An Empirical Study of Method Chaining in Java

Tomoki Nakamaru
nakamaru@csg.ci.i.u-tokyo.ac.jp
The University of Tokyo

Tomomasa Matsunaga
matsunaga@csg.ci.i.u-tokyo.ac.jp
The University of Tokyo

Tetsuro Yamazaki
yamazaki@csg.ci.i.u-tokyo.ac.jp
The University of Tokyo

Soramichi Akiyama
akiyama@ci.i.u-tokyo.ac.jp
The University of Tokyo

Shigeru Chiba
chiba@acm.org
The University of Tokyo

ABSTRACT

While some promote method chaining as a good practice for improving code readability, others refer to it as a bad practice that worsens code quality. In this paper, we first investigate whether method chaining is a programming style accepted by real-world programmers. To answer this question, we collected 2,814 Java repositories on GitHub and analyzed historical trends in the frequency of method chaining. The results of our analysis revealed the increasing use of method chaining; 23.1% of method invocations were part of method chains in 2018, whereas only 16.0% were such invocations in 2010. We then explore language features that are helpful to the method-chaining style but have not been supported yet in Java. For this aim, we conducted manual inspections of method chains that are randomly sampled from the collected repositories. We also estimated how effective they are to encourage the method-chaining style if they are adopted in Java.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; *Software design engineering*.

KEYWORDS

Method chaining, Repository mining, Quantitative analysis

ACM Reference Format:
Tomoki Nakamaru, Tomomasa Matsunaga, Tetsuro Yamazaki, Soramichi Akiyama, and Shigeru Chiba. 2020. An Empirical Study of Method Chaining in Java. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3379597.3387441

1 INTRODUCTION

Method chaining is a programming style in which multiple method invocations are chained in a single expression as follows:

```
new AlertDialog()
    .setTitle("Warning")
    .setMessage("Are you sure?").show();
```


MSR '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387441

Method chaining is promoted as a good practice that improves the readability of source code. By method chaining, redundant temporary variables and code repetitions are eliminated [8]; related method invocations are grouped into a single expression [34, 42]; an expression becomes easy to read from left to right as natural-language texts [13, 14, 18, 42].

However, at the same time, method chaining is often referred to as a bad practice. In the thread on StackOverflow [8], many posts claim that method chaining worsens the readability. For instance, a post says:

> If you do everything in a single statement then that is compact, but it is less readable (harder to follow) most of the times than doing it in multiple statements.

Another post mentions that method chaining makes code confusing since it hides the type of an object that a programmer operates upon. Negative opinions are not only about code readability. One post in the thread [8] states that method chaining is not reconcilable with most debuggers that offer line-level breakpoints:

> You can't put the breakpoint in a concise point so you can pause the program exactly where you want it. If one of these methods throws an exception, and you get a line number, you have no idea which method in the "chain" caused the problem.

Further, several posts in the thread [8] claim that method chaining violates the law of Demeter [27], a guideline for developing loosely coupled software.

The controversy above leads us to our first question: Is method chaining accepted widely by real-world programmers? To answer this question, we collected Java repositories on GitHub and analyzed historical trends in the frequency of method chaining. If method chaining is commonly considered as a bad practice, programmers would avoid chaining method invocations. As a result, the frequency would be the same or decrease over time. Conversely, if the frequency increases, that result will be supportive evidence for the wide acceptance of method chaining in the real world.

The answer to our first question is yes; the frequency of method chaining has increased considerably. Our analysis revealed that 23.1% of method invocations were part of method chains in 2018, whereas only 16.0% were such invocations in 2010. To help us better understand the trends, this paper also presents our investigation on the bias in the frequency and the categories of method chains.

Our second question then arises in response to the first answer: How can we extend the Java language to support this increasing trend (or what API design should library developers adopt)? To answer this question, we manually analyzed method chains that

Algorithm 1 Dataset construction

Input: Set of repositories *Repositories*
Output: Dataset *Dataset*
1: **for each** *repository* ∈ *Repositories* **do**
2: *Name* ← Name of *repository*
3: *Revisions* ← Year-end revisions of *repository*
4: **for each** *revision* ∈ *Revisions* **do**
5: *year* ← Year of *revision*
6: **for each** `.java` file *file* in *revision* **do**
7: *code* ← Content of *file*
8: Add (*code*, *name*, *year*) to *Dataset*

are randomly sampled from the collected repositories. We present language features (or API design) that we discovered are helpful for encouraging the method-chaining style in Java. Those features are supported in other languages but have not been supported yet in Java. We also estimated how effective they are if they are adopted in Java.

The contribution of this paper is summarized as follows:

- We present, to the best of our knowledge, the first quantitative study on the use of method chaining that is based on a large set of source code in the real world.
- We empirically show the increasing use of method chaining in Java, which has been claimed without empirical evidence in preceding studies [20, 21, 26, 33, 34, 42, 43].
- We present language features (or API design) that support method chaining but are not supported yet in Java. We statistically estimated how effective each feature/design would be.

The remainder of this paper is organized as follows: Section 2 describes the dataset and method that we use to answer our questions. Section 3 and 4 show the results of our analyses. Section 5 discusses the threats to validity of our results. Section 6 relates our work to preceding studies, and Section 7 concludes our paper.

2 MATERIALS AND METHODS

2.1 Dataset

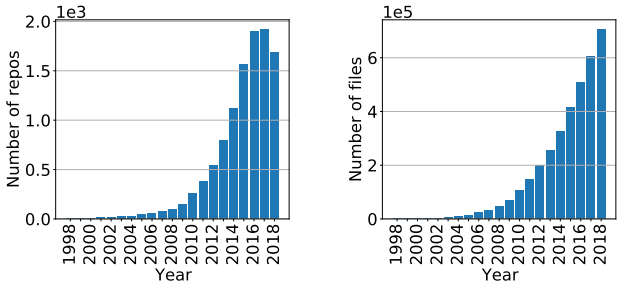
To build our dataset, we collected 2,814 Java repositories on GitHub. Those repositories are the ones that were listed at least once in the most-starred 1000 Java repositories on GitHub between Nov. 10th, 2019 and Dec. 21st, 2019. We collected them by monitoring the response of the GitHub API[1] every day during that period.

