IEICE TRANSACTIONS

on Information and Systems

VOL. E102-D NO. 12
DECEMBER 2019

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

IEICE TRANS. INF. & SYST., VOL.E102–D, NO.12 DECEMBER 2019

PAPER Special Section on Empirical Software Engineering

A Study on the Current Status of Functional Idioms in Java

Hiroto TANAKA[†a)], **Shinsuke MATSUMOTO**[†], *Nonmembers*, *and* **Shinji KUSUMOTO**[†], *Member*

SUMMARY Over the past recent decades, numerous programming languages have expanded to embrace multi-paradigms such as the fusion of object-oriented and functional programming. For example, Java, one of the most famous object-oriented programming languages, introduced a number of functional idioms in 2014. This evolution enables developers to achieve various benefits from both paradigms. However, we do not know how Java developers use functional idioms actually. Additionally, the extent to which, while there are several criticisms against the idioms, the developers actually accept and/or use the idioms currently remains unclear. In this paper, we investigate the actual use status of three functional idioms (Lambda Expression, Stream, and Optional) in Java projects by mining 100 projects containing approximately 130,000 revisions. From the mining results, we determined that Lambda Expression is utilized in 16% of all the examined projects, whereas Stream and Optional are only utilized in 2% to 3% of those projects. It appears that most Java developers avoid using functional idioms just because of keeping compatibility Java versions, while a number of developers accept these idioms for reasons of readability and runtime performance improvements. Besides, when they adopt the idioms, Lambda Expression frequently consists of a single statement, and Stream is used to operate the elements of a collection. On the other hand, some developers implement Optional using deprecated methods. We can say that good usage of the idioms should be widely known among developers.

key words: *functional idioms, java, lambda expression, stream, optional*

1. Introduction

Programming languages have been evolving in recent years [1], [2]. This evolution includes the progression of the *programming paradigm* [3], which can be defined as the fundamental style or manner of structuring and organizing programs [4]–[6]. Thus, language evolution includes not only idiom-level (e.g., adding sugar syntax) but also paradigm-level (e.g., new paradigm introduction) changes. Such paradigm-level enhancements trigger numerous drastic changes in source code structures, unit testing strategies, and programming experiences. Furthermore, it has recently become increasingly common for programming languages to adopt multi-paradigm structures, such as can be seen in the fusion of object-oriented [7] and functional programming [8] paradigm.

Java, known as a traditional object-oriented language, also has already become multi-paradigm. In 2014, Java 8 was released with some interesting and attractive new idioms such as Lambda Expression, Stream, and Optional. These idioms are inspired by the functional programming paradigm. This evolution allows developers to receive various benefits from both object-oriented programming and functional techniques. For instance, by using Stream APIs, developers can focus on what they want, rather than how to do it. Lambda Expression helps to avoid side effects in functions.

On the other hand, there are a number of valid criticisms against Java's functional idioms [9]–[14]. The most ideological criticism is that Java is still falling short of supporting some functional features [11], [14]. Lambda Expression is not a pure first-class object and just a syntax sugar to generate an anonymous inner class. Java has limited supports for closures compared to other languages. In terms of practical usage, it is claimed that Lambda makes harder for developers to debug programs [9], [10] and decreases program performance [11]. Exception handling in Stream with Lambda is harder to understand. In our best knowledge, little is known about *whether Java developers accept such arguable idioms or not?* and *how they actually use the idioms?*

In this paper, we investigate the current status of those idioms in Java projects. More specifically, by investigating changes to the source code used in Java projects, we will attempt to answer whether the idioms are being accepted and introduced by object-oriented developers, and how the developers use the idioms. The subject functional idioms are Lambda Expression, Stream, and Optional which are implemented to Java in March 2014. In our investigation, we collected 100 Java projects from GitHub which included more than 130,000 revisions and examined them.

The contributions of this study are as follows:

- We conducted an empirical study to determine how frequently functional idioms are used in Java.
- We collected practical knowledge that may support Java developers to decide whether to use functional idioms.
- Practical knowledge of actual usages can help Java developers start to introduce the idioms.

2. Functional Idioms

2.1 Lambda Expression

Lambda Expression represents an anonymous function object as its value. Lambda Expression consists of a

Manuscript received February 27, 2019.
Manuscript revised July 1, 2019.
Manuscript publicized September 6, 2019.

[†]The authors are with Osaka University, Suita-shi, 565–0871 Japan.
a) E-mail: h-tanaka@ist.osaka-u.ac.jp

DOI: 10.1587/transinf.2019MPP0002

comma-separated list of formal parameters, the arrow token (->), and the body which includes a simple expression or a block of statements. We show an example of this idiom.

```
List<User> users = getUsers();
users.forEach(u -> System.out.println(u.name));
```

In this example, Lambda Expression writes all user names of `users` variable to the standard output. We can write clear and concise code with this idiom. Additionally, the idiom can be given to method parameters. In other words, we can give not only values but also operations to method parameters.

