Understanding the use of lambda expressions in Java

DAVOOD MAZINANIAN*, Concordia University, Canada
AMEYA KETKAR*, Oregon State University, USA
NIKOLAOS TSANTALIS, Concordia University, Canada
DANNY DIG, Oregon State University, USA

Java 8 retrofitted lambda expressions, a core feature of functional programming, into a mainstream object-oriented language with an imperative paradigm. However, we do not know how Java developers have adapted to the functional style of thinking, and more importantly, what are the reasons that motivate Java developers to adopt functional programming. Without such knowledge, researchers miss opportunities to improve the state of the art, tool builders use unrealistic assumptions, language designers fail to improve upon their designs, and developers are unable to explore efficient and effective use of lambdas.

We present the first large-scale, quantitative and qualitative empirical study to shed light on how imperative programmers use lambda expressions as a gateway into functional thinking. Particularly, we statically scrutinize the source code of 241 open-source projects with 19,770 contributors, to study the characteristics of 100,540 lambda expressions. Moreover, we investigate the historical trends and adoption rates of lambdas in the studied projects. To get a complementary perspective, we seek the underlying reasons on why developers introduce lambda expressions, by surveying 97 developers who are introducing lambdas in their projects, using the firehouse interview method.

Among others, our findings revealed an increasing trend in the adoption of lambdas in Java: in 2016, the ratio of lambdas introduced per added line of code increased by two-fold compared to 2015. Lambdas are used for various reasons, including but not limited to (i) making existing code more succinct and readable, (ii) avoiding code duplication, and (iii) simulating lazy evaluation of functions. Interestingly, we found out that developers are using Java's built-in functional interfaces inefficiently, i.e., they prefer to use generic functional interfaces over the specialized ones, overlooking the performance overheads that might be imposed. Furthermore, developers are not adopting techniques from functional programming, e.g., currying. Finally, we present the implications of our findings for researchers, tool builders, language designers, and developers.

CCS Concepts: • **Software and its engineering → Language features**; *Functional languages*; *Multiparadigm languages*; Object oriented languages;

Additional Key Words and Phrases: Java 8, Lambda Expressions, Functional Programming, Empirical Studies, The Firehouse Interview Method, Multi-paradigm Programming

ACM Reference Format:
Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2017. Understanding the use of lambda expressions in Java. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 85 (October 2017), 31 pages.
https://doi.org/10.1145/3133909

*The first two authors of this paper had an equal contribution.

2475-1421/2017/10-ART85
https://doi.org/10.1145/3133909

1 INTRODUCTION

Lambdas are the core features of the *functional programming* paradigm. Over the last years, several mainstream object-oriented languages, such as Java 8, C# 3 and C++ 11, added lambda expressions to facilitate the functional programming style, and the parameterization of behavior.

The original intention of the Java language designers [Oracle 2013] behind the addition of lambda expressions was to provide a clear and concise way to represent one-method interfaces using an expression, improve the *collections framework* by providing an alternative way to iterate through, filter, and extract data from collections using streams, and improve performance in multi-core environments with new concurrency features using parallel streams. Since then, both researchers [Franklin et al. 2013] and IDE providers [Bull 2014; IntelliJ 2017; Wielenga 2014] developed tools to automatically refactor code to take advantage of lambda expressions. The adoption of these tools facilitates the migration from anonymous classes to lambda expressions, and from enhanced `for` loops to streams. Moreover, IDE providers improved substantially their debugging support for lambda expressions [Gee 2015], by allowing to set breakpoints for and step into lambda expressions.

Despite the growing tool support for lambda expressions in Java IDEs, there is still limited empirical evidence about the *adoption* of lambdas by Java developers, the actual *reasons* that motivate or discourage Java developers from using lambdas, and the *practices* applied by Java developers when using lambdas. The lack of this knowledge impacts negatively four audiences:

(1) *Researchers* are not aware of the research gaps (i.e., the actual unsolved problems faced by the developers), and thus miss opportunities to improve the current state of the art.
(2) *Language and library designers* do not know if the programming constructs and APIs they provide are effectively used by the developers, or are rather misused or underused.
(3) *Tool builders* do not know how to tailor their tools, such as recommendation systems and code assistants, to the actual needs and practices of the developers when using lambda expressions.
(4) *Developers* are not aware of the good and bad practices related to the use of lambda expressions.

In this work, we perform the first large-scale study on 241 open-source projects hosted on GitHub, containing 100,540 lambda expressions. In addition, we study the evolution history of these projects to discover trends in the adoption rate of lambdas. Moreover, to better understand the reasons motivating developers to use lambdas, we survey 97 developers of these projects right after introducing a lambda expression. Using the collected information, we answer the following research questions:

RQ1: *What* features of lambda expressions are adopted by Java developers? We found that developers mostly create their own custom functional interfaces rather than using the ones provided by Java in the `java.util.function` package, mainly due to the need for exception handling capabilities and extending other interfaces. In addition, we found that 20% of developers use unnecessary boxing and unboxing without thinking about the performance overhead. Moreover, test code has significantly higher lambda density than production code.

RQ2: *How* do Java developers introduce lambda expressions? We investigate the introduction rate of lambdas over time, starting from the first commit introducing a lambda until the last commit of the project. We found an increasing trend in the adoption rate of lambdas.

RQ3: *Who* introduces lambda expressions? We found that a small to moderate percentage of developers introduce most of the lambdas. Moreover, core contributors introduce slightly more lambdas than outsiders.

RQ4: *How* do Java developers introduce lambda expressions? We found evidence of migration efforts in several projects by converting anonymous classes to lambdas, replacing loops/conditionals with streams, and enhancing functionality by wrapping existing code to lambdas. Moreover, we found that most lambdas in new code are introduced without tool support.

RQ5: *Why* do Java developers introduce lambda expressions? We found 14 common reasons motivating developers to introduce lambdas, including improved readability, class creation avoidance, behavior parameterization, simulating lazy evaluation, callback implementation, and improved testability. Developers also employ lambdas to implement object-oriented design patterns and replace class inheritance with object composition.

Using this rich data, we reveal several actionable implications. Researchers can create novel applications of lambda expressions (e.g., in the GoF design patterns [Gamma et al. 1994]) and focus as well on code migration scenarios, thus increasing the adoption of lambdas. Language and library designers can find several pain points that developers wrestle with, which call for better language support. Among others, we highlight support for checked exception and optimization of generic boxed versions into specific primitives. Tool builders can improve tools to provide relevant recommendations and assistance intelligently, for example by suggesting currying refactorings. Developers can educate themselves on how to use lambdas wisely and prudently, for example by becoming aware of the performance implications when using boxing, or the effect of lambdas on code readability.

This paper makes the following contributions:

(1) To the best of our knowledge, this is the *first large-scale empirical study* to answer questions about the use of 100,540 lambda expressions by Java developers, analyzing the source code and its evolution.
(2) We designed and conducted a *survey* based on the firehouse interview method [Murphy-Hill et al. 2015] with 97 GitHub contributors to provide insights about the reasons motivating or discouraging Java developers from using lambdas.
(3) We developed *tools* to collect information about lambda usage and analyze the lambda evolution history of Java projects. We applied these tools on 241 open-source projects hosted on GitHub, and make the collected information publicly available for further research and reuse.
(4) We present *an empirically-justified set of implications* of our findings from the perspective of four audiences: researchers, tool builders, language designers, and developers.

2 BACKGROUND

Functional programming languages (e.g., LISP and Haskell) have been fundamentally designed based on the principles of Lambda Calculus [Church 1932]. However, several other languages that are not inherently functional (e.g., object-oriented languages, such as C++, C# and Java) have also adopted lambdas. In most of these programming languages, lambda expressions are declared in the form of anonymous functions which, similar to Lambda Calculus, can be passed to or returned by other functions. This essentially means that *behavior* can be passed as an argument using lambdas, thus enabling *behavior parameterization*.

In Java, lambdas are essentially constructs that can be used to implement interfaces with only one abstract method, i.e., *functional interfaces*. A lambda expression in Java consists of 1) a comma-separated list of formal parameters enclosed in parentheses (the types of the parameters can be omitted, and if there is only one parameter, the parentheses can be omitted as well), 2) the arrow token (`->`), and 3) the body of the lambda expression, which can be a simple expression or a block of statements. Figure 1 shows an example of a lambda expression in Java, from the project `intellij-community`. In this example, method `find()` is responsible for finding a specific object

```
@FunctionalInterface
interface Condition<T> {
   boolean value(T t);
}

public static <T> T find(
      @NotNull T[] array,
      @NotNull Condition<? super T> condition) {
   for (T element : array) {
      if (condition.value(element))
         return element;
   }
   return null;
}

template = find(list,
   new Condition<ProjectTemplate>() {
      @Override
      public boolean value(ProjectTemplate template) {
         return templateName.equals(template.getName());
      }
   }
);
```

(a) Anonymous class implementation

```
@FunctionalInterface
interface Condition<T> {
   boolean value(T t);
}

public static <T> T find(
      @NotNull T[] array,
      @NotNull Condition<? super T> condition) {
   for (T element : array) {
      if (condition.value(element))
         return element;
   }
   return null;
}

template = find(list,
   template1 -> templateName.equals(template1.getName())
);
```

(b) Lambda expression implementation

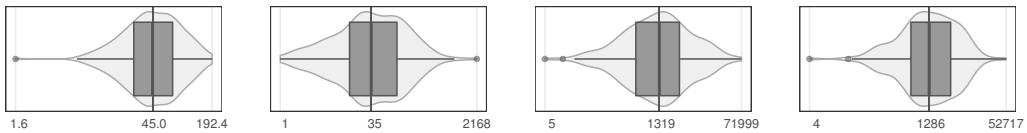
Fig. 1. A piece of Java code using lambda expression, and equivalent anonymous class implementation

in a list that meets certain criteria, and the callers can define the criteria using the `condition` parameter. In Figure 1b, a lambda expression is used to provide such criteria, by implementing the functional interface `Condition<T>`. The code inside the lambda expression's body (i.e., the condition) is evaluated when the abstract method of the corresponding functional interface is called (i.e., the `value()` method call inside the `for` loop). Without lambda expressions, the functional interface `Condition<T>` could alternatively be implemented using an *anonymous* class, as depicted in Figure 1a.

Like anonymous classes, a lambda expression in Java can only access local variables and parameters of the enclosing block that are *final* or *effectively final* (i.e., a non-final local variable or method parameter whose value is never changed after initialization). Therefore, lambda expressions in Java can be considered as syntactically shorter alternatives for anonymous classes; however, they are not completely equivalent. While anonymous classes are treated similarly to normal classes in bytecode, lambda expressions are translated and evaluated differently. In a nutshell, the body of a lambda expression is converted to a method, which is placed in the inner most class where the lambda expression appears, and is declared as a static method, if the lambda is *non-capturing* (i.e., it does not use any variables outside of its body), or as an instance method otherwise. The invocation of this method is done dynamically, using a specific JVM instruction called `invokedynamic` [Goetz 2012]. Consequently, lambdas may have different performance implications compared to their counterparts (i.e., anonymous classes). In this study we found that some developers are not aware of the performance implications of using lambdas versus anonymous classes, while others take the performance considerations seriously into account.

3 RESEARCH METHODOLOGY

In this study, we employ both quantitative and qualitative methods for answering our research questions. The qualitative approach helps to answer *why* Java developers use lambda expressions, while the quantitative approach allows answering other research questions. Indeed, the results of the



(a) Age (Months) (b) Authors (c) Commits (d) Java files

Fig. 2. Size metrics for the studied projects

qualitative study motivated several other research questions, that could be answered quantitatively. More importantly, blending the two methods has the advantage that the results can be triangulated [Easterbrook et al. 2008].

