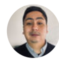
How do developers use the Java Stream API?

Conference Paper · August 2021

4 authors:


Joshua Nostas
Universidad Católica Boliviana San Pablo Cochabamba
3 PUBLICATIONS **4** CITATIONS
SEE PROFILE


Juan Pablo Sandoval Alcocer
Pontificia Universidad Católica de Chile
37 PUBLICATIONS **222** CITATIONS
SEE PROFILE


Diego Elias Costa
Concordia University Montreal
65 PUBLICATIONS **650** CITATIONS
SEE PROFILE


Alexandre Bergel
RelationalAI
229 PUBLICATIONS **1,805** CITATIONS
SEE PROFILE

How do developers use the Java Stream API?

Joshua Nostas[1][0000−0002−2924−8689], Juan Pablo Sandoval Alcocer[2][0000−0002−2924−8689], Diego Elias Costa[3][0000−0001−7084−2594], and Alexandre Bergel[4][0000−0001−8087−1903]

[1] Departamento de Ciencias Exactas e Ingenierías, Universidad Catolica Boliviana "San Pablo", Cochabamba, Bolivia
joshua.nostas@ucb.edu.bo

[2] Department of Computer Science, Pontificia Universidad Católica de Chile, Santiago, Chile

[3] Dept. of Computer Science and Software Engineering, Concordia University, Canada

[4] Department of Computer Science (DCC), University of Chile, Santiago, Chile

Abstract. Java 8 marked a shift in the Java development landscape by introducing functional-like concepts in its stream library. Java developers can now rely on stream pipelines to simplify data processing, reduce verbosity, easily enable parallel processing and increase the expressiveness of their code. While streams have seemingly positive effects in Java development, little is known to what extent Java developers have incorporated streams into their programs and the degree of adoption by the Java community of individual stream's features.
This paper presents a replication study on which we analyze the stream usage of 610 Java projects. Our findings show that the Java streams are used mostly by software libraries rather than regular applications. Developers rarely use parallel processing, and when they do so, they only superficially use parallelism features and most of the parallel streams are used on simple $forEach$ operations. The most common used pipelines involve *map*, $filter$ and *collect* operations. We carefully describe a number of stream idioms we identified, and detail how we addressed the challenges we faced to complete our study. Our findings will help developers at (i) making better decisions about which features to consider when improving the API and (ii) supporting stream-related IDEs features, such as refactoring.

Keywords: Software Quality · Software Maintenance · Java Streams · Empirical Study

1 Introduction

While the notion of stream processing has been around for decades [1], the Java 8 released in 2014 officially introduces this paradigm to Java programming with the stream library. The stream library provides a concise API for processing elements (objects and primitives) described as a *pipeline*, made of aggregated operations (*e.g.,* `map` and `filter`). Pipelines aim at supporting a declarative programming style: the code focuses on "what" it does as opposed to "how" it is supposed to do it. Through the API proposed by stream, a series of collections operations can be performed with just a few lines of code, increasing code comprehension and allowing developers to safely exploit multi-threaded processing through the use of parallel streams.

Java stream library overview. A *stream* is basically a view on a sequence of elements (objects or primitives), organized by the underlying data structure, a *stream source*. A stream source can be of any data type that implements the interface `java.util.Spliterator`, collections (including arrays), and I/O channels. Developers process a stream by defining *stream pipelines*, composed by functional *operations*, for which the typically are `filter` and `map`, as shown in the example of Listing 1.1.

```
int average = studentsList.stream()
                          .filter(Student::hasPassedFinalExam)
                          .mapToInt(Student::score)
                          .average();
```

Listing 1.1. A stream pipeline that calculates the average score of students that have passed the final exam.

Stream pipelines are designed to be used in a functional style, thus favoring operation composition. Each operation is implemented in a way that the stream source is never modified, and the result of a pipeline is stored on newly created object, making them natural to parallel processing. However, some operations may receive a *behavioral parameter*, a parameter that describes a user-specified behavior. In the example of Listing 1.1, the method `Student::hasPassedFinalExam` is such a behavioral parameter.

