lambda expressions.

We do not explain other migration patterns summarized in Table II in detail, and we summarize our findings as follows.

> **Finding #5:** *Lambda expressions that are inappropriately utilized are often migrated to conventional implementations using anonymous class, method reference, inner class instance, and so on. Besides, lambdas passed to the new Java 8 API methods are often migrated to conventional ones.*

E. Actionable Advice for Using Lambda Expressions

Based on the above findings, together with the responses from developers in our user study, we are able to make the following actionable advice for Java developers to better utilize lambda expressions in the future.

Avoid using lambdas in performance-critical code. Lambdas, especially in frequently invoked methods, can deteriorate the performance of programs, and thus should be refactored into implementations whose performance has been better optimized by the compiler. Specifically, in 29 out of the 117 collected issues, lambda expressions are removed for performance optimizations, some of which explicitly mention in their commit messages that *"lambdas are removed for optimization to reduce object allocations in critical code path"*. Meanwhile, five developers supported this point in their responses. For instance, as suggested by one developer, "*on a JVM with memory constraints or in a code base where the streams are going to be instantiated a comparable number of times to other*

Table II: Migration Patterns of Removing Lambda Expressions

ID	Types	Description	Frequency
1	Lambda passed to new Java 8 API methods ⇒ Conventional methods	Replace new Java 8 API, i.e., `Collections`, `Stream` and `Optionals`, with conventional `for loop`, `do while`, `enhanced for loops`, `if else`, etc.	24
2	Lambda ⇒ Method reference	Lambda expressions are refactored into method reference to improve readability or performance. In some cases, the lambda body is too large and should be first extracted into another function and then being invoked with method reference.	22
3	Lambda ⇒ Anonymous class	Lambda expressions are refactored into anonymous class.	17
4	Lambda ⇒ Inner class instance	The behavior existed in lambda expressions are wrapped into an newly defined inner class.	10
5	Method with lambdas are replaced with a new method	The method to which the lambda expression is passed, no longer exists and is replaced with a new method which does not accept lambdas any more.	6
6	Adding a type cast	Adding a type to provide more type information for type inference and overload resolution or implementing `Serializable`.	6
7	Existing method was changed to accept no lambdas	Parameters of the existing method are changed to accept no lambdas any more. The corresponding parameters are either changed to a new one which enclose the behavior of lambda expressions, or deleted (logics in removed lambdas are implemented in following code).	5

```
- Optional.ofNullable(securityExtension))
-    .map(Extension::hasAuthenticationMechanisms)
-    .filter(has -> has.equals(true))
-    .ifPresent(has -> Config.getFactory().register(...));
+ if (securityExtension.hasAuthenticationMechanisms()) {
+     Config.getFactory().register(...);
+ }
```

Listing 7: Replacing Lambda in `Optionals` with `if else` (`Commit#99d6f10`, Project TomEE)

objects one might consider replacing lambdas for normal for loops to avoid using extra memory resources".

Avoid using lambdas with new Java 8 API methods in branch logic processing. New Java 8 API methods with lambdas are unfriendly for branch logic processing. Two lambda removals that refactor new Java 8 API methods are caused by such a reason. Listing 7 displays an example, which shows the comparison between the implementation with `Optionals` and `if else`. The former is quite complex and can be easily refactored to the latter one. Besides, one developer in the user study also pointed out that: "*branch logic processing is hard to accomplish with existing stream processing mechanisms in Java. The degree of awkwardness to implement certain logic for me is a criteria for using lambda expressions.*"

Specify the type information in generic settings or overloaded methods. Since problems are common for lambda expressions in overload resolution and type inference, it is better to explicitly specify the type information when using lambdas. Otherwise, it is preferable to use an anonymous object instead. This has been supported by ten issues and three developers in our study. One developer responded that "*lambda methods do not provide enough information for automatic type extraction when Java generics are involved. An easy workaround is to use an (anonymous) class instead. Otherwise, the type has to be specified explicitly using type information* "