Specifically, first we mine the commit history of several popular (as counted by the number of stars) Java projects on GitHub, in order to understand the evolution trends in the usage of lambda expressions. Then we conduct a more thorough static analysis (by building the projects and resolving type binding information) of the source code in the latest revisions of these Java projects, to gain a better understanding about the characteristics of the lambda expressions developers introduce, and the ways they use them. In addition, we conduct a qualitative study, following the *firehouse interview* method [Murphy-Hill et al. 2015], to collect information from the developers right after they introduce a new lambda expression, and to find out the underlying motivations for adopting lambda expressions in Java projects. In the following subsections, we will discuss these methods in more detail.

3.1 Subject Systems

We collected the top 2000 Java projects on GitHub, ranked based on the number of stars (i.e., the *stargazers count*). Our dataset covers a wide range of Java projects, allowing to have diversity with respect to the application domain, size, development ecosystem, contribution governance, testing practices, and so on. This diversity is important to make sure that the collected data is representative for the study. First, we conducted an initial analysis on the latest revision of these projects to understand which ones have already started adopting lambda expressions. Particularly, we parsed the source code of the projects to generate Abstract Syntax Trees (ASTs), and then queried the ASTs to find uses of lambda expressions. All projects without any use of lambda expressions were filtered out, as they are not suitable subjects for our study. After the filtering phase, 241 projects remained for further analysis. Due to the space constraints, we cannot provide the complete list of these projects; however, the list is available online [Mazinanian et al. 2017].

Next, we monitored these projects for a period of 90 days (January 4^{th} - April 3^{rd}, 2017) on a daily basis to detect new lambda expressions introduced by developers. Out of these 241 projects, 223 projects had at least one commit during the period of our study, and in 136 projects, at least one lambda expression was introduced in the same period. To gain a better understanding about the studied projects, in Figure 2, we depict the violin plots [Hintze and Nelson 1998] for different size metrics of the 241 studied projects, including the age (in months), and the number of Java files, commits, and contributors. These plots confirm that our corpus is very diverse.

3.2 Mining Historical Data

We mined the source code histories of the subject systems, in order to study the introduction of new lambda expressions from different aspects:

- the historical trend of the introduction rate of lambdas (RQ2)
- the characteristics of the developers who introduce lambdas (RQ3)
- the ratio of lambdas used to migrate existing code, to those used to add new functionality (RQ4)

More specifically, we analyzed each revision of the projects, keeping track of the added lambdas to the source code. Accurately detecting newly-added lambda expressions is not straightforward, as a commit might include several changes, such as refactorings, where existing lambda expressions are moved to other places in code.

We compared each revision with its parent revision in the Git's directed acyclic graph (DAG), using the RefactoringMiner tool [Silva et al. 2016]. RefactoringMiner was originally designed to detect the refactoring operations applied in the commit history of a project. However, in order to detect refactoring operations, it first creates a mapping between the methods of a class in the current and its parent revision. During this process, it finds the methods that have been removed from the parent revision and those that have been added in the current revision, and then examines the similarity of the code inside their bodies to infer methods with updated signatures (i.e., method rename, parameter addition/deletion/replacement).

Using this information, we focus our analysis on the methods reported as newly added and the methods which have been mapped to an already existing one in the parent revision. In the case of newly added methods, we use an AST visitor to extract any lambdas that they might contain within their body, and record them as *newly added lambdas within new methods*. In the case of mapped methods, we use the same AST visitor to extract the lambdas contained within their body in the parent and current revision. We record as unchanged lambdas those that have an identical signature and body, and all remaining ones from the current revision we record as *newly added lambdas within existing methods*. In addition, using the refactoring information reported by RefactoringMiner, we exclude from our analysis the methods introduced due to refactoring (i.e., extracted and moved methods), because the lambdas contained within their body already exist in the parent revision, and thus should not be counted as newly added.

3.3 Static Source Code Analysis

We performed a thorough static analysis of the latest revisions of the source code, in order to investigate:

- the characteristics of the lambda expressions introduced by the developers, and discover good or bad practices in the way Java developers use lambdas (RQ1)
- the locations that developers tend to use more lambda expressions, e.g., production or test code (RQ1)

We implemented our static analysis tools on top of two AST Parser libraries, namely Eclipse JDT [development tools 2017] and JavaParser [JavaParser 2017]. Our static analysis extracts the syntactic information of the lambdas from the source code, namely the number of parameters, the information about the parent node of the lambda in the AST, and the location of lambda in source code. Eclipse JDT gives us further in-depth information from the compiler, namely the fully-qualified name of the type binding resolved for the functional interface in context, the exception handling capability of the functional interface, whether the functional interface is built-in (provided in Java 8 APIs), and the signature of the abstract method defined in the functional interface, including the type of its parameters and the return type.

To obtain precise binding information using Eclipse JDT, we had to build the studied projects. The majority of the projects in our corpus use a build system, such as Maven and Gradle, which makes the project building process rather straightforward by automatically downloading all required libraries. However, in several cases we found missing libraries that required considerable manual effort to be resolved. We successfully built 147 projects, making it possible for us to use Eclipse JDT to collect in-depth information from 59,984 lambdas. We used JavaParser to analyze the 94 remaining projects, giving us just the syntactic information for over 40,556 lambda expressions.

Overall, our study covers 100,540 lambdas found in the 241 studied projects, which gives us a large dataset to answer our research questions comprehensively.

3.4 Qualitative Study

The most reliable way to reveal the actual motivations behind the use of lambda expressions (RQ5) is to consult with developers that frequently use them. We adopted the *firehouse interview* research method [Murphy-Hill et al. 2015]. In this method, a phenomenon is studied right after it happens (e.g., observing the behavior of victims right after a house catches fire). In the same manner, we asked the developers of the examined projects to answer a few questions, right after they introduced a lambda expression in the code, with the aim of understanding *why* they introduced it. This kind of immediate interaction allows the developer to have her memory fresh when answering our questions, and thus provide more reliable answers. Indeed, Silva et al. [2016] showed that this method has a much high response rate, compared to other types of survey-based studies that require the participation of developers. Our qualitative study follows:

3.4.1 Monitoring Repositories for Identifying Newly-Added Lambdas. We monitored daily the examined projects hosted on GitHub, using the `pull` command to fetch the most recent commits to locally-cloned repositories. Every new commit was analyzed by RefactoringMiner to find newly added lambdas. We developed a convenient web application [Mazinanian et al. 2017] that allowed the authors to inspect the detected lambdas and generate an email to contact the developer, to conduct the firehouse interviews.

3.4.2 Filtering out Lambdas Introduced Due to Evident Motivations. In several cases the use of a lambda expression is required by the API being used. For instance, the *Stream* API introduced in Java 8, provides several methods that encourage the passing of a lambda expression as an argument. In such cases, the reason for the introduction of a lambda is evident (i.e., API requirement), and thus we decided to exclude these cases from sending an email to the corresponding developers. To avoid any error or bias when excluding such evident use cases, on each day, two authors of the paper independently examined every newly detected lambda expression to decide whether it should be excluded or not. The developers were contacted by email only when both authors agreed with each other that a lambda use case should be further investigated. Particularly, we excluded all lambda expressions (more precisely, 47,670 lambda expressions) belonging to the following categories:

- Lambdas passed as arguments to methods provided by the Java 8 API, including but not limited to the *Stream* API (e.g., `map()`, `filter()`), the *Collections* API (e.g., `forEach()`, `computeIfPresent()`, `computeIfAbsent()`) and the *Optional* class [Oracle 2014] (e.g., `ifPresent()`, `ifAbsent()`);
- Lambdas introduced from the conversion of anonymous classes;
- Lambdas used for implementing Single Abstract Method interfaces provided by the Java API, such as `Comparator`, `Runnable` and `ActionListener`.

In addition, to avoid spamming the same developers with multiple emails, we used a second filtering phase according to the following rules:

(1) If a developer introduced multiple lambdas in a single revision, we send an email for one of the detected lambdas, based on the consensus between the first two authors of this paper;
(2) If a developer has already responded to a previous email, we exclude all her subsequent lambdas;
(3) If a developer has been contacted for a lambda introduced in an older revision, but has not responded by the time we detect a lambda in the current revision, we send a final email for the newly introduced lambda.

The filtering process was performed through a web application that we specifically designed for this study. This web application also stored all communications with each developer (Section 3.4.3), and we used it for assigning labels (i.e., codes) to the developers' responses (Section 3.4.4) as well.

3.4.3 Contacting Developers. We contacted the developers introducing lambda expressions that we considered worthy of further investigation by sending an email to the address provided in their GitHub account. The body of each email message was automatically generated by the web application we developed, and included the following information:

- A short message introducing the research team and explaining the purpose of our study.
- The rank of the contacted developer among the developers of the project with respect to the number of introduced Lambda expressions in the project.
- The rank of the project among the examined projects with respect to the average density of lambdas per class in the project.
- A GitHub URL to the commit and the exact line where the lambda expression was introduced, so that the developer can easily find it and inspect it.
- Three questions that should be answered by the developer:

(1) Why did you introduce this lambda expression?
(2) Did you introduce it manually or used an automated tool (quick fix/assist, refactoring)?
(3) What IDE are you using?

The first question aims at discovering the actual motivations behind the introduction of lambdas as expressed by the developers themselves. The other two questions aim at understanding whether developers trust and use tool support for introducing lambdas in their project. This is important, as there is fairly mature IDE support for lambdas, e.g., refactoring anonymous classes to lambdas [Franklin et al. 2013].

In total, we sent 351 emails to developers, out of which 97 responded, bringing us to a 27.6% response rate. This is significantly higher than the usual response rate achieved in questionnaire-based software engineering surveys, which is around 5% [Singer et al. 2008].

3.4.4 Thematic Analysis. As the developers' responses are in free-form text (i.e., emails), a qualitative data analysis approach is needed to systematically extract useful information from them. *Thematic analysis* "is a method for identifying, analyzing, and reporting patterns (themes) within data" [Cruzes and Dybå 2011]. It is a widely-used qualitative approach that allows gaining a deeper understanding of the data [Wohlin and Aurum 2015]. We followed the steps required by thematic analysis as suggested by other researchers [Braun and Clarke 2006]. First, two authors of the paper independently read the developers' responses that were received every day, and replied back to the developers if further clarifications were needed. Any additional responses from the developers were also added to the dataset used for the thematic analysis.

The next step involves the coding process, where both authors independently read the responses carefully, sentence by sentence, and assigned one or more descriptive phrases (i.e., codes) to each sentence. Coding developers' responses yielded the *explicit codes*; however, the coders tried to capture *latent codes* [Braun and Clarke 2006] by looking at the lambda expressions and the surrounding source code. This is necessary, especially when the responses referred to specific places in code. Furthermore, latent codes could help in gaining a clearer picture about the context in which the lambda expressions were created.

To make sure that the coding process was consistent among the two coders, an initial meeting was held after a random sample (having around 25% of the data) was coded by both coders (in the literature, the suggested minimum size for this random sample is 10% [Campbell et al. 2013]). During the meeting, the coders investigated, discussed and assessed the coding process (i.e., whether the

coders conduct the coding correctly and with enough care and detail), and also negotiated any disagreements between the assigned codes. After 80% inter-coder agreement was achieved, the coders started assigning codes in the entire dataset of the developers' responses. Note that, there is no accepted minimum value for the inter-coder agreement level, and values ranging from 70% to more than 90% are suggested in the literature [Campbell et al. 2013].

After the coding finished, the coders held another meeting in order to finalize the codes and extract *themes* from them. Themes "capture something important about the data in relation to the research question, and represent some level of patterned response or meaning within the data set." [Braun and Clarke 2006]. Themes are extracted by finding a relationship between the codes (and possibly combining some of them). For instance, several responses contained the codes "*[lambdas are introduced] To avoid verbosity*", "*[lambdas are introduced] To have fewer lines of codes*", "*Lambdas are more precise*", "*Lambdas are laconic*", and "*Lambdas are terse*", and we used a theme like ***Terseness of lambdas*** to merge all these codes into a common one. The two authors reviewed the initial themes against the data several times, and refined their names and definitions until they both agreed that there were no further refinements possible.