2.2 Stream

Stream supports sequential and parallel aggregate operations. Stream consists of one operation to generate Stream, zero or more intermediate operations and one terminal operation. An example of this idiom is following.

```
List<User> users = ...;
users.stream()
     .filter(u -> u.age >= 20)
     .forEach(u -> System.out.println(u.name));
```

In this example, Stream is generated by `stream()` method. After generating, `filter()` method, one of the intermediate operations, extracts users who are 20 years old and over, and names of these users are written to the standard output by `forEach()` method which is a terminal operation. In addition, Stream API allows sequential processing to be parallel for Collection objects by only switching `stream()` method to `parallelStream()` method.

2.3 Optional

Optional is the object which may or may not contain a non-null value. For example:

```
List<User> users = ...;
int firstUserId =
    Optional.ofNullable(users.getFirstUserId())
            .orElseGet(-1);
```

In this example, `ofNullable()` method returns the ID of the first user of `users` variable with representing that the value might be null. -1 is given to `firstID` variable if the ID is null, the value of Optional is given otherwise. We can show that a value might be null without comments by using this idiom. Additionally, Optional forces us to write code which runs if a value is null. In other words, we can make safe programs with Optional.

2.4 Criticisms Against the Use of Functional Idioms in Java

Although functional idioms can provide some benefits to Java developers, there are several valid criticisms against their usage.

Debugging becomes more difficult [9], [10]: The stack trace becomes significantly longer when Lambda Expression is called in source code. This makes it harder for developers to debug.

Stream makes code run slower [11]: Consider an example in which we have multiple tasks and one of them takes significantly longer to complete than the others. In such a case, if the tasks are executed in parallel using Stream, the heavy task will degrade the overall performance.

Increased memory overhead [12]: When using Lambda Expression or Stream, the number of garbage collection executions is greater than when iterator is used because of the allocation of hidden objects. This may cause runtime performance problems.

Limitted support for monad [13]: Monad, a design pattern that allows structuring programs in functional programming [15], is not completely implemented in Java.

3. Research Questions

RQ1: Do Java developers accept functional idioms?

As previously mentioned, there are several common criticisms against the use of functional idioms in Java projects. However, the extent to which such functional idioms are accepted in Java projects and how frequently they are used remains unclear. By answering RQ1, we hope to gain an understanding of how widespread the use of such functional idioms has become in actual Java projects.

RQ2: Why do they accept or not accept the idioms?

While some Java projects accept the use of functional idioms, other developers do not. Accordingly, we conducted a qualitatively study to determine why functional idioms are being accepted or are not being accepted. The results of this study can be expected to help developers decide whether to introduce functional idioms to their projects in the future.

RQ3: How do they use the idioms?

We do not know how developers use functional idioms in Java projects. To see it, we focused on the latest revisions and studied actual usage of the idioms. The results of this study can be useful for developers when they try to use the idioms in their projects.

4. Research Methodology

Figure 1 shows an overview of our methodology to study the current status of the use of functional idioms. We conducted our research using the following procedure:

1. Collect 100 Java projects from GitHub.
2. Identify each functional idiom applied to the source code for each revision.
3. Calculate the *densities* for each functional idiom (details of this metric are described later.)

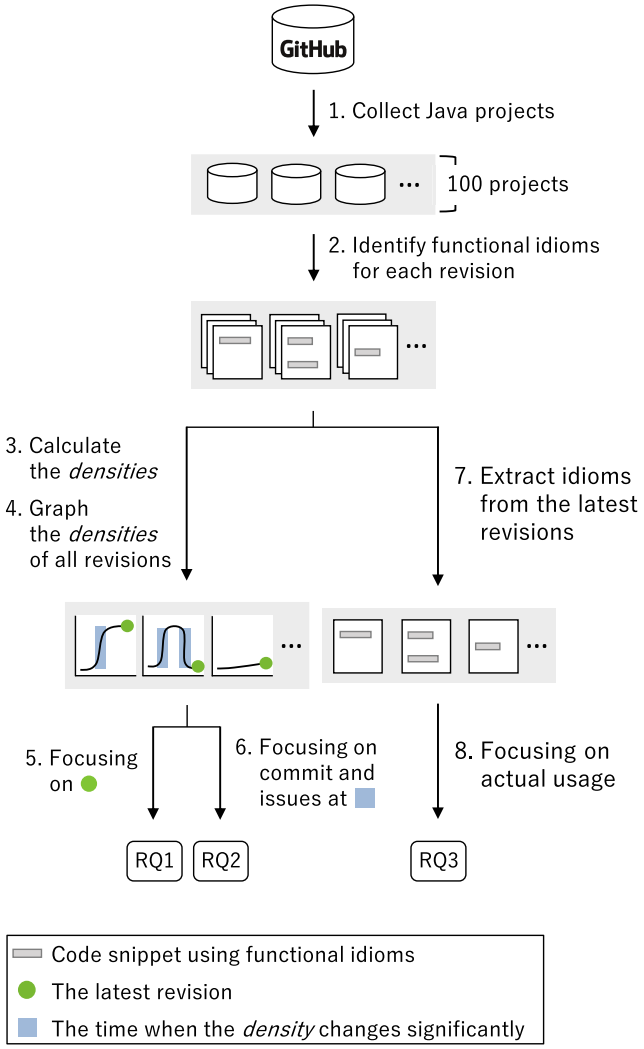


Fig. 1 Overview of our approach to answer RQs

4. Graph the *densities* for each revision.
5. Answer RQ1, focusing on the latest revisions.
6. Answer RQ2, focusing on the commit messages and issues discussed at times when the *densities* change significantly.
7. Extract functional idioms from the latest revisions.
8. Answer RQ3 focusing on actual usage.

