6-20-2013

A Large-scale Empirical Study of Java Language Feature Usage

Robert Dyer
Iowa State University, rdyer@iastate.edu

Hridesh Rajan
Iowa State University

Hoan Anh Nguyen
Iowa State University, hoan@iastate.edu

Tien N. Nguyen
Iowa State University, tien@iastate.edu

A Large-scale Empirical Study of Java Language Feature Usage

Abstract
Programming languages evolve over time, adding additional language features to simplify common tasks and make the language easier to use. For example, the Java Language Specification has four editions and is currently drafting a fifth. While the addition of language features is driven by an assumed need by the community (often with direct requests for such features), there is little empirical evidence demonstrating how these new features are adopted by developers once released. In this paper, we analyze over 23k open-source Java projects representing over 7 million Java files, which when parsed contain over 14 billion AST nodes. We analyze this corpus to find uses of new Java language features over time. Our study gives interesting insights, such as: the fact that while all features are used, there are still millions of more places they could potentially be used; all features are used before release; and features tend to be adopted by committers on an individual basis rather than as a team.

Keywords
Java, empirical study, language feature use

Disciplines
Programming Languages and Compilers

A Large-scale Empirical Study of Java Language Feature Usage

Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen

Iowa State University, Department of Computer Science, Technical Report 13-02
June 20, 2013

Keywords: Java, empirical study, language feature use

CR Categories:
D.3.3 [*Programming Languages*] Language Constructs and Features - Control structures

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

A Large-scale Empirical Study of Java Language Feature Usage

Robert Dyer · Hridesh Rajan
Hoan Anh Nguyen · Tien N. Nguyen

Abstract Programming languages evolve over time, adding additional language features to simplify common tasks and make the language easier to use. For example, the Java Language Specification has four editions and is currently drafting a fifth. While the addition of language features is driven by an assumed need by the community (often with direct requests for such features), there is little empirical evidence demonstrating how these new features are adopted by developers once released. In this paper, we analyze over 23k open-source Java projects representing over 7 million Java files, which when parsed contain over 14 billion AST nodes. We analyze this corpus to find uses of new Java language features over time. Our study gives interesting insights, such as: the fact that while all features are used, there are still millions of more places they could potentially be used; all features are used before release; and features tend to be adopted by committers on an individual basis rather than as a team.

Keywords Java, empirical study, language feature use

1 Introduction

The Java Language Specification (JLS) Gosling et al (1996, 2000, 2005, 2013) is the official specification for Java. New editions of the specification (JLS2–JLS4) are released as the language evolves to add new features. The official Java platforms (Java Runtime Environment (JRE) and Java Development Kit (JDK); Standard (SE), Mobile (ME), and Enterprise Editions (EE)) all implement the language based on this official specification.

Changes to the specification are driven by needs from the community. This need often comes in the form of an official request (a Java Specification Request (JSR)) using the Java Community Process (JCP). The JSR formally defines what the need is, why the current specification is lacking, and proposes a solution. Each new language feature has an accompanying JSR and each new edition of the language has an umbrella JSR to identify the new features.

Currently however, there is little quantitative evidence demonstrating how most of these new language features are used in practice. Previous studies have investigated the use of certain Java language features, e.g. Grechanik et al (2010) investigated the use of several object-oriented features in Java, such as class, interface, and method usage and Parnin et al (2011) investigated the use of generics in Java. Similarly, Livshits et al (2005), Callaú et al (2011), and Christensen et al (2003) investigated the use of non-language features such as reflection (which in Java, is supported by the runtime and not the language). However, these studies typically looked at a relatively small number of Java projects (around 20), investigated a very small subset of features, or did not investigate their adoption over time.

Robert Dyer · Hridesh Rajan
Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, IA 50010 USA
Hoan Anh Nguyen · Tien N. Nguyen
Department of Electrical and Computer Engineering, Iowa State University, 2215 Coover Hall, Ames, IA 50010 USA
E-mail: {rdyer,hridesh,hoan,tien}@iastate.edu

We wish to improve upon these existing studies in two ways. First, by studying most of the new language features introduced to Java after the first edition of the language specification. We are interested in answering several important research questions regarding Java language feature use, such as: how frequently each feature is used; are the features used before their official release; are the features generally added to new files or existing files; and is adoption limited to certain committers. The adoption of language features is measured along three dimensions: *projects*, *source code files*, and *committers using the features*. Second, we wish to improve on previous studies by answering these questions on a much larger number of Java projects.

To accomplish this, we utilize the Boa language and infrastructure Dyer et al (2013a,b); Rajan et al (2012) and write queries on Java source files. The dataset we query is over 23k Java projects from SourceForge (2013), representing over 7 million unique Java files, which when parsed contain over 14 billion AST nodes. Some interesting results from our study include:

- All studied features are used, however a few features are clearly the most popular, including: annotation use, enhanced for loops, and variables with generic types. Several features saw minimal use. We found many instances where features could have been used, but were not, indicating a need for better training or IDE support.
- All language features were used prior to their official release, indicating anticipation of such features.
- With the exception of JLS4 features, most language features appeared in newly added files in the repositories more often than in previously existing files.
- The most common generic types are part of the JDK's Collections framework. This result is consistent with a previous study Parnin et al (2011), but on 1,000 times more projects.
- Committers tend to adopt new features on an individual basis rather than in a team. This result is consistent with a previous study Parnin et al (2011), but with 100 times more committers.
- Most committers use only a small number of new features. A small number of committers account for the majority of new language feature uses.

In the next section, we give background on each edition of the JLS and the new language features. Then in Section 3 we pose the research questions our study aims to answer. We describe the approach used in our study in Section 4 and give the study itself in Section 5. In Section 6 we describe some previous studies regarding language feature use. Finally we conclude with future work in Section 7.

2 Background: Java Language Specifications (JLS)

Since the original edition of the Java Language Specification (JLS) Gosling et al (1996), there have been three updates. In this section we outline some of the changes to the language for each edition. Note that new language features are purely additive - each edition is fully backwards compatible with previous editions.

2.1 JLS2 New Language Features

The Java Language Specification, edition 2 (JLS2) Gosling et al (2000) was a relatively minor update in terms of new language features. This edition added one new language feature: *assert statements*.

Assertions allow programmers to verify properties they assume are true about code. If assertions are enabled at runtime and the property is false, an exception is thrown.

2.2 JLS3 New Language Features

The Java Language Specification, edition 3 (JLS3) Gosling et al (2005) added several significant language features, including: *annotation types*, *enhanced for loops*, *type-safe enumerations*, *generic types*, and *variable argument methods*.

2.2.1 Annotations

Annotations provide a declarative way of providing additional metadata about code. Annotations are declared similar to interfaces. For example:

```
public @interface Test { }
```

declares the annotation `Test`. Annotations can be added to certain declarations, such as methods, fields, and types. For example:

```
@Test void m() { .. }
```

declares that the method `m` represents a unit test. A unit testing framework would locate all methods containing this annotation and execute them.

2.2.2 Enhanced For Loops

Enhanced for loops provide an easy way to iterate over collections. Previously, programmers used the `Iterator` interface:

```
Iterator iter = items.iterator();
while (iter.hasNext()) {
  T val = iter.next();
  ..
}
```

With enhanced for loops however, the use of iterators is hidden from the programmer, which makes the code much more concise:

```
for (T val : items) ..
```

however it is not applicable for every use case. For example, if one needs access to the iterator (perhaps to remove an item) the original pattern should be used.

2.2.3 Type-safe Enumerations

In previous editions, a programmer would represent an enumerated type either by declaring a set of static, final integers:

```
public static final int N1 = 1;
public static final int N2 = 2;
..
```

or using a type-safe pattern, in which a class has a private constructor and instantiates final, static fields for each enumerated value. However, both of those approaches had problems and thus type-safe enumerations were added to the language. The previous example can be declared as:

```
public enum E { N1, N2, ..; }
```

which is type-safe, avoids the brittleness of previous approaches, and is less verbose.

2.2.4 Generics

Generics allow classes and methods to operate on different types while still maintaining the language's static type safety. A type, field, method, or variable may be declared with one or more generic type arguments. For example:

```
public interface List<T> { .. }
```

declares an interface that takes one type parameter. This allows creating lists of integers, lists of strings, etc which allows for better code reuse as well as avoiding the need to perform frequent casts.

