The indolent lambdification of Java

Understanding the support for lambda expressions in the Java ecosystem

Fernando Petrulio¹ · Anand Ashok Sawant² · Alberto Bacchelli¹

Accepted: 16 August 2021 / Published online: 1 October 2021
© The Author(s) 2021

Abstract

As Java 8 introduced functional interfaces and lambda expressions to the Java programming language, the JDK API was changed to introduce support for lambda expressions, thus allowing consumers to define lambda functions when using Java's collections. While the JDK API allows for a functional paradigm, for API consumers to be able to completely embrace Java's new functional features, third-party APIs must also support lambda expressions. To understand the current state of the Java ecosystem, we investigate (i) the extent to which third-party Java APIs have changed their interfaces, (ii) why or why not they introduce functional interface support and (iii) in the case the API has changed its interface how it does so. We also investigate the consumers' perspective, particularly their ease in using lambda expressions in Java with APIs. We perform our investigation by manually analyzing the top 50 popular Java APIs, conducting in-person and email interviews with 23 API producers, and surveying 110 developers. We find that only a minority of the top 50 APIs support functional interfaces, the rest does not support them, predominantly in the interest of backward compatibility. Java 7 support is still greatly desirable due to enterprise projects not migrating to newer versions of Java. This suggests that the Java ecosystem is stagnant and that the introduction of new language features will not be enough to save it from the advent of new languages such as Kotlin (JVM based) and Rust (non-JVM based).

Keywords Lambda expressions · Functional paradigm · API · Java

Communicated by: Denys Poshyvanyk

✉ Anand Ashok Sawant
asawant@ucdavis.edu

Fernando Petrulio
fpetrulio@ifi.uzh.ch

Alberto Bacchelli
bacchelli@ifi.uzh.ch

¹ University of Zurich, Zurich, Switzerland

² University of California, Davis, CA, USA

1 Introduction

Java 8 (released in 2014) introduced one of the most eagerly anticipated and controversial features, *i.e.*, support for functional interfaces and lambda expressions (Goetz 2013). Since this introduction, researchers have analyzed how and to what extent lambda expressions have been adopted by the Java development community. Mazinianian et al. investigated the usage of lambda expressions in 241 Java open-source software projects (Mazinianian et al. 2017). They observed an increasing trend in such usage and found that the primary reasons for developers to adopt lambda expressions were improved readability, reduced code duplication, and lazy evaluation of expressions.

For lambda expressions to be fully supported, Java developers should have the opportunity to develop end-to-end using functional features and lambda expressions. This can only happen if third-party APIs (in addition to the Java JDK API) do support lambda expressions.

In a previous study, Mazinianian et al. (2017) investigate how, when, and, why Java developers introduce lambda expressions when interacting with the Java collections. They find that there is an increasing trend of lambda usage and the main reason behind this is the increased maintainability of the code. Lucas et al. (2019) investigate the usage of the Java streams API in conjunction with lambda expressions to better understand how the code's maintainability is affected.

Their overall focus has centered around the adoption of lambda expressions when it comes to using the Java collections and streams APIs. When Java introduced lambda expression support, they did this to explicitly make it easier and faster to perform computation over large data collections. However, there is limited empirical evidence on how and to what extent the *larger Java ecosystem* has adapted to support the use of lambda expressions. We focus on this area by studying how and why third-party APIs have (or have not) adapted to Java 8 or higher¹ by investigating—from the API producer's perspective—as to what their decision making process is and how changes are implemented. We also investigate—from the API consumer's perspective—as to whether and how consumers have adapted to the changes in Java 8+.

This study is targeted at three main audiences: (1) API producers who have not yet migrated their APIs to Java 8+, (2) language designers (particularly Java ones), and (3) researchers. The first ones can learn from the strategies, rationales, and processes behind adapting to functional interfaces, as implemented by stable, historical, and popular APIs that have already changed. API producers can also learn about the challenges involved in making a change and the perceived advantages of making an API interface lambda compliant. Similarly, the API producers learn from those APIs that have not yet made any change as to why this is the case, seeing the stumbling blocks that prevent APIs from transitioning to a new version of Java. Simultaneously, our study provides data for researchers interested in better understanding and supporting this novel process of transitioning to a new version of Java and taking advantage of its functional features.

By investigating the consumer perspective on whether they currently (or would likely in the future) develop end-to-end using lambda expressions we intend on informing API producers on the demand that exists among the consumer community for lambda expression support. Java language designers can be better informed about the adoption of functional interfaces among consumers of APIs. Researchers can also be informed on the demand for

¹ In the context of this paper, we refer to this as Java 8+.

lambda support and help devise ways in which consumers can be stimulated to use lambda expressions.

To study the API producer perspective, we begin by investigating how many of the top 50 popular Java APIs² have migrated to Java 8+ and have declared their interfaces to be functional interfaces (*i.e.*, being lambda expression compliant). For those APIs that *do* make a change, we investigate with the aid of the API documentation and seven interviews with the API producers how and why certain changes were made. For the APIs that *do not* adapt to Java 8+, we conduct sixteen interviews with the API producers to understand what prevents an API producer from changing the API and making it Java 8+ ready. Finally, we investigate whether API consumers have reacted to the changes in the Java ecosystem by surveying 110 Java API consumers.

With our study, we make the following *main contributions*:

1. Evidence that only nine out of the top 50 popular Java APIs have introduced support for lambda expressions. This indicates how popular APIs are conservative when it comes to making major changes such as changing the version of Java being used.
2. A taxonomy of seven reasons behind API producers making a change to the API. These reasons are generally to benefit the API consumer - a fact echoed by the consumers themselves.
3. An enumeration of seven reasons why API producers *do not* introduce support for functional interfaces. Among these, the chief is the need for backward compatibility, since most of the Java world is still on Java 7. This suggests large inertia in the Java ecosystem.
4. Evidence that Java's support for several functional features is limited and does not fully implement the functional programming paradigm (*e.g.*, missing features such as monads) thus limiting the advantages of using lambdas.

2 The Lambdification of Java

Since its inception, Java has been an Object-Oriented Programming (OOP) language with no support for the functional programming paradigm. In the last decade, however, there has been a general shift to hybrid programming languages that have support for both programming paradigms (*e.g.*, Scala, Kotlin, and Swift), thus making Java face fierce competition. Introducing support for functional interfaces was a challenging endeavor and required multiple iterations over the proposal named Project Lambda (Goetz 2010).

As a result, Java now offers two ways to make lambda available for developers: (1) defining a new functional interface and (2) using objects belonging to `java.util.function`. A functional interface in Java is an interface with just one abstract method aside from the object methods (See Fig. 1, part a). This language enhancement also provided a specification for lambda expressions (or closures), which are anonymous functions. According to the Java language designers, there are two reasons behind making these enhancements: 1. anonymous classes in Java are currently bulky, confusing, inflexible, and lack the ability to abstract over control flow: in this regard, lambda expressions would aid in improving readability (Goetz 2010) and 2. Java programmers have become increasingly familiar with the lambda model of programming with other JVM based languages (Goetz 2013).

²ranked by the popularity of usage on GitHub

Objects belonging to `java.util.function` let developers avoid the implementation of a functional interface, providing a ready-to-use template for specific kinds of functions or structures, like assertions and predicates and collections. Streams are an example of their widespread use, where it's possible to filter data in a data structure by applying in-line lambda expressions. These features are useful in many circumstances, like the lazy evaluation by lambda. However, this practice exposes developers to Java built-in constraints, like the impossibility to define custom exceptions or specific control flows.

While the JDK API changed its interface and introduced explicit support for lambda expressions (Goetz 2013), the introduction of functional interface support in the Java language specification affects producers of third-party APIs as well. To make the entire ecosystem lambda compliant, API producers have to react to this changing landscape of Java and introduce support for the usage of lambda expressions. To do so, there are many ways in which APIs can change their interfaces, such as the one presented in part (b) of Fig. 1. In this example, we see that there are two ways in which a match can be computed on a string. In the case of the traditional eager approach, the `compute` function is invoked for both input parameters. In the case of the lazy evaluation, if the `compute` function errors out on the first parameter, the execution will immediately return a false without forcing a second evaluation, thus, resulting in a faster computation.

3 Methodological Overview

Our *goal* is to investigate the top 50 third-party and establish 1. the extent to which the Java ecosystem supports lambda expressions, 2. how and why such support was introduced by

Fig. 1 Impact of different changes on source code

API producers, 3. what factors prevent APIs from changing their interface to take advantage of the new Java language specification, and 4. to what extent consumers are ready for an ecosystem that fully supports the usage of lambda expressions. Based on the insights we discern from our investigation we would like to 1. inform API producers on how APIs introduce support for Java 8's features, 2. show those API producers contemplating a change to their API what the various challenges are when changing the API, 3. showcase to researchers what the major obstacles are to APIs changing and the entire Java ecosystem moving to the latest version of Java, and 4. provide evidence to the Java language designers on whether the changes implemented in Java 8 are sufficient and what the market demands in terms of new language features that support the functional paradigm.