We built our dataset by extracting syntactically valid `.java` files from the year-end revisions of each repository in the most-starred repository set. The year-end revision of a year is the latest revision made in that year. We marked a `.java` file as syntactically valid when JavaParser[2], a parser often used both in industry and academia, successfully parses the content of that file. To find year-end revisions, we use the command `git rev-list <branch>`, which lists all the revisions reachable from `<branch>`. We set `<branch>` to the default branch[3] of the repository. The default branch differs depending on the configuration of the repository on GitHub, but it is `master` in most repositories.

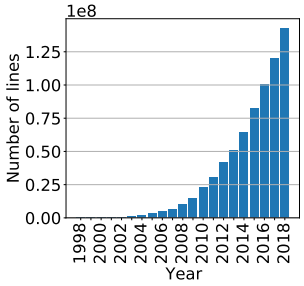
An entry of the dataset is a tuple (*code*, *name*, *year*), where *code* is the content of a source file, *name* is the name of the repository to which the file belongs, and *year* is the year of the revision to which the file belongs. Algorithm 1 shows pseudocode for constructing our dataset from a given set of repositories. A set of the most-starred repositories were used as the input to that algorithm.

Our dataset contains over three million Java files (approximately seven hundred million lines) in total. Figure 1 shows the number of repositories, files, and lines in each year. As seen, the amount of the collected code considerably varies from year to year. This property of our dataset indicates that it is inappropriate to directly compare the raw numbers of the method chains in each year.

[1] https://api.github.com/search/repositories?q=language:java&sort=stars
[2] https://javaparser.org
[3] https://help.github.com/en/articles/setting-the-default-branch



(a) Number of repositories (b) Number of files



(c) Number of lines

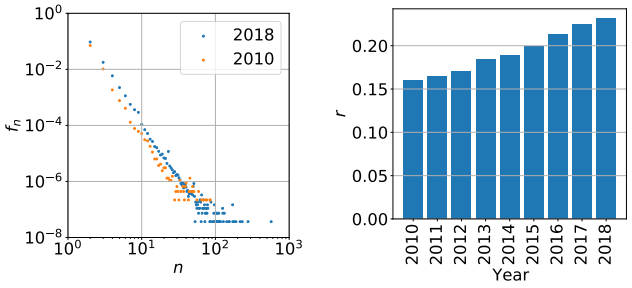
Figure 1: Number of repositories, files, and lines

```
List<String> list = new ArrayList();
list.add(
  createRandomString() // Length = 1
); // Length = 1
list.stream().map(s -> {
  return s.replace("foo", "bar")
          .replace("baz", "qux"); // Length = 2
}).forEach(s ->
  String output = String.format(
    "%d lines",
    s.split("\n").length // Length = 1
  ); // Length = 1
  System.out.println(output); // Length = 1
); // Length = 3
```

Figure 2: Method chains and their lengths

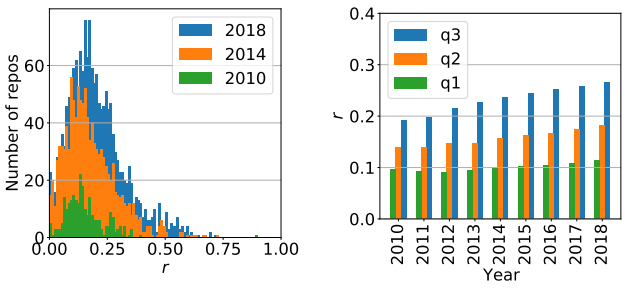
2.2 Definition of Method Chain

We define a method chain as a sequence of one or more method invocations joined by the "." symbol. We define the length of a



(a) f_n in 2010 and 2018 (b) r

Figure 3: Frequency of method chaining



(a) Distribution (b) Trend of quartiles

Changes of quartiles from 2010 to 2018

1st quartile	2nd quartile	3rd quartile	Average
+1.71%	+4.27%	+7.34%	+7.08%

Figure 4: Distribution and trend of per-repository r

method chain as the number of invocations in the sequence. For example, the Java code snippet shown in Figure 2 contains five chains of length 1, one chain of length 2, and one chain of length 3. We used JavaParser to parse a Java file and mined the parsing result for method chains.

We first enumerated the chains of length 1 (not-chained method invocations) from our dataset to obtain the baseline values. If the number of not-chained invocations increases at the same pace as the number of method chains longer than one, we cannot argue that the use of method chaining is growing. Note that, for convenience, we below regard a not-chained invocation as a method chain.

3 IS METHOD CHAINING ACCEPTED?

As we described in Section 1, we measure the frequency of method chaining to judge whether it is accepted in the real world. In our analyses, we use only the code that is newer than or in 2010. 2010 is the year in which the number of repositories exceeds 250 and in which the number of files exceeds 10^5 for the first time. We adopt this criterion to avoid that the programming styles of a small number of old repositories overly affect our analysis.

We use the following indicators f_n and r to measure the frequency:

$$f_n = m_n/m_1,$$
$$r = (\Sigma_{2\geq n} n m_n)/(\Sigma_{1\geq n} n m_n),$$

where m_n is the raw number of method chains of length n. The indicator f_n is the relative occurrence of chains of length n. The indicator r is the ratio of the method invocations that are part of method chains longer than 2, among all the method invocations. For example, the f_2 value of the code in Figure 2 is 1/5 (= 0.2) since the code includes five chains of length 1 and one chain of length 2. The r value of the example code is 5/10 (= 0.5) since the code includes ten method invocations in total and five of them constitute the method chains of length 2 or 3.

Figure 3a shows the plot of f_n for the code in 2010 and 2018. (The f_n values are computed over the total dataset and not the averages of per-file values.) The horizontal axis shows the value of n and the vertical axis shows the value of f_n. Note that we use logarithmic scales for both vertical and horizontal axes. Figure 3b shows the plot of r from 2010 to 2018. Although we omit the plot of f_n values from 2011 to 2017, similar distributions are observed in those years.