2.2.5 Variable Arguments (varargs) Methods

In previous editions, if a method needed to accept a variable number of arguments the programmer would simply make one argument be an array. In this edition, methods may declare one argument (it must be the last argument) to be of variable length. For example:

```
public void m(T... arg) { .. }
```

the method `m` declares a variable-length argument `arg`. Callers may pass in 0 or more arguments of type `T` when calling the method. The compiler automatically wraps the values into an array object and passes it to the method.

2.3 JLS4 New Language Features

The Java Language Specification, Java SE 7 edition (JLS4) Gosling et al (2013) made several changes, including: *binary literals syntax*, a *diamond operator* for generic type inference, *allowing catching multiple exception types*, *suppression of varargs warnings*, *automatic resource management*, and *underscores in literals*.

2.3.1 Binary Literals

This edition allows specifying literals using a new format. Previously, literals were specified in either base 10, base 8 (octal), or base 16 (hex). Integer literals may now be specified using base 2 (binary). For example:

```
final int FIVE = 0b101;
```

declares an integer with decimal value 5.

2.3.2 Type Inference for Generic Instance Creation (diamond)

As previously mentioned, the language allows generic types. When declaring a variable of a generic type however, the generic type arguments must be repeated. For example:

```
Map<K, V> m = new HashMap<K,V>();
```

declares a `HashMap` with keys of type `K` and values of type `V`. Note that the generic type arguments were repeated both in the variable declaration (left) and the object instantiation (right). This edition allows omitting the repeated generic type arguments in the instantiation (the so called *diamond operator*), thus changing the previous example to:

```
Map<K, V> m = new HashMap<>();
```

This new diamond operator can be used anywhere the compiler is able to infer the generic type arguments.

2.3.3 Catching Multiple Exception Types (multicatch)

This edition allows specifying more than one exception type inside a catch clause. The catch clause's body is then executed when either exception type is caught. For example, the statement:

```
try { .. } catch (E1 | E2 e) { .. }
```

will execute the catch statement's body if the try statement throws an exception of type `E1` or of type `E2`. This helps avoid code duplication.

2.3.4 Safe Varargs Warning Suppression

The variable number of arguments in methods feature previously added can lead to a large number of compile-time warnings when combined with generics. Often however the programmer knows that these warnings can safely be ignored, so a new ability to disable those warnings was added:

```
@SafeVarargs
@SuppressWarnings({"unchecked", "varargs"})
public static <T> void add (List<T> l, T... elems) { .. }
```

The use of either of these annotations will suppress compiler warnings at this method location.

2.3.5 Try with Resources

Certain resources, such as files, require manually releasing them when finished. This by itself is easy to forget, however even when programmers remember to close the resource, errors can still creep in. To ease the management of these resources, a new statement was introduced:

```
try (AutoCloseable ac = new ..) { .. }
```

This try statement declares a resource `ac` which is available within the try statement's body. Upon exiting the try statement (either through normal or exceptional program flow) the resource is automatically released.

2.3.6 Underscores in Literals

Another change in literals for this edition allows the use of underscores in the values. For example:

```
final int MILLION = 1_000_000;
```

declares an integer literal with the value 1,000,000. The underscores are used purely for improved readability.

3 Questions Regarding Language Features

The focus of our study is the usage of Java language features by open-source developers. In this section, we outline the specific research questions (RQ) we wish to answer.

3.1 RQ1: *How frequently is each language feature used?*

The first question deals with feature usage. The addition of language features is driven by needs from the community (often with direct requests for such features), yet to date there has been no study to see how most of Java's language features are being adopted by developers.

This question examines language features introduced in JLS2–JLS4. For each language feature, its use across our entire dataset is tracked. This data gives insight into how each feature was, and is, being used.

3.2 RQ2: *Do projects use new language features before the features are released?*

Often, especially with Java, an implementation of a requested feature is available before its release. This can take the form of an official beta/pre-release or someone implementing their own compiler.

We are interested in how often new language features are used prior to their official release. Such data can give an indication if a particular feature was anticipated and if providing implementations prior to release may be useful to the community.

3.3 RQ3: *Are new language features added to new or existing files?*

Another question we are interested in answering is whether new language features are usually added to new files or are they also added to existing files. This can help give insight into how features are adopted: by people adding new code into the system or by project maintainers.

3.4 RQ4: *How did committers adopt and use language features?*

Once a new set of language features is available, it takes time for developers to learn how and where to use them. Some developers may be excited and try using them as often as possible. Other developers may be content with solving problems with the old set of features, as that is what they are accustomed to. We wish to investigate to see how language feature adoption occurs for individual developers.

4 Approach: Tools and Dataset

In this section, we describe our approach for answering the previously identified research questions. Our approach relies on Boa Dyer et al (2013a,b); Rajan et al (2012) and its dataset from SourceForge (2013).

4.1 Background: Boa Language and Infrastructure

The Boa language and infrastructure was designed to abstract away the details of software mining and provide a platform for easily writing queries that execute efficiently against a very large set of software repository data. Boa contains data from SourceForge projects and supports a wide range of queries on that data.

The Boa language abstracts away most of the details of software mining. The Boa framework mines the software repositories (in this case, SourceForge) and transforms the data into a custom set of types. The language provides these domain-specific types, such as `Project`, `CodeRepository`, and `Revision` that allow users to perform queries against software repositories.

Boa currently processes CVS and Subversion repositories for Java projects. When it finds a change to a Java source file, it checks out the snapshot of that file at that revision and parses it using the Eclipse JDT parser. The parsed data is then translated into a custom representation. Types include `Namespace`, `Declaration`, `Method`, `Variable`, `Statement`, and `Expression`. The statement and expression types are union types, allowing each type to represent multiple cases.

The Boa infrastructure generates a Hadoop Apache Software Foundation (2012) MapReduce Dean and Ghemawat (2004) program to efficiently execute queries. This is also abstracted away and developers do not have to explicitly write any code to parallelize their queries.

4.2 Dataset Used in Our Study

The dataset used for our study is all Java projects on SourceForge that list at least one CVS or Subversion repository. The projects self-classify which means there may be some Java projects on SourceForge that we do not include in our dataset. The dataset also does not include Java projects with only Mercurial, Git, or Bazaar repositories. The total number of projects is over 34k (see Figure 1).

Metric	Count
Java Projects	*34,705*
Non-empty Projects	*25,649*
Studied Projects	**23,716**
Repositories	*36,194*
Revisions	**6,378,478**
Files	*27,446,128*
File Snapshots	*54,225,527*
Java Files	**7,662,807**
Java File Snapshots	**22,013,092**
AST Nodes	**14,487,845,808**

Fig. 1: Metrics for the SourceForge dataset.

However, not all of these projects are useful. We identified over 9k projects that, although listing a repository, contain no files at all. We filtered these projects out of our dataset. There were an additional 2k projects that contained files, but did not have at least one Java source file that parsed without error and we also filtered those projects. This leaves over 23k projects in the dataset for use in our study.

The dataset contains widely-used Java projects, including: Azureus/Vuze, Weka, Hibernate, JHotDraw, JabRef, JUnit, iText, FindBugs, JML, TightVNC, etc. This dataset represents over 6 million revisions by more than 50k Java developers. It also contains over 7 million unique Java files and a total of over 22 million snapshots of those files.

This represents (to the best of our knowledge) the largest empirical dataset to date for Java projects that contains both full history information of the source repositories with over a decade of history and also the full AST information from the Java source files.

For our research questions, the size of the Java projects (whether 1 file or 1k files) is irrelevant, as we are interested in investigating Java language features used by developers without constraining the study to any specific kind of developer. Thus we include small projects (perhaps written by novice developers) as well as large projects (perhaps written by experts). However for RQ4, smaller projects could affect our results and thus, as we mention later, for this research question we also filtered projects with few maintainers.