To embrace this newly introduced library, numerous tools have adopted lambdas and the stream library, and researchers have proposed new methods to facilitate the adoption of the benefits of stream processing. Modern Java IDEs, including IntelliJ IDEA, provide an extended support to refactor old Java code to use Java 8 functional features [4], like collapsing `for` loops to the more expressive stream pipelines. In research, some focus has been put to address the potential performance benefits of parallel streams [11]. Streams have also reported to be a major driver for the increase of Lambda functions in Java programming [16], due to their benefits in expressiveness and convenience.

Despite the extensive support of tools and methods, there is little empirical evidence on how developers have adopted stream processing into their Java programs and what are the most used features in practice. Some studies have investigated the broad scope of adoption of lambdas in Java [16], a critical aspect when using behavioral parameters and therefore streams. However, having an empirical study tailored to the usage of streams will help guide the Java community at better providing effective updates to the Stream API, foment better tools and help the research community at focusing on the pressing issues facing adopters of the stream library.

Partial replication. In 2020, Khatchadourian *et al.* [12] presented an empirical study on the use of the stream library in 34 Java projects and 719 code patches. To track streams and their attributes Khatchadourian *et al.* use a series of labeled transitions systems, static analysis and type-state analysis. A small set of project make possible to fully analyze stream usage in a reasonable amount of time. However, it may limit the generality of the findings. To address this threat, our paper partially replicate the effort of Khatchadourian *et al.*. In particular, we analyze 610 java project selected from a initial bag of 10,000 open source Java projects hosted on GitHub. The scope of our effort is slight different, as such, we limit our analysis to track stream pipelines fully declared within a method, excluding inter-procedural stream operations. In particular, we focus our analysis on answering the following research questions:

– *RQ1 - What is the trend of the stream library adoption in Java projects?* Getting a picture on how do Java developers are adopting the stream library is crucial to determine the importance of the library within the Java ecosystem.
– *RQ2 - How do Java developers use streams in their projects?* In particular, we are interested in the following questions: *What is the most commonly used stream operations? What are the most commonly used pipelines?,* and*, What are the common operations that developers do using streams?* Understanding how the API is used in practice is key for the stream library maintainers to understand expectations and concern from the Java community, in addition to improve the most used subset of operations and pipelines. Likewise, researchers that have the interest on researching the Streams API, may focus on the more prevalent cases.

Findings. Our study reveals findings related to stream pipelines usage that largely match and complement Khatchadourian *et al.* study:

– Similarly than Khatchadourian *et al.* we found that developers rarely rely on parallel processing, and that most of the parallel streams are used on simple `forEach` operations.
– The most common used non-parallel pipelines involve map, filter and collect operations. These results largely match with Khatchadourian *et al.*, which also shows that developers tend to favor more simplistic (linear) operations rather than more specialized non-scalar reductions.
– Complementary to Khatchadourian *et al.*, we find that most of the project under analysis are libraries or tools for developers, only 10% of the projects correspond to conventional applications.
– The most used Collector operations are `toList`, `toSet`, `joining`, and `toMap`. While the API provides more advanced collector operations, they are not commonly used by developers.

The following sections details the related work, the methodology we adopted, and our results.

2 Related Work

Numerous studies have investigated how developers adopt language and API features. In most of the works we discuss in this section, researchers adopt a similar methodology to ours, from selecting projects based on popularity criteria to mining software repositories to quantitatively assess the level of adoption of language features from developers. Particular effort has been put towards understanding how developers adopt unsafe language features, such as breaking type safety in Go [7], Rust [9] and using the infamous `goto` command in C [18]. In the context of Java programming, some studies have investigated the usage of Unsafe APIs [15], and dynamic type casting [14]. In both studies [14, 15], researchers report that the use of unsafe library features is widespread in the Java ecosystem. Developers often trade compiler checks for better flexibility, putting their programs at a higher risk of runtime errors. Other studies focuses on source code aspects related to memory consumption [3], performance regressions [20–22], or complexity [8, 17].

Some works that have investigated how developers use the Java collections API, a framework that is directly related to how developers use streams. Costa *et al.* [6] investigated how developers select Java collections in their projects. While the study showed that developers could benefit from using more specialized data structures for better performance, developers only rarely go beyond the general-purposed collection types. As such, several tools have been proposed to better guide developers in selecting their data structures in Java for better time and memory allocation [2, 5], and better energy consumption [19]. This finding that developers only rarely tune their collections has a similar parallel to our set of findings. In our study, we found that only in rare occasions developers make use of more complex stream operations and the parallelism of stream pipelines. Hence, there is the need for tools that can better guide developers at exploring the benefits of stream parallelism.