3.1 Research Questions

Our empirical study revolves around four main research questions. First, we aim to establish the extent to which the Java ecosystem supports lambda expressions. To do so, we study the most popular and well-maintained Java-based APIs to understand which of them has introduced support for lambda expressions. Thus, we ask:

RQ₁. *Which top 50 Java APIs support lambda expressions?*

After having found the APIs that transitioned we focus on understanding why they introduced support for lambda expressions and how this support has been introduced. This allows us to understand the API design decisions that have been taken for the changes. This understanding can inform other API producers faced with a similar decision to introduce lambda support in their APIs. We ask:

RQ₂. *How and why do certain producers of top 50 APIs introduce support for lambda expressions in their APIs?*

On the other hand, some APIs *do not* adapt their interfaces to conform to Java 8+. We investigate why these APIs do not change their interfaces, to help us understand what prevents an API from fully embracing functional interfaces in Java. We ask:

RQ₃. *Why do most producers among the top 50 APIs not introduce support for functional interfaces and lambda expressions?*

Investigating the API consumer adoption of API versions that have functional interfaces speaks to the acceptance of functional interfaces and lambda expressions in the larger Java ecosystem and can inform other consumers and language designers. We ask:

RQ₄. *How do API consumers perceive the transition to lambdas in the Java ecosystem?*

3.2 Research Method

Our study features a mixed-methods research approach (Johnson and Onwuegbuzie 2004), where software repository analyses are complemented with manual investigations, in-person as well as email interviews, and surveys with practitioners. We deem this approach appropriate for this study because we need to analyze an APIs repository to understand whether any change has been made. To understand *why* a change was made and *what* the design process is to implement the changes we have to interview the developers of the APIs. This study was conducted in compliance with human research guidelines of the respective institutions. Figure 2 depicts the various steps we performed to address our research questions.

In the first stage (initial filtering in Fig. 2), we select a large set of third-party Java APIs filtering by their popularity among Github projects (3.2.1). We choose to focus our investigation on the most *popular* Java APIs as we expect these APIs to be actively and carefully maintained due to their importance, quality, influence, and history of development. Then, we inspect their codebase (light blue segment in Fig. 2) to understand which of these introduced full compatibility with lambda (3.2.2). At this point, the study takes two different paths aimed at analyzing APIs that have (1) evolved providing or (2) not providing support for lambda. In the former case (light green segment in Fig. 2), we apply a rigorous method to create a taxonomy of reasons why lambda support has been explicitly provided (3.2.3). The knowledge outcomes are corroborated with seven interviews conducted with APIs maintainers (pink segment in Fig. 2) in charge of this transition (3.2.3). In the latter, negative case, we select a subset of APIs that have not transitioned to a lambda-compatible version yet and then we conduct interviews with sixteen API producers to understand the reasons behind this choice (3.2.4). In the end, to challenge our findings from the manual analysis and interviews from both sides (purple segment in Fig. 2), we conduct an online survey (3.2.5) among software developers.

3.2.1 Subject selection

To establish the popularity of Java-based APIs, we target active³ Maven-based projects on GitHub. Overall, we find and download 255,587 Maven build files from a total of 1,609,877 Java-based projects on GitHub. We extract the artifact and group ID for each of the dependency declarations in these build files to establish the popularity of each Java-based API.⁴ We limit ourselves to the top 50 APIs in terms of popularity because they would be the best maintained, actively developed, and have an active consumer and producer community that would make them appropriate cases for other APIs to follow. We consider APIs with differing artifact IDs (*e.g.*, `spring-webmvc`, `spring-context`, `spring-core`) but same group IDs as belonging to the same API family (*e.g.*, Spring Framework) and treat all these APIs as a single entity. In the case of APIs that belong to the Apache ecosystem, we treat each API as an independent entity as there is minimal overlap in the developer communities (Fielding 1999; Iqbal 2014). While we strive to select 50 APIs to study, the 50th API selected is the 300th most popular API due to the aforementioned unique selection constraints that we impose. Furthermore, by selecting a diverse sample of APIs that

³Based on data from GHTorrent (Gousios and Spinellis 2012).

⁴Similar to what is done in fine-GRAPE (Sawant 2017)

Fig. 2 Overview of the research method used in this study

originate from different sets of developers we aim to cover a large portion of the APIs used in the ecosystem. The full list of the considered APIs is available in our replication package (Blinded 2020).

3.2.2 RQ₁. Identification of APIs with changed interfaces

To uncover whether an API has made any changes or decided to change at all, we conduct a systematic manual analysis. We consider an API as having introduced support for lambda statements only if there is a release that has been made such that API consumers can depend on it. We do not consider APIs that are in the decision phase of introducing support as those that currently support lambda expressions. For each of the selected APIs using the following sources:

- Release notes: major changes in an API to support a new language version or language feature would be documented in the release notes.
- Website: the website of the API should mention the language version and features it supports and when this was introduced.
- Code Repository: API projects typically have README pages in their repositories on GitHub where language support can be mentioned.

3.2.3 RQ₂. Analysis of APIs with changed interfaces

Once we isolate the APIs that have introduced support for lambda expressions, we conduct a manual analysis to (1) understand which features have been changed and have introduced functional interface support, (2) why this was done and, (3) how the change was implemented. To uncover these aforementioned aspects, we conduct our investigation in three phases:

Phase 1 - Data collection. The first author of the paper manually analyzed the following sources:

Source code. To detect API elements that introduce support for lambda expressions by looking for the `@FunctionalInterface` annotation.

Commit history. To compare the release that introduced support for lambda expressions to the previous release of the API that did not have this support. This comparison yields a set of API elements that have either been added or changed in the new version of the API. We isolate these elements and manually parse the history of the files in question to isolate the commit where the change was made. For each of these commits, we collect the commit message, date of commit, and whether there was a referenced issue that could provide us with additional information.

Issue tracker. To investigate the rationale behind making a certain change and insights into the decision-making process of the API that led to the introduction of the lambda support.

API documentation. We investigate Javadoc and code comments to get a detailed rationale behind the changes made to an API. This documentation also provides the date and version of the change and can provide examples of how consumers are expected to use elements that support functional interfaces.

Community documentation. We perform an exploratory search using Google to search for additional sources of documentation that might provide insight into an API's decision. We look particularly for blog posts by API producers, mailing lists where such changes could be discussed, StackOverflow posts that discuss a certain API element that has changed, and online tutorials that show how to use the new interface.

Phase 2 - Synthesis of a taxonomy. The first author performed an iterative content analysis (Srivastava and Hopwood 2009) over the collected data. Specifically, the process is composed of two iterations:

Iteration 1: The author manually goes over each resource identified in the data collection phase and assigns initial codes to all relevant pieces of information (Coleman and O'Connor 2007). He first summarizes in a short phrase the essential topic of each relevant resource identified; then, he assigns explanatory codes to create emergent themes.

Iteration 2: Ambiguous cases (*i.e.*, a case does not clearly fit in one or more of the pre-existing codes) are discussed with the third author and another researcher who is not an author of this paper. Based on the discussion among the researchers, the first author re-classifies the information from the previous step (no contrasting/missing explanatory codes were found).

Phase 3. Validation. As an extra validation step, the second author of this paper independently classifies all the pieces of information having the identified codes as a base; this author is also allowed to define new codes if needed. We perform this extra step to strengthen the completeness of the assigned codes. In the end, no new codes were needed, thus making the inter-rater agreement 100%.

Interviews with API producers who transitioned After the manual analysis, we conduct semi-structured interviews to understand the API producers' perspective on why and how Lambda expressions have been supported in conjunction with their APIs.

We contact the maintainers of the APIs that have changed their interface via email, providing them with a high-level summary of our research, *i.e.*, we do not reveal the ultimate goals of our study to mitigate "interviewer bias" (Bailar et al. 1977); we ask them to discuss the current and future situations of their projects.

We could interview one person for each API that have made a change, with the exception of the Log4J and the Dropwizard metrics APIs (who did not respond to our request). We ask our interviewees to complete a pre-interview survey, used to gather their background and current occupation as API team member. Each interview is scheduled for 45 minutes and conducted via SKYPE. We ask the interviewees questions on (1) the motivation behind the introduction of lambda expression support, (2) design decisions taken when changing the API, (3) expectation regarding consumer adoption of new features and (4) challenges faced with Java's lambda implementation.

Before an interview with an API producer, we tailor our interview guideline based on the motivations and development process details that we learn from our manual analysis of the APIs as elucidated in Section 3.2.3. The interviews are then conducted by the first two authors of this paper and are recorded and transcribed. We follow an interpretive-descriptive approach (Thorne et al. 1997) - an approach that originates from the social sciences. As part of this interpretive-descriptive approach, each interview transcript is broken down into smaller parts and assigned a code, similar codes are clustered. When we encounter the same code in an interview, *i.e.*, saturation, we change our interview guide to explore new areas. The top part of Table 1 presents the characteristics of the interviewees for this research question.

3.2.4 RQ₃. Analysis of APIs that do not change interface

Since this is a contrary perspective, *i.e.*, one where the developers have not done something, we do not expect much data to be gleaned from the issue tracker or mailing list (confirmed by a cursory manual analysis). Instead, to gain this perspective, we conduct email interviews with the API producers.