JLS2	Assert
Uses	297,875
File	1%
Project	13.49%

JLS3	Annotation Declaration	Annotation Use	Enhanced For Loop	Enums	Generic Variable	Generic Method
Uses	22,704	5,603,688	1,700,162	113,000	5,406,221	166,756
File	0.29%	18.26%	7.32%	1.31%	16.34%	0.81%
Project	6.9%	51.01%	45.36%	28.68%	56.83%	13.03%

JLS3	Generic Type	Extends Wildcard	Super Wildcard	Other Wildcard	Varargs
Uses	154,845	254,682	65,988	530,569	151,179
File	1.79%	1.17%	0.15%	2.02%	0.88%
Project	19.79%	15.5%	2.59%	20.78%	14.92%

JLS4	Binary Literals	Diamond	MultiCatch	Safe Varargs	Try with Resources	Underscore Literals
Uses	49	3,116	443	25	526	35
File	0%	0.01%	0%	0%	0%	0%
Project	0.02%	0.27%	0.16%	0.03%	0.13%	0.03%

Fig. 2: Java language feature usage by total number of uses, by percent of all files, and by percent of all projects.

5 Study: Analyzing Language Feature Adoption

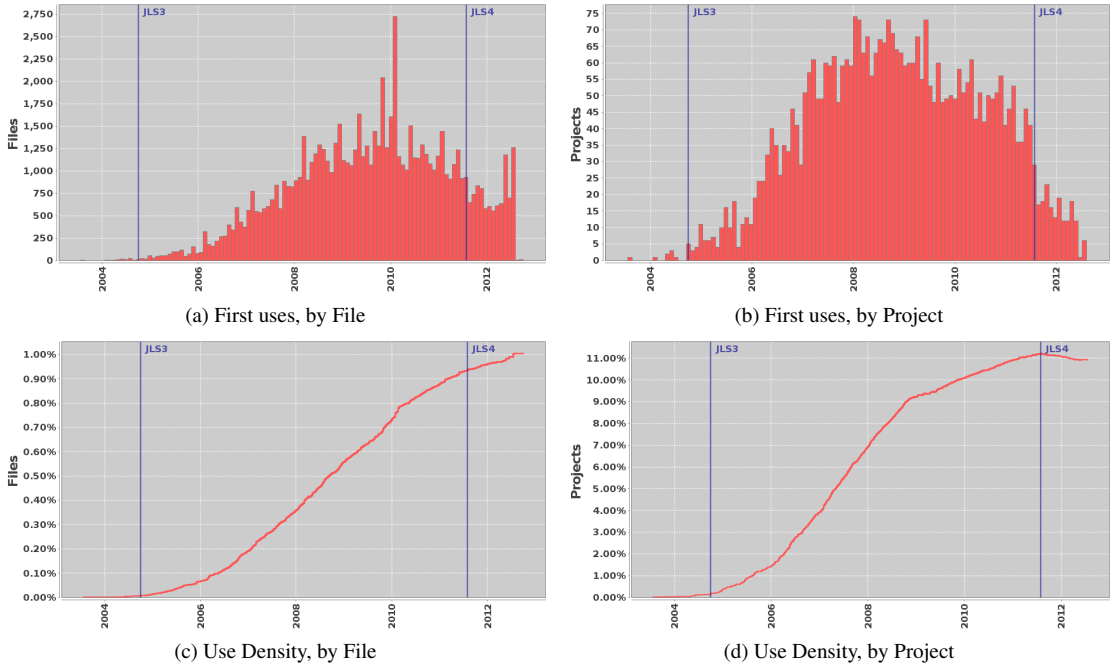
In this section we investigate Java language feature usage. We use Boa Dyer et al (2013a,b); Rajan et al (2012) to mine each language feature's use over time and answer several research questions using the SourceForge-based dataset.

5.1 RQ1: *How frequently is each language feature used?*

The addition of language features is driven by needs from the community. In this section, we quantitatively investigated how developers use these new features by looking at each unique Java source-file path in the system and taking the last existing snapshot of each. We then analyzed that set of snapshots and counted feature usage. For each file, we generated a mapping between each feature and the total uses in the file.

We show the results in Figure 2, first by total number of uses across the entire dataset, then by percent of Java files using the feature, and finally by percent of projects using the feature. The table clearly shows every feature is being used at least once. One trend that becomes readily apparent is that JLS4 features are not used very often, compared to JLS3 features.

Another observation that fits with what one expects is the ratio of uses to files. For example, the Annotation Declaration feature has a ratio close to 1[1]; there is roughly one annotation declaration per file. This is similar for Enums and Generic Type. These features represent types in Java and thus one generally expects to see one type per file. The ratios for the other features are higher (2–6) since they are expressions and statements. For example, the ratio of enhanced for loops is 3[2] meaning files using the feature use it around three times.



(a) First uses, by File

(b) First uses, by Project

(c) Use Density, by File

(d) Use Density, by Project

Fig. 3: Use of the *Varargs* language feature.

We plotted histograms of each feature's use, both by number of files and by number of projects. The histograms contain bins with 30-day time ranges. The first time a feature appears, it is added to the respective bin. See Figures 3a–3b, Figures 4a–4b, and Figures 5a–5b. The plots also contain marker lines to indicate the release date of each JLS.

We also plotted densities of each feature's use, both by number of files and by number of projects. Points in these charts represent the number of files/projects using a feature at that time, divided by the total number of files/projects at that time, to account for growing repositories. See Figures 3c–3d, Figures 4c–4d, and Figures 5c–5d.

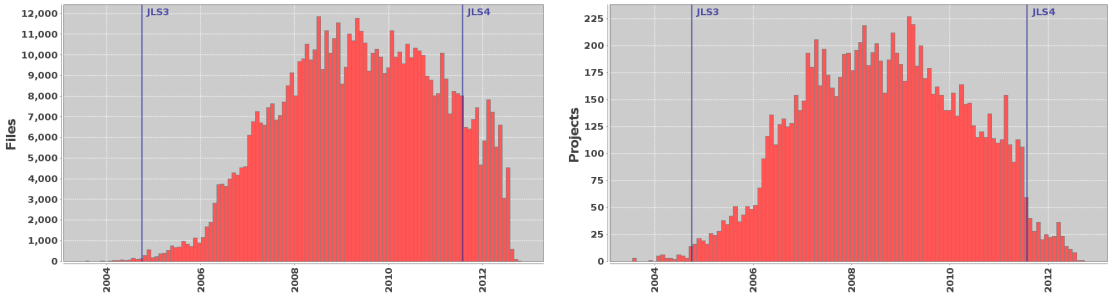
After examining these plots for each feature, we noticed similar trends among the features. They fell into two categories: JLS4 features and non-JLS4 features. Since the trends are similar across features, we picked representatives from each category.

For example, Figures 3 and 4 show two non-JLS4 features which have similar trends. The histograms all show increasing adoption of the features after release with peaks around 2009. Then the number of files/projects adopting the feature for the first time starts decreasing. To better understand this decrease, we investigate the density plots.

As can be expected, the files and projects in the system were increasing over time. The density plots try to remove this variable from our analysis, by computing the percent of feature use at each time. For example, when we look at Figures 3c–3d we can see that even as the total number of files and projects in the system increases,

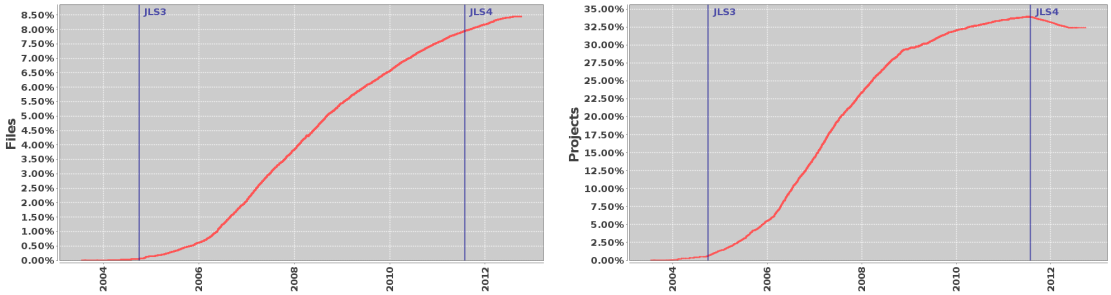
[1] This feature appears in 0.29% of files, or around 22k files, and is used around 22k times total.

[2] This feature appears in 7.3% of files, or around 560k files, and is used around 1.7m times total.