4 RESULTS

4.1 RQ1: *What* Features of Lambda Expressions Are Adopted by Java Developers?

4.1.1 Use of Custom Versus Built-In Functional Interfaces. Java 8 provides 43 built-in functional interfaces (in the `java.util.function` package) that can be used as types for lambda expressions. These functional interfaces cover a large number of input/output parameter combinations, so that the developers do not need to introduce their own custom functional interfaces. For example, the functional interface `Supplier<T>` represents lambdas that accept no parameter and return an object of type `T`. Conversely, `Consumer<T>` is used for lambdas that accept an object of type `T`, and return nothing. The question is, to what extent these built-in functional interfaces satisfy the developers' needs, and are they underused or misused?

We studied 59,984 lambda expressions used in 147 projects, by resolving the type that they are bound to at compile time. We found that 32% of lambda expressions use built-in functional interfaces, and 11% implement Single Abstract Method interfaces provided by the Java API, such as `Comparator` and `Runnable`. While the majority of the built-in functional interfaces (36 out of 43 in total) are specialized for primitive types (e.g., `IntToDoubleFunction` is a functional interface which accepts an `int` and returns a `double`), we found that only 7% of the lambdas are using this kind of specialized functional interfaces. However, our analysis revealed that 20% of the studied lambda expressions could have been replaced with a functional interface specialized for a primitive type. For example, `Function<Integer, T>`, where `T` can be any type, was used more than `IntToDoubleFunction`, `IntToUnaryFunction`, `IntToLongFunction` and `IntFunction` combined. We found similar trends for other primitive data types and functional interfaces.

In essence, Java 8 provides these specialized interfaces to improve performance. These functional interfaces use primitive types in the signature of the abstract method, consequently, when using these interfaces, there is no need for auto-boxing by converting primitive types to their corresponding object wrapper classes.

The data shows that developers are using functional interfaces *inefficiently*. We hypothesize different reasons for this inefficient use of built-in functional interfaces, e.g., lack of awareness of the available options, or the tediousness of finding the correct type of functional interface among the 43 options. To the best of our knowledge, no IDE can detect inefficient uses of functional interfaces, or allow refactoring an instance of a general functional interface to a specialized one.

Table 1. Reasons for defining a custom functional interface

Reason	Description	% Interfaces[1]	% Lambdas[2]
Throwing exceptions	The single abstract method of the functional interface needs to throw an exception, which is not handled by built-in functional interfaces.	23.01	42.04
Extending other interfaces	Functional interface extends another interface (e.g., `Serializable`)	25.15	63.53
Using annotations	Annotations (e.g., `@Nullable`) are defined for parameters/return type of the single abstract method or at method-declaration-level.	14.78	4.41
Default methods in the interface	Functional interface needs to have default methods besides the single abstract method.	12.95	2.61
Static methods in the interface	Functional interface needs to have static methods besides the single abstract method.	3.96	1.06
Static fields in the interface	Functional interface declares static fields besides the single abstract method.	1.98	1.01
External API SAM interface	The functional interface is declared in an external library to which the code depends, and thus cannot be migrated to built-in functional interface.	19.51	10.00
Other	Cases where a built-in functional interface could be used instead, but the reason for not doing so cannot be not known from the source code.	54.42	56.39

[1] Shows the percentage of *custom functional interfaces* introduced due to each reason. Note that, the percentages are not supposed to sum up to 100%, since there could be a custom functional interface that extends a built-in functional interface and at the same time declares a default method as well.
[2] Shows the percentage of all *lambdas* that were instantiated from the custom functional interfaces.

We also found that 57% of the lambdas are instantiated from *custom functional interfaces*, i.e., the ones that are defined by the developers. Why do developers need to define custom functional interfaces, given that Java provides so many built-in options?

One possible reason is that developers are *forced to*. As an example, there is no functional interface of which the single abstract method accepts more than two arguments, in the list of Java's built-in functional interfaces. The number of built-in functional interfaces would obviously grow exponentially, if Java language designers wanted to include functional interfaces capable of accommodating more parameters. However, we found that only 10% of the lambdas needed more than 2 parameters.

In functional programming, the need for larger arity is handled by currying, i.e., by converting functions of high arity to a chain of functions that accept only one parameter and return a function that partially applies that single parameter on the functions' body. To detect instances of currying, we first captured all the functional interfaces (built-in and custom) used in each project, using the binding information of the lambdas defined in that project. Then, we queried the ASTs of the project for identifying functional interfaces which accept one parameter and return another functional interface, which in turn accepts one parameter, but can return anything. Each implementation of these functional interfaces in the form of a lambda expression is essentially an application of currying. We found that only 1% of the total lambdas use currying.

In order to find other reasons for defining custom functional interfaces, we manually investigated 656 custom functional interfaces from a randomly-selected subset of the dataset (particularly, covering 32,268 lambda expressions), and summarized the results in Table 1. Notice that the most prominent reasons that *force* the developers to use custom functional interfaces are the lack of exception handling in the Java's built-in functional interfaces, and the need for extending other interfaces (e.g., `java.io.Serializable`).

Particularly, it turned out that 62.91% of the custom functional interfaces of which the single abstract method needed to throw a checked exception (accounting for 14.48% of all the studied functional interfaces), were exactly similar to built-in functional interfaces with an additional

Table 2. Where are lambda expressions used in code?

Parent type	Frequency	Percentage
Method invocation	87821	87.35
Initializer of variable declaration fragment	5165	5.14
`return` statement	3930	3.91
Class instance creation	2328	2.31
Assignment	680	0.68
`super()` constructor invocation	280	0.28
Lambda expression[1]	223	0.22
Enum constant declaration	54	0.05
Conditional expression	32	0.03
Array creation	27	0.03
Total	**100,540**	**100%**

[1] Cases like `a -> b -> c`, i.e., nested lambda expressions.

requirement to handle exceptions. In other words, these custom functional interfaces could be replaced with the built-in ones, if they had exception handling capability. We observed very little use of third-party functional interfaces (i.e., external libraries) to handle exceptions. We hypothesize that the developers find it easier to maintain their own checked functional interfaces, since they will have the freedom to extend other interfaces, add default methods, annotations, and static methods/fields.

In Table 1, for a significant portion of functional interfaces (i.e., 54.42%) we were not able to understand from the source code why the developers did not use built-in functional interfaces, while they definitely could (i.e., the single abstract method defined in the interface did not throw an exception, had less than 3 parameters, and there was no default or static methods or static fields declared in the interface). We can hypothesize different reasons for this behavior, for example, the developer might be unable to use a built-in functional interface because it is declared in legacy code that is difficult to be migrated, or because it is part of the public API of the system that should remain intact. Another possible reason is that the developer seeks a more expressive name for the interface, e.g., `Processor` instead of the built-in `Consumer`.

4.1.2 The Location of the Lambda Expressions. We were interested to see in what kind of expression or statement lambda expressions are used in the code. This can give us some insight into *why* lambdas are used, which will be further complemented by the qualitative study. Table 2 depicts the distribution of the type of the parent AST node of the lambda expressions.

Notice that most of the lambdas (89.94% accumulated) are passed as arguments to method invocations, class instance creations and super constructor invocations. The advantage of using lambdas at these locations is *behavior parameterization*, an important technique to eliminate duplicated code fragments having some behavioral differences. A recent study investigated the applicability of lambda expressions on a large dataset of Type-2 clones (i.e., structurally/syntactically identical code fragments with variations in identifier names, literal values, and types) and Type-3 clones (i.e., copied fragments with statements changed, added or removed in addition to Type-2 differences), and found that 58% of them could be merged and parameterized (i.e., refactored) only by using lambda expressions [Tsantalis et al. 2017]. Moreover, behavior parameterization was more applicable for the clones located in test code than production code (72% vs. 51% applicability).

Therefore, we studied the location of lambdas in the source code. We found that 60% of the lambdas are used in test code as opposed to production code. Figure 3 shows the distributions of the density of lambdas, as measured by the number of lambdas per file, corresponding to the

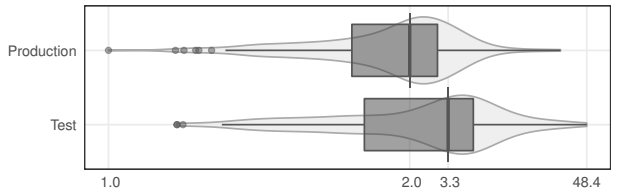


Fig. 3. Density of lambdas in production and test files

production code and test code of the examined projects, excluding the files not containing lambdas. The violin plots show that the density of lambdas in the test code is significantly higher than the production code. To assess if there is a statistically significant difference, we applied the Wilcoxon Signed-Rank test on the paired samples of lambda density in production and test code for each project. The test rejected the null hypothesis that the density of lambdas in production code is more than test code at the significance level of 5% (p-value = 1.835×10^{-10}).

We used the *Hodges-Lehman estimator* to quantify the difference between the density of lambda expressions in the test and production files, as it is appropriate to be used with the Wilcoxon Singed-Rank test. The value turned out to be 1.15, which is equal to the estimated median of the difference between the density of lambdas in a sample taken from the test files and a sample from the production files.

RQ1 Conclusions: Developers use built-in functional interfaces inefficiently. Java developers are forced to create custom functional interfaces for different reasons, e.g., where throwing checked exceptions or extending other functional interfaces is needed. However, developers do not use techniques from functional programming, like *currying*, to avoid creating custom functional interfaces. The behavior of test code is more parameterized with lambdas than production code.

4.2 RQ2: *When* Do Java Developers Introduce Lambda Expressions?

Is there a *trend* in the adoption of lambda expressions in Java projects? As lambdas are new in Java, it is expected that developers gradually start to take advantage of them. To examine this hypothesis, we analyzed the historical data collected during the *mining* study (explained in Section 3.2). More specifically, we computed the ratio of the number of lambda expressions introduced per added line of code in the project. The normalization is necessary, as it eliminates the adverse chance of observing increasing (or decreasing) trend due to the increase (or decrease) of the amount of code committed to the repository, for example if the project becomes more active (or inactive).

We found out that, across all the projects, the ratio of lambda expressions introduced per line of code in 2015 was increased by 204% in 2016. To conduct a more accurate and fine-grained trend analysis, we split the data of each project into monthly intervals, and applied a statistical trend test on it. Particularly, we used Mann-Kendal trend test [Mann 1945]; a non-parametric statistical test that examines the null-hypothesis that "there is no trend in the data". Obtaining small p-values (e.g., smaller than $\alpha = 0.05$) will lead to rejecting the null-hypothesis (i.e., there is a trend in the data). In this case, the test can also show the degree of the monotonicity of trend (i.e., the τ value). The τ value is ranging between $-1 < \tau < 1$, where -1 shows a perfectly decreasing trend, and 1 shows a perfectly increasing trend. While there are other ways to assess trends in the data (e.g., regression analysis), the Mann-Kendall test makes the fewest assumptions about the underlying data, making it suitable for our analysis. Indeed, this test has been used in previous software engineering studies, e.g., for examining the applicability of Lehman's Laws of Software Evolution [Amanatidis and Chatzigeorgiou 2016].

Running the test on the dataset revealed that, for 121 projects (i.e., 50.20% of all the projects which adopted lambdas), there was actually a trend in using lambda expressions (i.e., p-value < 0.05). In other projects, the null hypothesis is not rejected due to large p-value and no conclusion can be made for them.

For the projects where there was a trend, we investigated the values of τ, in order to see whether the trend is increasing or decreasing. As space constraints make depicting the results for the whole dataset impossible, we opted for aggregating the results and showing the distribution of the τ values in Figure 4. As it is observed, the median of τ in the projects having a significant trend in adopting lambdas is 0.39 (i.e., a moderate, positive trend). Except for only four projects (namely, `RxAndroidBle`, `java8-tutorial`, `speedment`, and `cyclops-react`), which are actually shown as outliers in Figure 4, the τ values were positive, with `intellij-community` having the strongest positive trend ($\tau = 0.78$).