We collected the top 100 Java projects currently on GitHub, ranked them by the number of assigned stars based on our expectation that those projects would be large-scale and widely known, and then identified Java source code subject revisions for each project for the period from September 18, 2013 (half a year before Java 8 release) to April 10, 2018.

For each functional idiom (Lambda Expression, Stream, and Optional), we calculated the *density*, which indicates how frequently these idioms are used in Java projects. The calculations were performed by normalizing the number of Java files using functional idioms by the number of all Java files. Dyer et al. [16] applied this metric in an investigation aimed at determining how frequently new Java language features are used. To identify the idioms, first, we construct Abstract Syntax Trees (ASTs) using Eclipse JDT (`org.eclipse.core.dom` package). Then, we detect AST nodes corresponded to each idiom. Lambda Expression can be easily detected by finding `LambdaExpression` node belonging to the JDT package. Since no AST node directly corresponds to Stream idiom, we detect all `SimpleName` nodes which are identifiers in Java like variable name and method name. Detected nodes are filtered by two conditions; nodes contain identifier name `stream`, nodes are bound to `java.util.Stream` package. Optional idiom is identified in a similar way: detecting all `SimpleName` nodes, filtering by identifier `optional`, and filtering by `java.util.Optional` package. Note that *densities* calculated for the functional idioms will be denoted as d_{lambda}, d_{stream}, and $d_{optional}$ hereinafter.

To answer the RQ1, we investigated how frequently functional idioms are accepted in Java projects at the latest revision and determined the *density* value transitions for all subject revisions based on the *density* which is calculated for each revision.

To answer the RQ2, we conducted a qualitative study to collect reasons why developers use the idioms or not. In this study, the first research subjects are commit messages and issues discussed at the time when the *density* changes significantly from the previous revision. The second subjects include README.md, CONTRIBUTING.md, wiki page on GitHub, and their web site. As an extra study, we manually checked commit messages, issues, and comment sentences in source code by combining the following queries in order to cover the period while the *density* does not change significantly. The queries are not runnable language such as SQL, but just a pseudo-language for an explanation. We extract subject commits, issues and comment statements from all Java projects on GitHub using equivalent queries through the web user interface of GitHub.

```
Scope  = ("java 8" OR "java8")
Fi     = ("lambda" OR "stream" OR "optional")
Action = ("use" OR "accept" OR "remove" OR
          "replace" OR "reason")
Query  = (Scope OR Fi) AND Action
```

Please note that, because of the manual effort required, we limited our investigations to 100 commits, 100 issues, and 100 comments extracted from the GitHub search results.

We conducted static source code analysis at the latest revisions with focusing on what kind of usages are mainly adopted by the projects, and answer the RQ3. Note that subject projects are not all projects but ones whose developers accept the idioms. We focused on what form developers use functional idioms, what method they select on the idioms and how they use the method, and studied the following four themes:

- Number of statements in Lambda Expressions
- Number of cases which each Stream method is used in
- Patterns of method chain on Stream

- Number of cases which each Optional method is used in

5. RQ1: Do Java Developers Accept Functional Idioms?

5.1 Results

Figure 2 shows the result of classifying 100 subject projects into either *accept* and *not-accept* based on the *density* at the latest revision. We define projects whose *densities* at the latest revisions are more than 1% as *accept* and all others as *not-accept*. As shown in the figure, we can see that Lambda Expression is the most accepted functional idiom, being accepted by 16% of the projects. In contrast, Stream and Optional are only accepted by 2 to 3% of the projects.

Figure 3 shows the *density* transition of the *accept* projects. Note that in the result of Lambda Expression, only the top-five projects, ranked based on the *density*, are shown in this figure. Since d_{lambda} is higher than d_{stream} and $d_{optional}$ in all cases, it can be said that Lambda Expression is more frequently used in Java projects than Stream and Optional. Focusing on the result of Lambda Expression, the idiom was rapidly introduced to the *vert.x*, and *spark* projects right after Java 8 was released, whereas *PocketHub* project introduced the idiom assertively at a particular point in time and *proxyee-down* began to use the idiom just after their project started. Taken together, we can say that the developers of these top five projects were interested in Lambda Expression. In contrast, the maximum values of d_{stream} and $d_{optional}$ do not exceed 3%. Furthermore, there has been little increase in the *density* of the idioms in any of the surveyed projects. We conclude that Stream and Optional were hardly ever used.

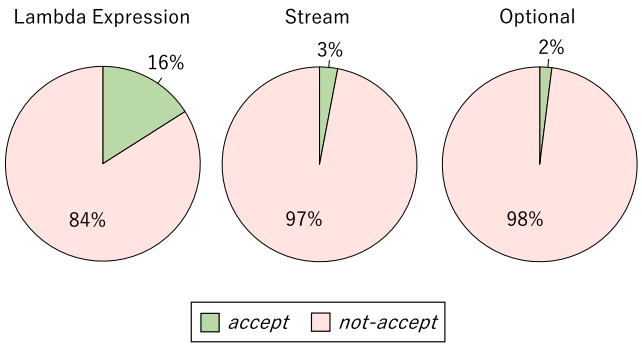


Fig. 2 Results of classifying 100 projects into either *accept* or *not-accept*

5.2 Discussion

We anticipated that there would be numerous opportunities to introduce functional idioms in source code because the idioms can be used as alternative code to frequent code snippet such as manipulating the collection element and defining anonymous functions. Furthermore, we set a very low threshold in defining *accept*. However, the percentage of projects accepting Lambda Expression is only 16% and those accepting Stream and Optional are no greater than 3%. Accordingly, it cannot be said that functional idioms are being frequently introduced into Java projects.