(a) First uses, by File

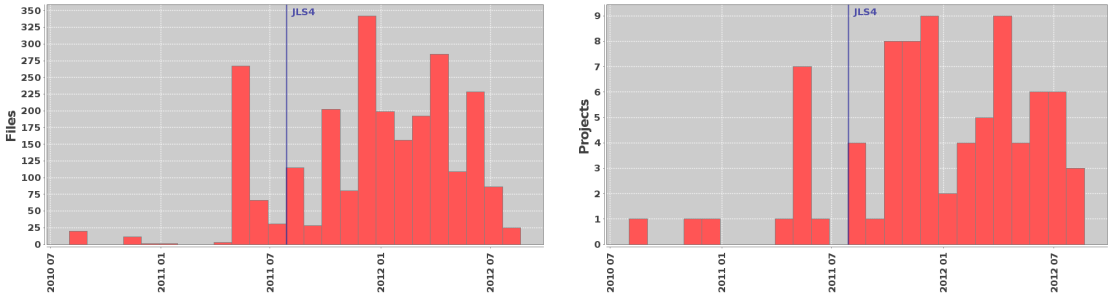
(b) First uses, by Project



(c) Use Density, by File

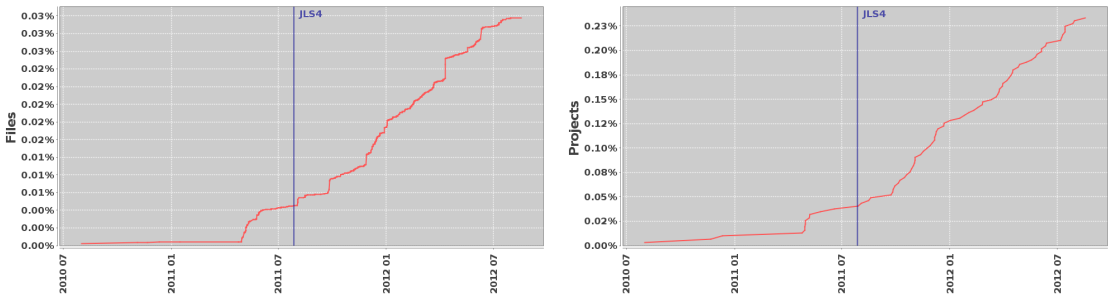
(d) Use Density, by Project

Fig. 4: Use of the *Enhanced-For* language feature.



(a) First uses, by File

(b) First uses, by Project



(c) Use Density, by File

(d) Use Density, by Project

Fig. 5: Use of the *Diamond* language feature.

the relative percent is increasing too. Thus we can see that over time, the use of features is increasing. This trend is apparent for all features.

Notice that Figure 5, a JLS4 feature, doesn't show as strong of trends as the previous two features discussed. In this chart, the histograms have less of an obvious trend to them, due to the relatively low number of total uses for this new feature. While the density graphs still show the same general trend of increasing use, both by files and by projects, there is less of a defined curve in these graphs.

5.1.1 Investigating Potential For More Use

In this section, we investigate where there may be more potential to use the new language features by mining the source code to find locations where new language features could potentially be used. For example, we mined to find integer literals with 7 or more characters that did not use underscores. We also mined generic instantiations that don't use the diamond pattern, expressions where the literal '1' was shifted left (which could use binary literals), try statements with more than one catch block having the same body, methods that take an array as last argument instead of a varargs argument, and methods that have as their first statement an if condition that if true throws an `IllegalArgumentException` (which could potentially be turned into an assert statement).

The results are shown in Figure 6. In the first row, we list potential uses in files that existed prior to the feature's release. These represent places where a maintainer could refactor code to use the new language feature. We found tens of thousands (to millions) of potential uses in old files.

	Assert	Varargs	Binary Literals	Diamond	MultiCatch	Underscore Literals
Old	74K	560K	48K	2.8M	292K	19.4M
New	242K	1.3M	4K	240K	14K	1.5M
All	316K	1.8M	52K	3M	307K	21M
Files	1.36%	12.56%	0.11%	11.83%	2.27%	19.92%
Projects	20.29%	90.05%	6.54%	55.94%	52.80%	90.36%

Fig. 6: Potential language feature uses, in old files (before feature release) and new files (after feature release).

The second line of the table shows potential uses in files that were added after the release of the feature. These are locations that developers had the option to use a language feature, but did not. Again, we found thousands of potential uses for each feature and even millions of potential uses for two features.

While we do not know why people avoided using the new features in these instances, the results clearly show a lot of potential use. It may be the case there should be more or better training of developers. Or perhaps better advertisement of new features. IDEs could also help more here, by providing suggestions to refactor code to use the new features. For example, if a user wrote "`int i = 3000000;`" the IDE could show a suggestion to refactor this code and use underscores for better readability.

5.1.2 Investigating Frequently Used Features

Most language features are used in a very small number of files (2% or less). The exceptions are Annotation Use, Enhanced For, and Generic Variable declarations. We investigated more to understand why these particular language features were so much more popular.

First we looked into the use of annotations, by collecting the annotation types named at each use. Figure 7 shows the ten most frequently used annotation types and the number of uses for each. As can be seen, over half of the annotation uses were the `@Override` annotation. Such widespread use of this annotation makes sense as IDEs such as Eclipse typically automatically add this annotation. The second most used annotation, `@Test`, is used by unit testing frameworks. In fact, the annotations listed in the table are almost all JDK or J2EE provided annotations. We anticipated high use of JDK annotations, as the Annotation Declaration language feature has less than 0.3% use across all Java files, but the clear domination of those annotations was surprising.

Next we looked into the generic variable declarations, by collecting the counts of each declared generic variable's type. Figure 8 shows the ten most frequent generic types used (top) and the top ten parameterized types (bottom). The results clearly show that the majority of generics are from collection types, the most common being

Annotation Name	Uses	Percent
@Override	3,336,141	52.76%
@Test	645,801	10.21%
@SuppressWarnings	396,539	6.27%
@SubL	134,718	2.13%
@Generated	114,060	1.80%
@Column	95,475	1.51%
@XmlElement	93,493	1.48%
@XmlAttribute	55,876	0.88%
@XmlAccessorType	52,048	0.82%
@XmlType	50,448	0.80%

Fig. 7: Annotation use counts. Percents are out of all annotation uses.

Generic Type	Uses	Percent	Generic Type	Uses	Percent
List	2,307,624	21.34%	List<String>	337,749	3.12%
ArrayList	1,353,309	12.51%	ArrayList<String>	262,501	2.43%
Map	755,309	6.98%	Class<?>	188,536	1.74%
HashMap	560,415	5.18%	Map<String, String>	131,299	1.21%
Set	530,861	4.91%	Set<String>	114,867	1.06%
Collection	408,788	3.78%	Map<String, Object>	101,267	0.94%
Vector	366,716	3.39%	HashMap<String, String>	93,462	0.86%
Class	353,568	3.27%	Vector<String>	82,045	0.76%
Iterator	334,986	3.10%	HashSet<String>	73,572	0.68%
HashSet	255,028	2.36%	HashMap<String, Object>	58,226	0.54%

Fig. 8: Variables declared with generic types.

`List<String>`. These results are consistent with the previously published study on generics use by Parnin et al (2011), although our study was on a thousand more projects.

5.2 RQ2: *Do projects use new language features before the features are released?*

If a feature is requested by the community, then most likely people will be excited to use it prior to its release. To see if this is true, first we needed to know the release dates of official implementations for each language specification. These released versions and their corresponding dates are shown in Figure 9.

Based on the dates from this table, we then analyzed each valid Java file to see if it used a particular feature. We filtered out any Java file that contained a parse error. Then we collected the timestamps of each file using each language feature and then filtered based on the particular language feature's release date. The results are shown in Figure 9 and include the list of features, the date of the first use of the feature, the number of projects that used the feature prior to release, and the total number of files that used the feature prior to release.

For the earliest uses, we manually investigated to verify the files identified actually used the particular feature and that the commit date matched our results. Based on this analysis we identified one project who's commit dates were clearly erroneous and we removed that project from our dataset. Interestingly, for one project that made heavy use of generics in 1998, the commit log referenced "switch[ing] to GJ", which is the language extension proposed by Bracha et al (1998) that eventually became the basis of Java's generics.