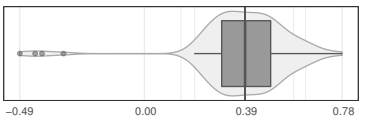


Fig. 4. Lambda adoption trend as measured by τ value computed with the Mann-Kendal trend test

In addition, we used the *Sen's estimator* [Sen 1968] to compute the *slope of the trendline* for the number of lambdas introduced per added line of code in the history of the whole dataset. This estimator is basically the median slope of the lines drawn between all the pairs of points, which turned out to be 2.06×10^{-05}, which essentially shows the amount by which the ratio of the lambdas introduced per added line is increased in each month within the analyzed period.

We then contacted the developers of the mentioned four projects where the trend of using lambda expressions was negative, to find possible reasons for this observation. Developers of three projects (namely, `RxAndroidBle`, `speedment`, and `cyclops-react`) responded to our query, but the developer of `speedment` could not pinpoint a specific reason for the observed negative trend. On the other hand, the developer of the project `cyclops-react` mentioned that *"The project simply being refactored, so, many lambda expressions are part of the code that has been just extracted and placed in another repository"*. Moreover, the developer of the project `RxAndroidBle` mentioned two possible reasons for the negative trend in lambda adoption: *"(1)* `Retrolambda` *(which we have used) is incompatible with* `Jack` *that was meant to be the toolchain for Android and (2) Lambdas create more methods in the* `.dex` *file than anonymous classes"*.

Interestingly, one of the developers of the project `cassandra` with fairly strong positive trend ($\tau = 0.58$) mentioned that *"...Unfortunately, we quickly realized that Streams and Lambdas were pretty bad from a performance point of view. Due to this fact, we stopped using them in hot path."*.

Note that, in general, there is no consensus on the real impact of using lambdas on the performance of Java 8 streams. Some experiments comparing the performance of lambdas over their counterpart implementations [Kuksenko 2013; Zhitnitsky 2015] show that traditional Java programming style with `Iterators` and enhanced `for` loops significantly outperforms new implementations made available by Java 8. Other experiments found that streams perform slightly better when the size of the collections is large, but for small collections, traditional `for` loops significantly outperform streams [Mazur 2017]. Interestingly, according to Mazur [2017], irrespective of the size of the collections, parallel streams are much more efficient than traditional for loops. Recent studies have discussed the problems associated with the state-of-the-art approaches used in performance analysis experiments. For example, Barrett et al. [2017] recently showed that there are a number of problems in the approach used by the current VM benchmarking methodologies, as many benchmarks fail

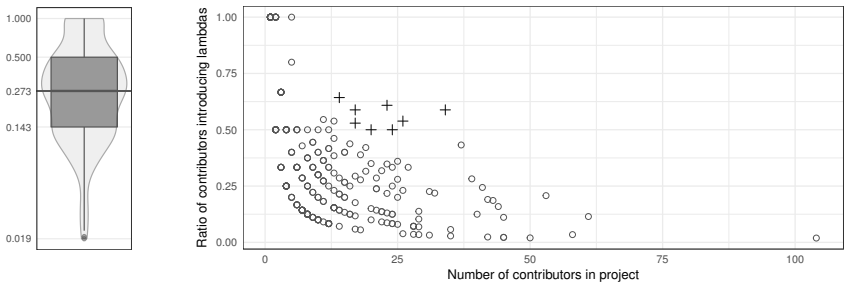
to reach a steady state of peak performance, and thus might draw unreliable conclusions. As a result, these experiments are not necessarily conclusive, and more research is needed to study the performance of lambdas used with streams in different scenarios.

Nevertheless, our qualitative study showed that developers seem to be unaware of these performance implications, since very few of the developers we surveyed mentioned it. Developers should become more aware of these performance implications, and make better-informed implementation decisions, especially for performance-critical parts of their software.

RQ2 Conclusions: The adoption of lambda expressions in Java programs has a increasing trend among several of the top projects in the open source community.

4.3 RQ3: *Who* Introduces Lambda Expressions?

4.3.1 Ratio of Developers Who Introduce Lambda Expressions. In Figure 5a, we illustrate the distribution of the percentage of developers who introduced lambdas in the code over all projects. Overall, we noticed that the median percentage of developers who introduced at least one lambda expression into the code base of the examined projects is 27%, and in half of the projects the percentage is ranging between 14% and 50%. Note that, we only considered the authors that pushed a commit after the first lambda expression was introduced into the code base of each project. Moreover, we found that in 31 projects having 1 to 5 developers, all developers introduced at least one lambda expressions in the code. In Figure 5b, we depict the ratio of developers introducing lambdas over the total number of contributors in each project. We observe a power-law distribution, meaning that in projects with smaller number of contributors the majority introduces lambdas, while in projects with larger number of contributors the minority introduces lambdas. However, we can see that in some projects with a relatively large number of developers 50% to 64% of them contribute lambdas, as highlighted in Figure 5b. These projects enhance functional programming features of Java (`javaslang`, `functionaljava`, `jOOL`, `retrolambda` and `gs-collections`), provide asynchronous libraries (`armeria`) and testing frameworks (`junit5`).



(a) Ratio distribution (b) Scatter plot of ratio over team size

Fig. 5. Ratio of developers introducing lambdas

4.3.2 Outsiders Versus Core Developers. A common practice in open-source software development, which is actually one of the most important strengths of it [Raymond 2001], is the contribution of volunteers who are outside the circle of core developers, i.e., the *outsiders*. Outsiders usually submit a solution to a bug, or an implementation of a new feature to the core developers, in the form of a *pull request*. The submitted code is then reviewed by a core developer, and is accepted to be merged, if it meets the quality standards of the project.

When we found that three of the surveyed developers, who happened to be outsiders in the projects they contributed to, mentioned that they introduced lambda expressions merely because

the code they were contributing to was already using lambdas, we were motivated to see whether there is a significant difference between the use of lambda expressions among the core developers and outsiders. Intuitively, to maintain a consistent code base, there should not be any significant difference in the usage of lambdas between outsiders and core developers. To investigate this, we first compared the ratio of lambda expression per line of code introduced by core developers and outsiders. The normalization eliminates the effect of the amount of code contributed by the two groups.

An eyeball analysis of the results, shown in Figure 6, shows that core developers tend to create more lambdas per line of added code. This is also statistically supported by employing the Mann-Whitney U test, comparing the ratios between the two groups at the significance level of 5%. The test accepted the alternative hypothesis that *outsiders introduce less lambda expressions per lines of code* with p-value $< 2.2 \times 10^{-16}$. We also tried to quantify the difference between the two groups. Again, the Hodges-Lehman estimator is appropriate to be used along with the Mann-Whitney U test, to estimate the difference between the medians of samples drawn from the two groups under analysis. For our analysis, this value turned out to be 0.007. In other words, if we draw samples from the two groups, we can estimate that per 1000 lines of code, core developers introduce 7 lambdas more than outsiders.

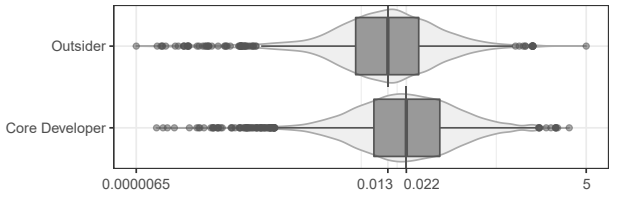


Fig. 6. Comparison of lambda ratio per lines of code added

4.3.3 Lambda Enthusiasts. We also tried to see whether there are *lambda enthusiasts*, i.e., developers who adopt lambdas significantly more than other developers, or whether everyone in the project contributes equally to the introduction of lambdas. Knowing who are the lambda enthusiasts can be useful in many cases. For instance, the code introduced by these developers should probably be reviewed by the other (possibly, core) developers to spot possible misuses of lambda expressions (e.g., using slow `Stream` API calls with lambdas in performance-sensitive parts of the code).

We extracted the total number of lambda expressions introduced by every developer in each project, normalized by the total number of lines of code added by that developer, in the entire history of subject systems. We then tried to identify the *outliers* in these numbers, i.e., the developers who introduced considerably higher number of lambdas per line of code, compared to the rest of the developers in the same project. Outliers in data are essentially considered as *abnormal* values, and their existence may reveal important information about the phenomenon being measured. For instance, previous studies used outliers of quality metric values, extracted from box plots of data, to detect design flaws in software systems [Marinescu 2004]. In this approach, values which are greater than the *upper inner fence* of the box plot, defined as $Q_3 + 1.5 \times (Q_3 - Q_1)$, are considered as outliers, where Q_1 and Q_3 are the first and third quartiles of data.

This analysis found lambda enthusiasts in 83 projects (i.e., 34.44% of projects that used lambda expressions). In total, there were 158 lambda enthusiasts out of the 1,503 lambda-introducing developers (10.51%). In Figure 7, for the 83 projects in which we found lambda enthusiasts, we depict the percentage distribution for the lambda enthusiasts over all the lambda-introducing developers. Notice that, across these projects, the median percentage for the lambda-introducing developers who are also lambda enthusiasts is 14.3%.

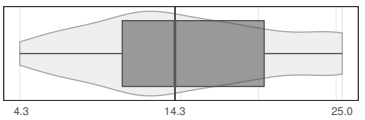


Fig. 7. Percentage of lambda enthusiasts versus all the lambda-introducing developers

In other projects, developers who introduced lambdas did not differ significantly, with respect to the ratio of added lambdas per line of code added. Moreover, it turned out that 39% of lambda enthusiasts were outsiders. As mentioned, it is necessary to pay more attention when reviewing the code by these outsiders, as they might be misusing lambda expressions in certain parts of the code.

RQ3 Conclusions: The median percentage of developers who introduce lambdas in the studied projects is 27%. This accounts for both core developers and outsiders, while core developers introduce more lambdas than outsiders. Moreover, in 34.44% of the projects, lambda enthusiasts were found, i.e., developers who contribute significantly more lambdas than other developers.

4.4 RQ4: *How* Do Java Developers Introduce Lambda Expressions?

4.4.1 Migration to Lambdas. A lambda expression can be introduced within the new code, or by refactoring existing code to take advantage of lambdas. Across all the lambdas introduced in the history of the subject systems, we found that 26% were introduced in existing methods. We hypothesized that developers might apply massive number of refactorings to *migrate* the existing code base to take advantage of lambda expressions.

We can identify migration activities by investigating the history of the subject systems, looking for the commits with unusually large number of lambdas introduced within existing code, i.e., the *outlier commits*. The outliers can be computed similar to RQ3, using the box plots of the values. We observed that in 51 projects (i.e., 21.16%), there were such outlier commits. To find evidence of lambda migration efforts, for each of the 51 projects we selected the outlier commit with the largest number of introduced lambdas for further manual investigation.

We found the highest number of introduced lambdas in a single commit in the `intellij-community` project, with more than 5000 lambdas. Inspecting the code and the commit message showed that the developer refactored all the usages of anonymous classes to lambdas. We suspect that such a massive introduction of lambdas was done with an automated refactoring tool. We found similar anonymous class to lambda refactorings in 8 other projects. In the remaining projects, we found several different kinds of refactoring activities, which we summarize in Table 3.