Table 1 Participants to the interviews

ID	Project	Experience (years)	ID	Project	Experience (years)
Why do top API producers introduce support for Lambda Expressions?					
P1	Guava	24	P5	Spring Security	21
P2	Spring Framework	23	P6	Spring Boot	19
P3	JUnit	11	P7	Spring Data	13
P4	Mockito	7			
Why do top API producers not introduce support for Lambda Expressions?					
P8	Apache Commons	32	P16	Eclipse Jetty	15
P9	Apache Commons	13	P17	H2	9
P10	Apache Commons	10	P18	Hamcrest	10
P11	Apache Commons	8	P19	Hamcrest	7
P12	Apache Hadoop	31	P20	Hibernate	16
P13	Apache Maven	31	P21	Project Lombok	20
P14	Apache Maven	17	P22	SLFJ	24
P15	Apache HttpComponents	20	P23	TestNG	15

Interviews with API producers who did not transition As opposed to the previous research question, in this phase, we conduct interviews via email as opposed to in person. The advantage of conducting email interviews is that the process is asynchronous, *i.e.*, the interviewee does not have to block one specific period to talk to the interviewer (Meho 2006). Moreover, email interviews allow us to reach a far larger audience and obtain perspectives from a more diverse set of developers than with in-person interviews. We ensure that we follow up on each email interview to ask for clarifications, challenge responses from the interviewee, and explore certain answers in depth. For projects that have not migrated to Java 8+, we ask the developers the reason and whether there are any plans to do so in the future, especially to support lambda expressions. For those APIs that have migrated to Java 8+, but have not introduced support for functional interfaces, we inquire why this is the case and whether it would be beneficial to introduce such support. The bottom part of Table 1 presents the characteristics of our interviewees for this research question. Out of all the APIs that fall in this category, nine APIs have no mailing list and we obtain no response from seven other APIs. For the Apache commons projects, we have four developers who respond and cover the topic of transitioning the commons packages to supporting lambda expressions for all projects under the commons umbrella.

3.2.5 RQ₂ - RQ₄. Survey with API Producers and Consumers

To challenge our findings we conduct a survey with API producers and consumers. Our survey is composed of 21 questions (included in Appendix A). From the producers, we seek their motivations to introduce functional interface support and how far they are willing to go to change their API. This seeks to explore whether there are any motivations beyond what an earlier phase of our study uncovers. From the consumers, we investigate how they perceive these changes in the ecosystem and whether they perceive the importance of adopting versions of APIs that support functional interfaces such that they can develop end-to-end using the functional paradigm.

Survey Design We first gather demographics such as (i) programming experience; (2) expertise with Object-oriented programming, functional paradigm, and lambda expressions; (3) development practice in Java 8+ and, (4) experience with traditional functional languages (*e.g.*, Lisp). Based on their role (*i.e.*, producer of APIs, consumer of APIs, or both), we direct respondents to a dedicated part of the survey.

The survey from the producer perspective asks whether an API producer has introduced support for Java 8+ and its lambda expressions. If so, we ask to indicate (on a 5-point Likert scale) to what extent the motivations uncovered from our interviews and manual analysis have influenced them. From those producers who have not introduced support, in a free form text box, we ask why this is the case. We also ask these producers for their take on the lambdas in Java and ask them more about their future plans to introduce further support for lambda expressions.

To API consumers, we ask their frequency of usage of lambda in Java. This question is general and does not refer to the use of lambda expressions in conjunction with third-party APIs; we include it to contrast our results with the findings previously achieved on the general usage of lambda in Java (Mazinanian et al. 2017). Then, we ask (on a 5-point Likert scale) what factors have influenced them to use lambda expressions with APIs and what reasons play a role in them not adopting an API that supports functional interfaces.

Respondents' recruitment Following the guidelines provided by Flanigan et al. (Flanigan et al. 2008), we limit common issues possibly arising in survey studies and affecting the response rate: We limit the length of the survey, respect the anonymity of participants, and prevent our influence in the responses. The survey is advertised through professional contacts and published on practitioners blogs like REDDIT (Reddit 2019), TWITTER and JAVA CODE RANCH (Java Code Ranch 2019). As a result, we collect 252 responses, but we filter out 142 because incomplete. Among the remaining 110 respondents, the average programming experience is over 9 years. Out of these 110 respondents, 65 respondents are API producers and answer the API producer perspective segment of the survey and all 110 respond to the API consumer perspective segment. Our respondents originate from 21 different countries from 3 different continents. 28 of our respondents work in both open source and industrial contexts, 30 work only on open source projects, and the rest are based in industry. Only 8 of our respondents do not develop with Java 8+ and the majority (76 out of 110) report a 'good' or higher experience with functional programming.

4 RQ₁: Lambdas among top APIs

Although 31 of the top 50 APIs have migrated to a version of Java 8+ (the other 19 APIs continue to adhere to Java version 7 or 6), only nine of them have introduced support for functional interfaces and lambda expressions. Table 2 presents an overview of all 50 APIs from our manual analysis indicating whether or not they have changed their interface and whether they support Java 8+.

Through our manual analysis, we found three APIs currently working on introducing support for functional interfaces. Hibernate is going to start supporting lambda expressions and Java 8+ from version 6 (currently under development with an alpha release made in April 2020⁵); SLF4J (the most popular logging library) has had a long discussion on

⁵<https://github.com/hibernate/hibernate-orm/tree/wip/6.0>

Table 2 Top 50 APIs: Support for functional interface (FI) and Java 8+

API	FI Support	Java 8+	API	FI Support	Java 8+
log4j	✓	✓	mockito	✓	✓
spring security	✓	✓	spring data	✓	✓
spring boot	✓	✓	spring framework	✓	✓
google guava	✓	✓	junit	✓	✓
dropwizard metrics	✓	✓	hibernate	✗	✓
h2database	✗	✗	testng	✗	✓
selenium hq	✗	✓	project lombok	✗	✓
mybatis	✗	✓	mysql	✗	✓
json	✗	✓	hsqldb	✗	✓
hamcrest	✗	✗	eclipse jetty	✗	✓
codehaus jackson	✗	✗	aspectj	✗	✓
apache tiles	✗	✗	apache poi	✗	✓
apache maven	✗	✗	apache httpcomponents	✗	✗
apache hadoop	✗	✓	apache commons	✗	✓
apache camel	✗	✓	jstl	✗	✗
joda-time	✗	✗	javax validation	✗	✓
javax servlet jsp	✗	✗	javax.servlet	✗	✓
javax inject	✗	✗	javax	✗	✓
commons logging	✗	✗	commons lang	✗	✗
commons io	✗	✗	commons fileupload	✗	✗
commons dbcp	✗	✓	commons collections	✗	✓
commons codec	✗	✗	slf4j	✗	✓
google code gson	✗	✗	fasterxml jackson core	✗	✓
alibaba	✗	✗	qos logback	✗	✗
cglib	✗	✓	sun jersey	✗	✗

introducing support for lambda expressions since 2016.⁶ As of June 2019, version 2.0.0 of SLF4J will change its interface and support lambda expressions, however, as of May 2020, only an alpha version has been released. TestNG has been beta testing its latest version (7.0.0) for the last year,⁷ this version introduces support for functional interfaces. In our study, we define these APIs to be in an intermediate state.

In our survey, we ask a broader set of API producers to indicate whether the API they work on has introduced Java 8+ support. 91% (59 out of 65) of the respondents indicate that their API has made the change to support Java 8+ (*i.e.*, they develop on Java 8+ and support consumers that develop on Java 8+). This is in contrast to the comparatively lower 62% of the top 50 Java APIs which we observed to support Java 8+. This difference could be a result of the population of survey participants who might work on closed source APIs or APIs that

⁶<https://jira.qos.ch/plugins/servlet/mobile#issue/SLF4J-371>

⁷[https://groups.google.com/forum/#!searchin/testng-users/7.0\\$20beta%7Csort:date/testng-users/rkQP6ouYPXI/yY2IWdbbBwAJ](https://groups.google.com/forum/#!searchin/testng-users/7.0$20beta%7Csort:date/testng-users/rkQP6ouYPXI/yY2IWdbbBwAJ)

are not popular enough to figure in our dataset. Alternatively, a self-selection bias might be at play where developers who have made a change preferred to respond to the survey.

We ask the producers whose APIs have migrated to Java 8+ as to whether their API supports lambda functionality and functional interfaces. 61% (36 out of 59) indicate that their API has changed its interface to enable lambdas. A further 22% of the respondents indicate that the API they work on is currently in the process of implementing support for lambda expressions. This is also higher than what was observed for the top 50 APIs, where only 10% of the APIs who have migrated to Java 8+ plan on introducing support for lambda expressions.

Finding 1. A total of 41 among the top 50 Java APIs have *not* introduced support for lambda expressions or functional interfaces. Only nine APIs support both Java 8+ and lambda expressions. This shows that (out of the limited subset under study) a small proportion of the Java ecosystem supports Java’s functional features.

5 RQ₂: TOP APIs Introducing Lambdas

In this section, we focus on the angle of those APIs that introduced lambda support. Table 3 shows that the majority of the APIs has been around for more than 10 years. Most of these APIs release frequently: even the younger APIs (Spring Boot and Spring Data) have over one hundred releases. The Dropwizard metrics API is the smallest in our dataset with only 29,605 lines of code, while Guava is the largest API with over 750,000 lines of code. The Spring projects expose a large number of public methods, whereas Log4J has the most limited functionality. Spring Framework and Spring Boot are further along in introducing support for functional interfaces, while the other APIs are still in a nascent stage.

5.1 Reasons for introducing lambda support

Our analysis and interviews with API producers uncovered seven motivations behind introducing support for lambda expressions, which we order by the popularity of applicability to APIs as indicated by third-party API developers in our survey (Fig. 3).