Similarly to the Streams API, lambdas have been introduced in Java 8, in an effort to enable Java developers to use functional idioms. The work of Mazinanian *et al.* have investigated the usage of lambda functions in Java program, including their usage in stream pipelines [16]. This study reported that streams are frequently used as replacement to for-loops, as they provide a more concise and easy-to-read idiom. However, the profiling of stream features used by developers remained out the paper's scope.

The work that is most related to ours is the work of Khatchadourian *et al.* [12], in which we partially replicate in this study. Khatchadourian *et al.* have investigated the use and misuse of the java stream API over 34 java projects [12]. Similarly to our findings, they have reported that parallelization is seldom used and that developers frequently use ordered streams, which are not optimal for parallelism. Khatchadourian *et al.* also reported that developers favor more straightforward stream pipelines, which corroborated with our findings that the most frequently used pipelines contain map-collect or filter-collect operation chains. Developers use functional-like streams to process data but collect them back to the imperative programming style they are most accustomed to. We complement their work by confirming part of their results with an analysis a the analysis of a large set of 610 projects. Furthermore, we investigate what types of projects use Java streams which help us understand what profile of projects tend to use streams.

3 Experimental Setup

This section highlights some aspects of the methodology we used to answer the two research questions stated above.

3.1 Methodology

To carry out the mining of stream API usage in a large corpus of Java applications, we use the following 3-steps process:

1. *Project selections* – Our very first step is to select GitHub repositories of Java software projects, which we consider relevant for the scope of our study.
2. *Detect* – To detect the stream pipeline usage, we analyze the abstract syntax tree (AST) of each Java file for all selected projects. Stream usage, expressed in term of pipelines, are extracted from the AST. In particular, the source code mining focuses on extracting components of a pipeline: stream factories (`studentsList.stream()` in Listing 1.1), intermediate operations (`filter, mapToInt`), and terminal operations (`average`).
3. *Categorize* – We quantitatively and qualitatively analyze stream pipelines and characterize their usage.

The following sections details each one of these steps.

3.2 Challenges of Using Type Inference

We consider a stream API usage as a method call performed on a stream object. As such, determining whether or not an object is a stream is key in our analysis. Since most GitHub repositories contain application source code and rarely contain the result of a compilation process, we need to identify stream API usage by solely inspecting source code. Type inferencers are tools designed to determine the type of each object and the signature of each method call. Using a type inferencer to identify stream usage is therefore appealing. Despite their solid theoretical foundation, using type inferencers in our context suffers from two different aspects:

Project Dependencies. The precision of the type inference heavily depends on the resolution of dependent libraries used by each project. However, many projects in GitHub do not have their dependencies explicitly declared. This fact may limit the accuracy of our study. Previous studies show that automatically download all the dependencies of great portion of the projects is an open problem [13].

Overhead. Infering the type of objects from the source code of a large project is time consuming. Consider the JavaSymbolSolver type inferencer[5]. Computing

[5] `https://github.com/javaparser/javasymbolsolver`

the type of each variable and the signature of each method call takes hours for any sizeable number of Java source code files. A rough estimation based on an initial estimation indicates that type inferencing 10k projects takes up to 48 days of computation[6].

Because of the aforementioned challenges, we adopted a sequentially staged process to investigate stream usage in a large number of projects. The goal of this style of methodology is to reduce the overhead of running type inference analysis, by removing projects that clearly do not rely on streams. We describe this in more detail in Section 3.3.

3.3 Sequential Staged Project Filtering

We start our study with a initial sample of 10,000 most starred Java projects from GitHub. Their stars range from 142 to 44,779 stars. Stars on GitHub are used as a token of appreciation from their users and also indicates how often people tagged projects for later exploration. Hence, stars are considered as a good proxy for project popularity and is commonly used on empirical studies to filter out unpopular projects [6, 7].

Initial project filtering. To discard toy or small projects, we initially select for further analysis projects that meet following criteria:

- Projects not archived, disabled or forked, as these projects do not have current development activity.
- Projects with 3 or more contributors.
- Projects with more than 50 commits.

After applying the filter, from the initial 10,000 projects, we were left with 5,386 projects for further analysis.