We found that it is quite common to wrap existing code into lambdas. In most of these cases, lambdas were mostly responsible for deferring the evaluation of the code being enclosed in a lambda to a certain time, e.g., for handling the thrown exceptions, or checking some preconditions before the behavior is executed. For instance, consider the code in Figure 8a, from project `speedment`. The `finallyClose()` method was supposed to make possible the reuse of the behavior of closing a stream, whether an exception is thrown or not by an expression, in several places of the code base (one of which being illustrated). However, this method would not work as it was expected, since the exception would be always thrown outside the `finallyClose()` method, when the passed argument is evaluated. Indeed, this was reported as a bug in the project's issue tracker. In one commit, the developer attempted to fix the bug, by changing the code to what is depicted in Figure 8b. Here, the method `finallyClose()` accepts a `Supplier` instead of a normal parameter, so that the execution of the behavior is moved into the method (i.e., when the `get()` method is

Table 3. Kinds of code migration to use lambdas

#Projects[1]	Kind	#Introduced lambdas	Description
14	New lambda wraps the existing behavior	510	The behavior existed already in the code, however, it is now enclosed in a new lambda which is passed to another method for specific treatment, including lazy evaluation, or special handling of the thrown exceptions
9	Anonymous class to lambda refactoring	5855	All the uses of anonymous classes are replaced with lambdas
6	Replacement of loops and conditionals with constructs that employ lambdas	433	Existing code using `for`, Iterators or the enhanced `for` is converted to use the `forEach()` method calls or the `Stream` API, and conditionals with `ifPresent()` and `ifAbsent()` calls
2	Clone refactoring using lambdas	22	Clones are extracted into a method, where the varying behavior across the clone instances is parameterized using lambdas
2	Method reference changed to lambda	49	Method references[2]are replaced with `Suppliers`
1	Existing method was changed to accept lambdas	17	A new parameter is introduced to parameterize parts of an existing method
1	Method accepting a lambda is inlined	19	A method that accepted a lambda is inlined; the clients can pass new behavior (through a lambda) instead of the hard-coded lambda in the previous method
12	No refactorings	1717[3]	The newly-added lambdas are in the same methods, but they add extra functionality to the existing code
4	False Positives	169	The tool incorrectly reported newly added lambdas, because the body of existing lambdas changed in the commit

[1] Note that the commits with the largest number of introduced lambdas per project are considered for this analysis.
[2] Method references in Java are in the form of `Type::method`, and can be used wherever a lambda of type `Supplier` with the same signature can be used.
[3] In one case (project `javaslang`), 1464 lambdas were found to be introduced in one commit. The commit basically included uncommenting a large number of lines of code in one file, which happened to contain all the lambdas. Consequently, they were marked as new lambdas.

```
protected <T> T finallyClose(T t) {
   try {
      return t;
   } finally {
      close();
   }
}

return finallyClose(streamTerminator.average(pipeline()));
```

(a) Before

```
protected <T> T finallyClose(Supplier<T> s) {
   try {
      return s.get();
   } finally {
      close();
   }
}

return finallyClose(() -> streamTerminator.average(pipeline()));
```

(b) After

Fig. 8. Bug fix by wrapping behavior into a lambda (specifically, a `Supplier`)

called inside the `try` block). The developer needed to wrap all the previous arguments into lambda expressions, leading to a massive introduction of lambda expressions.

In addition, lambdas are useful to refactor instances of duplicated code that require the same logic enclosing varying behavior [Tsantalis et al. 2017]. Interestingly, we observed such refactoring efforts in two projects, one of which is illustrated in Figure 9. The refactoring in this commit included extracting the duplicated code from the body of the methods `notifyInterfaceLinkStateChanged()` and `notifyInterfaceAdded()`, into a new method `invokeForAllObservers()`. Notice that there were several instances of the same logic, but we are showing only a pair of those, due to space constraints. The unified method uses a functional interface, namely `NetworkManagementEventCallback`, so that the methods that previously contained the cloned instances can pass the behavior corresponding to the statements which are different across the two clone instances.

4.4.2 Use of Automated Tools for Introducing Lambdas. As mentioned before, lambda expressions in Java can be replaced with anonymous inner classes and vice versa. There is quite mature tool and IDE support for refactoring existing anonymous classes to lambda expressions. Some IDEs (e.g., Eclipse) even provide a feature for automatically converting all anonymous classes to lambda

```
private void notifyInterfaceLinkStateChanged(String iface, boolean up) {
  final int length = mObservers.beginBroadcast();
  try {
    for (int i = 0; i < length; i++) {
      try {
        mObservers.getBroadcastItem(i).interfaceLinkStateChanged(iface, up);
      } catch (RemoteException | RuntimeException e) {}
    }
  } finally {
    mObservers.finishBroadcast();
  }
}
private void notifyInterfaceAdded(String iface) {
  final int length = mObservers.beginBroadcast();
  try {
    for (int i = 0; i < length; i++) {
      try {
        mObservers.getBroadcastItem(i).interfaceAdded(iface);
      } catch (RemoteException | RuntimeException e) {}
    }
  } finally {
    mObservers.finishBroadcast();
  }
}
```

(a) Before

```
private void invokeForAllObservers(NetworkManagementEventCallback ec){
  final int length = mObservers.beginBroadcast();
  try {
    for (int i = 0; i < length; i++) {
      try {
        ec.sendCallback(mObservers.getBroadcastItem(i));
      } catch (RemoteException | RuntimeException e) {}
    }
  } finally {
    mObservers.finishBroadcast();
  }
}
private void notifyInterfaceLinkStateChanged(String iface, boolean up){
  invokeForAllObservers(o -> o.interfaceLinkStateChanged(iface, up));
}
private void notifyInterfaceAdded(String iface) {
  invokeForAllObservers(o -> o.interfaceAdded(iface));
}
@FunctionalInterface
private interface NetworkManagementEventCallback {
  public void sendCallback(INetworkManagementEventObserver o)
    throws RemoteException;
}
```

(a) After

Fig. 9. Clone refactoring using lambda expressions

expressions when saving or building a Java project. Investigating how developers usually introduce lambdas (i.e., manually, or using automatic tools and IDE support) is very interesting, as it may reveal, for instance, the challenges that developers face when manually introducing lambdas (e.g., because the developer might not know the types or number of the parameters of the lambda being introduced), or the shortcomings associated with existing tool support.

To find this out, we relied on the answers collected in the conducted qualitative study. We asked the developers who created lambdas during the study period what IDE they are using to understand what automated refactoring tools that were available to them, when they introduced a lambda expression, and whether they used any automated refactoring tool or quick assists for creating lambda expressions.

The results (Figure 10) show that most of the developers (59.6%) that we contacted used IntelliJ IDEA. We noticed that none of the developers using NetBeans or Eclipse took advantage of the automated refactoring/assistance offered by these IDEs. On the other hand, 11% of developers using IntelliJ IDEA and 50% using Android Studio, which is an extension of IntelliJ IDEA tailored for developing Android mobile applications, used the IDE tool support for introducing lambda expressions. A similar study performed by Silva et al. [2016] investigating the refactoring operations applied in GitHub projects, showed that IntelliJ IDEA was the most popular IDE and had the highest percentage of automated refactoring applications (around 70%). Both findings confirm that IntelliJ IDEA is trusted more than the other IDEs by the developers for refactoring their code.

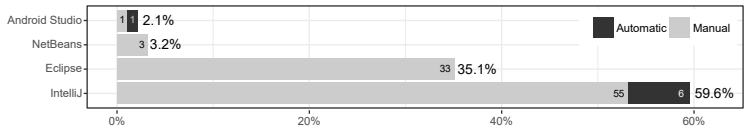


Fig. 10. IDEs used by the surveyed developers and degree of automation

Overall, we found that only 11% of the lambdas were introduced automatically, regardless of the used IDE. This finding is in agreement with previous studies that investigated the use of automated tool support for refactoring. Murphy-Hill et al. [2009] found that only 10% of developers using the Eclipse IDE took advantage of automated tool support for refactoring. Silva et al. [2016] conducted a study involving developers using the same four IDEs as in our study, and found that only 38% of the developers performed refactorings in a fully automated manner.

Despite the relatively low percentage of automatically introduced lambdas, 20% of the developers mentioned that they also consider using automatic transformations provided by the IDE and apply them if the suggestion fits their needs, or it improves the readability of the code. In particular, one developer mentioned that *"... sometimes tools suggest lambda reduction which is not a real improvement in code clarity or comprehension"*. This definitely calls for improving the code transformations applied by existing lambda refactoring tools.

RQ4 Conclusions: It is quite common practice to migrate existing code in Java projects to take advantage of lambdas. Regarding newly introduced lambdas in a project, Java developers tend to write them manually, as the lambda code generated by tools might not fully accommodate the developers' needs and expectations.

4.5 RQ5: *Why* Do Java Developers Introduce Lambda Expressions?

We relied on the results of the conducted qualitative study explained in Section 3, in order to identify the underlying reasons for introducing lambda expressions in Java projects. We have summarized the results of the survey in Table 4. It is worth mentioning that, in some cases, there were more than one theme emerging from an email correspondence, for a specific lambda expression. For instance, there were several cases where a developer mentioned the *goal* of using a lambda (e.g., *implementing a listener*), in addition to *why* a lambda was used instead of an alternative solution (e.g., *the code with lambda is considered to be more readable*).

Table 4. Reasons for introducing lambdas in code, as reported by the surveyed participants

Theme	Description	Frequency
Terseness of lambdas	Lambda expressions are considered more readable, more concise, and easier to write than the equivalent code using, e.g., anonymous inner classes	57
Avoiding the implementation of new classes	Lambdas are used mainly to avoid introducing any kind of class, including normal or nested (static or non-static) classes	27
Behavior parameterization and reusability	Behavior is implemented using a lambda so that it can be reused, or can be passed as the varying step of an algorithm which is frequently used, to avoid duplicated code (i.e., clones)	21
API convention enforcement	Methods in API accept functional interfaces (e.g., `Suppliers` and `Consumers`), or the API documentation persuades using lambda expressions.	18
Migration To Java 8	The existing code is migrated, or new code introduced in a way that the new Java 8 features with lambdas can be used, so that an obsolete dependency can be removed, or to allow clients to have the freedom to invoke API methods using lambdas	8
Implementing listeners and callbacks	Listeners and callbacks are implemented using lambda expressions	7
Lazy evaluation	Lambda expressions are used to deffer evaluating an expression	7
Maintaining consistency	Lambda expressions are introduced (or copy-and-pasted) to maintain the consistency of the existing code that already used lambda expressions	7
Object-oriented design patterns	Specific patterns (e.g., Adapter, Visitor, Decorator, Façade) are implemented using lambdas	6
Performance	Lambdas are used in the places where their performance was superior compared to the alternative solutions (e.g., anonymous classes)	6
Simplify testing	Lambdas are introduced to unify and reuse repetitive testing logic	5
Favor composition over inheritance	Lambdas are used as instance variables, or passed to class constructors, and are used instead of constructing an inheritance hierarchy to achieve polymorphic behavior	2
Multi-threading	The behavior is wrapped into a lambda and passed to specific methods that take care of dispatching the behavior to a certain thread, or handle race conditions	2
Exception handling	The behavior is wrapped into a lambda and passed to a specific method that handles checked exceptions, and if necessary, convert the exception to unchecked (so that it doesn't need to be handled)	1

In the following subsections, we provide a detailed explanation for the themes along with real examples from source code and quotes from the developers.

4.5.1 Terseness of Lambdas. In most of the cases developers mentioned that they favored lambda expressions over the possible equivalent implementations, because lambda expressions are much shorter. For example, a lambda expression can be alternatively implemented using an anonymous class (as mentioned in Section 2). An anonymous class implementation would need 1) the `new` operator, 2) the name of the Functional Interface being implemented followed by parentheses, and its body which should be distinguished with curly brackets, 3) the full declaration of the method being overridden, including its access modifier, return type, method name, type and name of each of the formal parameters enclosed by parentheses, and the body of the method. The method can also be optionally preceded with the `@Override` annotation (which is usually inserted when the developer applies a suggestion from the IDE). A lambda, on the other hand, is much shorter, as it only needs the name of the parameters, parentheses (can be omitted if there is only one parameter), the arrow token (i.e., `->`), and the lambda body.

