Understanding Java Usability by Mining GitHub Repositories

Mark J. Lemay
Boston University, Boston, MA, USA
lemay@bu.edu

Abstract

There is a need for better empirical methods in programming language design. This paper addresses that need by demonstrating how, by observing publicly available Java source code, we can infer usage and usability issues with the Java language. In this study, 1,746 GitHub projects were analyzed and some basic usage facts are reported.

2012 ACM Subject Classification Human-centered computing → Empirical studies in HCI

Keywords and phrases programming languages, usability, data mining

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2018.2

Acknowledgements Thanks to my advisor Hongwei Xi for the encouragement to publish this research, the anonymous reviews who provided immense constructive feedback and to Stephanie Savir for correcting numerous errors.

1 Introduction

What makes a good programming language? While nearly every programmer has an opinion on what makes a programming language good, finding objective answers to this question is hard. While theoretical studies, like those in type theory, are important for the future of programming, theoretical properties like type safety and powerful constructs like dependent types have made little impact on mainstream software engineering. Theory may be necessary for "good" programming languages, but it is clearly not sufficient.

Another approach to measuring the "goodness" of languages comes from user studies. These studies generally take real people and have them perform some specific task using the language technology in question. While this approach has significantly improved some aspects of the mainstream programming experience[2], and hinted at interesting ways to develop a language[16] the scope of user studies is necessarily limited.

This paper proposes another way to measure the quality of programming languages: by analyzing publicly available source code artifacts such as those available on GitHub[1]. This approach alleviates many of the problems with user studies: very large samples are possible, the contributors are more likely to be experienced developers and projects are frequently large and realistic[2]. However, the data mining approach brings about new issues. We cannot directly ask users about their experiences, so there must be additional interpretation. Are programmers avoiding some features they find confusing and error prone? Or are they using an inconvenient feature frequently because the language is forcing them to? Aside from

[1] `https://github.com/`, GitHub is popular site for open source projects based on the git version control system

[2] This study includes popular libraries like spring-boot, guava, selenium, jenkins, junit and projects from organizations such as Netflix, Oracle, Paypal, Facebook and Google.

this, the programming language features we are interested in analyzing maybe underutilized for reasons other than their inherent usability: there might be a lack of education or features might be used indirectly through libraries. For instance, when observing the looping constructs of Java we see that the `do while` loop is very unpopular. This may be because of a lack of awareness of the future, rather then its inherent awkwardness. While conducting this research I found obscure Java features I was unaware of. Underlying language paradigms can also drastically change the usability of a feature. For instance, Haskell has no inherent notion of state, so a primitive "while" construct would not make sense. Hopefully data mining can provide a vastly different perspective from usability studies and theory that can help independently inform programming language design.

In this paper I mined the 1,746 most popular Java projects from GitHub. From this sample we can conclude a number of basic but novel facts about Java language usage. These facts will then be used to draw conclusions about the usability of different Java features, and suggest pain points that future languages should address. Additionally this paper demonstrates a simple method for analyzing Java files through the Eclipse IDE's parser[3].

2 Methodology

Java is one of the most popular programming languages and it has a large ecosystem of projects that can be analyzed. This makes Java a good candidate for data mining[4]. In addition, the Eclipse IDE's Java parser allows very precise information to be drawn from even very malformed files. Java is a relatively conservative language and invests heavily in backwards compatibility, so projects using very old versions of Java can be analyzed with little ambiguity.

In this study, the top Java GitHub repositories determined by star count[5] were selected by the GitHub search API and downloaded using an archive link. The most popular projects where chosen to avoid the many forks and copies of projects, and because it is likely that popular projects are more widely used and maintained by experienced developers. Some projects were randomly skipped over because of pagination issues with the search API[6]. Every repository that was available had each of its Java files parsed by the Eclipse IDE's parser into a traversable AST with the parser's best guess at partial type information. Because the Eclipse parser is designed to work with malformed files, it avoids several the issues other data mining projects have suffered from. This includes needing to know how to build the project, needing to resolve the correct version of library dependencies, and needing to find the correct version of the Java run time and Java language version (which is often not disclosed by build tools). Feature usages were then queried and aggregated.

3 Results

1,746 projects containing a total of 614,816 `.java` files and 97,758,514 lines of code was analyzed. The average Java file is 159 lines long.