Either of Figure 3 shows that the increasing use of method chaining. The relative number of method chains has increased from 2010 to 2018 in almost all lengths. The r value has increased from 16.0% to 23.1%. Further, the maximum length of a chain has also been increased from 2010 to 2018.

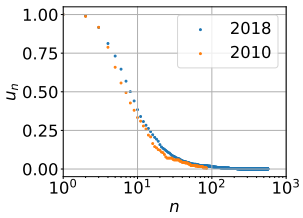
3.1 Power-law Distribution

As seen in Figure 3, f_n decreases almost linearly in the log-log scale plot. Such a linear decrease is observed when a distribution has a heavy tail [35]. The heavy tail indicates that extremely long chains often appear and their occurrences are not exceptional, unlike a normal distribution. Such distributions are found in several measures of source code such as change sizes [22, 28, 41], component sizes [25], in-degree and out-degree in dependency networks [29], and the number of subclasses [40].

We performed the Kolmogorov-Smirnov (KS) test to see whether the right tail of f_n in 2018 is generated by a power-law distribution, a well-known heavy-tailed distribution. The test reported 9 for x_{min} and 0.937 for p-value. The p-value is greater than the commonly used significance level 0.05. These results of the KS test indicate that it is consistent to assume that the observed values for $n \geq 9$ are generated by a power-law distribution. Although it is interesting to discover the generation model of such a distribution, we leave it for future work.

3.2 Bias in Frequency

Since the two indicators shown in Figure 3 represent average values for the repositories in our dataset, we observed only the average trend in our dataset. Thus we then examined a hypothesis, that only a small number of large repositories in our dataset might contain a large number of method chains and increase the total number of chains in our dataset while others contain a small number of chains. To reveal the bias of the increase found in Figure 3, we computed the r value for each repository and carried out the analysis on their distribution.



n	u_n in 2018	u_n in 2010
1	100%	100%
8	50.1%	42.7%
9	44.1%	38.0%
41	5.10%	4.31%
42	4.98%	4.31%

Figure 5: Ratio containing chains longer than or equal to n

Figure 4a shows the histograms of per-repository r in 2010, 2014, and 2018. The horizontal axis shows the value of r and the vertical axis shows the number of repositories. Figure 4b shows the trend of their quartiles.

Figure 4 shows that the overall r value is increased not only by a few repositories. If the increase has occurred only in a small number of repositories, we would not have observed changes in the first and second quartiles. However, the increase of overall r value is increased significantly by repositories containing a lot of method chains since the change in the third quartile is much larger than the change in the first quartile.

To see the widespread use from a different perspective, we calculated the following value:

u_n: The ratio of repositories that contain one or more method chains whose length is longer than or equal to n.

The left chart in Figure 5 shows the plot of u_n in 2010 and 2018. The horizontal axis shows the value of n and the vertical axis shows the value of u_n. Note that we use logarithmic scales for the horizontal axis. The table on right shows the values that we refer to in later analyses.

Figure 5 shows that more than 50% of repositories contain at least one chain longer than length 7. Since chains of length 8 are unlikely to be composed by programmers who tend to avoid method chaining, this result is another supportive evidence for the widespread use of method chaining.

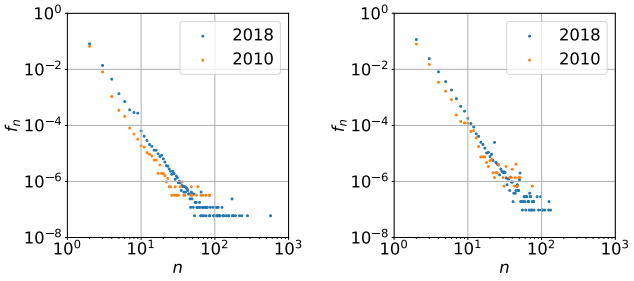
We also investigated the difference in the use of method chaining between testing code and non-testing code. Figure 6 shows the plot of f_n values and the changes in r values in those code sets.

The figures show that the testing code contains more method chains longer than 2. The increasing amount in the testing code (+8.57%) is larger than the one in the non-testing code (+5.37%). On the other hand, the maximum length in the non-testing code is longer than the one in the testing code. Although there are those differences, the increasing trends and the heavy-tailed distributions can be seen in both code sets. From these results, we concluded that method chaining is used not only in either code set but in both sets.

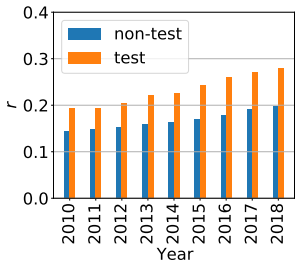
3.3 Categories

To better understand the trends in method chaining, we manually categorized the method chains in 2018 and 2010 by their behaviors.

For this analysis, we divided the set of method chains into three groups by their length: Short, Long, and ExtLong. Table 1 shows the range of lengths and the number of chains for each group. We



(a) f_n of non-testing code **(b) f_n in testing code**



(c) r

Figure 6: Non-testing code vs. Testing code

Table 1: Groups of Method Chains

	Short	Long	ExtLong
Length range	$1 < \text{len.} \leq 8$	$8 < \text{len.} < 42$	$42 \leq \text{len.}$
# of chains in 2018	3,343,781	19,084	280
# of chains in 2010	384,549	1,106	27

chose the border length 8 and 42 in consideration of u_n values, the ratio of repositories that contain one or more method chains whose length is longer than or equal to n. In 2018, more than 50% of repositories contain chains longer than 8, and less than 5% of repositories contain chains longer than 42, as shown in Figure 5.

Since it is not feasible to manually inspect all the chains in Short and Long, we analyze randomly-sampled 280 method chains in each of those groups. We analyze all the chains in ExtLong.

We categorized a method chain into either of Accessor, Builder, Assertion, and Others. Figure 7 describes the first three of them by example. Others is the category for chains that do not match any of Accessor, Builder, and Assertion.

Figure 8 illustrates the ratios of each category in Short and Long. The black bars in the figure indicate the margin of errors due to the sampling at a 95% level of confidence. Since we did not find any Accessor chain in the samples of Long, no blue bars are drawn in Long. The error bar for Assertion in Long in 2010 is not drawn since the number of chains is too low to compute its valid margin. All the chains in ExtLong are categorized into Builder in both 2010 and 2018.