In contrast to the majority of developers, two developers mentioned that this terseness actually hampers readability. Particularly, one developer stated that *"I also like to experiment with the expressions and see if the code flows better or is easier to read. Often it does not"*. The other developer mentioned that *"Although anonymous classes are more descriptive, lambdas make code shorter and therefore easier to read and understand as a whole"*. We further asked these two developers to elaborate their answers. The first developer replied that *"For someone not used to lambdas, this [i.e., the lambda] syntax here takes a while to wrap your head around"*, pointing to this lambda expression from their source code: `(config, mode) -> config`. The second one stated that *"Lambda in Java doesn't have a name unless you store it in a variable. Both return type definition and parameter types are inferred. Because of that the purpose of a lambda can only be obvious or totally unknown"*. Moreover, the developer provided an example of such a case, which we show in Figure 11.

```
RowArrayAdapter adapter2 =
   new RowArrayAdapter(HeaderItem.class, new RowFactory() {
      @Override
      @NonNull
      public Row create(ViewGroup __) {
         return null;
      }
});
```

(a) Anonymous class implementation

```
RowArrayAdapter adapter =
   new RowArrayAdapter(HeaderItem.class, __ -> null);
```

(b) Lambda expression implementation

Fig. 11. Lambda terseness hampers maintainability

The developer adds *"Both examples are obviously wrong, because* `RowFactory` *cannot return a null value. It's impossible to tell that by just looking at the lambda. The parameter is also a bit mysterious - I'm not sure what I can do with it without using autocompletion or documentation. Anonymous class is longer, but has all the information on the spot, despite of using meaningless parameter name"*.

4.5.2 Behavior Parameterization and Reusability. A method can accept lambda expressions to parameterize behavior. 21 developers mentioned that they used lambdas to avoid introducing *boilerplate code* (i.e. small pieces of copy-and-pasted code with few differences in statements), or to pass a small behavior (e.g., a boolean decision based on a given set of inputs), which is not available beforehand and should be passed from the client to other parts of the code. Previous work has looked into using lambda expressions to refactor existing duplicated code, where behavior parameterization is needed [Tsantalis et al. 2017].

Notice that in "behavior parameterization" lambdas inject variations of behavior and promote code reusability, which can in turn avoid boilerplate code. On the other hand, "terseness" (which was discussed in the previous subsection), happens when lambdas replace longer syntactic constructs

with shorter alternatives that have the same semantics (i.e., reusability is not the main concern in such cases).

4.5.3 Lazy Evaluation. One of the most important characteristics of lambda expressions is that they are evaluated only when it is required. This makes lambda expressions suitable when, for instance, an algorithm needs data which might not be available at the time or location that it is defined, or its computation is costly so it should be done lazily. For instance, in project `junit5`, there was a method responsible for creating a specific exception object, when a predicate was evaluated to false (Figure 12).

```
public static void condition(boolean predicate,
      Supplier<String> messageSupplier)
         throws PreconditionViolationException {
   if (!predicate) {
      throw new PreconditionViolationException(messageSupplier.get());
   }
}
```

(a) Using Supplier to evaluate behavior when necessary

```
Preconditions.condition(allSet.containsAll(names),
    () -> "Invalid enum constant name(s) found in annotation: " +
       enumSource + ". Valid names include: " + allSet
);
```

(b) Providing a Supplier using a lambda

Fig. 12. Lazy evaluation using a Supplier

The method `condition()` in Figure 12a uses a built-in Functional Interface (i.e., a `Supplier<String>`), which does not need a parameter and returns a `String`. If the value of `predicate` is `false`, the provided behavior through the lambda expression (e.g., the string concatenation in Figure 12b) is evaluated. If, instead of the `Supplier<String>`, a normal `String` was used, the string concatenation (or any expensive computation) would be evaluated even if the value of the parameter `predicate` was `true` and the evaluation was unnecessary.

4.5.4 Object-Oriented Design Patterns. Some developers mentioned the use of lambdas when implementing object-oriented design patterns, e.g., Visitor, Decorator, Adapter, and Façade [Gamma et al. 1994]. For instance, the lambda expression in Figure 13 essentially acts as a Façade, as the developer writes *"In this case we have a fairly big class that is the root of a bunch of code that manages the state of the cluster. Some other bits of code need to know how to get certain bits of the cluster state like the node's id but they have not business mucking around with the rest of the cluster state. Lambdas offer a very simple way to do that"*. Here, the lambda expression is of type `Supplier<Integer>`, which hides all the methods of the `DiscoveryNode` class (returned by the `localNode()` method) and delegates the behavior of retrieving the ID of the node to the `DiscoveryNode#getId()` method.

```
client.initialize(injector.getInstance(new Key<Map<GenericAction, TransportAction>>() {}),
                  () -> clusterService.localNode().getId());
```

Fig. 13. Use of a Façade in project `Elasticsearch`

4.5.5 Avoiding the Implementation of New Classes. Developers mentioned various reasons for avoiding using new classes in the code, being either a normal class, or a nested class. For instance, a developer mentioned that *"... the reference to* `this` *is clearer than it is with the expanded version."* By expanded version, the developer refers to an anonymous class. Note that, anonymous classes in Java are not static, i.e., they have access to the fields and methods of the enclosing class they are defined in. When a field of the enclosing class is required inside the anonymous class, one should refer to it using `EnclosingClass.this.field` expression, since the reference to `this` will be bound to the anonymous class, rather than the enclosing class. However, inside a lambda expression, the `this` reference can be referred without ambiguity to the enclosing class. Another developer mentioned that *"In general, I tend to introduce lambdas because they allow me to keep small methods in the same*

place as the code that's calling them" which is unavoidable, if a new normal class is used instead of a lambda expression.

4.5.6 API Convention Enforcement. 18 developers mentioned that they introduced lambdas, because they wanted to follow the API's documented conventions, or the API accepted built-in Functional Interfaces, where using lambda expressions are *"idiomatic"* or *"the most straightforward solution"*.

4.5.7 Migration to Java 8. For eight developers, introducing lambda expressions was solely due to the fact that they wanted to migrate their code base to take advantage of lambda expressions in new Java 8 API methods, e.g., the streams, collections, and optionals. Among them, two developers mentioned that the methods that accepted lambda expressions bring *"client convenience"* and *"flexibility"*, as the clients can use lambda expressions, instead of being forced to use alternative solutions which were available before Java 8.

4.5.8 Implementing Listeners and Callbacks. Specifically for listeners and callbacks, using lambdas seemed to be *natural* by several developers. For instance, one developer stated that *"Vert.x programming model uses callbacks extensively so lambdas fit very well in the code"*. An older style in implementing listeners is to make the existing outer class (i.e., the class that contains the object of which the event is handled) implement the listener interface, and pass the `this` reference to the corresponding method that adds the listener (usually named `addListener()`). One developer mentioned that this approach *"exposes the listener method in the public API of the outer class (and requires additional jumping around in the code to learn about all relevant pieces)"*, i.e., the *API bloat*. The other alternatives, i.e., implementing a new class, or using anonymous classes, would suffer from the issues mentioned in Section 4.5.5, e.g., boilerplate code. In particular, one developer mentioned that *"With lambdas, this trade-off between boilerplate and API bloat is no longer relevant since a lambda expression allows doing the same thing without exposing any public API and with less boilerplate than either of the alternatives available in Java 7."*.

4.5.9 Maintaining Consistency. The existence of lambda expressions in a piece of code can make developers to prefer using lambda expressions when contributing to the same code. For instance, we received responses from developers like *"I was just trying to unify my test code with other code in the test suite which started using this pattern earlier, so it was just refactoring to make the code more unified"*, or *"I was fixing a bug in the project affecting a component that was contributed by another group here; there were a couple of other lambdas in that component already, so I thought it would have been nicer to keep on using them instead of anonymous classes"*.

4.5.10 Performance. As mentioned, the developers of project `cassandra` stopped using lambda expressions with methods of the `Stream` API, due to their inferior performance compared to traditional loops. Also, we discussed that using lambdas with generic Functional Interfaces (e.g., `Function<Integer, Double>`) instead of specialized primitive ones (e.g., `IntToDoubleFuntion`) can lead to deteriorating the performance, due to the need for auto-boxing and unboxing.

On the other hand, several developers mentioned that they actually used lambdas because they have better performance than their alternatives: *"[lambdas] read much easier than anonymous classes and their performance is considered better"*. Indeed, it turns out that, depending on where and how lambda expressions are used, they can improve or degrade performance [Oaks 2014]. For instance, if an anonymous class is declared inside a loop, an object of that class is instantiated to and removed from the memory at each iteration; however, lambda expressions don't have this overhead.

4.5.11 Simplifying Testing. A developer of the project `Hibernate` writes *"I introduced the Lambda expression to simplify testing"*. Often, frameworks provide special testing APIs so that the clients can develop unit tests for the parts of their code that depend on the framework's core API. For example, when developing unit tests, the clients of `Hibernate` need to call certain methods in a specific order, e.g., to start and rollback database transactions. The test code (i.e., data manipulation and test assertions) must appear within this chain of calls to work properly. Developers should know the exact order in which these methods should be called, and also they should probably repeat the same logic for calling those methods in their test suits. `Hibernate` provides a testing API to solve this problem. The methods of the test API hide the chain of methods that should be called from the clients, and allow them to pass their test code using lambda expressions. This makes the test code much shorter for the clients, and the correct order of API calls is always guaranteed.

4.5.12 Favoring Composition over Inheritance. Favoring object composition over class inheritance is a recommended practice in object-oriented paradigm [Bloch 2008; Gamma et al. 1994], where code reuse is achieved by calling methods of an object (often defined as an instance variable), instead of extending the class that contains the code. The reused code might also be defined in (or, using the constructor of the class, passed to) the class that needs it, in the form of a lambda expression: *"So [in the* `pac4j` *project], the clients are composed of a few components, each component does one thing and thus only has one method (and in that case, it can be modeled by a lambda). When these components have a complicated algorithm and some parameters in the constructor, I designed them as classes, but for small logic components, I prefer to use lambdas"*. These lambdas effectively replace polymorphic calls, which would be necessary if inheritance was used.

4.5.13 Multi-Threading. It is often required that a thread-unsafe behavior is passed to a method that handles synchronization issues, to avoid race conditions and deadlocks. In this case, the behavior can be wrapped as a lambda expression to be passed. Particularly, such cases are found in applications having GUI, for instance: *"[passing the lambda] is a trick that is common in JavaFX to make sure the code in the lambda is executed on the JavaFX thread. In this case, I introduced it to solve a bug where the method in question was invoked from multiple threads causing an exception in rare cases."*.

4.5.14 Exception Handling. We found out that in one case, lambda expressions were introduced for wrapping and passing a behavior that threw an exception, to a method that was responsible for handling the exception. In that case, the thrown exception was converted to a `RuntimeException`: *"...[the behavior] is wrapped in an Exception Softener provided by* `cyclops-react` *in which Checked Exceptions get converted into Unchecked exceptions"*. The softener method is shown in Figure 14. As it is observed, the method accepts a specific `Supplier` that throws a checked exception, and returns another `Supplier` (represented as a lambda expression) that invokes the input lambda and, in case of an exception, softens it using the `throwSoftenedException()` method. The reason that the method is returning a lambda is to send the control of executing the exception-throwing input lambda back to the caller.

```
public static <T> Supplier<T> softenSupplier(final CheckedSupplier<T> s) {
   return () -> {
      try {
         return s.get();
      } catch (final Throwable e) {
         throw throwSoftenedException(e);
      }
   };
}
```

Fig. 14. Exception softener that accepts and returns lambda expression

Note that, exception softening has different uses, e.g., if a method should be changed to throw a checked exception while it is called deep in a call hierarchy, where the exception cannot be caught or thrown by the intermediate methods in the hierarchy (e.g., when the code cannot be changed as it is a public API, or when changing all the intermediate methods requires a huge effort), the thrown exception can be softened to a `RuntimeException`, which then can be handled wherever needed.