Table 3 Overview of the subject APIs that introduced lambda support (FI: Functional Interfaces)

Project	Domain	Users	1st Release	Releases	LOC	Public methods	FI
Guava	Collections	190,130	Apr, 2010	90	761,055	509	11
Spring Framework	Dependency Injct.	142,761	Dec, 2005	174	672,374	6,435	142
JUnit5	Testing	65,603	May, 2007	46	82,589	1,068	29
Mockito	Testing	89,029	Apr, 2008	490	58,906	814	5
Log4j	Logging	365,065	Nov, 2005	73	37,222	165	3
Dropwizard Metrics	Profiling	8,042	Feb, 2010	89	29,605	269	6
Spring Boot	Environment Mgmt.	59,190	Apr, 2014	151	332,935	2,773	169
Spring Security	Authentication	93,866	Apr, 2008	145	258,079	415	31
Spring Data	Data Access	13,964	Feb, 2013	111	53,444	564	6

Fig. 3 Motivations (from 65 API producers) influencing the introduction of lambda support in APIs

The survey uncovered no new reasons to introduce lambda expression support in an API. All seven reasons mentioned by our interviewees proved to be exhaustive. The first reason was by far the most popular and agreed upon by all our interviewees; this is expected given it is one of the original reasons behind introducing support for lambda expression in Java. Surprisingly, among the other reason to introduce lambdas, improving performance ranked only number six. Only one API that we analyzed - Log4J decided to change (according to its documentation) its logging interface to allow their consumers to perform what they term as ‘lazy logging’. This a performance enhancement, where log messages are only published on the evaluation of a lambda expression that wraps a method invocation whose result is to be logged when the right logging level is set and not every time. This ensures that the method invocation to be logged is only lazily evaluated.

All our interviewees mentioned the second as well, as they wanted to migrate their API to a newer version of Java. However, with such a change, Guava had to remove certain existing functionality (*i.e.*, they had to standardize their features, reason 4 in Fig. 3). Before the advent of Java 8, Guava had put a workaround in place to offer functional support: “The syntax sucks. At the same time, this stuff is now, has always been and will always be nothing but a stopgap measure until the right language change can come along, at which time we can finally really decide on the optimal syntax and have functional-style programming start actually making lives better in Java for once”. This had to be removed after the migration to Java 8 and thus changed the interface. Migrating to Java 8 also has its advantages (reason 5): “lambdas make a sort of style of programming possible within Java which previously was not really possible and that’s really the callback-based programming”.

Interestingly, the third reason was mentioned only by Guava. In the functional programming paradigm, the execution of a function should have no side effects. One interviewee mentioned that in Guava they “added methods to all [immutable collections] named toImmutable[Type]() (e.g. ImmutableList. toImmutableList()) which return a Collector for collecting a Stream into an immutable collection/map object, thus ensuring no mutation of the original collection.

‘Checking exception’ was the least popular reason, only mentioned by one interviewee: “assert throw assertion made it much, much clearer that all the code that’s basically wrapped

in this lambda is supposed to be throwing in an exception”_[P3]. It appears to be more of an outlier of a use case of lambda expression with APIs in Java.

5.2 Design decisions

We investigate the design decisions and their implementation by top APIs (in the case of the Spring framework, Spring boot, Spring security, and Spring data we consider them under Spring projects).

Spring projects. The Spring framework (which acts as a base to all other Spring projects) has typically been conservative about changing versions of Java. For example, up until Spring 4, they supported only Java 7. Only with the release of Spring 5 (September 2017) did they, for the first time, require their consumers to use Java 8. This was due to the end-of-life support for Java 7. Both P5 and P6 mention that Spring security and boot are both dependent on the Spring framework making a change before they can also make any kind of change.

The move to support Java 8 in the Spring framework did not require major changes to the API: “it was a very natural transition because we cannot refactor, we cannot break existing APIs so we didn’t add anything we just added functional interface types”_[P2]. One such example can be seen in Fig. 4 where the existing interface simply accepts a functional interface as a parameter, which results in a more readable and compact code. P5 echoed the word of P2 by saying that in Spring security they “didn’t need to make any formal changes around this since single method interfaces are implicitly function interfaces”. P2 and P6 mentioned that the typical pattern of interaction with Spring was to use anonymous classes inside of method invocations. Transitioning away from this pattern was easy as P2 puts it: “basically there were abstract single method classes right types so we annotated a bunch of those with functional interfaces as well retroactively”. **JUnit.** JUnit decided in 2016 to introduce support for Java 8 and its features instead of Java 5 which was the standard at the time. Which meant a long debate on how to support Java 8 as elucidated by P3: “[the] first meeting [the JUnit team] had we had a full week basically where we met and kicked off [JUnit Jupiter]”.

```
public void executeQueryWithPreparedStatementAndRowMapper() {

    \\\ without lambda
    template.query("SELECT name, age FROM person WHERE dep = ?", new PreparedStatementSetter() {
        @Override
        public void setValues(PreparedStatement ps) throws SQLException {
            ps.setString(1, "Sales");
        }
    }, new RowMapper<Person>() {
        @Override
        public Person mapRow(ResultSet rs, int i) throws SQLException {
            return new Person(rs.getString(1), rs.getString(2));
        }
    });

    // with lambda
    template.query("SELECT name, age FROM person WHERE dep = ?", ps -> ps.setString(1, "Sales"),
        (rs, rowNum) -> new Person(rs.getString(1), rs.getString(2)));
}
```

Example of lambda
Expressions with Spring

Fig. 4 Example of new usage pattern with Spring

```

// Without Lambda Expression
@Test
public void isOlderThan30() throws Exception {
    assertTrue(example.matches(new Predicate<Person>() {
        @Override
        public boolean test(Person person) {
            return person.getAge() > 30;
        }
    }));
}

// Using JUnit Assertion with Lambda Expressions
@Test
public void isOlderThan30WithOneLineLambda() throws Exception {
    assertTrue(example.matches((Person person) -> person.getAge() > 30));
}

```

Example of lambda Expressions with JUnit

Fig. 5 Example of new usage pattern with JUnit

To support Java 8 JUnit had to completely redevelop its interface. They created a new assertions framework that could take lambda expressions as an argument as seen in Fig. 5. They also created a new way to test exceptions in Java with the aid of lambda expressions. This implied that a consumer could not easily migrate from JUnit 4 to JUnit Jupiter. To overcome these compatibility issues, JUnit provides the JUnit Vintage Engine which “basically ... supports running JUnit 3 and JUnit 4 tests without any changes in the test code so you basically don’t have to change your tests you don’t have to switch them to a new API”.

Mockito. Mockito has not officially made any move to support Java 8+. At the time of our analysis, Mockito still developed on Java 6. The last major release of Mockito was version 2 and it was released around the time when Java 8 was still new. As P4 put it: “Java 8 was like very new and because Mockito is used in so many corporate environments, having Java 8 is just not reliable”.

Despite not moving to make Java 8 support official, Mockito has still started to support lambda expressions. According to P4, these changes were facilitated by “compatibility between the Java versions so that we can run our Java 6 implementation with Java 8 features”_[P4]. The main change has been in “the old ugly answer API, it’s nicer to write a lambda with the ugly API than to write a whole anonymous class ... so it’s just less code”_[P4].

Guava. Guava moved to support Java 8 as soon as possible. P1 explained the principal reason for this: “Google’s internal engineering migrates from version to version all at once. The decision to go for Java 8 was a no-brainer since developers loathe verbose anonymous classes”_[P1]. The principal reason for this is “[valuing] code simplicity and readability a LOT here”. Backward compatibility is not of major concern for Guava: “We try to create the impression that people should move off of Guava onto standard


```

// Using a for-loop (external iteration)
public Iterable<String> getBestStudentsOfYear(Iterable<Student> students, int year) {

    List<String> result = new ArrayList<>();

    for (Student s : students) {
        if (s.getGraduationYear() == year && s.getScore() >= 9) {
            result.add(s.getName());
        }
    }

    return result;
}

// Using Google Guava with lambda expressions
public Iterable<String> getBestStudentsOfYear(Iterable<Student> students, int year) {

    Iterable<Student> bestStudents = Iterables.filter(students,
        (Student s) -> s.getGraduationYear() == year && s.getScore() >= 9);

    Iterable<String> names = Iterables.transform(bestStudents, s -> s.getName());
    return names;
}

```

Example of lambda
expressions with Guava

Fig. 6 Example of new usage pattern with Guava

libraries when they can. We know that most clients won't bother to change working code". Internally they do try to aid in the automated transition from one version to another of Guava, however, this does not work on code outside of Google.

The biggest change that Guava made was to introduce new features that allow for a lambda definition to convert a collection from one type to another. Since Guava is a collections library, they added stream support in conjunction with lambda support for each of the collections as seen in Fig. 6. They also removed features that were in place to emulate functional behavior but only acted as workarounds.

5.3 Challenges When Transitioning

We asked our interviewees whether they faced any major challenges when introducing support for lambda expressions. In the case of Spring, P2 very proudly informed us that "I think the foundation is flexible enough and the Spring foundation has shown to be flexible enough to support ... functions because the abstractions are designed correctly". Echoing this statement, P6 mentioned that he "didn't see it as a challenge".

In the case of Mockito, the challenge was not related to implementation, but the struggle within the development team. P4 mentioned to us that he was the only one who wanted to see more lambda support in Mockito. However, most of the other core maintainers of Mockito were hesitant to make the change due to the mantra of "if it is not broken then why fix it?".

According to P3 of JUnit “One of the main reason we didn’t use Java built-in functional interfaces was they did not allow to throw checked exceptions”. To overcome this, JUnit had to implement its own functional interfaces as a workaround.

Only two survey respondents rated the effort to transform the API to support functional interfaces a five on a five-point effort scale. One respondent indicated that this is because the API is a byte code compiler, meaning that there was an increased level of difficulty when it came to parsing code with lambda expressions.