Figure 8 shows that approximately 80% of chains in Short are categorized into Accessor in 2010. In 2018, the ratio of Accessor

```
// Access
jfc.getCategoryPlot().getRangeAxis();
miniCluster.getNameNode().getNamesystem()
    .getBlockManager().getDatanodeManager()
    .getNumStaleNodes();
histogram.getBuckets().get(0).getKey();
getSubscriptionAttributes()
    .getInterestPolicy().isCacheContent();
// Access for operation
index.getLibraries().add(libIndex);
getSupportActionBar()
    .setDisplayShowTitleEnabled(false);
```

Accessor. A chain where all methods perform data access except for the last method. Although a chain in this category violates the law of Demeter [27], it frequently appears in the real world code.

```
// java.lang.StringBuilder
sb.append("New").append(kindName).append("Array");
// com.google.common.base.MoreObjects
MoreObjects.toStringHelper(this)
    .add("iLine", iLine)
    .add("lastK", lastK)
    .add("spacesPending", spacesPending)
    .add("newlinesPending", newlinesPending)
    .add("blankLines", blankLines)
    .add("super", super.toString())
    .toString()
```

Builder. A chain that builds an object. It often ends with the invocation of a method named like `buildFoo` or `toFoo`.

```
// Mockito
when(myHttpClient.execute(capt.capture()))
    .thenReturn(myHttpResponse);
verify(map).containsKey("testOk");
// AssertJ
assertThat(kunaTimeTicker.getTicker()).isNotNull();
Assertions
  .assertThat(actualObj.has("outline_colors"))
  .isTrue();
```

Assertion. A chain that describes expected behaviors of an object. Understandably, such chains are written in test code. We found usages of the two libraries Mockito and AssertJ in the sampled set for this category.

Figure 7: Categories of Method Chains

decreases to approximately 55%, and the ratios of Builder and Assertion increase accordingly. These changes in the ratios could be explained by the general acceptance of fluent interfaces, API design that encourages its users to chain method invocations [18, 23]. An Accessor chain can be composed even when the library is not fluent. On the other hand, a Builder/Assertion chain needs a fluent interface to compose. In Java, object building and assertions are used to be written in the non-chaining style as follows:

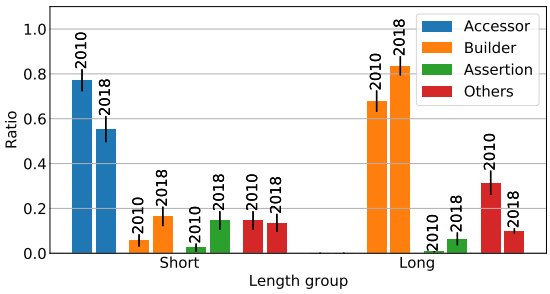


Figure 8: Constitution ratio of each category

```
// new Builder().setFoo().setBar().build();
Builder b = new Builder();
b.setFoo();
b.setBar();
Object o = b.build();
// assertThat(list).contains("a").contains("b");
assert list.contains("a");
assert list.contains("b");
```

Thus, the increase in the ratio of Builder/Assertion chains indicates the increasing use of fluent interfaces. The same trends can be seen in the changes in Long. The ratio of Others considerably decreases as the use of Builder and Assertion chains increases. The increasing use of fluent interfaces is supportive evidence for the wide acceptance of method chaining.

Although we expected to frequently encounter the use of the Stream API in Others, we found only two such chains (0.71%) in Short and three chains (1.07%) in Long in 2018. We found no Stream API usages in 2010 as the API is not introduced into Java in 2010.

3.4 Extremely Long Chains

Since we did not immediately see why and how such long chains exist in the real-world code, we conducted further inspection of the chains in ExtLong in 2018.

3.4.1 How much of the ExtLong chains are in testing code? We found 140 chains (50% of ExtLong) in testing code. As mentioned above, all the chains in ExtLong are composed to build an object. Thus, half of the ExtLong chains build objects for testing.

3.4.2 Are the ExtLong chains machine-generated? We found only five generated chains (1.79% of ExtLong). All of those chains are generated by `aws-java-sdk-code-generator`. Most chains in ExtLong are very likely to be written by human programmers.

3.4.3 Which libraries produce the ExtLong chains? To answer this question, we checked the package name of the first method invocation of each chain. We found 71 different package names, which indicates that various libraries are used to compose extremely long chains. However, a large bias exists in the number of appearances of each library: 53.5% of the libraries are used only once; Three libraries constitute 43.9% of the appearances. The following summarizes the most-used three libraries:

Elasticsearch[4] We found 75 chains (26.8%) using `XContentFactory` or `XContentBuilder` in this library. Those classes are for building data used in Elasticsearch.
Guava[5] We found 30 chains (10.7%) using immutable collection builders (e.g. `ImmutableSet.Builder`) in this library.
Java Std. Lib. We found 18 chains (6.43%) using `StringBuilder` or `StringBuffer` in `java.lang`

3.4.4 Are the ExtLong chains styled nicely? To improve the readability of extremely long chains, programmers often introduce semantical indentations as follows:[6]

```
String jsonString = new PrettyJSON()
  .array()
    .object()
      .key("method")  .value("POST")
      .key("to")      .value("...")
      .key("body")
        .object()
          .key("key").value("ID")
          .key("value").value("fra")
          ... # Our comment: Omitted
        .endObject()
      .key("id")      .value(0)
    .endObject()
    ... # Our comment: Omitted
  .endArray().toString();
```

Another technique is to insert empty lines and comments to group semantically related part as follows:[7]

```
return new SpacingBuilder(
  settings, BallerinaLanguage.INSTANCE)
  // Keywords
  .around(IMPORT).spaceIf(true)
  .around(AS).spaceIf(true)
  .around(CHECK).spaceIf(true)
      # Our comment: Empty line for segmentation
  .around(ABORTED).spaceIf(true)
  .around(COMMITTED).spaceIf(true)
  .around(LISTENER).spaceIf(true)
  ... # Our comment: Omitted
```

We counted the number of chains that are styled nicely as shown above. Our inspection revealed that 184 chains are nicely-styled, which is 66.9% of handwritten chains.