Data Cleaning. To discard educational or example projects, the first three authors (with experience in Java development) manually read the project name and description to categorize each one of these 5,386 projects. We discard projects that were categorized by at least two authors as toy, educational, or example projects. In total, we discard 317 projects. Therefore, after this step our sample is reduced from 5,386 projects to 5,096 projects.

Sequential Staged Filtering. Due to the two challenges described in (Section 3.2), we cannot blindly run a type inference on our projects. Instead, we adopt a *sequential staged filtering* of the projects: from the set of 5,096 Java projects, we apply a sequence of filtering and expansion techniques: a sequence of keyword filtering, data cleaning, resolving dependencies, and type inference filtering.

Keyword Filtering. Downloading the dependencies and inferring the type of all method calls in our initial set of projects is too costly to carryout. Instead,

[6] It takes 28 hours to process 7,700 Java source code files, and the 10k projects contain 317,032 source files.

we perform a *keyword filtering* on the projects that consists in reviewing each method call of all the Java projects. The Java stream library provides a number of ways to directly create Stream objects, the most common are: `of`, `stream` and `parallelStream`. This keyword filtering step consists in detecting projects that have at least one method call with these keywords. As a result, we found 2,189 projects that meet this criteria.

Download Dependencies. We automatically download all project dependencies that use *Gradle* and *Maven*. Our automation uses two standard commands to download the dependencies[7]. From the 2,189 projects, 1,193 use *Maven* for dependency management, 908 use *Gradle*, and 88 use another project management tool. Using our process, we successfully download the dependency of 851 projects, 682 in Maven, and 169 in Gradle. Since our analysis highly depends on being able to determine if a method call is related to a stream object, we use these 851 project as a base of our study.

Type Inference Filtering. We use the JavaSymbolSolver type inferencer to determine the type of each method call in the 851 projects. From total method calls, 17% of them cannot be inferred. The reason is because the current type inference libraries have also a number of limitations in particular cases (*e.g.,* high polymorphic operations and reflection). After the type inference, we found that 241 projects do not contain any method call related to stream. Therefore these are considered false positives and were discarded from our study. Finally, we consider the remaining **610 projects** as the target projects for our stream usage study. You will find the information about the projects under study online [8].

3.4 Detecting Stream API Usage

We use the Java Parser library to analyze the projects under study. We parse all the files of each project, build an Abstract Syntax Tree and look for all method calls nodes in the tree that are performed on a stream object.

Stream Usage Detection. We categorize the method calls as follows: 1) *Stream factory*, a method call that returns a stream but the receiver object is not a stream; 2) *Stream intermediate operation*, a method call that returns a stream and the receiver is a stream, 3) *Stream terminal operation*, a method call that does not return a stream but the receiver is a stream.

Pipeline Detection. We define a pipeline a sequence of method calls that follow the pattern: stream factory, followed by a sequence of intermediate operations, and ended by a terminal operation.

[7] `mvn dependency: copy-dependencies` and
`gradle dependencies`

[8] `https://bit.ly/2UPiGIO`

3.5 Categorization

To answer our research questions, we categorize the 610 projects and the stream usage as follows:

Projects Categorization. Two authors carefully read the project title, description and project page to categorize the projects into one of the two categories:

- *Frameworks-Libraries-&-Tools*: programs that developers use to create, debug, maintain, or otherwise support other programs, and programs esigned to be reused by other programs, often for software development.
- *Applications:* conventional applications, which do not fit in any of the other categories; this is the overwhelming majority

First, authors categorize projects separately. Then, they compare their categories and when to a consensus about which categories assign to each project.

Pipeline Categorization. We consider a pipeline as a method call chain on a stream object. For this categorization, we consider the set of consecutive operations done over a stream. This categorization was done automatically using a java parser and a type inference. We consider only operations performed over a stream object as receiver.

4 Results

4.1 What is the stream usage trend?

We present in Figure 1 a stacked bar plot showing (i) the number of projects created by year and (ii) the portion of these projects resulting from the keyword filtering and type filtering. Our approach **estimate that the portion of the projects that use at least once the stream API is about 21%** $(2,189/10,000)$. Note the real proportion of stream adoption is likely to be smaller since we cannot efficiently avoid false positives without downloading dependencies. The figure also indicates that the share is in constant increase over the year, reflecting **an incremental and steady adoption of the stream library by practitioners**.