The results in the table clearly show that every language feature was used prior to its official release date. Notice however that the total number of projects using a feature prior to release never exceeded 65 and was as little as 2, which is substantially lower than the number of projects using those features after release. So while it is clear that language features are used prior to their release, they typically are only used by a small number of projects. Regardless, these results show that new language features will have early adopters.

5.3 RQ3: *Are new language features added to new or existing files?*

When do new language features show up in a class? Are they typically added to only new classes or do the features also show up by modifying existing classes? In this section, we investigate these questions.

JLS2 (JSR 59) - Released 05 Feb 2002			
Feature	**Earliest Use**	**Projects**	**Files**
Assert	08 Feb 1998	111	1,065
JLS3 (JSR 176) - Released 29 Sep 2004			
Feature	**Earliest Use**	**Projects**	**Files**
Annotation Declaration	11 Nov 2003	7	130
Annotation Use	11 Nov 2003	11	1,112
Enhanced For	18 Jul 2003	43	606
Enums	18 Jul 2003	19	151
Generic Variable	01 Jul 1998	64	3,688
Generic Method	04 May 1999	21	919
Generic Type	01 Jul 1998	30	2,042
Extends Wildcard	24 Jul 2003	17	565
Super Wildcard	24 Jul 2003	3	426
Other Wildcard	24 Jul 2003	22	641
Varargs	23 Jul 2003	10	70
JLS4 (JSR 366) - Released 27 Jul 2011			
Feature	**Earliest Use**	**Projects**	**Files**
Binary Literals	04 Nov 2010	2	4
Diamond	01 Aug 2010	12	400
MultiCatch	01 Aug 2010	9	91
SafeVarargs	30 Apr 2011	3	17
Try with Resources	04 Nov 2010	8	107
Underscore Literals	04 Nov 2010	2	2

Fig. 9: Language features are used before their release.

Language Feature	Added Files	Added %	Changed Files	Changed %	Total Files	Total %
Assert	55,884	0.74%	31,459	0.41%	87,343	1.15%
Annotation Declaration	19,171	0.25%	2,833	0.04%	22,004	0.29%
Annotation Use	1,110,282	14.62%	455,343	6.00%	1,565,625	20.62%
Enhanced For	422,101	5.56%	221,111	2.91%	643,212	8.47%
Enums	84,270	1.11%	24,931	0.33%	109,201	1.44%
Generic Variable	973,758	12.83%	423,879	5.58%	1,397,637	18.41%
Generic Method	47,336	0.62%	23,855	0.31%	71,191	0.94%
Generic Type	109,830	1.45%	37,362	0.49%	147,192	1.94%
Extends Wildcard	64,902	0.85%	41,194	0.54%	106,096	1.40%
Super Wildcard	7,809	0.10%	5,509	0.07%	13,318	0.18%
Other Wildcard	113,401	1.49%	73,944	0.97%	187,345	2.47%
Varargs	50,604	0.67%	25,803	0.34%	76,407	1.01%
Binary Literals	7	0.00%	2	0.00%	9	0.00%
Diamond	967	0.01%	1,495	0.02%	2,462	0.03%
MultiCatch	218	0.00%	226	0.00%	444	0.01%
SafeVarargs	21	0.00%	43	0.00%	64	0.00%
Try with Resources	215	0.00%	189	0.00%	404	0.01%
Underscore Literals	12	0.00%	2	0.00%	14	0.00%

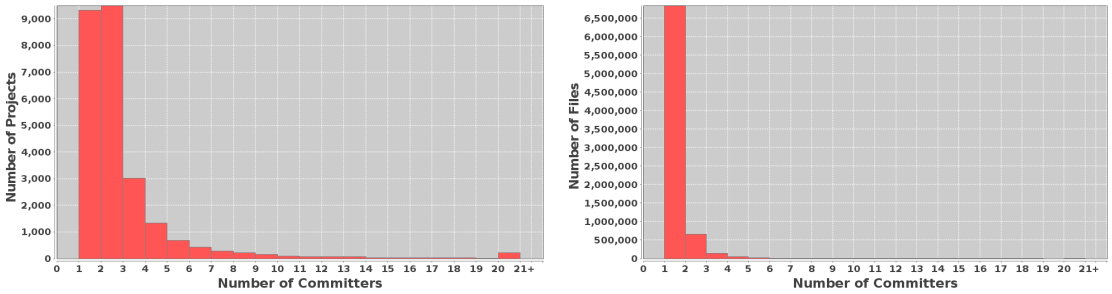
Fig. 10: Counting first snapshot of each Java file where a language feature appears, for new and existing files.

For each unique Java file, we determined the first time a language feature is used and collected whether the file was being added or changed. Note that unlike RQ1, we are not looking at the last snapshot of each file but rather the first snapshot where the feature occurs. If the feature is later removed from a file, it will still show in this list (but would not be counted in RQ1). Thus the total counts are different.

Figure 10 shows the results for all language features and includes how many added files contained the feature, the percent that represents out of all unique Java files, how many changed files contained the feature, and the total number of files. With the exception of Diamond, MultiCatch, and SafeVarargs, new language features appear more often in added files.

5.4 RQ4: *How did committers adopt and use language features?*

While in RQ1 we showed that all features are adopted in terms of files and projects, and RQ2 showed they are even adopted before their release, so far we have only evaluated feature adoption in terms of files and projects. In this section, we wish to evaluate if similar adoption trends also apply in terms of committers. Specifically, we also wish to study the adoption behavior of individual committers.



(a) Number of committers in a project

(b) Number of committers editing a file

Fig. 11: Number of committers per-project and per-file in SourceForge.

In order to do that, for each changed or added file that was recognized as containing a feature, we collected its commit time and author. For each commit that has changed files containing the use of a feature for the first time, the corresponding author is counted as one committer using that feature. The number of committed files containing the new features are also recorded and counted toward the number of uses for the corresponding committer. The threat to this method of counting is that if a committer uses a feature in a file which has already contained that feature (introduced by some other committer), they would not be counted. However, in our dataset a file is usually owned and edited by one or a few committers (as shown in Figure 11b), thus our results are not affected much.

5.4.1 RQ4.1: How many committers adopted and used new features over time?

Figures 12, 13 and 14 show the result for the number of committers using three different features over time. Each bar in a graph shows the number of users in the corresponding month. Even though the features appeared at different times, all three show the same trend of adoption: a few committers used the feature before its release, then the number of users increases to a peak, and finally decreases. This is consistent with the adoption trend for projects and files seen in RQ1.

Among the committers using a feature, we counted the ones who used that feature for the first time (the lower part of a bar - in red color) and the ones who had used that feature before (the upper part of a bar - in blue color). As seen in the figures, after the release date more committers adopted the new features. Once a feature is used for the first time, many committers kept using it in later commits (in blue). After a while, the number of first-time users (in red) decreases. This trend is the same for all the features in our study.

Comparing the charts, it can be seen that the number of committers using Annotations is much higher than that for Assert and Diamond. This result is also consistent with RQ1 where we measure the adoption in terms of projects and files.

5.4.2 RQ4.2: How much did committers use each feature?

We answer this question by counting the number of uses of each feature for each committer. Since different features are used at different levels of granularity in the source code, e.g. annotation `@Override` can be used only at the method level, while enhanced for loops can be used multiple times in the body of the method, we

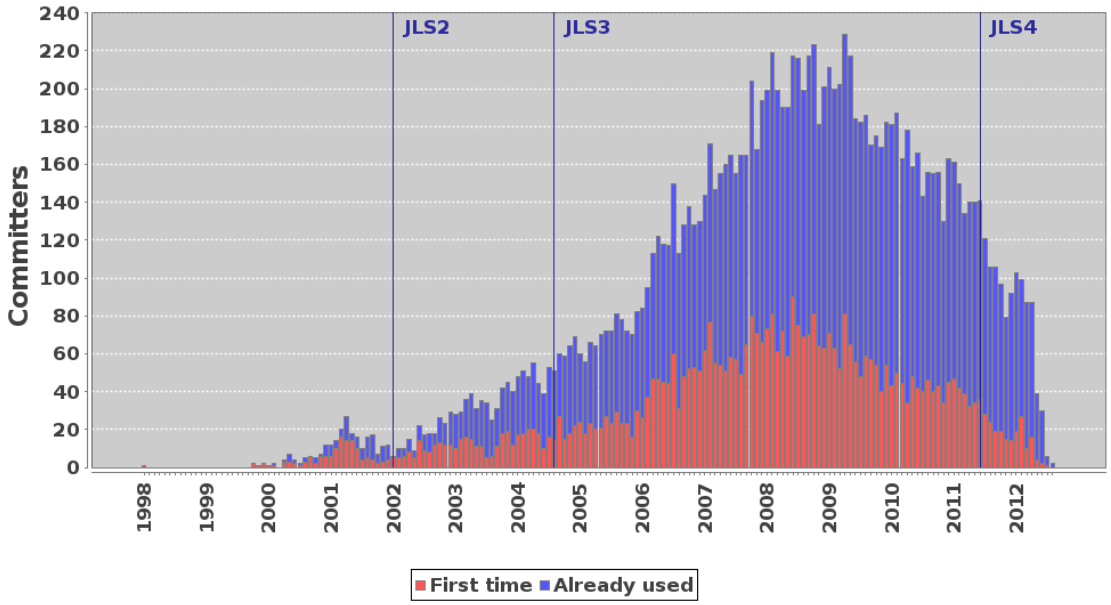


Fig. 12: Committers use of Assert over time.