RQ: Is Method Chaining Accepted in the Real World?
Answer: Method chaining is increasingly used in the real world. In 2018, 23.1% of method invocations are part of chains longer than 2, while 16.0% are such invocations in 2010. More than 50% of repositories contain at least one chain that is longer than seven. Approximately 5% of repositories contain chains that are longer than 42. Further, the increase is not caused by a few repositories that heavily use method chaining.

We also observed the increasing use of fluent interfaces, which is a supportive result for the wide acceptance of method chaining. In 2010, approximately 80% of chains are accessor chains, those that can be composed without fluent interfaces. The ratio of accessor chains decreased to approximately 55% in 2018, and the ratio of builder/assertion chains – those that require fluent interfaces to compose – increased accordingly.

All the chains longer than 42 are builder chains. 98.2% of them are very likely to be handwritten, and 65.7% are styled nicely with indentations, empty lines, and comments. The variety of such extremely long chains is unexpectedly wide. We found that 71 packages are used for composing 280 extremely long chains.
Implications: Our results are supportive evidence for the wide acceptance of method chaining in the real world. If method chaining is commonly considered as a bad practice, the use of method chaining would be the same or decreasing. However, to clearly state that method chaining is accepted, user studies need to be conducted. Such studies are our primary future work.

The above-mentioned implication will motivate the developers of fundamental software such as languages, libraries, and integrated development environments (IDEs). It will be a supportive and quantitative ground in the discussion of adding new functions for method chaining. For example, library developers can claim that adding fluent interfaces is beneficial for their users; IDE developers can discuss the priority of supporting breakpoints between method invocations in a chain and of a code formatting feature for long chains. The answer also motivates researchers of tools for developing safe fluent interfaces [20, 21, 26, 33, 34, 42]. The researchers can quantitatively state that their tools and further studies on the tools are beneficial for a number of real-world programmers.

4 HOW CAN WE EXTEND JAVA?

To investigate what language features (or library design) needs to be supported to encourage the use of the method-chaining style, we manually analyzed randomly-sampled 385 chains and the code around them. Since we are interested in why method invocations are not chained, the population of the random sampling was the chains of all lengths (including non-chained invocations) in 2018.

We attempted to find code patterns that could be transformed into the method-chaining style if Java supported an appropriate language feature or the library supported appropriate API design. In what follows, we present the found patterns with their description by example and the number of their occurrences in the randomly-sampled dataset. We then present appropriate language features or API design for those patterns. At the end of this subsection, we summarize the estimated ratios of those patterns in the population.

4.1 NullExceptionAvoidance

When a method invocation may return `null`, a programmer cannot chain all related method invocations; for example, as follows:[8]

```
JAXBMapping mapping = jaxbModel.get(qname);
```

[4] https://github.com/elastic/elasticsearch
[5] https://github.com/google/guava
[6] https://github.com/neo4j/neo4j/blob/a43b26fac61c59da813ec9302a24dd86f6657537/community/server/src/test/java/org/neo4j/server/rest/BatchOperationIT.java#L501
[7] https://github.com/ballerina-platform/ballerina-lang/blob/27292e84b9f661da89b6c66840802f2196decb0d/tool-plugins/intellij/src/main/java/io/ballerina/plugins/idea/formatter/BallerinaFormattingModelBuilder.java#L328
[8] https://github.com/corretto/corretto-8/blob/32a35a24e2791bc810a0b4d89ad685c97e4485fa/src/jaxws/src/share/jaxws_classes/com/sun/tools/internal/ws/processor/modeler/wsdl/JAXBModelBuilder.java#L119

```
if (mapping == null){
  return null;
}
return mapping.getType().getTypeAnn();
```

A `NullPointerException` may be thrown by `get(qname)` if he or she simply chains all the method invocations `get(qname)`, `getType()`, and `getTypeAnn()`. We classified a chain into NullException-Avoidance when `null`-checking has to be performed on the receiver side of the first invocation in a chain. We found nine chains (2.34%) of this pattern in the sampled dataset.

The safe call syntax in Kotlin [7] is helpful for this pattern.

```
return jaxbModel.get(qname)? // Safe call
    .getType().getTypeAnn();
```

The code above invokes `getType()` only when `get(qname)` returns a non-null object. With this syntax, a programmer can group semantically related invocations into a single chain. Furthermore, the following negative opinions in the thread on StackOverflow [8] will be addressed by introducing the safe call syntax:

> Chaining different objects can also lead to unexpected null errors. ... there's no guarantee (as an outside developer looking at the code) that getSchedule will actually return a valid, non-null schedule object.

Equivalent syntax also exists in Swift [9] and TypeScript [11]. The syntax is called optional chaining syntax in those languages.

Library developers can also provide better user experience by using `Optional<T>` for methods that possibly return `null`. The example code of this pattern can be transformed into the following if `get(qname)` returns `Optional<JAXBMapping>`:

```
return jaxbModel
    .get(qname) // returns Optional<JAXBMapping>
    .map(qname -> qname.getType().getTypeAnn())
    .orElse(null);
```

The code above would be easier to understand since all the related invocations are grouped into a chain.

4.2 RepeatedReceiver

Different methods are often invoked on the same object as follows:[9]

```
event.getPresentation().setEnabled(true);
event.getPresentation().setVisible(true);
```

This code repeats the expression `event.getPresentation()`. This repetition is not preferable in terms of code readability. We classified a chain as RepeatedReceiver when the receiver of the last invocation is the same as the previous/next chain. We excluded a chain from this pattern if the receiver is `this` or a class since no code repetition exists in such cases. We also excluded chains where programmers avoid chaining on purpose; for example as follows:[10]

```
// These two statements can be written as a chain,
// however, they are written separately.
// sb: java.lang.StringBuilder
sb.append("hasFilter:");
sb.append(this.hasFilter);
```

In the sampled dataset, we found 33 chains (8.57%) of this pattern.