---

[3] `https://www.eclipse.org/jdt/core/`

[4] I spent several years as a Java developer so I was experienced in the nuances of the language and the ecosystem.

[5] At the time of the download the most popular project had 37432 stars. The least popular project had 52 Stars.

[6] Fewer than 2% of projects were skipped. More careful scripts could avoid most of this error, but there will always be potential issues pulling data that is changing in real time while also respecting GitHub's rate limit.

Table 1 Control flow constructs.

Construct	Count/File[a]	Count
Return	6.2	3,825,353
If	4.7	2,878,814
Throw	0.74	455,898
Try	0.72	442,698
Catch Clause	0.64	396,475
For	0.52	317,699
Enhanced For[b]	0.44	271,766
Break	0.38	230,681
While	0.18	111,966
Switch	0.11	72,995
Continue	0.078	48,136
Synchronized	0.061	37,436
Do While	0.016	9,948
Labeled	0.0072	4,415

[a] This assumes that the count is averaged over all files in the sample, it is very likely some features are clustered together in non uniform ways.

[b] Added in Java 5, this variant of for loop allows collections to be traversed by element without an index.

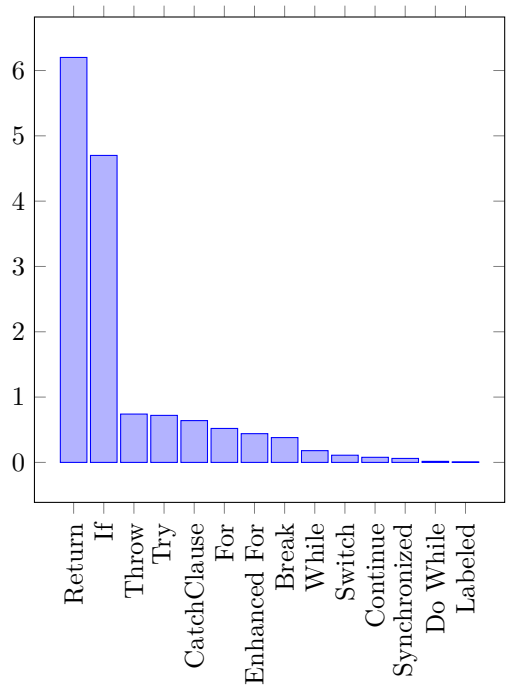

Figure 1 Control flow constructs, by Count/File

3.1 The Java Language

3.1.1 Control flow constructs

Java allows for several control flow constructs such as `for` loops, `switch` statements, `throw` and `catch` statements, and `return` statements. Table 1 shows the count of each construct from every `.java` file in the sample.

`return` is essentially required for writing Java functions, unsurprisingly it sees the heaviest usage.

`for` loops are by far the most popular looping construct. `while` loops are much less popular, though still used. Language authors should consider not including `do while` loops, since they seem to be avoided in practice. The obscure loop labeling construct that allows specific breaking of nested loops should be avoided in future languages.

It is interesting how much more popular the `if` statement is then the `switch` statement. Though, since `if` statements can be chained together to have switch like behavior, a direct comparison is questionable. This turns out not to be an issue, 82% of if statements have no else block, another 16% of `if`'s only have an `else` (with no directly nested `if`). `switch` statements eventually become more popular than `if else` chains, but usages of either is rare. This may mean that language authors should consider not including a `switch` construct, or instead include a more powerful pattern matching construct like those in functional languages like Haskell or Scala.

Table 2 Literal Usage.

Kind of Literal	Count/File	Count
Number	11.9	7,335,479
String	11.7	7,195,704
Null	3.4	2,098,983
Boolean	2.4	1,479,122
Character	0.35	214,443

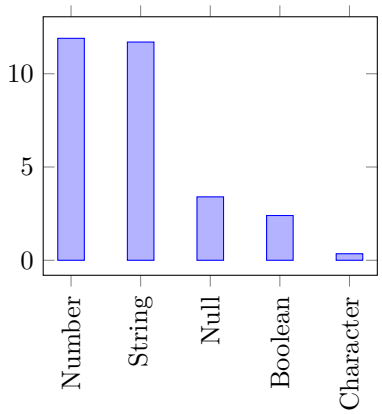

Figure 2 Literal Usage, by Count/File.

3.1.2 Literals

Literals are special syntactic constructs that a programmer may put in their code (for instance `"hello world"`, `'c'` , and `7`). Table 2 shows the count of each literal.