Regarding the type of projects that use streams, we present in Table 1 the distribution of the 610 projects that use streams across the two categories. Our results indicate that the **majority of projects (90%) that use streams are Frameworks, Libraries and Tools, rather than Java applications.** Note that the proportion may differ in the 2,189 projects.

4.2 What are the most used pipelines?

We present in Figure 2 and Figure 3 the most used stream pipelines from the 610 projects in our dataset. We found that only 113 (18%) of them create at least one parallel stream object, while the remaining 497 (82%) use solely non-parallel streams. Furthermore, from all the stream factories in these 113 projects,

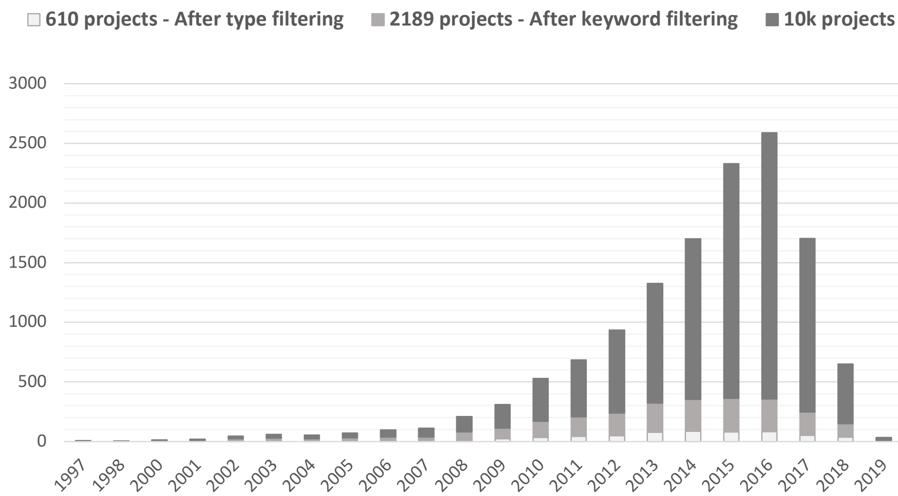


Fig. 1. Projects by Creation Year

Table 1. Number of projects by category that have at least one stream pipeline.

Category	**# Projects**	**%**
Frameworks, Libraries and Tools	549	90%
Applications	61	10%

only 20% of them create a parallel stream. This finding suggests that even in projects that use parallel streams, its usage is very infrequent. We conclude that **developers rarely rely on parallel processing** despite the large number of features for parallel computing provided by the Java Stream API. Figure 2 lists the five most popular sequential pipelines and Figure 3 lists the five most popular parallel pipelines. The figure also indicates that **most of the parallel streams are used on simple `forEach` operations**, thus avoiding a large number of expressive features for parallelism.

On the other hand, Figure 2 also shows that **the most common used non-parallel pipelines involve `map`, `filter` and `collect` operations**. These are arguably the most straightforward stream operations in the API, showing that developers prefer simple pipelines that are easy to understand and maintain. The most frequently used pipeline `stream-map-collect` indicates that developers frequently use streams to process data from one collection to the other. For instance, consider the following example:

```
private boolean checkRole(User user, Set<Role> expectedRoles) {
  Set<String> roleNames = expectedRoles.
  stream().map(Role::getRoleName).collect(Collectors.toSet());
  return authorityRepository.checkRole(user.getUsername(), roleNames);
```

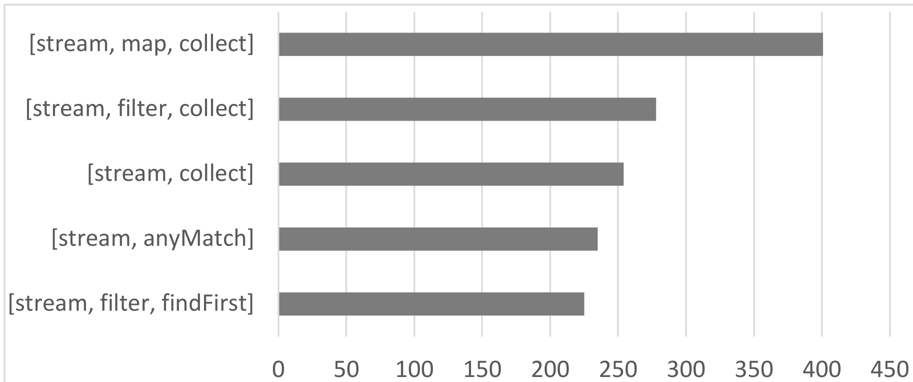


Fig. 2. Most commonly used stream pipelines with sequential processing. The bar width represents number of project that have at least one instance of the pipeline.