The method cascading syntax in Smalltalk [12] and Dart [6] is useful for removing such repetitions. With this syntax, the example code of this pattern can be written as follows:

```
event.getPresentation()
    .setEnabled(true)
    ..setVisible(true);  // Dart-style syntax
```

The repetition can also be removed by setting the return value of `setEnabled(...)` to `this`. However, when considering the descriptive role of return types, some may think that it is not desirable to return a value in a method named like `setFoo`. In that case, it might be a good convention to name a normal setter (returning nothing) as `setFoo` and a fluent setter (returning `this`) as `withFoo`. Such a naming convention is adopted in `TemporaryCredential` in AWS SDK[11].

4.3 DownCast

When the invocation of a method needs downcasting, a chain is frequently split as follows:[12] to make code easily understandable

```
firstBtn = (Button) findViewById(R.id.firstBtn);
firstBtn.setText(START);
```

If a programmer wants to write a single chain for this operation, he or she needs to write nested parentheses as follows:

```
((Button) findViewById(R.id.firstBtn))
    .setText(START);
```

However, the nested parentheses worsen the readability. We classified a chain as DownCast when it contains cast operations. Six chains (1.56%) of this category are found in the sampled dataset.

Providing a method for downcasting relieves the problem in the DownCast chains.

```
// When destination types are practically known
findViewById(R.id.firstBtn).asButton()
    .setText(START);
// When destination types are unknown
findViewById(R.id.firstBtn).as(Button.class)
    .setText(START);
```

With such a method, an expression can be read from left to right easily. Although the solution described here is for library developers, it would be unnecessary if the top type `Object` provides the downcasting method `as(...)`. This is a candidate for language extension to Java for supporting method chains.

[9] https://github.com/eclipse/che/blob/e4d0f9987db58f3d46a3a727b88e601e84a5749b/ide/che-core-ide-app/src/main/java/org/eclipse/che/ide/processes/actions/StopProcessAction.java#L73

[10] https://github.com/apache/incubator-pinot/blob/09eb0150dec47a28d5a4517e4930183eb5dfd0af/pinot-common/src/main/java/com/linkedin/pinot/common/request/QueryType.java#L567

[11] https://github.com/aws/aws-sdk-java/blob/dafccf5a1241b5655c542a45eae05a582c3225de/aws-java-sdk-opsworks/src/main/java/com/amazonaws/services/opsworks/model/TemporaryCredential.java#L186

[12] https://github.com/yaowen369/DownloadHelper/blob/a27944d175cc48ddbe06151db8ad7cb415e9fa60/sample/src/main/java/com/yaoxiaowen/download/sample/MainActivity.java#L144

Table 2: Estimated Ratio of Pattern

Pattern	Ratio	$c = 95\%$	$c = 99\%$
NullExceptionAvoidance	2.34%	±1.51%	±1.98%
RepeatedReceiver	8.57%	±2.80%	±3.67%
DownCast	1.56%	±1.24%	-
ConditionalExecution	2.34%	±1.51%	±1.98%
	14.8%	±3.55%	±4.66%

4.4 ConditionalExecution

Some methods are invoked only when certain conditions are satisfied; for example as follows:[13]

```
if (buildLogger.isInfoEnabled()) {
  buildLogger.info(message, throwable);
}
```

We classified a chain as ConditionalExecution when it is conditionally executed as shown above. We found nine chains (2.34%) of this pattern in the sampled dataset.

The code in this pattern can be transformed into a single chain if the library provides a method that takes a lambda expression as its argument:

```
buildLogger.ifInfoEnabled(
  logger -> logger.info(message, throwable));
```

This workaround is largely adopted in the JavaParser library. Other Java libraries could adopt this workaround.

4.5 Estimated Ratios

We statistically estimated the ratio in the population from the results obtained from the analysis of the randomly-sampled dataset. Table 2 summarizes the estimated ratios. The columns "$c = n\%$" show margins of errors at $n\%$ level of confidence. We put the symbol "-" when the ratio in the sampled dataset is too low to compute the valid margin of errors at that level of confidence. The last row of the table shows the values for the sum of those discovered patterns.

As shown in the table, approximately 15% of chains can be combined into a single chain with other invocations around that chain if the appropriate language features or API design are provided. Since we counted the number of chains conservatively, the real ratios might be larger than the values shown in the table. For example, we did not classify the following chain as NullExceptionAvoidance:

```
JAXBMapping mapping = jaxbModel.get(qname);
updateFoo();  // Possibly changes states
if (mapping == null){
  return null;
}
return mapping.getType().getTypeAnn();
```

By exchanging the first and second statements, the above code can be transformed into a single chain if the safe call syntax is available. However, this exchange possibly changes the semantics of the code. Thus, we did not count such code for NullExceptionAvoidance.

[13] https://github.com/raphw/byte-buddy/blob/9364421492e830b883d10d9718d6586480e35747/byte-buddy-dep/src/main/java/net/bytebuddy/build/BuildLogger.java#L535

RQ. How Can We Extend Java?
Answer: We found two language features and four API designs that allow programmers to chain more method invocations. Some readers might think it is obvious that those features and design are useful for method chaining. However, what is important is that our quantitative analyses revealed how effective they are if adopted. At least 10% of not-chained invocations can be transformed into a chain. This quantitative result will be a basis for the design decision when adding those features or adopting the library design.
Proposal for Language Designers: Language designers can save the development effort of their users by implementing the safe call syntax and method cascading syntax. The users can write compact code by chaining method invocations with such syntax. In Java, which is a language that does not support the syntax, approximately 10% of chains suffer from `null`-checking and writing receiver objects repeatedly.
Best Practices in API Design: The following summarizes best practices that we propose for Java library developers:

- Return `Optional<T>` when a method may return `null`.
- When creating a method returning a boolean value such as `isFoo()`, create the method `ifFoo(...)` that takes lambda expressions.
- When creating a setter method such as `setFoo(...)`, return `this` in that method instead of returning nothing.
- When creating a public non-final class, provide the method for downcasting.

Tool developers can save these efforts of library developers by creating code generators for the boilerplate code.

5 THREATS TO VALIDITY

5.1 Internal Validity

The validity of ratios shown in Section 3.3, 3.4, and 4 highly depends on our manual inspection of method chains. To discuss the validity openly, we made our results of the inspection publicly available. The details on the data are provided in Appendix A.