RQ1 conclusions

It cannot be said that functional idioms are being frequently accepted into Java projects.

6. RQ2: Why Do They Accept or not Accept the Idioms?

6.1 Results

The numbers of revisions, while the *density* was significantly changed, were 87 for Lambda Expression, 53 for Stream and 69 for Optional. We manually investigated both the above revisions and extra revisions retrieved by GitHub search.

Table 1 (a) summarizes the primary reasons why Java developers accept functional idioms. The ✓indicates which idioms were accepted by the described reason. In *guava*, developers accepted Lambda Expression and Stream because the improvements they provide make collections perform

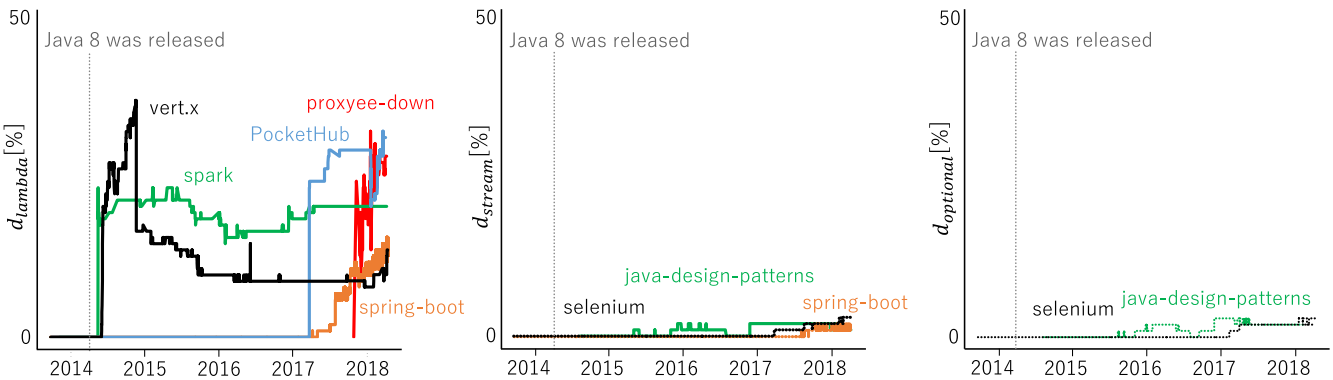


Fig. 3 The *density* translation of the *accept* projects. Note that only the top-five projects, ranked by the *density*, are shown in the graph of d_{lambda}

Table 1 Reasons for accepting or not accepting functional idioms

(a) Reasons for accepting

Project	Reason	Lambda Expression	Stream	Optional	Reference	Date
spark	To make the code cleaner	✓			pull/134	2014/04/07
selenium	To reduce the final jar size	✓			issues/4867	2014/11/01
guava	To make Guava's collections perform better with Stream	✓	✓		discussion†	2016/11/05
retrofit	To make converters for wrapping values into Optional			✓	commit/e985d	2017/03/12
realm-java	To support RxJava 2, which uses Lambda Expression	✓			commit/9ac68	2017/09/12
selenium	To write code more easily	✓	✓		pull/3495	2017/12/03

(b) Reasons for not accepting

Project	Reason	Lambda Expression	Stream	Optional	Reference	Date
GraalVM	To prevent recursion from overflowing the stack prematurely		✓		commit/bca7c	2014/09/09
guava	To avoid complications when handling exceptions	✓			issues/1670	2014/11/01
presto	To avoid decreasing performance		✓		README.md	2015/10/10
RxJava	To maintain backward compatibility	✓	✓	✓	commit/000a1	2016/02/04
Hystrix	To maintain backward compatibility	✓	✓	✓	commit/e102e	2016/08/19
selenium	To avoid making it harder to call methods			✓	commit/4c38c	2017/03/30
lottie-android	To keep support for JDK6/7	✓			commit/fa239	2017/04/08

better. However, it should be noted that the kinds of performance improvements that result from the idioms have not been specified. For *realm-java*, developers decided to introduce functional idioms for supporting *RxJava 2* which uses functional idioms in method APIs. *Selenium* developers used Lambda Expression and Stream in order to make writing code easier, and *spark* developers accepted Lambda Expression for the same reason. Additionally, *Spark* developers used Lambda Expression in order to reduce the final jar size. In *retrofit*, developers introduced Optional for the purpose of making converters that can wrap values into Optional because the tool that they use in *retrofit* does not allow null values in Stream.