Avoid lambdas if you want to throw a checked exception. Throwing a checked exception in the lambda body is quite tricky, which should be avoided. This has been supported by two issues in our study and three developers in the user study. Throwing a checked exception in lambda bodies is implemented by either declaring it in the functional interface to throw this exception, or wrapping the checked exception inside a `RuntimeException`, and then throw the wrapped unchecked

```
public void removeHivePluginFrom(Iterable<Drillbit>
            drillbits) throws PluginException {
  try {
    drillbits.forEach(bit -> {
      try {
        bit.getContext().getStorage().remove(pluginName);
      } catch (PluginException e) {
        throw new RuntimeException("...", e);
      }
    });
  } catch (RuntimeException e) {
    throw (PluginException) e.getCause();
  }
}
```

Listing 8: Throwing a Checked exception with a Wrapper

```
public void removeHivePluginFrom(Iterable<Drillbit>
            drillbits) throws PluginException {
  for (Drillbit drillbit : drillbits) {
    drillbit.getContext().getStorage().remove(name);
  }
}
```

Listing 9: Throwing a Checked Exception with For Loop

exception instead. However, it is sometimes inconvenient to modify functional interfaces (i.e., built-in functional interface). Using wrappers will perplex the programs, which contradicts the intention of using lambda expressions to simplify code. Therefore, developers are prone to remove lambda expressions for such cases. Listing 8 shows an example as mentioned by one developer from the project *Apache Drill* [33]. Specifically, the `remove()` method may throw a checked exception: `PluginException`. The implementation with `enhanced for loop` in Listing 9 is quite simple compared with that using a wrapper with lambdas as shown in Listing 8.

Avoid constructing large lambdas. For a lambda which is too large or encapsulates too much logic, it is better to be refactored to facilitate code comprehension. This has been suggested by six developers in the responses towards the scenarios they would avoid using lambda expressions. As mentioned by one developer, "*a lambda expression encapsulates too much logic, and it would be easier to follow the code if broken out into a named function, or refactored some other way.*"

V. Threats to Validity

This study can suffer from the following threats to validity.

First, our collected dataset might contain noise and may not be generalizable enough. In our dataset, there can be a lambda

expression that is not removed for fixing an issue because it is challenging to accurately infer the semantics of all code changes in large-scale projects. To mitigate this threat, we tried our best to set several rigorous rules to reduce the potential noises in our dataset. Besides, our empirical findings might not be generalizable to other non-ASF projects since we mainly collected the removed and kept lambda expressions from the ASF open-source projects. To reduce such a threat, we selected 103 large-scale open-source Java projects from different categories including *big data*, *cloud computing*, *databases*, *mobile application*, *network client/server*, and etc. to ensure our dataset is diverse and representative enough.

Second, the issues collected for exploring the concerns and actions of developers may not cover all aspects. To tackle this problem, we collected our dataset from multiple sources, including Apache JIRA, GitHub Issues, and code commits. Furthermore, we also referred to the mailing lists of our subjects and communicated with the developers who removed lambdas for further information. Collecting data from various sources can help enrich the generality of our dataset.

Third, our study mainly focuses on lambda removals, which may miss issues caused by lambda modifications. We chose those removed lambdas since they are more likely to cause side effects. On the contrary, lambda modifications are mostly associated with functionality improvement or project maintenance, which may introduce potential noises if we consider such cases. However, important issues that are worth exploring can also be caused by lambda modifications, and thus, in the future, we will try to devise more sophisticated methods to collect such data for further studies.

Forth, the side effects summarized in our findings are not only caused by the lambda expressions themselves, but also can be collectively caused by the lambdas and the surrounding contexts, such as the `Stream` APIs, to which the lambdas are passed as parameters. Since the common usages of lambdas are frequently mixed with these APIs, we take such cases into consideration to enrich our scope as explained in Section II-B. To avoid misunderstanding by readers, we differentiate the syntactic constructs of lambda expressions and the APIs that rely on them while clarifying our findings.