5.2 External Validity

We did not apply any filter (e.g. filter by project domains) to the collected repositories. This supports the generalizability of our results. However, the trends in other languages would be different especially when a language provides special constructs to build a domain-specific language (DSL). Since method chaining is often regarded as a technique to design a DSL embedded in a host language, method chaining may not be used if the language provides such special constructs (e.g. [1, 5]). Our results are more likely to be applied to a language that does not provide such a construct (e.g. PHP and JavaScript). The empirical study of this hypothesis is future work.

6 RELATED WORK

Heavy-tailed distributions are found in a number of source code measures [16, 22, 25, 28, 29, 32, 40, 41]. For example, the study [22] shows that such distributions are found in the lexical properties of source code such as the number of lines and changed lines. The

study [29, 40] shows that they are found in the structural properties such as the number of subclasses and the in-degree and out-degree in dependency networks. The paper [28] says

> if one were to analyze the distribution of another measure of software, it would be most surprising to find it not following a power law or other heavy-tailed distribution.

However, none of those empirical results can describe the power-law distribution we observed in the number of method chains.

While a number of power-law distributions have been reported, the generation model of those distributions is less studied. Turnu et al. proposed a modified Yule process to model the evolution of object-oriented system properties [40]. Lin and Whitehead proposed a model based on preferential attachment and self-organized criticality [28]. The model proposal for the number of method chains is future work that is needed to deeply understand method chaining.

The use of language features is often empirically studied. In Java, the use of generics [17, 36], lambda expressions [31], annotations [17], and cast operators [30] has been studied. In the study on lambda expressions [31], Mazinanian et al. investigated not only the use in the real-world code but also the reason by interviewing the authors of the source code. Such a study would be beneficial to better understand the advantage of method chaining. In the literature [39], Tanaka et al. analyze the use of method chaining from the view point of functional idioms in Java.

The study [24] empirically shows that the violation of the law of Demeter has a negative impact on software quality. As pointed out in the Stackoverflow thread [8], chaining method invocations violates the law in most cases. Considering these facts, our results imply that real-world software increasingly becomes error-prone. However, method chaining is said to cooperate well with method completion systems in IDEs and let programmers write code easily and quickly [34, 42]. Further research needs to be carried out that inspects problems and merits in method chaining for the future development of language features and static analyzers addressing those problems.

High development cost is a well-known drawback of a fluent interface [14]. The cost becomes significantly higher when the developers choose to implement typed chaining [19] since a number of class definitions are required to achieve typed chaining. The tools and techniques proposed in the paper [13, 15, 20, 21, 26, 34, 42] help library developers to create a fluent interface instantly from grammar definitions.

There are more features that help programmers to chain method invocations although only a small number of languages provide those features. D [4] and Nim [37] have the uniform function call syntax, which allows chaining the invocation not only of methods but also functions. The scope functions in Kotlin [10] can be used to compose a chain with conditional statements and loops without storing an intermediate state into a temporary variable. The class extension mechanism without inheritance help programmers to use a fluent library to mitigate the extensibility problem pointed out in the blog posts [14, 38], Swift [3] and Kotlin [2] offer such a mechanism. Assume that we provide the library for counting numbers implemented as follows:

```
class Counter {
  n = 0;
  Counter increment() { n++; return this; }
}
```

The user of this library can count up a number by chaining the method call `increment()`. Suppose that the user need to add the method `decrement()` to this library. Programmers would use inheritance, a feature implemented in most object-oriented languages, to extend existing code:

```
class ExtCounter extends Counter {
  ExtCounter decrement() { n--; return this; }
}
```

However, this approach does not work as expected since the method `increment()` returns an instance of `Counter`:

```
new ExtCounter().increment() // Returns Counter
                .decrement(); // Method not found!
```

The user can avoid this problem by overwriting the existing methods in `Counter`, but it is too tedious to overwrite all the existing methods when a given library provides tens of methods. With the class extension, the user can add `decrement()` as expected.

7 CONCLUSION

This paper presented our empirical study of method chaining in Java. Our analysis quantitatively revealed the widespread and increasing trend of method chaining. To provide information for future language/library development, we estimated how effective it would be to introduce language features and API design for method chaining.

Although our results support the acceptance of method chaining in the real world, user studies need to be carried out to assert the acceptance. It is also interesting and beneficial for a better understanding of method chaining to investigate why real-world programmers prefer (or do not prefer) method chaining. The issues mentioned in the StackOverflow thread [8] would be helpful to start such an investigation. All of those studies are our primary future work.

REFERENCES

[1] [n.d.]. Domain-Specific Languages. http://docs.groovy-lang.org/docs/latest/html/documentation/core-domain-specific-languages.html. Accessed on 08/23/2019.
[2] [n.d.]. Extensions - Kotlin Programming Language. https://kotlinlang.org/docs/reference/extensions.html. Accessed on 08/23/2019.
[3] [n.d.]. Extensions - The Swift Programming Language (Swift 5.1). https://docs.swift.org/swift-book/LanguageGuide/Extensions.html. Accessed on 08/23/2019.
[4] [n.d.]. Functions - D Programming Language. https://dlang.org/spec/function.html. Accessed on 08/23/2019.
[5] [n.d.]. Higher-Order Functions and Lambdas - Kotlin Programming Language. https://kotlinlang.org/docs/reference/lambdas.html#passing-a-lambda-to-the-last-parameter. Accessed on 08/23/2019.
[6] [n.d.]. Method Cascades in Dart. http://news.dartlang.org/2012/02/method-cascades-in-dart-posted-by-gilad.html. Accessed on 08/23/2019.
[7] [n.d.]. Null Safety - Kotlin Programming Language. https://kotlinlang.org/docs/reference/null-safety.html. Accessed on 08/23/2019.
[8] [n.d.]. OOP - Method chaining - why is it a good practice, or not? - Stack Overflow. https://stackoverflow.com/questions/1103985/method-chaining-why-is-it-a-good-practice-or-not. Accessed on 08/23/2019.
[9] [n.d.]. Optional Chaining - The Swift Programming Language (Swift 5.1). https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html. Accessed on 08/23/2019.
[10] [n.d.]. Scope Functions - Kotlin Programming Language. https://kotlinlang.org/docs/reference/scope-functions.html. Accessed on 08/23/2019.