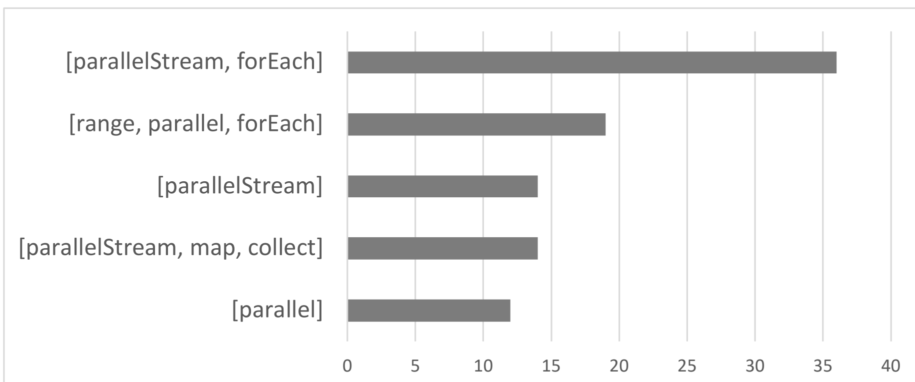


Fig. 3. Most commonly used stream pipelines with parallel processing. The bar width represents number of project that have at least one instance of the pipeline.

```
}
```

The example shows a snippet containing a stream pipeline that creates a stream from a set of role objects and return it as a set. Therefore, the stream is used as an intermediate step, and the rest of the execution use a collection as a main data structure.

Figure 3 also shows that **the most commonly used parallel pipelines involve `forEach`, `map` and `collect` operations**. Once again, parallelism is used in its most simple form, the most frequent pipeline `parallelStream-forEach` is used to process elements using a single behavioral function. For instance, consider the following example:

```
private void stopAllBrowsers() {
    if (allSessions == null) {
```

```
        return;
    }
    allSessions.getAllSessions().parallelStream()
        .forEach(session -> {
          try {
            session.stop();
          } catch (Exception ignored) {
            // Ignored
          }
        });
}
```

The example depicts a snippet where developers create a parallel stream from a set of session objects and stops each of them.

Stream Collectors. The Java Collector class provides a number of expressive reduction operations. These operations are commonly used together with the `collect` operations. The most used Collector operators are `toList` (9079), `toSet` (2000), `joining` (1326), `toMap` (981) and `groupingBy` (319). Although the library offers an efficient `mapping` function, developers use a `map` followed by a `collect`, thus representing a missed opportunity for using a dedicated optimization.

Partial Stream pipelines. There are a number of partial pipelines. We consider partial stream pipelines to a method call chain that do not end with a terminal operation. For instance, `of` and `stream`, these represent program statements where a stream was instantiated, but none terminal operation was performed on them. Normally, these streams are used in different methods. We plan to consider inter-procedural stream operations as future work.

5 Conclusion & Future Work

As far as we are aware of, this paper describes the largest effort in mining Java stream usage. We initially analyzed 10,000 popular Java projects and considered 610 projects for a deeper analysis. Results of such a large scale analysis can be used to complement recent effort in studying Java streams [12]. For example, recent works [10] have proposed a sophisticated refactoring tooling of the pipelines `sort-collect`, `parallel-map-collect`, and `unordered-distinct`. Our empirical study shows that these pipelines are extremely rare, thus mitigating the possible impact of such a refactoring. As future work, we plan to pursue our effort in considering partial stream pipelines.

Acknowledgments. Bergel thanks ANID Fondecyt 1200067 for partially sponsoring this work.