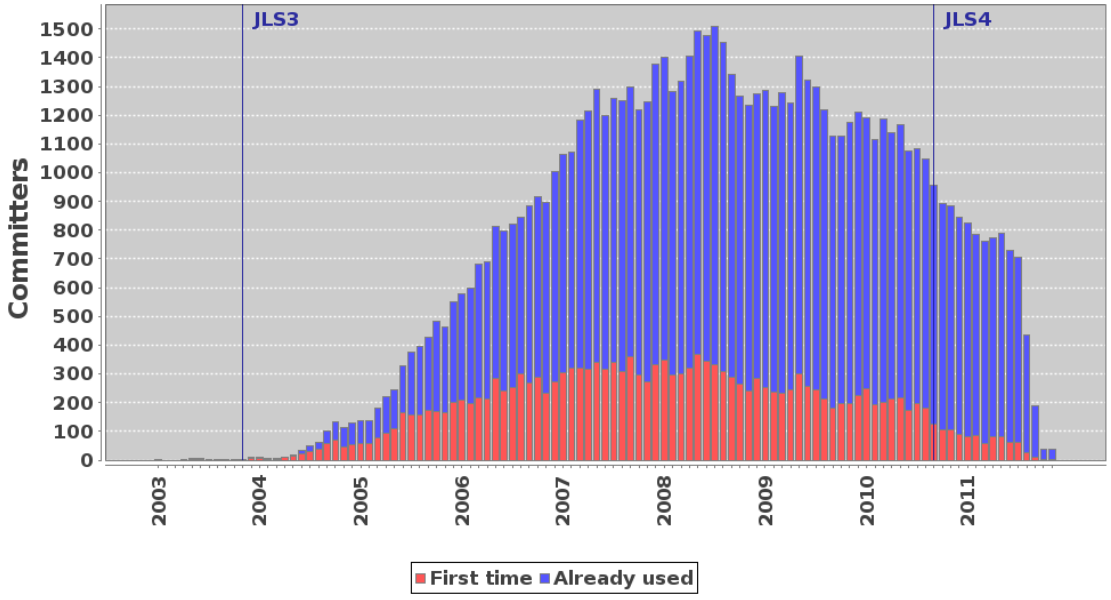


Fig. 13: Committers use of Annotations over time.

used files to compute the number of uses. That is, the number of uses of a feature for a committer is the number of files to which that committer was the first one introducing that feature.

Figure 15 shows the result for four features: Diamond (15a), Super Wildcard (15b), Extends Wildcard (15c), and Annotation Use (15d). In each chart, the x-axis represents the committers ranked by their number of uses and the y-axis (logarithmic scale) represents the number of uses. Each bar represents the number of uses for a single committer. The four charts consistently show an exponential trend in the number of uses for different features. As seen from the charts, the number of uses is highly skewed. A small number of committers accounts for a large number of uses of features. About half of the number of committers introduced a feature to less than

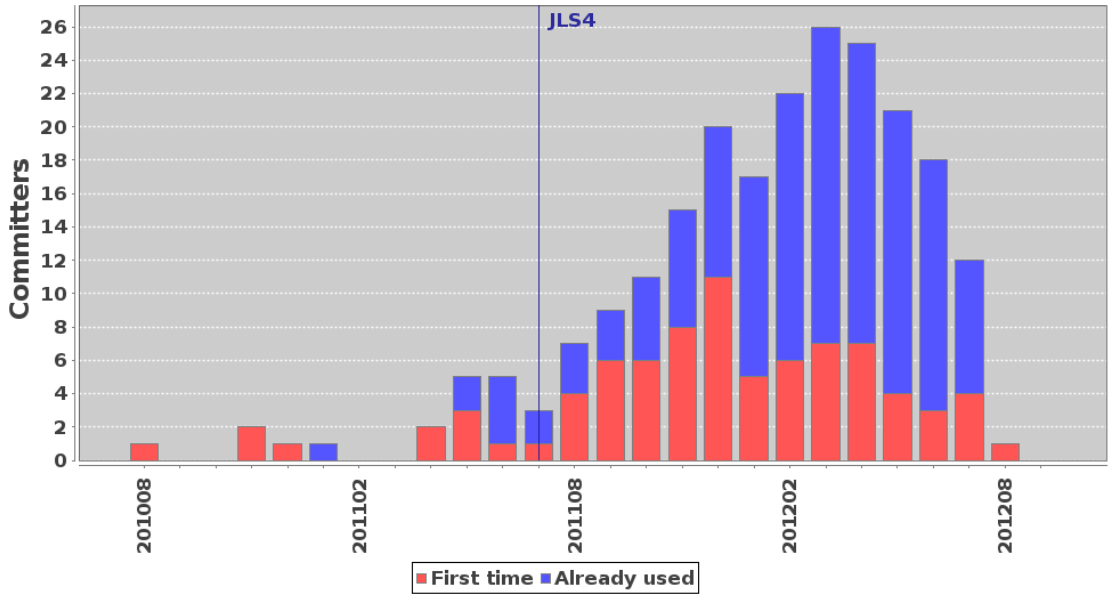
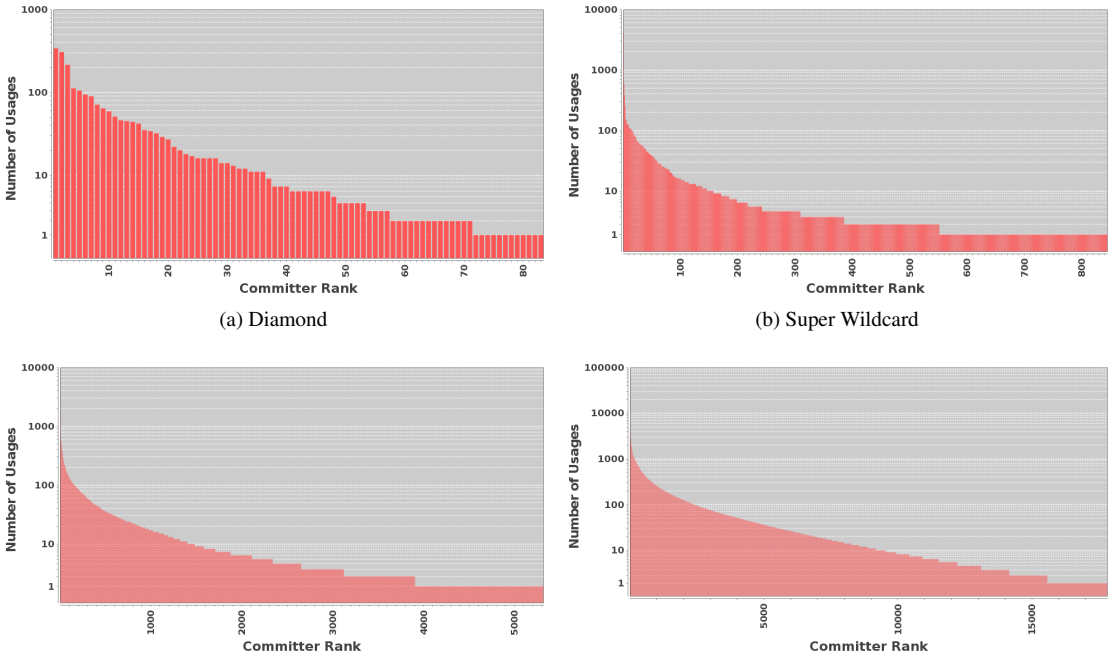


Fig. 14: Committers use of Diamond over time.