RQ5 Conclusions: Most of the developers believe that lambda expressions are more readable than equivalent solutions, i.e., anonymous inner classes, and that is the reason they introduce them in the code. While lambdas can be used for several purposes (e.g., to avoid introducing new classes, for behavior parameterization, for code reuse, to enable lazy evaluation of expressions, to implement callbacks, listeners, and design patterns), they might be introduced in code merely to keep the current code base consistent.

5 IMPLICATIONS

With the introduction of lambda expressions and functional interfaces in Java 8, we observed a proliferation of functional style into an imperative object-oriented language. To help developers leverage the power of functional programming, we draw several implications that we group according to the needs of researchers, language designers, tool builders, and software developers.

5.1 Researchers

The findings from RQ5 reveal several novel research opportunities:

R1. The third most frequent reason developers use lambda expressions is to avoid code duplication or eliminate code clones with behavioral differences. A great portion of clones have differences in method calls and object creations [Tsantalis et al. 2015]. Parameterizing such behavioral differences with regular parameters may cause a change in the program behavior, due to *side-effects*. On the other hand, lambda expressions make possible the parameterization of such behavioral differences without changing the execution order of the original duplicated code. Researchers can further explore the effectiveness of lambda expressions when refactoring software clones.

R2. Among the top-10 reasons developers use lambda expressions is to enhance the traditional implementation of object-oriented design patterns. Design patterns rely heavily on inheritance and polymorphism, which in some cases might have some negative impact on the understandability, maintainability, and performance of the software system [Ampatzoglou et al. 2015; Bieman et al. 2003; Khomh and Guéhéneuc 2008; Ng et al. 2007; Noureddine and Rajan 2015; Prechelt et al. 2001; Vokáč et al. 2004]. In other cases, a design pattern might not be applicable due to constraints coming from the overall design of a system. For example, when eliminating code duplication, introducing a Template Method design pattern [Gamma et al. 1994] is not an option, because declaring the common superclass as abstract is not always feasible [Tsantalis et al. 2017]. Therefore, lambda expressions can be used as an alternative solution to polymorphic method calls and abstract classes. Researchers can expand the catalogue of design patterns by documenting pattern implementations that use lambda expressions instead, and designing techniques for the migration of traditional pattern implementations to those based on lambdas.

5.2 Language and API Designers

The findings from RQ1 provide several implications:

L1. In most cases, Java developers prefer to use generic functional interfaces instead of specialized ones for primitive types. This could be due to the lack of awareness of all the available options, or the tediousness of finding the correct type of functional interface among a long list of options. Thus, developers use auto-boxing unnecessarily, which degrades the performance of the software. The Java compiler could perhaps optimize the code by replacing general functional interfaces to more specialized ones at bytecode level.

L2. We found that 42% of the custom functional interfaces implemented by developers contained methods throwing a checked exception. 63% of these functional interfaces meet the input/output requirements of the existing Java built-in functional interfaces. However, they cannot be currently replaced with built-in functional interfaces because none of them supports exception throwing. This finding calls for Java API designers to extend built-in functional interfaces in order to support exception throwing. Currently, some external libraries [TouK 2017] add this missing feature to functional interfaces.

L3. We found that Java projects use currying, a fundamental functional technique, sparingly. Since a developer needs to explicitly specify the type of functional interfaces involved in currying, this becomes very verbose. Language designers can make the syntax currying-friendly by applying type inference techniques [Oracle 2015b].

5.3 Tool Builders and IDE Designers

T1. The under-utilization of the Java built-in specialized functional interfaces supporting primitive types (RQ1), clearly points out the need for tools that can convert generic functional interfaces to more specialized ones, thus avoiding unnecessary auto-boxing.

T2. RQ1 highlights that 10% of the studied lambdas use custom functional interfaces requiring more than 2 parameters. Such cases could be alternatively implemented with currying of existing built-in functional interfaces, thus avoiding the need to create custom interfaces. Automated code assistance and refactoring could suggest appropriate built-in functional interfaces for currying, and convert existing custom functional interfaces to alternative implementations taking advantage of currying.

T3. RQ5 highlights that Java developers mostly use lambdas to make the code more terse and readable. However, in many cases, the use of lambdas eliminates important information from the source code, such as the annotations used in the signature of functional interface methods. In the example shown in Figure 11a, the `@NotNull` annotation, which appears in the signature of method `create(ViewGroup)` informs the developer that the method should not return `null`. However, when a lambda expression is used to implement the same functional interface, the annotation information is lost, as shown in Figure 11b, thus increasing the chance of a bug by a developer who is not aware that this method should not return `null`. IDEs should find ways to convey such missing information to the developers when coding lambda expressions.

T4. In some cases, we saw that the use of lambdas can deteriorate performance (e.g., replacing iterators and for-each loops with streams). IDEs could detect such cases as performance code smells, inform the developer about their implications on performance, and recommend their replacement with a more efficient implementation.

5.4 Software Developers

S1. in RQ1, we found that Java developers seem to be unaware of the performance implications around lambda expressions (e.g., iterators and for-each loops might outperform streams in certain situations), since very few of the developers we surveyed mentioned this issue.

S2. In addition, RQ1 showed that Java developers tend to select more general built-in functional interfaces, thus performing a lot of unnecessary boxing and unboxing operations, which

deteriorate performance. These performance implications should be more extensively discussed in books and tutorials, so that developers can make better-informed implementation decisions, especially for performance-critical parts of their software.

6 THREATS TO VALIDITY

6.1 Internal Validity

The findings of our study highly depend on how accurately we detected the introduction of lambda expressions over the evolution history of the studied projects. We mitigated this threat by extending RefactoringMiner [Silva et al. 2016], so that it reports newly added lambda expressions in existing and new methods. RefactoringMiner detects refactoring operations with a high precision (98%) and recall (93%), by creating a mapping between the methods of a class in the current and its parent revision based on their code similarity. RefactoringMiner considers the remaining unmatched methods in the parent revision as deleted, and the remaining unmatched methods in the current revision as added. Our analysis focuses on the matched (i.e., existing) and newly added methods. Moreover, our use of RefactoringMiner makes our analysis immune to the noise created by refactorings such as extract method, rename, and move method.

To avoid any *bias* when collecting the surveys, when filtering out cases with an evident motivation we sought the agreement of the two first authors of the paper. In addition, we achieved an inter-coder agreement of 80% in assigning codes to the developers' responses, before labeling the entire dataset.

6.2 External Validity

We studied 241 out of the top 2000 projects on GitHub, which met the criterion of having at least one lambda expression in their code base. This accounts for a wide range of application domains, making the results of the study *generalizable* to other projects in similar domains. However, a study of proprietary code-bases might reveal different results.

6.3 Verifiability

We make available online all developed tools, collected data, and R scripts used for statistical analysis [Mazinanian et al. 2017], so that the study is fully *reproducible*.

7 RELATED WORK

We group the related work into three areas: (1) empirical studies on language features, (2) studies bridging the functional and imperative paradigms, and (3) studies and tools targeting lambda expressions.

7.1 Empirical Studies on Language Features

There have been several empirical studies on various language features of Java. Costa et al. [2017] conducted an empirical study on the execution time and memory consumption of the Java Collections Framework and found that alternative implementations provided by third-party libraries greatly decrease memory consumption while offering comparable or even better execution time. Similarly, we found that using custom functional interfaces, either provided by third-party libraries (e.g., jOOλ [Eder 2014]) or defined by developers, provides more freedom to add extra capabilities to the lambda expressions, e.g., exception handling, using annotations, adding default or static methods, or adding static fields, which built-in functional interfaces do not provide.

Bono et al. [2014] propose the *interface-as-trait* approach where *default methods* in Java 8 can be used to improve code modularization. Similarly, we study another Java 8 feature, i.e., lambda

expressions; but instead of discussing *how to use lambda expressions*, we study *how they are used in practice*, by analyzing top GitHub repositories and interviewing the contributors of lambda expressions to get a qualitative and quantitative viewpoint on this newly-added language feature.

Parnin et al. [2011] evaluate several hypotheses, each based on the assertions made by prior researchers, about how and why Java developers use generics. This includes the impact of generics on the source code, who in the project uses generics, the existence of large scale migrations to generics, and IDE support for generics. In this study, we found answers to similar questions relating to lambda expressions. For example, we observed that 89.94% of the lambdas are passed as arguments to method invocations, constructor invocations, class instance creations and super constructor invocations, implying behavior parameterization that results in the reduction of code duplication. Parnin et al. [2011] also found similar results supporting the hypothesis that the introduction of generics reduces duplicated code. Moreover, similar to Parnin et al. [2011] who found that half of the analyzed projects did not use generics, we found that only 241 out of 2000 projects used lambda expressions. However, while Parnin et al. [2011] observed that only one or two champions per project introduce generics, we found that, on average, 27% of the developers in each project introduce lambdas frequently.

In their empirical study of inheritance in Java, Tempero et al. [2013] observed that almost 22% of inheritance instances are used for *external reuse* and not for *subtyping*, and thus, they can be replaced by composition. In our qualitative analysis, we found that one of the reasons why developers are using lambda expressions is that they favor composition over inheritance, since using lambdas facilitate implementing composition.

Dyer et al. [2014] analyzed billions of AST nodes and found that several language features like generics and annotations are less widely used. This is similar to what we found for lambda expressions in Java: only 12.05% of the top 2000 repositories have adopted lambda expressions. The authors also observed that developers convert existing code to utilize new language features after their release. As we discussed, we found similar trend in migration from anonymous classes to lambda expressions. On the other hand, while Dyer et al. [2014] report that committers tend to adopt new features on an individual basis rather than in a team, we found that 27% of the projects' contributors use lambda expressions.

7.2 Studies Bridging the Functional and Imperative Paradigms

The studies by Meijer and Finne [2001], Coblenz et al. [2016], Robbes et al. [2015], Schärli et al. [2003], Setzer [2003], Dekker [2006], and Pankratius et al. [2012], explore the possibility of leveraging functional programming in Java and Java-like languages (e.g., Scala), and give insight into the advantages of functional features in performance, understandability, flexibility, and extensibility. In contrast, we study how functional features have been *already* adopted in an open-source community of 19,770 Java developers.

7.3 Studies and Tools that Target Lambda Expressions

Uesbeck et al. [2016] conducted one of the first qualitative studies, investigating *the impact of* lambda expressions on the development, debugging, and testing effort in C++. Our study, in contrast, focuses on *understanding the use of lambda expressions*, answering different questions including, but not limited to, *why* and *how* lambdas are used in Java.

The most popular Java IDEs, namely Eclipse, IntelliJ IDEA, and Netbeans have added support for migrating code to use lambda expressions [Bull 2014; Franklin et al. 2013; Gyori et al. 2013; IntelliJ 2017; Wielenga 2014]. In this study, we found that there were 5,855 instances of migration from anonymous classes to lambda expressions. In some cases, the migration was done in a single commit; we hypothesize that it was done using the afore-mentioned tools. However, we observed

that there are other kinds of migration to lambda expressions, which might not be supported by current IDEs, e.g., using lambdas to remove duplicated code, which was addressed by Tsantalis et al. [2017].

Moreover, we present several new implications for a new generation of tools that treat lambdas intelligently for improving the performance and maintainability of the code. These implications can definitely complement the existing set of guidelines and best practices [Baeldung 2016; Oracle 2015a] that help in using lambda expressions in Java more efficiently.

8 CONCLUSIONS

Java 8 retrofitted lambda expressions, one of the hallmarks of functional programming. However, without understanding the adoption of lambdas among Java developers, there can be no improvement. In this paper we use complementary empirical methods (mining 241 software repositories containing over 100,000 lambdas, and conducting 97 firehouse surveys) to answer fundamental questions such as *when, what, who, how*, and *why* are Java developers adopting lambdas.