References

1. Abelson, H., Sussman, G.J.: Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA, USA, 2nd edn. (1996)

2. Basios, M., Li, L., Wu, F., Kanthan, L., Barr, E.T.: Darwinian data structure selection. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 118–128. ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA (2018)
3. Bergel, A., Infante, A., Maass, S., Sandoval Alcocer, J.P.: Reducing resource consumption of expandable collections: The pharo case. Science of Computer Programming **161**, 34–56 (2018), advances in Dynamic Languages
4. Blog, I.I.: Intellij idea inspection settings for refactoring to java 8, (Accessed on 01/14/2020)
5. Costa, D., Andrzejak, A.: Collectionswitch: A framework for efficient and dynamic collection selection. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization. p. 16–26. CGO 2018, Association for Computing Machinery, New York, NY, USA (2018)
6. Costa, D., Andrzejak, A., Seboek, J., Lo, D.: Empirical study of usage and performance of java collections. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering. p. 389–400. ICPE '17, Association for Computing Machinery, New York, NY, USA (2017)
7. Costa, D., Mujahid, S., Abdalkareem, R., Shihab, E.: Breaking type-safety in go: An empirical study on the usage of the unsafe package. IEEE Transactions on Software Engineering (01), 1–1 (feb 5555)
8. Crasso, M., Mateos, C., Zunino, A., Misra, S., Polvorin, P.: Assessing cognitive complexity in java-based object-oriented systems: Metrics and tool support. Computing and Informatics **35** (01 2014)
9. Evans, A.N., Campbell, B., Soffa, M.L.: Is rust used safely by software developers? In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. p. 246–257. ICSE '20, Association for Computing Machinery, New York, NY, USA (2020)
10. Khatchadourian, R., Tang, Y., Bagherzadeh, M., Ahmed, S.: [engineering paper] a tool for optimizing java 8 stream software via automated refactoring. In: 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 34–39 (Sep 2018)
11. Khatchadourian, R., Tang, Y., Bagherzadeh, M., Ahmed, S.: Safe automated refactoring for intelligent parallelization of java 8 streams. In: Proceedings of the 41st International Conference on Software Engineering. p. 619–630. IEEE Press (2019)
12. Khatchadourian, R., Tang, Y., Bagherzadeh, M., Ray, B.: An empirical study on the use and misuse of java 8 streams. In: Wehrheim, H., Cabot, J. (eds.) Fundamental Approaches to Software Engineering. pp. 97–118. Springer International Publishing, Cham (2020)
13. Martins, P., Achar, R., V. Lopes, C.: 50k-c: A dataset of compilable, and compiled, java projects. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR). pp. 1–5 (May 2018)
14. Mastrangelo, L., Hauswirth, M., Nystrom, N.: Casting about in the dark: An empirical study of cast operations in java programs. Proceedings of the ACM on Programming Languages **3**(OOPSLA) (Oct 2019)
15. Mastrangelo, L., Ponzanelli, L., Mocci, A., Lanza, M., Hauswirth, M., Nystrom, N.: Use at your own risk: The java unsafe api in the wild. SIGPLAN Notices **50**(10), 695–710 (Oct 2015)
16. Mazinanian, D., Ketkar, A., Tsantalis, N., Dig, D.: Understanding the use of lambda expressions in java. Proc. ACM Program. Lang. **1**(OOPSLA) (Oct 2017)

17. Misra, S., Cafer, F., Akman, I., Fernandez-Sanz, L.: Multi-paradigm metric and its applicability on java projects. Acta Polytechnica Hungarica **10**, 203–220 (01 2013)
18. Nagappan, M., Robbes, R., Kamei, Y., Tanter, E., McIntosh, S., Mockus, A., Hassan, A.E.: An empirical study of goto in c code from github repositories. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. p. 404–414. ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA (2015)
19. Oliveira, W., Oliveira, R., Castor, F., Fernandes, B., Pinto, G.: Recommending energy-efficient java collections. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). pp. 160–170 (2019)
20. Sandoval Alcocer, J.P., Bergel, A.: Tracking down performance variation against source code evolution. In: Proceedings of the 11th Symposium on Dynamic Languages. p. 129–139. DLS 2015, Association for Computing Machinery, New York, NY, USA (2015)
21. Sandoval Alcocer, J.P., Bergel, A., Valente, M.T.: Learning from source code history to identify performance failures. In: Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering. p. 37–48. ICPE '16, Association for Computing Machinery, New York, NY, USA (2016)
22. Sandoval Alcocer, J.P., Bergel, A., Valente, M.T.: Prioritizing versions for performance regression testing: The pharo case. Science of Computer Programming **191**, 102415 (2020)