Table 1 (b) summarizes the reasons for rejection. In *GraalVM*, Stream was not accepted, because the idiom causes the recursion to overflow the stack prematurely. As for *guave*, developers did not use Lambda Expression because it makes it necessary to create complex code to deal with checked exceptions. Project *presto* developers sometimes do not implement Stream for the same reason as a general criticism [11]. They avoid using Stream in inner loops and performance sensitive sections in order to decrease performance. Since we cannot catch checked exceptions thrown by Lambda Expression on the outside, it is necessarily needed to wrap the exceptions on the inside and unwrap the same type on the outside. Furthermore, *Hystrix*, *lottie-android*, and *RxJava* developers decided to reject functional idioms for backward compatibility reasons. One of the advantages of using Optional is that it prevents NullPointerException. However, some *selenium* developers decided not to use the idiom, because they claim that Optional does not prevent NullPointerException when used as parameters.

†https://groups.google.com/d/msg/guava-announce/o954PqvaXLY/7ss96X6sAwAJ

6.2 Discussion

From the results shown in Table 1 (a), we broadly categorize the reasons for the acceptance into the following three: to make source code simpler, to improve the performance of programs, and to facilitate compatibility with tools which use functional idioms. On the other hand, the rejection reasons can be classified into: to ensure backward compatibility, to facilitate debugging maintainability, and to prevent complications when handling checked exceptions.

RQ2 conclusions

Developers introduce functional idioms mainly with hoping to make simple and high performant programs. On the other hand, they do not use the idioms just because they should keep Java compatibility.

7. RQ3: How Do They Use the Idioms?

7.1 Results

- Number of statements in Lambda Expressions
  Figure 4 shows the result of calculating the number of statements in Lambda Expression. From the result, we can see that 1 statement is the most popular case for Lambda Expression. It is as twice as the second most popular case, 2 statement case. On the other hand, there is one case that Lambda Expression is composed of 29 statements.
- Number of cases which each Stream method is used in
  Figure 5 shows how frequently each Stream method is used. From the result, `map()` method is used in 129 cases, and it can be said that the method is the most popular among all Stream methods. This method applies the given function to the elements of Stream, and it is one of the intermediate operations. The second most