Developers rarely specify character literals. In fact, strings of length 1, occur 3 times as often as character literals. Language designers should consider not having special syntax for characters, instead relying on string syntax (as Python does).

The popular usage of `null` is interesting, and we will revisit this later.

3.1.3 Operators

Java does not allow operator overloading, so the 19 infix operators provided by the language are the only infix operators available. Were they well chosen? Table 3 shows the count of each operator.

Arithmetic and logic operators are very popular, but the bitwise operators are relatively unpopular. This is weak evidence that `x ^ y` might have been better used as the math power operator (instead of the rarely used XOR operator), though calls to `java.lang.Math::pow` occur less frequently.

3.1.4 Nulls

It turns out that the popularity of the `null` literal and the `==` operator are related.

In fact, over half of all equality checks are really `null` checks. This explains 59% of the null literals that occur in practice. Further inspection of `null` literals shows that 13% are used in method invocations, 13% are directly assigned or used in a declaration, and 7% are used in `return` statements. This weakly supports the popular idea that `null` references are a broken programming feature [8] and justifies special syntax for `null` checks in Kotlin, and the `Maybe` monad in Haskell.

3.2 The Java Standard Library

3.2.1 Most common method calls

Table 5 shows the most popular method call by name followed by the type that was most often resolved at the call site (methods with different signatures but the same name were counted the same for the sake of simplicity). The table shows that the collections libraries and string

Table 3 Infix operator usage.

Operator	Count/File	Count
x == y	1.98	1,216,367
x + y	1.89	1,164,466
x != y	1.39	855,485
x < y	0.70	430,412
x && y	0.68	415,305
x * y	0.58	355,789
x - y	0.56	342,409
x || y	0.38	231,446
x > y	0.31	189,792
x / y	0.24	149,703
x & y	0.15	92,747
x >= y	0.15	90,951
x <= y	0.12	74,888
x ^ y	0.08	46,641
x << y	0.06	35,205
x | y	0.04	26,638
x % y	0.04	23,412
x >> y	0.03	18,899
x >>> y	0.02	11,860

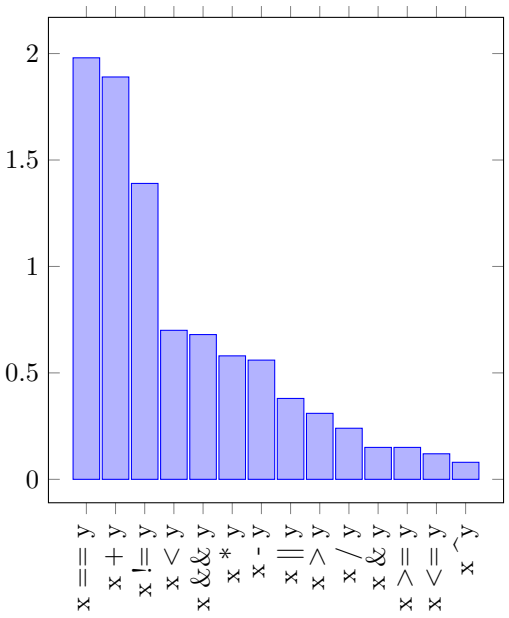

Figure 3 Infix operator usage, by Count/File.

Table 4 `null` checks.

Operator[a]	Count/File	Count
x != null	1.03	634,786
x == null	0.99	609,510
x == y	0.99	606,857
x != y	0.36	220,699

[a] `x == null` and `null == x` where counted the same.

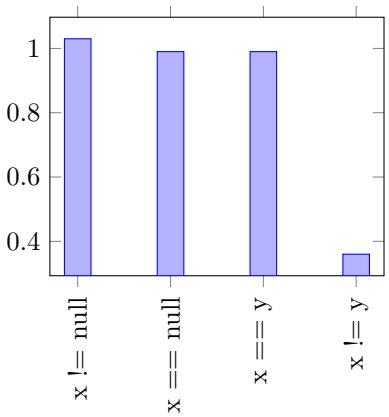

Figure 4 `null` checks, by Count/File.

Table 5 standard library calls.