(a) Diamond

(b) Super Wildcard

(c) Extends Wildcard

(d) Annotation Use

Fig. 15: Use of language features by committers.

10 files, while a few committers used the feature in tens, hundreds, to thousands files. This trend holds for all features.

Comparing the four charts, we can see that the number of committers are quite different: about 85 for Diamond (15a), 850 for Super Wildcard (15b), more than 5,000 for Extends Wildcard (15c), and about 18,000 for

Annotation Use (15d), even though the last three features were all released in JLS3. In addition, the number of committers with the same number of uses varies among features. For example, at 10 uses, there are about 34, 160, 1,400 and 9,000 committers. This suggests that there are some popular feature(s) which are widely-used (e.g. Annotation) and others which are popular among another group of committers (e.g. Generics).

(a) Top-1 Assert committer (b) Top-2 Assert committers (c) Top-3 Assert committers

(d) Top-1 Annotation Use committer (e) Top-2 Annotation Use committers (f) Top-3 Annotation Use committers

Fig. 16: Proportion of feature uses in projects.

5.4.3 RQ4.3: Did committers adopt features on an individual basis or as a team?

We sought to answer this question by studying how many team members adopted a feature in a project. We first collected the set of committers for each project and identified who had used the features. We also counted the number of uses for each committer and ranked the committers in each project based on their number of uses. Then, for the top-k committers (k=1,2,3), we computed the proportion of their uses over the total number of uses in the whole project.

In Figure 11a, we learned that the distribution of the number of committers in a project is right-skewed. That is, many projects have only a few committers. In those projects, only one or two committers contribute to almost 100% of the uses. To avoid that bias and to study the team culture, we filtered out the projects having less than six committers.

The result is shown in Figure 16. Each chart shows the histogram of the proportions of feature usage in projects adopting that feature. The bins are the ranges of 1-10%, 11-20%, ..., and 91-100%. We show the result for only two features. The charts on the top, Figures 16a–16c, show the uses of Assert and are representative for the set of 15 less widely used features. The charts on the bottom, Figures 16d–16f, show the uses of annotations. They are representative for the 3 most widely used features (Enhanced For and Generic Variable).

Figure 16a shows the result for feature Assert. One committer contributes 100% uses in almost 300 projects and more than 80% uses in the majority of projects. In Figure 16c, when considering the top-3 users of Assert, the number of 100% uses increases to more than 550 projects and are almost all the projects using Assert. The same trend repeats for the uses of Annotation and the other 16 features (not shown).

This result indicates that a feature is not widely adopted by all members of the team, but instead are mainly championed by a small number of members. This is also consistent with the finding by Parnin et al (2011) even though they studied only 20 projects while we studied almost 1,800 projects, each of which has at least 6 committers.

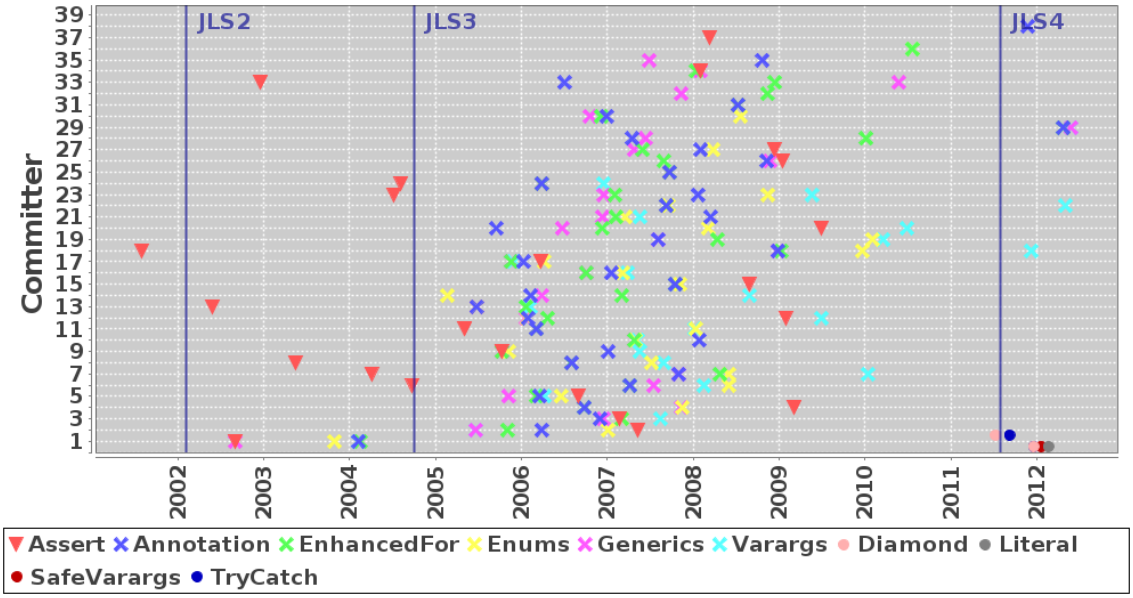


Fig. 17: Tracking features used by committers.

5.4.4 RQ4.4: Did committers use all new features?

We answered this question by tracking the feature uses of a group of "active" committers, who routinely committed code over a long enough period. Since the most new features were released in JLS3, we started with the set of committers in SourceForge at the release time of JLS3. We kept all committers that had routinely committed code at least every 3 months in the time between releases of JLS3 and JLS4. The remaining set contained 48 committers, among which only 38 had used at least one language feature in our study. The scatter graph in Figure 17 shows their uses over time.

For better visualization, we group related features from the same edition into groups, i.e. annotations define and use into Annotation, all generics features into Generics, Binary and Underscore Literals into Literal, and Try with Resources and MultiCatch into TryCatch. Each horizontal line from 1 to 38 shows the use over time for one committer.

As seen from the graph, among these 38 committers, only committer 2 adopted features from all three editions. Most committers used features from JLS2 and JLS3. JLS4 was recently released and thus has been used by only committers 1 and 2 (points in the lower right corner). Most of the committers used Assert, the only new feature in JLS2, however, they started late after its release. Meanwhile, the committers adopted JLS3 quite early and most of them used several different features. In terms of individual feature uses, up to now, no committer has used all studied features. Committers have used at most 7 among 10 different grouped features, mainly in JLS3.

5.5 Threats

We identified a threat regarding who commits code versus who actually wrote that code. Someone may commit a file they did not write, perhaps adding a file from another library so it is local in their own repository. Our analysis would attribute the source of that file to the person who committed it which is why we focused on committers, not developers.

A similar threat relates to the timestamps of committed code. If someone commits a file they did not write, the timestamp of the commit may be wrong. It is possible that features were actually used earlier than identified in RQ2.

We identified an external threat to our study regarding the generalizability of our results. Since we only studied open-source software, the results may not necessarily represent Java language feature usage by non-open source developers, such as those in industry. We also do not know the experience level of committers, which may vary greatly and limits our ability to generalize. We avoid generalizing our results and instead focus on if the trends we observed are similar to the trends the previous study by Parnin et al (2011) observed.

6 Previous Language Feature Studies

Grechanik et al (2010) performed a large-scale study on Java features on 2k projects from SourceForge. Their study looked at features such as: classes (abstract, nested, etc), methods (arities, return types, etc), fields, conditional statements, try/catches, etc. The majority of the features studied are object-oriented language features available since JLS1. They did not study newer language features in JLS3 or JLS4. Their study also focused on releases of projects and not the full history of the repositories.

Parnin et al (2011) mined the history of 20 open-source Java projects to evaluate how Java generics were integrated and adopted into open source software. As we already showed, our finding on the most popular generic types is consistent with their empirical result. It is also true for the finding that generics are usually adopted by individuals championing for the features, rather than all committers in the team. Basit et al (2005) performed an empirical study on two projects regarding how Java generics and C++ templates can help in code refactoring.