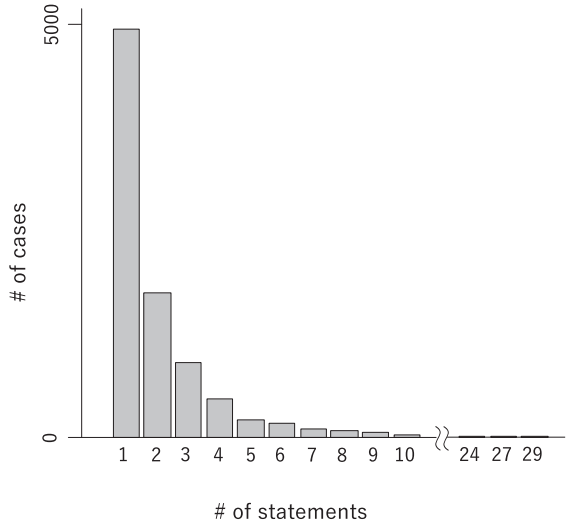


Fig. 4 Number of statements in Lambda Expressions

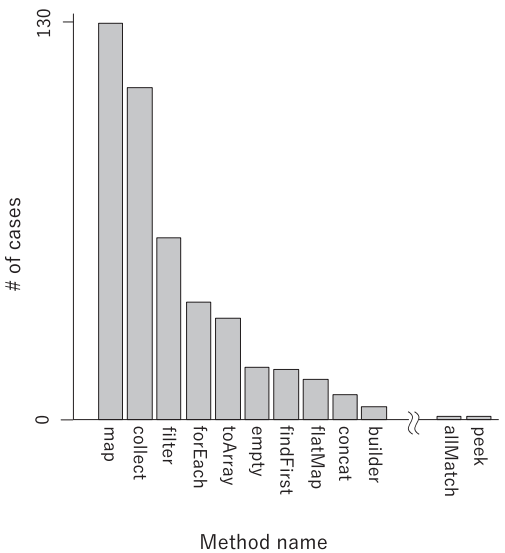


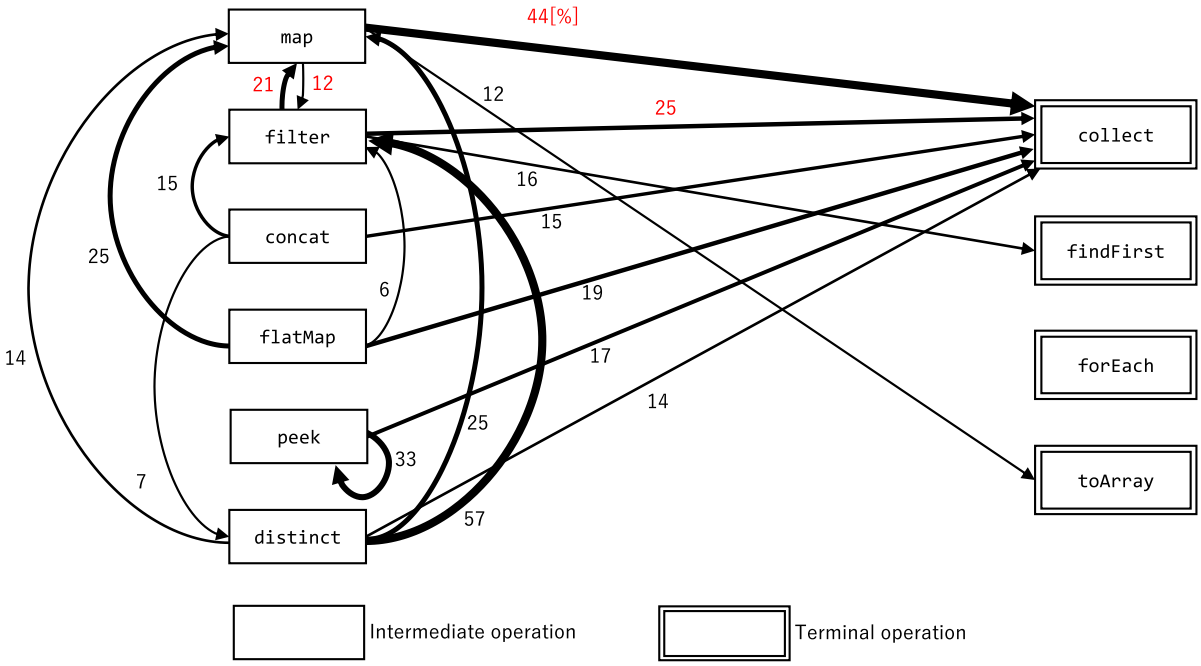
Fig. 5 Number of cases which each Stream method is used in

pervasive intermediate operation is `filter()` that returns elements which match the given condition. We can also see that `collect()` is the second most frequently used of all methods. This method aggregates elements of Stream, and it is one of the terminal operations. From there results, It is expected that these three methods are often combined and used to operate elements and aggregate them. On the other hand, `peek()` method is the most unfamiliar intermediate operation and `allMatch()` method which is one of the terminal operations is hardly used. `peek()` method returns Stream consisting of the elements at the time of the method being called. This method exists mainly to support debugging. Therefore, it might not be used frequently in released source code. `allMatch()` returns `true` when all Stream elements match the provided predicate. Otherwise, it returns `false`. This method returns `true` also when the Stream is empty. However, this behavior cannot be matched with the one expected by developers. Therefore, `allMatch()` method may not be used in many cases.

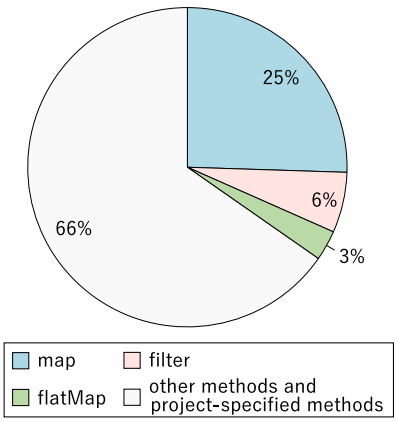
- Patterns of method chain on Stream
  Figure 6 (a) shows the transitions of Stream methods. The numbers corresponded the arrows are the percentage of transitions from one node to another. We can see that the most frequent chain pattern is composed of `map()`, `filter()` and `collect()` method. Figure 6 (b), 6 (c) shows which method is most commonly used in the start or end of method chain. We defined the first immediate operation which is run after generating Stream as the start of method chain. Note that only top-three methods are mentioned in both figures. From these three results, there are many cases which developers modify elements of Stream by `map()` and `filter()`, and aggregate the elements by `collect()`. In other words, we can say that developers use Stream mainly to operate its elements.
- Number of cases which each Optional method is used in
  Figure 7 shows how frequently each Optional method is used. From the result, `empty()` is used on 29 cases, and it may be the most popular method of all Optional methods. This method returns an empty Optional object. It is expected that this method might be commonly used when developers initialize Optional objects. On the other hand, `equals()` is used in only 1 case, and we can say that it is not popular among developers. `equals()` determine if two Optional objects are equal. We expect that this method is not commonly used because there might be many cases of comparing the elements of Optional, instead of the Optional objects. `orElse()` is also in only 1 case. It returns the value given to its parameter when the element wrapped into Optional is not present. However, the method is called also when the element is present. This may differ from the expectation of developers. Therefore, this method might not be used frequently. In some cases, `get()` and `isPresent()` are adopted. However, it is generally said that developers should not use these methods [17], [18].

7.2 Discussion

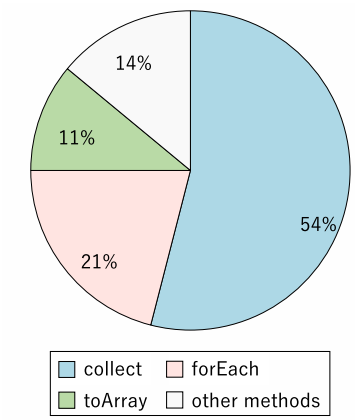
There are many cases using Lambda Expression composed of a single statement. Generally speaking, good usage of the idiom is not deeply nested and written in one line [19]. Thus, it can be said that Java developers keep this manner in their mind. Stream is mainly used to operate the elements of collections. Therefore, we can say that this usage is the accepted one among Java developers, and might be a good usage of the idiom. In contrast, there are several Lambda Expression consisting of a large number of statements. This use is against the benefit that Lambda Expression enables us to make code clear. Additionally, some developers introduce Optional with methods which they should not use. We expect that the tools which suggest refactoring deprecated usage are needed.