Method	Count/File	Count	Most Common Example	Count/File	Count
append	0.67	412,426	java.lang.StringBuilder::append	0.57	347,825
get	0.52	320,393	java.util.List::get	0.17	105,124
add	0.48	294,697	java.util.List::add	0.28	173,803
put	0.34	208,357	java.util.Map::put	0.26	160,172
equals	0.3	187,119	java.lang.String::equals	0.21	130,928
size	0.27	163,659	java.util.List::size	0.16	100,976
toString	0.21	131,916	java.lang.StringBuilder::toString	0.08	48,729
println	0.18	110,867	java.io.PrintStream::println	0.15	90,464
length	0.11	70,160	java.lang.String::length	0.09	55,591
getName	0.11	68,646	java.lang.Class::getName	0.08	47,940
valueOf	0.1	64,421	java.lang.String::valueOf	0.03	21,129
hashCode	0.09	57,261	java.lang.String::hashCode	0.05	32,212
format	0.09	56,865	java.lang.String::format	0.08	46,710
isEmpty	0.08	51,011	java.util.List::isEmpty	0.03	19,546
contains	0.08	50,778	java.lang.String::contains	0.03	19,271
asList	0.08	47,444	java.util.Arrays::asList	0.08	47,432
getMessage	0.07	46,099	java.lang.Exception::getMessage	0.04	22,045
substring	0.06	38,717	java.lang.String::substring	0.06	38,071
next	0.06	36,884	java.util.Iterator::next	0.05	30,500
remove	0.06	35,878	java.util.Map::remove	0.01	7,849

operations make up a large fraction of method calls, and should be considered important for languages and standard libraries. Efficient string composition should be prioritized in future languages: optimizing string concatenation (with the `+` operator) would have made Java programs that use the appending function more readable. Almost every listed standard library call was more popular than `>>>`, the least popular infix operation.

4 Threats to Validity

There are some reasons to be concerned with this analysis

- Most software development is proprietary, and the open source projects on GitHub may be unrepresentative of non-open source projects.
- The most popular open source Java projects may not be representative of open source projects in general. For instance there was at least one satirical project in the sample[7].

[7] `https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition`

- Though it is frowned upon to check in generated code, it is still likely to happen in practice. Generated code could cause a bias in favor of constructs produced by the generation procedure.
- Projects that themselves try to parse Java (like the IntelliJ IDE[8]) will often include pathological test cases that contain extreme examples of Java syntax.
- It is also possible that some unknown bias was introduced though the Eclipse parser. Results should be compared against future versions of the parser.

5 Prior Work

There has been a large amount of work in mining software repositories[4]. While most of this work has focused on answering questions unrelated to language usability, there have been some studies worth mentioning.

5.1 Notable Java data mining projects

- In [11], Java projects from the open source repository SourceForge were mined to analyze 3rd party library usage and migration. This paper improves on their methodology by using the Eclipse parser, so projects can be analyzed in a reliable way without needing build information.
- [9] analyzed 22,730 artifacts in the maven package repository, to understand the landscape of the Java library ecosystem. Since projects in the maven system are expected to be used as libraries and require some basic level of quality to be accepted, this may not be a representative sample of Java projects in general.
- The Boa infrastructure[6] has the most similar methodology to this paper. It uses the Eclipse parser to analyze a snapshot of all Java projects with full git histories and uses Hadoop[9] to quickly mine large numbers of projects. The Boa infrastructure also includes the Boa programming language designed for non-expert computer users as in interface to the data mining infrastructure and Eclipse parser. The language and snapshots are made publicly available online[10]. In [7] the Boa infrastructure was used to analyze the adoption of Java features before Java 8. While Boa has some clear improvements over the methodology in this paper in terms of speed and sample size, only [7] dealt with Java usability directly. This paper does improve on a few details: Boa's latest publicly available snapshots are from 2015 at the time of this writing, while the projects in this paper were pulled in August 2018.

5.2 Other studies that address usability through data mining

There are several papers that have looked into usability of specific language features. There has been extensive research into usage of Java's exception handling mechanism[10, 13, 1, 15, 3]. Dikstra's skepticism of the `goto` construct[5] has been empirically tested with 384 C files from GitHub[12]. The usage of Scala's implicit parameter feature was analyzed in 120 GitHub projects to inform how to extend the feature[14].