Livshits et al (2005) focused on the reflection feature in Java. They introduced a static-analysis based reflection resolution algorithm that uses points-to analysis to approximate the targets of reflective calls as part of the call graph. Callaú et al (2011) studied the reflection feature in Smalltalk. They reported that such a feature is mostly used in specific kinds of projects: core system libraries, development tools, and tests, rather than in regular applications.

Richards et al (2011) performed a large-scale study on the use of `eval` in JavaScript applications. `eval` is used to transform text into executable code, allowing programmers the ability to dynamically extend applications. They studied large-scale execution traces with 550,358 calls to the `eval` function exercised in over 10,000 websites. They found that it is often misused and many uses were unnecessary and could be replaced with equivalent and safer code. Earlier, Richards et al (2010) analyzed a smaller set of JavaScript programs and concluded the popular usage of `eval` and reported the degree of dynamism in those programs. Ratanaworabhan et al (2010) reported on an existing benchmark for JavaScript and focused on two aspects of JavaScript runtime behavior 1) functions and code and 2) events and handlers. Yue and Wang (2009) performed an empirical study on 6,805 unique websites regarding insecure practices of JavaScript inclusion and dynamic generation. They reported that over 44.4% of the measured websites dangerously use `eval`.

Gorschek et al (2010) performed a large-scale study on how developers use object-oriented concepts. Tempero (2009) studied how fields are used in Java and reported that it is common for developers to declare non-private fields, but then not take advantage of that access. Tempero et al (2008) found higher use of inheritance than expected and variation in the use of inheritance between interfaces and classes. Muschevici et al (2008) studied multiple dispatch in several languages and compared its uses.

While these previous studies have looked at various language features, most are limited to studying a few features, looked at a relatively small number of projects, or did not look at the full history of the software studied. Our study looks at most of Java's new language features, studies over 23k Java projects, and uses each file's full history.

Linstead et al (2009) describe the the Sourcerer project which provides a relational database of mined software artifacts. Their dataset contains over 18k Java projects from SourceForge and Apache. The data is modeled as entities, such as classes, methods, or fields, and relationships among those entities. The dataset contains the source code from the latest snapshot of each project. Grechanik et al (2010) also provided a framework for source code analysis of Java projects. Their dataset contains 2k Java projects from SourceForge. They also provide a relational database, though they contain full entity information from the source code all the way down to the expressions. Their dataset contains releases of projects.

Compared to these works, the focus of the Boa infrastructure Dyer et al (2013b,a); Rajan et al (2012) with over 23k projects is to provide the full history information of all Java files in the dataset, as well as entities down to the expression level. These features were essential for our study.

7 Future Work and Conclusion

Programming languages evolve over time to meet the assumed needs of developers. What was needed is a study to see how those features actually are used by developers. In this paper we investigated language feature usage for Java's three newest editions.

Our results showed that every feature is indeed used. The most used features we studied were the enhanced-for loops, declaring variables of generic type, and using pre-defined annotations. The first two features are related and our analysis indicated their heavy use is influenced by the Collections classes provided by Java's runtime. The heavy use of annotations, but relative lack of custom annotations, indicated the use was mostly by automated tools such as IDEs or code generators.

We also found millions of places features could have been used but were not. This included old files that could be refactored as well as new code. This suggests that IDEs can do a better job, perhaps by providing extra warnings and refactorings to use the new features. There may also be a need for better training and more advertising of new language features.

Our analysis also revealed that some developers were eager to use these features, even using them as far as six years before their release. Every feature was used prior to its release by at least one project. We found that in each project's team, there were usually only a few developers who championed adopting new features and as few as one to three members who contributed almost 100% of new feature uses.

In the future it would be interesting to perform a survey of the developers that appeared in this study to see why they chose to start using certain features when they did. Perhaps there is a need for better education/outreach to inform developers of these new features.

Acknowledgments

This work was supported in part by the National Science Foundation grants CCF-11-17937, CCF-10-17334, and CCF-10-18600.

References

Apache Software Foundation (2012) Hadoop: Open source implementation of MapReduce. `http://hadoop.apache.org/`

Basit HA, Rajapakse DC, Jarzabek S (2005) An empirical study on limits of clone unification using generics. In: In Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05), pp 109–114

Bracha G, Odersky M, Stoutamire D, Wadler P (1998) Making the future safe for the past: adding genericity to the Java programming language. SIGPLAN Not 33(10)

Callaú O, Robbes R, Tanter E, Röthlisberger D (2011) How developers use the dynamic features of programming languages: the case of Smalltalk. In: Proceedings of the 8th Working Conference on Mining Software Repositories, ACM, MSR '11, pp 23–32

Christensen AS, Møller A, Schwartzbach MI (2003) Precise analysis of string expressions. In: Proceedings of the 10th international conference on Static analysis, Springer-Verlag, SAS'03, pp 1–18

Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: OSDI

Dyer R, Nguyen H, Rajan H, Nguyen TN (2013a) Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: ICSE'13: 35th International Conference on Software Engineering, pp 422–431

Dyer R, Rajan H, Nguyen TN (2013b) Supporting fine-grained source code mining with full history on billions of AST nodes. In: TOSEM (in submission)

Gorschek T, Tempero E, Angelis L (2010) A large-scale empirical study of practitioners' use of object-oriented concepts. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ACM, ICSE '10, pp 115–124

Gosling J, Joy B, Steele G (1996) Java(TM) Language Specification, 1st edn. Addison-Wesley Longman Publishing Co., Inc.

Gosling J, Joy B, Steele G, Bracha G (2000) Java(TM) Language Specification, 2nd edn. Addison-Wesley Longman Publishing Co., Inc.

Gosling J, Joy B, Steele G, Bracha G (2005) Java(TM) Language Specification, 3rd edn. Addison-Wesley Professional

Gosling J, Joy B, Steele G, Bracha G, Buckley A (2013) Java(TM) Language Specification, Java SE 7 edn. Prentice Hall

Grechanik M, McMillan C, DeFerrari L, Comi M, Crespi S, Poshyvanyk D, Fu C, Xie Q, Ghezzi C (2010) An empirical investigation into a large-scale Java open source code repository. In: ESEM'10: International Symposium on Empirical Software Engineering and Measurement, pp 11:1–11:10

Linstead E, Bajracharya S, Ngo T, Rigor P, Lopes C, Baldi P (2009) Sourcerer: mining and searching internet-scale software repositories. Data Mining and Knowledge Discovery 18

Livshits B, Whaley J, Lam MS (2005) Reflection analysis for Java. In: Proceedings of the Third Asian conference on Programming Languages and Systems, Springer-Verlag, APLAS'05, pp 139–160

Muschevici R, Potanin A, Tempero E, Noble J (2008) Multiple dispatch in practice. In: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, ACM, OOPSLA '08, pp 563–582

Parnin C, Bird C, Murphy-Hill ER (2011) Java generics adoption: how new features are introduced, championed, or ignored. In: MSR

Rajan H, Nguyen TN, Dyer R, Nguyen HA (2012) Boa website. `http://boa.cs.iastate.edu/`

Ratanaworabhan P, Livshits B, Zorn BG (2010) Jsmeter: comparing the behavior of JavaScript benchmarks with real web applications. In: Proceedings of the 2010 USENIX conference on Web application development, USENIX Association, WebApps'10

Richards G, Lebresne S, Burg B, Vitek J (2010) An analysis of the dynamic behavior of JavaScript programs. In: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, ACM, PLDI '10

Richards G, Hammer C, Burg B, Vitek J (2011) The eval that men do: A large-scale study of the use of eval in JavaScript applications. In: Proceedings of the 25th European conference on Object-oriented programming, Springer-Verlag, ECOOP'11, pp 52–78

SourceForge (2013) Sourceforge website. `http://sourceforge.net/`

Tempero E (2009) How fields are used in Java: An empirical study. In: Proceedings of the 20th Australian Software Engineering Conference, ASWEC'09, pp 91–100

Tempero E, Noble J, Melton H (2008) How do Java programs use inheritance? An empirical study of inheritance in Java software. In: Proceedings of the 22nd European conference on Object-Oriented Programming, Springer-Verlag, ECOOP '08, pp 667–691

Yue C, Wang H (2009) Characterizing insecure JavaScript practices on the web. In: Proceedings of the 18th international conference on World Wide Web, ACM, WWW '09, pp 961–970