(a) Percentage of method transitions from one method to another. Note that this figure shows only top-three transitions for each node.



(b) Percentage of intermidiate operations which are run for the first time after generating Stream. Note that only top-three methods are mentioned.



(c) Percentage of terminal operations which are run at the end of method chain. Note that only top-three methods are mentioned.

Fig. 6 Patterns of method chain on Stream

RQ3 conclusions

While Lambda Expression and Stream are used frequently in the form of good usage, Optional is applied with deprecated methods. Thus, it is needed to develop tools to suggest refactoring bad usage.

8. Threats to Validity

The most serious threat to the validity of this study is that the definition of accepting idioms is ambiguous. Herein, we focused on the value of *density* at the latest revision and assessed the likelihood that the examined projects accept functional idioms. However, this method is unable to precisely determine whether functional idioms were adopted.

As for the threat to external validity, our subject was 100 Java projects collected from GitHub. Nevertheless, even though this subject provided a somewhat wide range of application domains, it might not be considered sufficient to produce generalizable results. Another threat to generalization is that we studied limited themes for answering RQ3. Therefore, our results may not cover all actual usages. In other words, if we had included a wider range of projects and themes, we might have obtained different results.

In both RQ1 and RQ2, we do not identify whether each idiom usage is appropriate or not. Our expectation is that the latest version of the master branch is more likely to contain appropriate usage because of some git-relating practice such as GitHub flow and pull-requests. If a project strictly

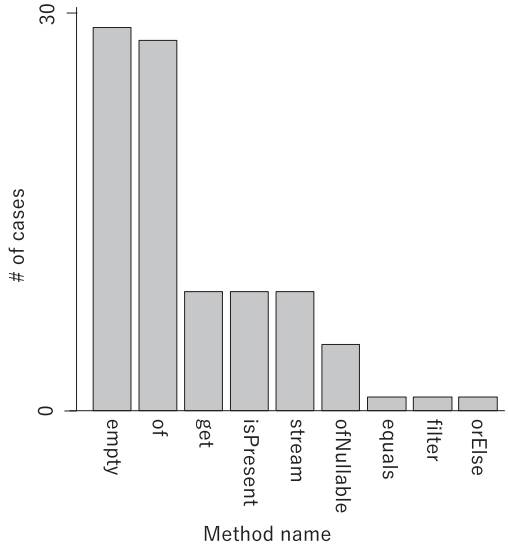


Fig. 7 Number of cases which each Optional method is used in

follows GitHub flow, the master branch is always reviewed and approved by at least one other developer. However, sometimes, inappropriate API usages may occur due to a lack of developer's knowledge. The results of the study are threatened by the problem. Identifying inappropriate usage would enable to organize bad practice.

The *density* metric used in RQ1 is calculated based on the information about whether there are functional idioms in source code or not. Thus, the metric ignores the numbers of using a specific idiom in a single Java file. Due to this, the *density* sometimes may not reflect how frequently the functional idioms are used.

The manual investigation for RQ2 was conducted in a limited number of commits and issues by focusing on *density* changes. Thus, it cannot be said that this study covers all of the reasons for accepting or not accepting functional idioms. It must be acknowledged that any discussions outside of our subjects will not be reflected in our results.

9. Related Work

There have been several previous studies on programming language evolution [3], [16], [20]–[22]. For example, Dyer et al. [16] studied Java feature adoption over time by analyzing over 18 billion abstract syntax tree nodes, while Parnin et al. [20] examined the adoption and use of generics, which were introduced into Java in 2004.

However, research into the introduction of functional idioms is becoming more active these days [23], [24]. For example, Mazinanian et al. [23] have conducted a large-scale empirical study of Lambda Expression to answer how Java developers introduce Lambda, and what are the reasons that motivate Java developers to use the Lambda. Our study shares some findings of their work on the positive reasons (e.g., the terseness of Lambda is the most general reason for accepting). On the other hand, our work examined both positive and negative reasons which motivate or prevent the use of Lambda. As a result, the most frequent negative reason is from backward compatibility rather than the powerful benefits of Lambda. This result might be one of the decision-making factors for developers who expect their developing system (especially library) to be used on various JVM versions. Usebeck et al. [24] conducted qualitative studies to determine the impact of Lambda Expression in C++ by comparing it to iterator. In our current study, we examine the use of all functional programming-inspired features including Lambda Expression, Stream and Optional in Java and survey the reasons why developers accept or do not accept the use of those idioms.

10. Conclusion

In this study, we investigated the current status of the use of functional idioms in Java projects and found that the idioms are not being frequently used, primarily because Java project developers avoid their use in order to facilitate backward compatibility and maintainability. On the other hand, some Java projects accept these idioms because they can improve performance and produce short, clear, and readable code. They use Lambda Expression frequently in the form of a single statement and adopt Stream to operate collection. However, in several cases, developers introduce Optional with using methods which they should not use.

According to the previous works, it has been revealed that a small number of developers account for the majority of using new language feature [16], and such fact can be seen also in Lambda Expression [23]. However, it should be considered the extent to which applying functional idioms affect development activities in projects and developers. By revealing the activeness change, we can grasp from a different point of view whether the idioms are accepted.