[8] `https://github.com/JetBrains/intellij-community`
[9] `https://hadoop.apache.org/`
[10] `http://boa.cs.iastate.edu/boa`

5.3 Unpublished work from 2015

In unpublished work from 2015, I used a similar methodology to scan 1,970 GitHub repositories to understand adoption of Java 8. At the time only 14% of the projects had observably adopted Java 8 (by using one of Java 8 features). Of those projects, there was a noticeable decrease in single method anonymous classes, the Java 7 feature that most closely resembled the lambdas that appeared in Java 8. Lambdas were by far the most popular syntactic construct introduced by Java 8.

6 Future Work

There are a number of interesting directions to take this research

- Compare language usage across project types. There are many different uses for Java in practice: Java is heavily used in web development, Java was the primarily supported language for phone development under Android, and Java is used extensively for analytics with projects like Hadoop. It is now much easier to categorize project types since GitHub added "topics" in 2017[11]. Topics are tags that are generated automatically via machine learning[12] to project repositories and then curated by the project owner. This would offer a straightforward way to see how different kinds of projects use a language and its standard library differently.
- Reproduce the results of this paper under the Boa infrastructure. This would help increase confidence in this study as well as making the methodology more reproducible.
- Extend the analysis to different languages and paradigms. It would be interesting to see how Java usage compares to Python. It would be very interesting to see if a functional language like Haskell has similar usage.
- Run a similar analysis on popular libraries. Popular libraries like the Apache Commons[13] might give insight into features that should be incorporated into future standard libraries out of the box. An analysis like this could be very helpful for the library authors as well.
- Observe changes in usage over time. Since git keeps a record of a repositories history it would be interesting to see if some features become more or less popular over time. Do users upgrade to newer versions of the language and libraries? How quickly are new features adopted?
- Analyze the "real" types of literals such as strings. In Java strings are often used to to represent specific languages like regular expressions, SQL, or English. Analyzing the string literal usage would allow language designers to know when or if these languages should be made into separate first class languages, or should be handled in the standard library with string interpolation.
- More analysis should be done into the usage of `null` in practice. Does it very by project type? and has it become less popular over time? It would also be interesting to see how frequently the `notNull` annotation is used.

7 Conclusion

Data mining has the potential to inform many aspects of future language and library design; this paper barely scratches the surface. Hopefully, these techniques will suggest how future languages should be designed to make programming a more productive, safe, and enjoyable experience.

[11] `https://help.github.com/articles/about-topics/`
[12] `https://githubengineering.com/topics/`
[13] `https://commons.apache.org/`

References

1. Muhammad Asaduzzaman, Muhammad Ahasanuzzaman, Chanchal K Roy, and Kevin A Schneider. How developers use exception handling in Java? In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 516–519. ACM, 2016.
2. Steven Clarke. Chapter 29. How Usable Are Your APIs? In Andy Oram and Gregn Wilson, editors, *Making software: What really works, and why we believe it*. O'Reilly Media, Inc., 2010.
3. Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. Unveiling exception handling bug hazards in Android based on GitHub and Google code issues. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 134–145. IEEE, 2015.
4. Valerio Cosentino, Javier Luis, and Jordi Cabot. Findings from GitHub: methods, datasets and limitations. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 137–141. ACM, 2016.
5. Edsger W Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.
6. Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE'13, pages 422–431, 2013.
7. Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N Nguyen. 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*, pages 779–790. ACM, 2014.
8. Tony Hoare. Null references: The billion dollar mistake. *Presentation at QCon London*, 298, 2009.
9. Vassilios Karakoidas, Dimitris Mitropoulos, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. Generating the blueprints of the Java ecosystem. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 510–513. IEEE Press, 2015.
10. Mary Beth Kery, Claire Le Goues, and Brad A Myers. Examining programmer practices for locally handling exceptions. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 484–487. IEEE, 2016.
11. Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1317–1324. ACM, 2011.
12. Meiyappan Nagappan, Romain Robbes, Yasutaka Kamei, Éric Tanter, Shane McIntosh, Audris Mockus, and Ahmed E Hassan. An empirical study of goto in C code from GitHub repositories. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 404–414. ACM, 2015.
13. Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. Analysis of exception handling patterns in Java projects: An empirical study. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 500–503. ACM, 2016.
14. Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicitly: foundations and applications of implicit function types. *Proceedings of the ACM on Programming Languages*, 2(POPL):42, 2017.
15. Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. Understanding the exception handling strategies of Java libraries: An empirical study. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 212–222. ACM, 2016.
16. Andreas Stefik and Susanna Siebert. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)*, 13(4):19, 2013.