VI. Discussion and Future Work

Lambda Removal Recommender. We have explored a set of static features for lambdas (see Section III-C) and summarized several inappropriate ways of lambda usages along with common migration patterns (see Section IV-C), which are helpful for constructing automated tools to detect certain lambda misuses and recommend a more appropriate implementation instead. For instance, the tool can recommend migrating lambda expressions into method references for readability enhancement if these lambdas are too complex (as measured by the features we extracted in the qualitative study). Besides, we also found that lambda expressions have a higher probability to cause type inference failures when the corresponding functional interfaces are generic. Therefore, the tool can recommend specifying the generic type for such cases. In the future, we will explore the above ideas and build automated tools to facilitate appropriate lambda usages.

Study on more Java Functional Idioms. Along with lambda expressions, some other functional idioms [8] (i.e., `Collections`, `Stream`, and `Optionals`) are also introduced in Java 8 to facilitate functional programming in Java. In this study, we inspected two side-effects of these functional idioms: poor performance, and unfriendly to branch logic processing. Further efforts will be made to comprehensively understand the advantages and disadvantages of applying these functional idioms, thus providing more reliable and useful guidance on functional programming with Java.

Applicability to other Programming Languages. Lambda expressions are also supported by other Object-Oriented languages, such as C++, Python, etc. Some side effects that we observed in Java lambda expressions also exist in other programming languages. For instance, usages of lambdas in Python can also lead to poor readability when lambda expressions are too long [34] or do not have names for developers to understand their functionalities [35]. Therefore, some of our findings are also applicable to other Object-Oriented languages. It is certain that the characteristics of lambda expressions might be diverse in different languages, while the methodology of our studies including both the quantitative analysis and qualitative analysis can be similarly adapted to investigate other languages. It is also worth studying how functional programming paradigms impact Object-Oriented Programming and the differences among different OOP languages. However, following existing studies [2], [8], [9], [11], we put our focus mainly on the Java language in this study, and left the studies across different languages as our important future works.

VII. Conclusion

Based on our observation that lambda expressions can often incur side effects or even severe bugs, we are motivated, in this study, to understand inappropriate usages of lambda expressions in Java. Specifically, we performed a quantitative and a qualitative study in open-source Java projects to investigate the characteristics of removed lambda expressions as well as the concerns of developers when removing them. Specifically, we explore four code-level characteristics of lambdas that are removed by developers, which provide useful information on the behind reasons. On the other hand, we summarized seven major reasons for the removal of lambda expressions along with the associated migration patterns. Based on all our findings, we finally come up with five pieces of actionable advice to help developers free from the side effects of lambda expressions as much as possible.

Acknowledgment

We sincerely thank all anonymous reviewers for their valuable comments. This work was supported by the National Natural Science Foundation of China (Grant No. 62002125 and No. 61802164) as well as the Fundamental Research Funds for the Central Universities (HUST) No.2020kfyXJJS076.

REFERENCES

[1] M. Vanags and R. Cevere, "The perfect lambda syntax," *Baltic Journal of Modern Computing*, vol. 6, no. 1, pp. 13–30, 2018. [Online]. Available: https://doi.org/10.22364/bjmc.2018.6.1.02

[2] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig, "Understanding the use of lambda expressions in java," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–31, 2017. [Online]. Available: https://doi.org/10.1145/3133909

[3] "Lucene-9478," 2021, accessed: 2021-4-12. [Online]. Available: https://issues.apache.org/jira/browse/LUCENE-9478

[4] "Apache camel," 2021, accessed: 2021-4-12. [Online]. Available: https://camel.apache.org/

[5] "Apache geode," 2021, accessed: 2021-4-12. [Online]. Available: https://geode.apache.org/

[6] "Apache calcite," 2021, accessed: 2021-4-12. [Online]. Available: https://github.com/apache/calcite

[7] "Stream," 2021, accessed: 2021-4-12. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

[8] H. Tanaka, S. Matsumoto, and S. Kusumoto, "A study on the current status of functional idioms in java," *IEICE Transactions on Information and Systems*, vol. 102, no. 12, pp. 2414–2422, 2019. [Online]. Available: https://doi.org/10.1587/transinf.2019MPP0002

[9] W. Lucas, R. Bonifácio, E. D. Canedo, D. Marcílio, and F. Lima, "Does the introduction of lambda expressions improve the comprehension of java programs?" in *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, 2019, pp. 187–196. [Online]. Available: https://doi.org/10.1145/3350768.3350791