In the future, it will be interesting to define examples of bad usage of functional idioms in Java by conducting a survey of actual cases in which developers found it necessary to rewrite idioms. This would enable us to develop a tool which detects bad usage and suggest ways that developers could refactor idioms.

Acknowledgments

This work was supported by JSPS/MEXT KAKENHI Grant Number 16H02908 and 18H03222.

References

[1] P.J. Landin, "The next 700 programming languages," Communications of the Association for Computing Machinery, vol.9, no.3, pp.157–166, 1966.
[2] D. Spinellis, P. Louridas, and M. Kechagia, "The evolution of c programming practices: A study of the unix operating system 1973–2015," Proc. 38th International Conference on Software Engineering, pp.748–759, 2016.
[3] J.-M. Favre, "Languages evolve too! changing the software time scale," Proc. 8th International Workshop on Principles of Software Evolution, pp.33–42, 2005.

[4] R.W. Floyd, "The paradigms of programming," Communications of the Association for Computing Machinery, vol.22, no.8, pp.455–460, 1979.
[5] D.A. Watt, Programming Language Concepts and Paradigms, Prentice-Hall, Inc., 1990.
[6] D.M. Simmonds, "The programming paradigm evolution," Computer, vol.45, no.6, pp.93–95, 2012.
[7] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Object-oriented Modeling and Design, Prentice-Hall, Inc., 1991.
[8] P. Henderson, Functional programming: application and implementation, Prentice-Hall, Inc., 1980.
[9] R. Fischer, Java Closures and Lambda, ch. 7, Apress., 2015.
[10] T. Weiss, "The Dark Side Of Lambda Expressions in Java 8 | OverOps Blog." https://blog.takipi.com/the-dark-side-of-lambda-expressions-in-java-8/, (accessed Jan. 5, 2019).
[11] A. Zhitnitsky, "The 6 biggest problems of Java 8 - JAXenter." https://jaxenter.com/java-8-problems-112279.html, (accessed Dec. 26, 2018).
[12] Y. Cheon and A. Torre, "Impacts of java language features on the memory performances of android apps," tech. rep., University of Texas at El Paso, 2017.
[13] P.Y. Saumont, "What's Wrong in Java 8, Part IV: Monads - DZone Performance." https://dzone.com/articles/whats-wrong-java-8-part-iv, (accessed Jan. 5, 2019).
[14] S. Rauh, "Is Java 8 a Functional Programming Language?." https://www.beyondjava.net/java-8-functional-programming-language, (accessed June 26, 2019).
[15] P. Wadler, "The essence of functional programming," Proc. 19th Principles of Programming Languages, pp.1–14, 1992.
[16] R. Dyer, H. Rajan, H.A. Nguyen, and T.N. Nguyen, "Mining billions of ast nodes to study actual and potential usage of java language features," Proc. 36th International Conference on Software Engineering, pp.779–790, 2014.
[17] M. Winnicki, "Optional isPresent() Is Bad for You - DZone Java." https://dzone.com/articles/optional-ispresent-is-bad-for-you, (accessed Dec. 21, 2018).
[18] M.P. Gioiosa, "Java 8 Optional - Replace Your Get() Calls - DZone Java." https://dzone.com/articles/java-8-optional-replace-your-get-calls, (accessed Dec. 21, 2018).
[19] V. Subramaniam, "Java 8 idioms: Why the perfect lambda expression is just one line." https://www.ibm.com/developerworks/library/j-java8idioms6/index.html, (accessed Dec. 26, 2018).
[20] C. Parnin, C. Bird, and E. Murphy-Hill, "Java generics adoption: How new features are introduced, championed, or ignored," Proc. 8th Working Conference on Mining Software Repositories, pp.3–12, 2011.
[21] L.A. Meyerovich and A.S. Rabkin, "Empirical analysis of programming language adoption," Special Interest Group on Programming Languages Notices, vol.48, no.10, pp.1–18, 2013.
[22] M. Hoppe and S. Hanenberg, "Do developers benefit from generic types?: An empirical comparison of generic and raw types in java," SIGPLAN Not., vol.48, no.10, pp.457–474, 2013.
[23] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig, "Understanding the use of lambda expressions in java," Proc. Association for Computing Machinery on Programming Languages, pp.85:1–85:31, 2017.
[24] P.M. Uesbeck, A. Stefik, S. Hanenberg, J. Pedersen, and P. Daleiden, "An empirical study on the impact of c++ lambdas and programmer experience," Proc. 38th International Conference on Software Engineering, pp.760–771, 2016.

Hiroto Tanaka received the BI degree from Osaka University in 2018. He is a master course student at Osaka University. His research interests include mining software repositories.

Shinsuke Matsumoto received the ME and Ph.D degrees from Nara Institute of Science and Technology in 2008 and 2010, respectively. He is currently an assistant professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include empirical software engineering.

Shinji Kusumoto received the BE, ME, and DE degrees in information and computer sciences from Osaka University in 1988, 1990, and 1993, respectively. He is currently a professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include software metrics and software quality assurance technique. He is a member of the IEEE, IEICE, and JFPUG.