5.4 Issues With Lambda Expressions

As lambda expressions are a new concept in Java, we asked our interviewees to express their thoughts on whether there were any major issues with lambda expressions and whether this would have an impact on their consumers. Although performance benefits from using lambda expressions were one of the main motivations behind the Java language designers introducing functional interfaces in Java 8, P6 stated: “There is a known performance problem attached to it that makes us avoid using them in code paths that are frequently invoked”.

Throwing a checked exception from a lambda expression caused issues for the migration of JUnit to Java 8+; P6 explained: “I am personally annoyed by the lack of support when the body of the lambda invokes code that can throw a checked exception”_[P6].

Our interviewees also mention that debugging lambda expressions is hard because stack traces are unreadable: “The saddest thing about them, though, is their obfuscated appearance in stack traces and other debug (toString) output. It would be cool if that can be improved somehow”_[P1]. However, P4 does say that developers should use more robust debugging techniques such as breakpoints.

Finding 2. We found seven reasons among API producers for introducing support for lambda expressions and functional interfaces. ‘Improving readability’ and ‘migrating to Java 8+’ are the most popular reasons. Challenges when making the transition regarded both social and technical aspects. The technical challenges were exacerbated by the implementation of lambda expressions in Java which required API producers to implement workarounds.

6 RQ₃: TOP APIs Without Lambdas

Mostly through email interviews, we investigate the reasons why 41 out of the top 50 APIs did not transition to Java 8+ or introduce support for lambda expressions. Out of these 41 APIs, for nine projects we could not find a development mailing list, seven had only a private one, and six did not respond to our emails. Developers from the remaining 19 APIs provided us with rich information. In the following, we present our findings ordered by the frequency with which they were mentioned by the interviewees on their reasons for *not* transitioning or introducing lambda support.

Backward compatibility. According to some API producers (P8, P12, P21), backward compatibility was very important to their APIs. These API producers did migrate their

API to Java 8, however, they do not change their interface to support lambda expressions as this is considered to be too big a burden to impose on consumers. The API producers prefer to prioritize ensuring backward compatibility over adopting new language features. This indicates that most top APIs are conservative when it comes to making changes and try to respect their consumers' needs. This sentiment was echoed by one survey respondent, who said that their API would not introduce support for lambdas to maintain backward compatibility.

Limited developer manpower. One limiting factor mentioned by some producers (P10, P15, P23) was the lack of enough available manpower. Due to the open-source nature of these projects, they were dependent on volunteers actively contributing code. Even if these projects wanted to make a change to the latest version they could not. We found this surprising since these APIs are among the most popularly used and actively maintained.

Most consumers use Java 7. According to some API producers who have not migrated their APIs to Java 8+ (P9, P14, P15), there is no need to support Java 8+. In their view, Java 7 and older versions of Java are still the most used versions. These API producers do poll their consumers to see whether a change in Java version supported would be acceptable, and in most cases, the consumers vote for staying with the older version of Java. This is surprising since Java 7 has already reached End Of Life (EOL). In the producers' view, Java 7 being declared EOL meant nothing. P10 asserts that Oracle would never force everyone to new versions because enterprise clients have remained on older versions of Java. This forces Oracle to continue releasing security patches for older versions. These producers do feel the Java ecosystem will move to newer versions, but only in several years. One of our survey respondents echoed this stating that it was a hard requirement to remain on Java 7 due to their consumer base.

No new useful language features. In the view of the API producers (P9, P10, P15), new language features did not matter when it came to deciding to change Java versions. They felt that consumer codebases can migrate to the latest version of the language to take advantage of new programming language features, but they would simply be restricted in the case of some of these third-party APIs. The API producers feel that such a shortfall is acceptable. This finding may suggest that the new features that Java language designers have been releasing at an increased release cadence over the last two years have little motivational impact on APIs migrating to the new versions of the language to support newer features.

Lambdas are not applicable. P18 mentioned that introducing lambda expressions in the Hamcrest API would make no sense because the API does not take any kind of lambda-compatible input. Hamcrest is typically used in conjunction with a testing API such as JUnit; if that API supports lambda expressions, Hamcrest can be used with lambda expressions as well. Similarly, according to P17, the H2 database API is constrained by the JDBC (Java Database Connector) specification, thus introducing lambda expressions in this context would not make sense. In the survey, one respondent indicated that in their development context it made no sense to migrate to Java 8+, another indicated that despite moving to Java 8+ lambda support did not apply to them.

Lambdas do not improve anything. P15 and P16, feel that the advantages of lambda expressions and functional interfaces improving performance and readability are overstated. P16 felt strongly that having less code with more lambda expressions made the code unintelligible. While this appears to be a subjective take on the implementation of lambdas in Java, this view had a direct impact on their decision-making.

Java is limited. One of the primary concerns of P10 and P11 was that Java and its current type system and generics are simply too limited to support the functional programming paradigm. The lack of monads, dependent types, and functors makes functional programming in Java limited. In fact, within the Apache ecosystem, there was a move to create a project called Apache Commons Functor, which would try to make Java more functional than it is. Currently, Java's implementation of lambda expressions and functional interfaces feels to our respondents more like syntactic sugar rather than a powerful functional tool. Despite this negative view of Java and its functional capabilities, the developers do concede that certain aspects of the Apache Commons ecosystem can take advantage of providing lambda support in certain parts of their various APIs.

Finding 3. Migrating to Java 8+ is not of priority for the majority of the top Java APIs. These APIs prefer to emphasize backward compatibility since several enterprise projects are still on Java 7. This landscape creates inertia to change in the Java ecosystem.

7 RQ4: The Consumers' Adoption

In our survey, we ask API consumers the frequency with which they have used a lambda expression in Java (not only limited to the APIs we study but also the JDK API). Only 12 out of 110 indicate that they never use lambda expressions. Instead, 45% and a further 38% report that they use lambda expressions occasionally and every time they can, respectively. The results of this question are in line with those reported by Mazinanian et al. (2017).

We ask the consumers as to what motivation behind an API introducing lambda expression support motivated them to adopt the latest version of the API and use its features (results in Fig. 7). We see that API supporting Java 8 (86% positive) and improving readability (84% positive) were the main drivers; this is in line with the motivations expressed

Fig. 7 Why API consumers (based on responses from 110 API consumers) adopt lambda expressions

Fig. 8 What influenced consumers (based on responses from 110 API consumers) to *not* adopt lambda expressions in the APIs they use.

by the API producers. Overall, these results confirm that API producers and consumers are aligned.

We ask consumers what has prevented them from adopting versions of an API that introduced support for lambda expressions (Fig. 8). Overall, other than concerns about the development effort, consumers report no other issue. Contrary to the findings by Nielebock et al. (2019), API consumers do not report that the use of lambda expressions slows down their code.

Finding 4. Consumers of APIs indicate that they use lambda expression in their Java code for the increased readability and standardizing of features.

8 Discussion

In the following, we reflect on some of our most compelling findings.

8.1 Java ecosystem has stagnated

Previous research on API breaking changes has shown that breaking changes are quite prevalent in development ecosystems (Bogart et al. 2016; Xavier et al. 2017; Linares-Vásquez et al. 2013; Raemaekers et al. 2014). API producers have generally not been

averse to changing the interface of some of their features, removing features, or deprecating features in their API.

Surprisingly, we observe that API producers are hesitant, instead, to change build platforms, *i.e.*, changing the minimum version of Java that they support. API producers prioritize backward compatibility over changing the API. Some of the API producers mention that supporting older versions of Java was a priority as many enterprise projects have not yet transitioned to a newer Java version. This issue is exacerbated by the fact that Oracle (the maintainer of Java) refuses to terminate support for the enterprise edition of Java 7 even though the standard edition has reached its end of life support (Oracle 2015).

In the case of APIs that have transitioned to Java 8+ and introduced support for Java's functional features (such as Spring projects and Mockito), these projects too have preferred to prioritize backward compatibility (in fact, Mockito officially still supports Java 6). They do not want to have their consumers make wholesale changes to how they develop. The changes they make to their interface are lightweight and allow their consumers to stick to the OOP paradigm and not introduce a single lambda expression.

This study indicates the beginning of an alarming trend in the Java ecosystem where Java projects appear to not keep up with the latest language developments by, instead, preferring to focus on backward compatibility. This intense focus on backwards compatibility is dangerous as over time it could lead to legacy systems (Bennett 1995) with few developers able to maintain them (Bennett and Rajlich 2000) and with developers moving to newer languages with the latest language features such as Kotlin which is Java Virtual Machine (JVM) based - thus retaining the benefits of the using the JVM while simultaneously having modern language design. There is precedence for this: the COBOL ecosystem has already gone through such an evolution. In this day and age, it is virtually impossible to find a developer fluent in COBOL (a fact that has recently had immense practical implications on the American economy) (Silverthorne 2020).

Recommendations and implications Researchers and Java language designers can learn from the inertia in the Java ecosystem. Java APIs appear hesitant to change their interfaces due to the fear of no longer being backward compatible. This points to a stagnation in the Java ecosystem. Researchers and Java language designers need to devise a way to arrest this behavior and entice Java developers to adapt to newer versions of Java with minimal fuss. This does not imply that change should be forced on Java developers. Instead, automatic semantic aware migration tools can help developers get over their hesitancy to adapt and spur increased adoption of new language features.

In the current state, based on our observations, Java APIs have not been agile in transitioning to support new Java language features. This inertia coupled with developers increasingly opting for new JVM-based languages such as Kotlin, might hurt Java's popularity, which is already sliding (TIOBE 2021).