[10] A. Gyori, L. Franklin, D. Dig, and J. Lahoda, "Crossing the gap from imperative to functional programming through refactoring," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 543–553. [Online]. Available: https://doi.org/10.1145/2491411.2491461

[11] A. Alqaimi, P. Thongtanunam, and C. Treude, "Automatically generating documentation for lambda expressions in java," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 310–320. [Online]. Available: https://doi.org/10.1109/MSR.2019.00057

[12] N. Tsantalis, D. Mazinanian, and S. Rostami, "Clone refactoring with lambda expressions," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 60–70. [Online]. Available: https://doi.org/10.1109/ICSE.2017.14

[13] "Myfaces-4337," 2021, accessed: 2021-7-20. [Online]. Available: https://issues.apache.org/jira/browse/MYFACES-4337

[14] "Flink-18075," 2021, accessed: 2021-4-20. [Online]. Available: https://issues.apache.org/jira/browse/FLINK-18075

[15] "Flink-20147," 2021, accessed: 2021-4-20. [Online]. Available: https://issues.apache.org/jira/browse/FLINK-20147

[16] "Asf project list," 2021, accessed: 2021-4-12. [Online]. Available: https://projects.apache.org/projects.html?number

[17] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982

[18] "Hunks," 2021, accessed: 2021-4-12. [Online]. Available: http://www.gnu.org/software/diffutils/manual/html_node/Hunks.html

[19] K. Herzig and A. Zeller, "The impact of tangled code changes," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 121–130. [Online]. Available: https://doi.org/10.1109/MSR.2013.6624018

[20] S. Kim and E. J. Whitehead Jr, "How long did it take to fix bugs?" in *Proceedings of the 2006 international workshop on Mining software repositories*, 2006, pp. 173–174. [Online]. Available: https://doi.org/10.1145/1137983.1138027

[21] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the dependency conflicts in my project matter?" in *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 319–330. [Online]. Available: https://doi.org/10.1145/3236024.3236056

[22] K. Pearson, "X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 50, no. 302, pp. 157–175, 1900.

[23] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[24] "State of the lambda," 2021, accessed: 2021-4-12. [Online]. Available: https://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html

[25] "Is it worth to use if-else statement as java optional pattern?" 2021, accessed: 2021-7-20. [Online]. Available: https://codereview.stackexchange.com/questions/223277/is-it-worth-to-use-if-else-statement-as-java-optional-pattern

[26] J. W. Creswell and C. N. Poth, *Qualitative inquiry and research design: Choosing among five approaches*. Sage publications, 2016.

[27] "Pr 476, project solr," 2021, accessed: 2021-7-20. [Online]. Available: https://github.com/apache/lucene-solr/pull/476/files

[28] "Commit f555aa6, project presto," 2021, accessed: 2021-7-20. [Online]. Available: https://github.com/prestodb/presto/commit/f555aa69

[29] "How and why to serialize lambdas?" 2021, accessed: 2021-7-20. [Online]. Available: https://dzone.com/articles/how-and-why-to-serialialize-lambdas

[30] "The java® language specification," 2015, accessed: 2020-10-28. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se8/html/index.html

[31] "Tinkerpop-1230," 2021, accessed: 2021-7-20. [Online]. Available: https://issues.apache.org/jira/browse/TINKERPOP-1230

[32] "Cassandra-13905," 2021, accessed: 2021-7-20. [Online]. Available: https://issues.apache.org/jira/browse/CASSANDRA-13905

[33] "Apache drill," 2021, accessed: 2021-4-12. [Online]. Available: https://github.com/apache/drill

[34] "lambda function in different lines," 2021, accessed: 2021-7-20. [Online]. Available: https://stackoverflow.com/questions/57969592/lambda-function-in-different-lines

[35] "Which is more preferable to use: lambda functions or nested functions ('def')?" 2021, accessed: 2021-7-20. [Online]. Available: https://stackoverflow.com/questions/134626/which-is-more-preferable-to-use-lambda-functions-or-nested-functions-def