We found that Java developers are increasingly adopting lambdas for replacing anonymous classes, replacing external with internal iterators, and for behavior parameterization. However, we found they sometimes do this inefficiently and are not leveraging the full power of the functional paradigm. We found that a core subset of developers are introducing most of the lambdas, mostly manually.

These results call for improvements on the language design and more intelligent tool support, such as detecting inefficient use and suggesting appropriate code changes, and judiciously replacing inheritance with composition.

Currently, we are developing refactoring tools to eliminate the misuses of lambda expressions. Our first target is to remove auto-boxing from lambda expressions by using specialized primitive functional interfaces instead of the generic ones. In the future, we plan to assist a developer in exception handling for lambda expressions, preventing object mutation, and eliminate other lambda misuses.

We hope that our paper inspires a symbiotic ecosystem where researchers, language designers, and tool builders work together to increase the adoption of functional constructs.

ACKNOWLEDGMENTS

We are grateful to Joshua Bloch, Doug Lea, Eric Walkingshaw, Sruti Srinivasa, and the anonymous reviewers for their insightful and constructive feedback for improving the paper. We also thank all the developers who participated in the qualitative study. This research was partially supported by NSF grants CCF-1439957 and CCF-1553741, a Google Faculty Research award, and NSERC grant 435480-2013.

REFERENCES

Theodoros Amanatidis and Alexander Chatzigeorgiou. 2016. Studying the evolution of PHP web applications. *Information and Software Technology* 72 (2016), 48 – 67. DOI: http://dx.doi.org/10.1016/j.infsof.2015.11.009

A. Ampatzoglou, A. Chatzigeorgiou, S. Charalampidou, and P. Avgeriou. 2015. The Effect of GoF Design Patterns on Stability: A Case Study. *IEEE Transactions on Software Engineering* 41, 8 (Aug 2015), 781–802. DOI: http://dx.doi.org/10.1109/TSE.2015.2414917

Baeldung. 2016. Lambda Expressions and Functional Interfaces: Tips and Best Practices. (November 2016). http://www.baeldung.com/java-8-lambda-expressions-tips

Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017).

James M. Bieman, Greg Straw, Huxia Wang, P. Willard Munger, and Roger T. Alexander. 2003. Design Patterns and Change Proneness: An Examination of Five Evolving Systems. In *Proceedings of the 9th International Symposium on Software Metrics (METRICS '03)*. 40–49. http://dl.acm.org/citation.cfm?id=942804.943777

Joshua Bloch. 2008. *Effective Java: A Programming Language Guide.* Addison-Wesley Professional.

Viviana Bono, Enrico Mensa, and Marco Naddeo. 2014. Trait-oriented programming in Java 8. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014.* 181–186. DOI: http://dx.doi.org/10.1145/2647508.2647520

Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. DOI: http://dx.doi.org/10.1191/1478088706qp063oa

Ian Bull. 2014. Eclipse Support for Java 8. (March 2014). http://eclipsesource.com/blogs/2014/03/25/eclipse-support-for-java-8/

John L. Campbell, Charles Quincy, Jordan Osserman, and Ove K. Pedersen. 2013. Coding In-depth Semistructured Interviews. *Sociological Methods & Research* 42, 3 (2013), 294–320. DOI: http://dx.doi.org/10.1177/0049124113500475

Alonzo Church. 1932. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics* 33, 2 (1932), 346–366. http://www.jstor.org/stable/1968337

Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. 2016. Exploring Language Support for Immutability. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16).* 736–747. DOI: http://dx.doi.org/10.1145/2884781.2884798

Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. 2017. Empirical Study of Usage and Performance of Java Collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17).* 389–400. DOI: http://dx.doi.org/10.1145/3030207.3030221

Daniela S. Cruzes and Tore Dybå. 2011. Research synthesis in software engineering: A tertiary study. *Information and Software Technology* 53, 5 (May 2011), 440–455. DOI: http://dx.doi.org/10.1016/j.infsof.2011.01.004

Anthony H. Dekker. 2006. Lazy Functional Programming in Java. *SIGPLAN Notices* 41, 3 (March 2006), 30–39. DOI: http://dx.doi.org/10.1145/1140543.1140549

Eclipse Java development tools. 2017. JDT Components. (2017). http://www.eclipse.org/jdt/

Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014).* 779–790. DOI: http://dx.doi.org/10.1145/2568225.2568295

Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. *Selecting Empirical Methods for Software Engineering Research.* Springer London, 285–311. DOI: http://dx.doi.org/10.1007/978-1-84800-044-5_11

Lukas Eder. 2014. jOOλ - The Missing Parts in Java 8. (2014). https://github.com/jOOQ/jOOL

Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. 2013. LAMBDAFICATOR: From Imperative to Functional Programming Through Automated Refactoring. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13).* 1287–1290.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional.

Trisha Gee. 2015. Debugger Improvements. (November 2015). https://www.youtube.com/watch?v=rimzOolGguo

Brian Goetz. 2012. Translation of Lambda Expressions. (April 2012). http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html

Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. 2013. Crossing the gap from imperative to functional programming through refactoring. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE).* 543–553.

Jerry L. Hintze and Ray D. Nelson. 1998. Violin Plots: A Box Plot-Density Trace Synergism. *The American Statistician* 52, 2 (1998), 181–184.

IntelliJ. 2017. Migrating to Java 8. (April 2017). https://www.jetbrains.com/help/idea/2017.1/migrating-to-java-8.html

JavaParser. 2017. Java 9 Parser and Abstract Syntax Tree for Java. (2017). http://javaparser.org/

Foutse Khomh and Yann-Gaël Guéhéneuc. 2008. Do Design Patterns Impact Software Quality Positively?. In *12th European Conference on Software Maintenance and Reengineering.* 274–278. DOI: http://dx.doi.org/10.1109/CSMR.2008.4493325

Sergey Kuksenko. 2013. JDK 8: Lambda Performance study. (2013). http://www.oracle.com/technetwork/java/jvmls2013kuksen-2014088.pdf

Henry B. Mann. 1945. Nonparametric Tests Against Trend. *Econometrica* 13, 3 (1945), 245–259.

Radu Marinescu. 2004. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM '04).* 350–359.

Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2017. Supplemental material. (2017). http://dmazinanian.me/conference-papers/oopsla/2017/07/04/oopsla17.html

Bartłomiej Mazur. 2017. Performance of Java, part 2. (2017). https://blog.gotofinal.com/java/benchmark/performance/2017/07/09/performance-of-java-1.html

Erik Meijer and Sigbjorn Finne. 2001. Lambada, Haskell as a better Java. *Electronic Notes in Theoretical Computer Science* 41, 1 (2001), 91 – 119. DOI: http://dx.doi.org/10.1016/S1571-0661(05)80549-3

Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How We Refactor, and How We Know It. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. 287–297. DOI: http://dx.doi.org/10.1109/ICSE.2009.5070529

Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. 2015. The Design Space of Bug Fixes and How Developers Navigate It. *IEEE Transactions on Software Engineering* 41, 1 (Jan 2015), 65–81. DOI: http://dx.doi.org/10.1109/TSE.2014.2357438

T. H. Ng, S. C. Cheung, W. K. Chan, and Y. T. Yu. 2007. Do Maintainers Utilize Deployed Design Patterns Effectively?. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. 168–177. DOI: http://dx.doi.org/10.1109/ICSE.2007.33

Adel Noureddine and Ajitha Rajan. 2015. Optimising Energy Consumption of Design Patterns. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. 623–626. http://dl.acm.org/citation.cfm?id=2819009.2819120

Scott Oaks. 2014. *Java Performance: The Definitive Guide: Getting the Most Out of Your Code.* O'Reilly Media.

Oracle. 2013. Java SE 8: Lambda Quick Start. (December 2013). http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html

Oracle. 2014. Optional (Java 8 Documentation). (2014). https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html

Oracle. 2015a. Lambda Expressions. (2015). https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Oracle. 2015b. Type Inference. (2015). https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html

V. Pankratius, F. Schmidt, and G. GarretÃşn. 2012. Combining functional and imperative programming for multicore software: An empirical study evaluating Scala and Java. In *2012 34th International Conference on Software Engineering (ICSE'12)*. 123–133. DOI: http://dx.doi.org/10.1109/ICSE.2012.6227200

Chris Parnin, Christian Bird, and Emerson Murphy-Hill. 2011. Java Generics Adoption: How New Features Are Introduced, Championed, or Ignored. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*. 3–12. DOI: http://dx.doi.org/10.1145/1985441.1985446

L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L. G. Votta. 2001. A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. *IEEE Transactions on Software Engineering* 27, 12 (Dec. 2001), 1134–1144. DOI: http://dx.doi.org/10.1109/32.988711

Eric S. Raymond. 2001. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Romain Robbes, David Röthlisberger, and Éric Tanter. 2015. Object-oriented software extensions in practice. *Empirical Software Engineering* 20, 3 (2015), 745–782. DOI: http://dx.doi.org/10.1007/s10664-013-9298-0

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. 2003. Traits: Composable Units of Behaviour. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03)*. 248–274. DOI: http://dx.doi.org/10.1007/978-3-540-45070-2_12

Pranab Kumar Sen. 1968. Estimates of the Regression Coefficient Based on Kendall's Tau. *J. Amer. Statist. Assoc.* 63, 324 (1968), 1379–1389.

Anton Setzer. 2003. Java as a Functional Programming Language. In *: International Workshop of Types for Proofs and Programs (TYPES 2002)*. 279–298. DOI: http://dx.doi.org/10.1007/3-540-39185-1_16

Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. 858–870. DOI: http://dx.doi.org/10.1145/2950290.2950305

Janice Singer, Susan E. Sim, and Timothy C. Lethbridge. 2008. Software Engineering Data Collection for Field Studies. In *Guide to Advanced Empirical Software Engineering*, Forrest Shull, Janice Singer, and Dag I. K. Sjøberg (Eds.). Springer London, Chapter 1, 9–34.

Ewan Tempero, Hong Yul Yang, and James Noble. 2013. What Programmers Do with Inheritance in Java. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP 2013)*. 577–601. DOI: http://dx.doi.org/10.1007/978-3-642-39038-8_24

TouK. 2017. ThrowingFunction. (2017). https://github.com/TouK/ThrowingFunction

Nikolaos Tsantalis, Davood Mazinanian, and Giri P. Krishnan. 2015. Assessing the Refactorability of Software Clones. *IEEE Transactions on Software Engineering* 41, 11 (Nov 2015), 1055–1090. DOI: http://dx.doi.org/10.1109/TSE.2015.2448531

Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. 2017. Clone Refactoring with Lambda Expressions. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. 60–70. DOI: http://dx.doi.org/10.1109/ICSE.2017.14

Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. 2016. An Empirical Study on the Impact of C++ Lambdas and Programmer Experience. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. 760–771. DOI: http://dx.doi.org/10.1145/2884781.2884849

Marek Vokáč, Walter Tichy, Dag I. K. Sjøberg, Erik Arisholm, and Magne Aldrin. 2004. A Controlled Experiment Comparing the Maintainability of Programs Designed with and Without Design Patterns – A Replication in a Real Programming Environment. *Empirical Software Engineering* 9, 3 (Sept. 2004), 149–195. DOI: http://dx.doi.org/10.1023/B:EMSE.0000027778.69251.1f

Geertjan Wielenga. 2014. Smart Migration to JDK 8 in NetBeans IDE. (January 2014). https://netbeans.org/kb/docs/java/jdk8-migration-screencast.html

Claes Wohlin and Aybüke Aurum. 2015. Towards a decision-making structure for selecting a research design in empirical software engineering. *Empirical Software Engineering* 20, 6 (2015), 1427–1455. DOI: http://dx.doi.org/10.1007/s10664-014-9319-7

Alex Zhitnitsky. 2015. Benchmark: How Misusing Streams Can Make Your Code 5 Times Slower. (2015). http://blog.takipi.com/benchmark-how-java-8-lambdas-and-streams-can-make-your-code-5-times-slower/