8.2 Popular Java APIs are slowly adapting to Java 8

Java 8 was released in 2014, and it was one of the most eagerly anticipated releases as it introduced functional interfaces and lambda expressions (Oracle 2014). With this release, the JDK API was adapted to support lambda expressions when used with the streams abstraction added to the collections API. However, for Java developers to be able to develop end-to-end using lambdas, third-party APIs would also have to support lambda expressions. A lack of such support would make it difficult for lambda expressions in Java to be fully embraced.

Rogers' theory on the diffusion of innovations (Rogers 2010) sets out to explain the rate and extent of diffusion of a novelty. In the context of this work, we seek to understand the diffusion of Java 8's lambdas by determining the extent that top APIs support Java 8's functional interfaces. We observe that only nine out of the top 50 Java APIs support lambda expressions, in the five years since the release of Java 8. According to this theory, this does not necessarily indicate that the diffusion of lambdas has failed, because diffusion of innovations can take a long time (Bass 1980).

According to the diffusion of innovation theory, we are in the phase with early adopters. This is where certain APIs have made the change and adopted the latest features. One potential reason behind this is that making changes to the API is non-trivial. In the case of JUnit, the entire API had to be rewritten and Guava had to introduce major breaking changes to their API. It seems possible that API producers do not want to make the effort to start supporting Java 8+ for what are perceived as minor benefits if the API needs major changes. APIs such as Spring and Mockito might simply be outliers and can take advantage of the domain that they are in or that the design of the API ensured its lack of need to modify the interface. Other third-party APIs might simply not have this luxury.

This phase of early adoption has lasted 6 years (at this point) and might go on longer, which might suggest that the innovation might have failed. However, we cannot as yet definitively speak about failed diffusion of lambda expressions in Java. Based on the theory of diffusion of innovations, it is within the realms of possibility that—if this analysis would be repeated in a few years—one might observe that more APIs change their interface and introduce support for Java's functional features. We see that APIs such as Hibernate, SLF4J, and TestNG are currently in the process of making a change. In the Apache Commons ecosystem, there has been a debate about migrating to Java 8+ as well. Overall, there appears to be a very slow adoption of the new features.

From the consumer perspective, we observe that most consumers are ready to adopt new versions of the API that support functional interfaces. Furthermore, most consumers also report that they currently already develop using lambdas, thus indicating a familiarity with the functional programming features within Java. Blame for slow adaption of API interfaces cannot be blamed on the lack of demand/familiarity from the consumer end.

This finding is a potential stumbling block to the new features that Java plans on rolling out on a more frequent schedule. Java has moved to a six-month release cadence, which has resulted in older versions of Java reaching EOL earlier than before (*e.g.*, Java 10 has already reached EOL and we are currently on Java 14). With each new release of Java, new features such as the `var` keyword for lazy type inference, leftover features for lambda expressions, enhanced switch statements, and records are being introduced. Observing this slow adoption of Java 8+, the new features might also be adopted gradually (or not at all), which would imply that this newfound impetus from the Java language designers might be for nothing.

Recommendations and implications API producers who have not yet transitioned to Java 8+ will have to start considering making the move to newer versions of Java soon. This will be important as the early majority phase of adoption begins and the entire Java development ecosystem transitions to Java 8+. API producers can learn from this study about the cost and benefit of changing the API to support Java 8. For example, in the case of APIs such as Spring, the layered design of the API (where changes in the internals of the API did not change the API semantics or the external facing contracts) prevented the API from having to be rewritten to fully support Java 8 features. On the other hand, we observe that JUnit completely reimplemented their API in version 5. APIs wanting to change their

interface could choose either extreme when it comes to making a change or maybe a strategy in between.

We found evidence on how API design can have an impact on the number of breaking changes introduced by an API. API consumers could take this into account when adopting an API and use only those APIs whose interface is not volatile.

We provide researchers the first insight into the rate of change of interfaces to support the functional aspects of Java. Moreover, based on our findings on the pain points of adopting functional interfaces, further research can investigate how these can be resolved. Our study speaks to the larger acceptance of a new language feature and can inform future research to assess ecosystem-wise language feature adoption.

8.3 Drawbacks of lambdas in Java

When Java introduced lambda expressions in Java 8, the stated primary motivation was 1. improving code readability and 2. providing developers with performance benefits when developing in concurrent context (Oracle 2014; Goetz 2010, 2013). In this study, we confirm that improved readability is a motive behind APIs changing their interface.

However, Java's promise of performance improvement has fallen short of its goals. Even among the APIs that have changed their interface, we observe that some API producers complain about the performance aspect of Java's lambda expressions. This finding confirms that of previous work by Nielebock et al. (2019) which shows that developers often avoid lambda expressions in performance-dependent parts of their system.

Aside from performance issues, API producers who adapted to Java 8+ also complained about the fact that there was an inability to throw a checked expression from a lambda expression. This required a workaround on the part of the API producers. Furthermore, debugging issues in lambda expressions is non-trivial due to the obfuscated nature of the stack trace.

Among those API producers that do not change their API, there is skepticism surrounding the new features introduced. These API producers complain that there are still certain functional programming features that Java lacks: (1) limited type system, (2) support for monads, (3) support for functors and, (4) support for higher-order functions. In the opinion of these producers, the current implementation of lambda expressions in Java is nothing more than syntactic sugar.

Given both sets of API producers (those that have adapted to Java 8+ and those that have not) feel that there are limitations to Java's support for functional features, this suggests that Java's functional language features need further work. This does not necessarily imply that the current implementation is flawed beyond repair as lambda expressions are still being used in conjunction with the JDK API (Mazinanian et al. 2017). Java is also not the only programming language to have introduced lambda expressions and have them not work out as expected from the outset—the implementation of lambda expressions in C++ too had its flaws (Uesbeck et al. 2016).

Recommendations and implications Our research provides Java language designers with the needs and issues of the top API producers. Addressing these can only help spur the rate of adoption of lambda expressions in Java. Furthermore, implementing support for functional concepts (*e.g.*, monads) can help Java keep up with newer languages (*e.g.*, Kotlin) that already provide this on the JVM.

9 Related Work

Recently, researchers have focused on understanding the adoption and impact of programming language features and in particular lambda expressions. Meyerovich and Rabkin (2012) proposed a sociologically-grounded programming language theory, investigating which features have been adopted by programming languages over time and what was the outcome in terms of their popularity. As a result, they came up with a set of recommendations for language designers on how to incorporate new features. Several researchers (Dyer et al. 2014; Qiu et al. 2017; Wu et al. 2015; Khatchadourian and Masuhara 2018) have complemented this work by studying how Java programming language features have been adopted by developers. These works all find that in most cases the majority of the programming language features have not been adopted. Parnin et al. (2011) analyzed how Java generics have been integrated into open-source projects and showed that they are scarcely adopted. Hoppe and Hanenberg (2013) investigated the effect of Java generics on developer's productivity, discovering that they improve documentation and reduce extensibility, despite not reducing the time required for fixing type errors. Callaú et al. (2013) analyzed the usage of dynamic features in Smalltalk uncovering the reasons driving developers to use them, while Capek et al. (2015) explored the success of programming language features in the .NET ecosystem.

Our work focuses primarily on the adoption of lambda expressions in Java. In this vein, Mazinianian et al. (2017) studied the history of 241 Java projects to uncover the trends of adoption of lambda expression in the codebase. They complemented their findings with the opinions of developers gathered through surveys and interviews. This study found that developers prefer to use lambdas as they felt that it improved the readability of the code, helps avoid code duplication, and simulates lazy evaluation of functions. In contrast to this study, we focus on how and why third-party APIs have changed their interface to better support functional interfaces.

Uesbeck et al. (2016) compared the usage of lambda expressions in the replacement of explicit loops and functions in C++, finding that there is an initial negative impact in using lambda expressions, especially for novice developers. The difficulties that novice programmers have with lambda expressions was also highlighted by Hanenberg (2015). Myers et al. (2016a, b) identified a number of programmers' needs that involve the usage of lambda expressions and the usability of APIs in general. Nielebock et al. (2019) investigated the usage of lambda statements in thread-based concurrent systems, finding that their usage is very infrequent in such a context. Finally, Costa et al. (2017) investigated the performance of Java collections, finding that alternative implementations involving lambda expressions could save up to 50 times in terms of memory usage.

10 Limitations

In this section, we discuss the main limitations of our study and how we mitigated possible threats to validity.

Interviewer bias When performing semi-structured interviews, our own bias might have led interviewees to provide answers closer to our research expectations (Hildum and Brown 1956). To mitigate this issue, we triangulated our findings with a survey with API producers and consumers, finding similar conclusions. Given that we sent this survey out through various social media channels we cannot establish the exact response rate for the survey, but we can report that out of 252 people who started the survey, 110 concluded it appropriately.

Credibility Social desirability bias (Furnham 1986), *i.e.*, an interviewee's tendency to provide a socially acceptable response to appear in a positive light, may have impacted the developers' responses during the interviews and survey. To mitigate this issue, we informed participants that the responses would be anonymous and evaluated only in an aggregated form.

Manual Analyses A part of our study consisted of manual investigations aimed at building a taxonomy of how and why API producers support lambda expressions. This might introduce human error, which we mitigated by having the second author of the paper double-checking each step done by the first author when manually investigating the APIs. Cases of disagreement were discussed among all the authors to avoid interpretation errors and subjectiveness.

Originality Previous studies on lambda introduction in Java have only studied this from the perspective of the consumers of the JDK API. None of these studies have investigated whether lambda expressions have permeated the entire Java ecosystem or the issues and needs of API producers in this domain. We provide the first insight into this phenomenon along with a roadmap for other API producers to follow.