[11] [n.d.]. TypeScript 3.7 · TypeScript. https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-7.html. Accessed on 12/30/2019.
[12] Kent Beck. 1997. *Smalltalk Best Practice Patterns.*
[13] Eric Bodden. 2014. TS4J: A Fluent Interface for Defining and Computing Typestate Analyses. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis.*
[14] Yegor Bugayenko. [n.d.]. Fluent Interfaces Are Bad for Maintainability. https://www.yegor256.com/2018/03/13/fluent-interfaces.html. Accessed on 08/23/2019.
[15] Arvid Butting, Manuela Dalibor, Gerrit Leonhardt, Bernhard Rumpe, and Andreas Wortmann. 2018. Deriving Fluent Internal Domain-specific Languages from Grammars. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering.*
[16] S. Cook, R. Harrison, and P. Wernick. 2005. A simulation model of self-organising evolvability in software systems. In *IEEE International Workshop on Software Evolvability (Software-Evolvability'05).* 17–22.
[17] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features. In *Proceedings of the 36th International Conference on Software Engineering.* 779–790.
[18] Martin Fowler. [n.d.]. FluentInterface. https://www.martinfowler.com/bliki/FluentInterface.html. Accessed on 08/23/2019.
[19] Steve Freeman and Nat Pryce. 2006. Evolving an Embedded Domain-specific Language in Java. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications.*
[20] Yossi Gil and Tomer Levy. 2016. Formal Language Recognition with the Java Type Checker. In *Proceedings of 30th European Conference on Object-Oriented Programming.*
[21] Yossi Gil and Ori Roth. 2019. Fling — A Fluent API Generator. In *Proceedings of 30th European Conference on Object-Oriented Programming.*
[22] AA Gorshenev and Yu M Pis'mak. 2005. Punctuated Equilibrium in Software Evolution. *Physical review. E, Statistical, nonlinear, and soft matter physics* 70 (2005), 067103.
[23] Radu Grigore. 2017. Java Generics Are Turing Complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages.*
[24] Y. Guo, M. Wuersch, E. Giger, and H. C. Gall. 2011. An Empirical Validation of the Benefits of Adhering to the Law of Demeter. In *2011 18th Working Conference on Reverse Engineering.*
[25] L. Hatton. 2009. Power-Law Distributions of Component Size in General Software Systems. *IEEE Transactions on Software Engineering* 35, 4 (2009), 566–572.
[26] Tomer Levy. 2017. *A Fluent API for Automatic Generation of Fluent APIs in Java.* Ph.D. Dissertation. Israel Institute of Technology.
[27] K. Lieberherr, I. Holland, and A. Riel. 1988. Object-oriented Programming: An Objective Sense of Style. *SIGPLAN Not.* 23 (1988), 323–334.
[28] Z. Lin and J. Whitehead. 2015. Why Power Laws? An Explanation from Fine-Grained Code Changes. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories.*
[29] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. 2008. Power Laws in Software. *ACM Trans. Softw. Eng. Methodol.* 18, 1 (2008), 2:1–2:26.
[30] Luis Mastrangelo, Matthias Hauswirth, and Nathaniel Nystrom. 2019. Casting about in the Dark: An Empirical Study of Cast Operations in Java Programs. *Proc. ACM Program. Lang.* (2019).
[31] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2017. Understanding the Use of Lambda Expressions in Java. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 85:1–85:31.
[32] Chris Myers. 2003. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical review. E, Statistical, nonlinear, and soft matter physics* 68 (2003), 046116.
[33] Tomoki Nakamaru and Shigeru Chiba. 2020. Generating a Generic Fluent API in Java. *The Art, Science, and Engineering of Programming* (2020).
[34] Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. 2017. Silverchain: a fluent API generator. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences.*
[35] Mark EJ Newman. 2005. Power laws, Pareto distributions and Zipf's law. *Contemporary physics* 46, 5 (2005), 323–351.
[36] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. 2011. Java Generics Adoption: How New Features are Introduced, Championed, or Ignored. In *Proceedings of the International Working Conference on Mining Software Repositories* (proceedings of the international working conference on mining software repositories ed.).
[37] Dominik Picheta. 2017. *Nim in Action.*
[38] Marco Pivetta. [n.d.]. Fluent Interfaces are Evil. https://ocramius.github.io/blog/fluent-interfaces-are-evil/. Accessed on 08/23/2019.
[39] Hiroto TANAKA, Shinsuke MATSUMOTO, and Shinji KUSUMOTO. 2019. A Study on the Current Status of Functional Idioms in Java. *IEICE Transactions on Information and Systems* (2019).
[40] I. Turnu, G. Concas, M. Marchesi, S. Pinna, and R. Tonelli. 2011. A modified Yule process to model the evolution of some object-oriented system properties. *Information Sciences* 181, 4 (2011), 883 – 902.
[41] J. Wu, R. C. Holt, and A. E. Hassan. 2007. Empirical Evidence for SOC Dynamics in Software Evolution. In *2007 IEEE International Conference on Software Maintenance.*
[42] Hao Xu. 2010. EriLex: An Embedded Domain Specific Language Generator. In *Objects, Models, Components, Patterns.*
[43] Tetsuro Yamazaki, Tomoki Nakamaru, Kazuhiro Ichikawa, and Shigeru Chiba. 2019. Generating a fluent API with syntax checking from an LR grammar. *Proc. ACM Program. Lang.* (2019).

A DATA

The collected method chains and the results of our manual inspections are publicly available as an archive file[14]. The archive file contains the following materials:

`data.txt`: List of all the collected chains.
`metadata.txt`: List of files and the number of lines for each file.
`rq1_{short,long,extlong}_201{0,8}.md`: Sampled chains for RQ1.
`rq1_201{0,8}.csv`: Result of our manual inspections in RQ1.
`rq2.md`: Sampled chains for RQ2.
`rq2.csv`: Result of our manual inspections in RQ2.

[14] https://doi.org/10.5281/zenodo.3697939