Generalizability We based our study on the top 50 most popular Java-based APIs. This was done because we expect these APIs to be stable, actively maintained, and have large consumer and producer communities. However, this set of 50 APIs does not represent all Java APIs. Another sample of selected APIs (for example 50 APIs that are seldom used or APIs internal to software companies) could result in a different set of observations. To mitigate this, we endeavored to send our survey out to a diverse set of API producers to see to what extent they agree with our interviewees. We found that there was a large degree of agreement between the producers interviewed and those surveyed.

Survey sampling We spread our survey using platforms such as Code Ranch, Twitter, and Reddit. Therefore the results based on our sample may not generalize to developers that work in other settings. Among our respondents, we observe a large geographical diversity (participants originate from three different continents), an average of nine years of experience, and the majority working on industry projects and not just open-source projects.

11 Conclusion

In this study, we have investigated the changes that the top 50 (ranked in terms of popularity of usage on GitHub) third-party Java APIs have made to introduce support for lambda expressions. We found that nine out of these 50 APIs have changed their interface. There are seven reasons behind these changes, the chief among which is that the usage of lambda expressions helps improve the readability of code.

At the same time, we found that 41 of the 50 top third-party APIs do not support Java 8+ and its features. There are seven reasons behind this behavior. The principal reason behind APIs not changing their interface is that they prefer to remain backward compatible. The API producers we interviewed mention that a lot of enterprise projects remain on Java 7, and hence being backward compatible is essential.

In addition to backward compatibility, the Java ecosystem has not adapted to the new Java 8+ features, due to the flaws with the implementation of lambda expressions in the

language. Despite these flaws and the low rate of diffusion of lambda expressions support, API consumers appear more than ready to develop using Java's functional features.

Overall, we found evidence that the Java ecosystem lacks the impetus to change. The emphasis on backward compatibility has resulted in a very slow pace when it comes to adopting new language features. This does not bode well for the Java language designers who are releasing new language features at a rapid rate. This behavior could result in Java becoming a legacy language (not the JVM) as developers move to new languages such as Kotlin (already adopted by the Android world) and Swift (adopted by the Apple ecosystem). This opens up the challenge for both researchers and practitioners to devise ways in which the Java ecosystem can be raised from its current stupor.

Appendix A

API Lambda Developers' Survey - Producer Perspective

Welcome

ID: 3

Dear Java Developer,

We are a team of researchers investigating Lambda Expressions and their introduction in APIs after the release of Java 8. We would like to understand your perception on the main challenges, strengths and weakness of adapting to the introduction of lambda expressions.

About the survey:

- It lasts 5 minutes
- Answers are collected and analyzed in an anonymized form

Thank you for your help with our scientific work!

Kind Regards.

ID: 101

Can we also share your anonymized answers in a publicly available dataset?*

(Selecting "yes" allows other researchers and the public to benefit from your answers and effort)

☐ Yes

☐ No

Programming language preference

Page exit logic: Skip / Disqualify Logic**IF:** #1 Question "Do you actively develop in Java?" is one of the following answers ("No") **THEN:** Disqualify and display: "Sorry, you do not qualify to take this survey."

ID: 65

1) Do you actively develop in Java?*

☐ Yes

☐ No

About you

Page exit logic: Skip / Disqualify Logic**IF:** #6 Question "When developing in Java, which of the following would best describe you?" is one of the following answers ("I am a consumer of APIs (I often use a libraries, APIs, or third-party component when I develop)") **THEN:** Disqualify and display: "Sorry, you do not qualify to take this survey."

ID: 8

2) How many years of Java programming experience do you have?

ID: 14

3) What setting do you work in?

☐ Open Source

☐ Industrial/Proprietary

☐ Other:: _____

ID: 15

4) How do you rate your expertise with:

	Poor	Fair	Good	Very good	Excellent
Object Oriented Paradigm	☐	☐	☐	☐	☐
Functional Paradigm	☐	☐	☐	☐	☐
Lambda Expressions	☐	☐	☐	☐	☐

ID: 100

5) Do you actively develop using Java 8 or higher?

- ☐ Yes
☐ No

ID: 69

6) When developing in Java, which of the following would best describe you?*

- ☐ I am a producer of APIs (I often create/maintain libraries, APIs, or third-party components)
☐ I am a consumer of APIs (I often use a library, API, or third-party component when I develop)
☐ Both of the above

ID: 20

7) Do you have experience with developing in a functional programming language?

- ☐ Lisp

- ☐ Scala
- ☐ Erlang
- ☐ Haskell
- ☐ JavaScript
- ☐ Others: _____

Page entry logic: This page will show when: #6 Question "When developing in Java, which of the following would best describe you?" is one of the following answers ("I am a producer of APIs (I often create/maintain libraries, APIs, or third-party components)", "Both of the above")

API Producer perspective on Lambda Expressions

Please answer the following questions from the perspective of an API producer.

Logic: Show/hide trigger exists.

ID: 46

8) Does the API you work on support Java 8+?*

- ☐ Yes
- ☐ No

Logic: Hidden unless: #8 Question "Does the API you work on support Java 8+?" is one of the following answers ("No")

ID: 83

9) Why do you not adopt Java 8+?

Logic: Show/hide trigger exists. Hidden unless: #8 Question "Does the API you work on support Java 8+?" is one of the following answers ("Yes")

ID: 50

10) Can your consumers use lambda expressions when interacting with your API?*

- ☐ Yes
- ☐ No
- ☐ Work in progress
- ☐ Other: _____*

ID: 92

11) What influenced you to introduce support for lambda expressions in your API?

	Not at all influential	Slightly influential	Somewhat influential	Very influential	Extremely influential	Not applicable
We wanted to have full support of Java 8 features	☐	☐	☐	☐	☐	☐
We wanted to standardize the functional support in our API	☐	☐	☐	☐	☐	☐
We wanted to provide our consumers with a performance improvement (e.g. lazy loading of log messages)	☐	☐	☐	☐	☐	☐

We wanted to improve the readability of the code	☐	☐	☐	☐	☐	☐
We wanted to introduce design patterns in our API	☐	☐	☐	☐	☐	☐
We wanted to throw checked exceptions from functional interfaces to make code more concise	☐	☐	☐	☐	☐	☐
We wanted to preserve functional purity (i.e. avoiding side effects)	☐	☐	☐	☐	☐	☐

Logic: Show/hide trigger exists. Hidden unless: #10 Question "Can your consumers use lambda expressions when interacting with your API?" is one of the following answers ("Yes", "Work in progress")

ID: 59

12) How much effort did it take to introduce support for lambda expressions in your API?

Effortless
☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
High degree of effort

Logic: Hidden unless: #12 Question "How much effort did it take to introduce support for lambda expressions in your API?" is one of the following answers ("4","5")

ID: 102

13) Why did the introduction of lambda support require this much effort?

Logic: Hidden unless: #10 Question "Can your consumers use lambda expressions when interacting with your API?" is one of the following answers ("Yes","Work in progress")

ID: 60

14) How likely are you to introduce lambda support to a larger extent in the future?

☐ Extremely unlikely ☐ Unlikely ☐ Neutral ☐ Likely ☐ Very likely

Logic: Hidden unless: #10 Question "Can your consumers use lambda expressions when interacting with your API?" is one of the following answers ("No")

ID: 64

15) Why do you not support lambda expressions in your API?

Debriefing

ID: 31

16) Feel free to add any further comment on this survey, topic, etc.

ID: 32

17) If you are available to be contacted for further research on this topic, please enter your email address.

Thank You

ID: 1

Thank you for taking our survey!

Appendix B

Producers' Interview Field Guide

Introduction

Introduction of the team

1. "Hi, my name is — from the ZEST lab at UZH University. These are my colleagues — and we are very thankful for your participation in this interview. Most of the questions we want to ask you are about your perceptions about this topic, so feel free to say all you want, in the way you wish.

Introduction of research topic at a high level

2. "We are conducting an empirical study on lambda expressions and their introduction in Java."

Overview & Background

Expected: 5 minutes

"Before starting to dive deeply in this argument, we would know more about you."

3. "Can you tell us the role you have in the XXX project and your experience during last years?"
4. "Can you talk about the evolution of the XXX project in the last four years? Purposes, challenges..."
5. "Are you working on other projects as well?"
6. "What is your main contribution to these projects? Do you usually have a specific role or deal with a specific part in these projects (like concurrency, security, design)?"

Exploring Specifics

Expected: 20 minutes

Motivation

7. "So, what can you say about the introduction of Functional Interfaces in the XXX Project? Can you recall the reasons behind introducing them?"
8. "what benefits do you think your project gets from providing Functional Interfaces? Are there any dark sides or drawbacks?"

JUnit Examples "Basically, it seems the main functional interfaces in JUnit, like `Executable`, `ThrowingSupplier`, and `ThrowingConsumer`, have been introduced to let developers to throw different kinds of Exception. Is it true? Are there any other reason behind?" "In JUnit 5 your team merged Jupiter in JUnit. Can you explain why? Did you face any problems in doing it?"

Spring Examples "Many of your functional interfaces and methods accepting lambda expressions seems to be oriented to replace ordinary Callback implementation. Can you talk about these?" "There are some functional interfaces used to change the type of an Object. We are talking about `Serializer`, `Deserializer` and `Converter`. Can you explain the advantage of using lambda expressions with them?" "Is there a connection between Lambda expression and asynchronous execution?"

Context & Environment

9. "Was there any aspect you considered before refactoring your code to introduce Functional Interfaces? Did you discuss this refactoring with your team?"
10. "Did your team introduce Functional Interfaces as soon as it was possible or did you wait for another moment? (if the second, why?)"
11. "Have you done something for the clients that use old releases of your APP? Do you support any kind of backward compatibility?"

JUnit Examples "Can you talk about JUnit Vintage?"

12. "What about the Java ecosystem? Are there any particular needs or lacks related to lambda expressions introduction?"

Effort vs Expectation

12. "what were the challenges faced when changing the nature of the interface? what were alternatives considered? and what was the impact of the decision taken?"
13. "Which expectations did/did you have regarding the use of Lambda Expressions by the clients of your API?"

Reflection & Projection

Expected: 5 minutes

14. "How do you see the role of Lambda Expressions in the evolution of your system?"
15. "Do you think Java/Oracle will do something to encourage the use of Lambda Expressions? Would you approve that? Why?"
16. "Do you think there are some aspects to improve in lambda expressions? What Java/Oracle should do? Why?"

Wrap-up

"So, we're going to close."

15. "Did we miss something? Is there anything you want to tell us?"
16. "Ok. Then, thank you again for your participation. We enjoyed this conversation, and we hope in a future collaboration with you and your team."

Appendix C

Email interview guide

1. Could you please explain what major changes have been made to in the last four years?
2. Can you explain as to how much of a challenge it was to adapt to Java 8 and why a decision was made to switch to Java 8+?
3. What major API changes were made when adopting Java 8's lambda expressions (with an example)?
4. Did your team introduce Java 8's Functional Interfaces as soon as it was possible or did you wait for another moment? (if the second, why?)
5. What benefits do you think your project gets from providing Functional Interfaces? Are there any drawbacks?
6. How do you see the role of Lambda Expressions in the evolution of your system?
7. To what extent do you expect your clients to adopt the Java 8 based features as opposed to older Guava way of doing things?
8. Do you think that lambda expressions in Java 8 are completely usable and fully mature? Can Java/Oracle do something to improve it and why?
9. To what extent is the decision making process on adopting new language features similar every time (for example was the discussion the same when talking about Java 8's lambdas and Java 9's project jigsaw)?
10. Would your project immediately adopt new (Java) language features and (Java) language versions?
11. How does your project ensure backwards compatibility (despite adhering to newer language standards)?

Acknowledgements Alberto Bacchelli and Fernando Petruccio gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529. Anand Ashok Sawant gratefully acknowledges support from NSF CISE (SHF LARGE) Grant No. 1414172, and from Sandia National Laboratories.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence,

and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Goetz B (2013) State of the lambda. [Online]. Available: <http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html>
- Mazinanian D, Ketkar A, Tsantalis N, Dig D (2017) Understanding the use of lambda expressions in java. *Proc ACM Programm Lang* 1(OOPSLA):85
- Lucas W, Bonifácio R, Canedo ED, Marcílio D, Lima F (2019) Does the introduction of lambda expressions improve the comprehension of java programs? In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, pp 187–196
- Goetz B (2010) Project lambda. [Online]. Available: <https://openjdk.java.net/projects/lambda/>
- Goetz B (2013) State of the lambda: Libraries edition. [Online]. Available: <https://cr.openjdk.java.net/~briangoetz/lambda/lambda-libraries-final.html>
- Johnson RB, Onwuegbuzie AJ (2004) Mixed methods research: a research paradigm whose time has come. *Educ Res* 33(7):14–26
- Gousios G, Spinellis D (2012) Ghtorrent: Github's data from a firehose. In: *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, pp 12–21
- Sawant AA (2017) A Bacchelli fine-grape: fine-grained api usage extractor—an approach and dataset to investigate api usage. *Empir Softw Eng* 22(3):1348–1371
- Fielding RT (1999) Shared leadership in the apache project. *Commun ACM* 42(4):42–42
- Iqbal A (2014) Understanding contributor to developer turnover patterns in oss projects: a case study of apache projects. *ISRN Software Engineering* 2014
- Blinded (2020) The indolent lambdification of java: Understanding the support for lambda expressions in the java ecosystem. [Online]. Available: https://www.dropbox.com/s/ld2cwkm4g41rl3v/replication_package.zip?dl=0
- Srivastava P, Hopwood N (2009) A practical iterative framework for qualitative data analysis. *Int J Qual Methods* 8(1):76–84
- Coleman G, O'Connor R (2007) Using grounded theory to understand software process improvement: a study of irish software product companies. *Inf Softw Technol* 49(6):654–667
- Bailar B, Bailey L, Stevens J (1977) Measures of interviewer bias and variance. *J Mark Res*:337–343
- Thorne S, Kirkham SR, MacDonald-Emes J (1997) Interpretive description: a noncategorical qualitative alternative for developing nursing knowledge. *Res Nursing Health* 20(2):169–177
- Meho LI (2006) E-mail interviewing in qualitative research: a methodological discussion. *J Amer Soc Inf Sci Technol* 57(10):1284–1295
- Flanigan TS, McFarlane E, Cook S (2008) Conducting survey research among physicians and other medical professionals: a review of current literature. In: *Proceedings of the Survey Research Methods Section, American Statistical Association*, vol 1, pp 4136–47
- Reddit (2019) <http://www.reddit.com>
- Java Code Ranch (2019) <http://www.https://coderanch.com/forums>
- Nielebock S, Heumüller R, Ortmeier F (2019) Programmers do not favor lambda expressions for concurrent object-oriented code. *Empir Softw Eng*:1–36
- Bogart C, Kästner C, Herbsleb J, Thung F (2016) How to break an api: cost negotiation and community values in three software ecosystems. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, pp. 109–120
- Xavier L, Brito A, Hora A, Valente MT (2017) Historical and impact analysis of api breaking changes: A large-scale study. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, pp 138–147
- Linares-Vásquez M, Bavota G, Bernal-Cárdenas C, Di Penta M, Oliveto R, Poshyanyk D (2013) Api change and fault proneness: a threat to the success of android apps. In: *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. ACM, pp 477–487
- Raemaekers S, Van Deursen A, Visser J (2014) Semantic versioning versus breaking changes: A study of the maven repository. In: *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, pp 215–224

- Oracle (2015) Java 7 Information, https://java.com/en/download/faq/java_7.xml
- Bennett K (1995) Legacy systems: Coping with success. *IEEE Softw* 12(1):19–23
- Bennett K, Rajlich VT (2000) Software maintenance and evolution: a roadmap. In: *Proceedings of the Conference on the Future of Software Engineering*, pp 73–87
- Silverthorne V (2020) Gitlab: How can we help solve the COBOL programmer shortage? <https://about.gitlab.com/blog/2020/04/23/cobol-programmer-shortage/>
- TIOBE (2021) Tiobe index may 2021. [Online]. Available: <https://www.tiobe.com/tiobe-index/>
- Oracle (2014) Java 8 release notes. [Online]. Available: <https://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html>
- Rogers EM (2010) *Diffusion of innovations*. Simon and Schuster
- Bass FM (1980) The relationship between diffusion rates, experience curves, and demand elasticities for consumer durable technological innovations. *J Bus*:S51–S67
- Uesbeck PM, Stefik A, Hanenberg S, Pedersen J, Daleiden P (2016) An empirical study on the impact of c++ lambdas and programmer experience. In: *Proceedings of the 38th International Conference on Software Engineering*. ACM, pp 760–771
- Meyerovich LA, Rabkin AS (2012) Socio-plt: Principles for programming language adoption. In: *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, pp 39–54
- Dyer R, Rajan H, Nguyen HA, Nguyen TN (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*. ACM, pp 779–790
- Qiu D, Li B, Barr ET, Su Z (2017) Understanding the syntactic rule usage in java. *J Syst Softw* 123:160–172
- Wu D, Chen L, Zhou Y, Xu B (2015) An empirical study on the adoption of c++ templates
- Khatchadourian R, Masuhara H (2018) Proactive empirical assessment of new language feature adoption via automated refactoring: The case of java 8 default methods, arXiv:1803.10198
- Parnin C, Bird C, Murphy-Hill E (2011) Java generics adoption: how new features are introduced, championed, or ignored. In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, pp 3–12
- Hoppe M, Hanenberg S (2013) Do developers benefit from generic types?: an empirical comparison of generic and raw types in java. *ACM SIGPLAN Not* 48(10):457–474
- Callaui O, Robbes R, Tanter É, Röthlisberger D (2013) How (and why) developers use the dynamic features of programming languages: the case of smalltalk. *Empir Softw Eng* 18(6):1156–1194
- Capek P, Kral E, Senkerik R (2015) Towards an empirical analysis of .net framework and c# language features' adoption. In: *2015 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, pp 865–866
- Hanenberg S (2015) Empirical, human-centered evaluation of programming and programming language constructs: Controlled experiments. In: *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, pp 45–72
- Myers BA, Ko AJ, LaToza TD, Yoon Y (2016a) Programmers are users too: Human-centered methods for improving programming tools. *Computer* 49(7):44–52
- Myers BA, Stylos J (2016b) Improving api usability. *Commun ACM* 59(6):62–69
- Costa D, Andrzejak A, Sebock J, Lo D (2017) Empirical study of usage and performance of java collections. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, pp 389–400
- Hildum DC, Brown RW (1956) Verbal reinforcement and interviewer bias. *J Abnormal Soc Psychol* 53(1):108
- Furnham A (1986) Response bias, social desirability and dissimulation. *Person Individ Differ* 7(3):385–400

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.