Porting the NetBeans Java 8 Enhanced for Loop Lambda Expression Refactoring to Eclipse

Md Arefin Raffi Khatchadourian
City University of New York
md.arefin@mail.citytech.cuny.edu rkhatchadourian@citytech.cuny.edu

Abstract

Java 8 is one of the largest upgrades to the popular language and framework in over a decade. However, the Eclipse IDE is missing several key refactorings that could help developers take advantage of new features in Java 8 more easily. In this paper, we discuss our ongoing work in porting the enhanced for loop to lambda expression refactoring from the NetBeans IDE to Eclipse. We also discuss future plans for new Java 8 refactorings not found in any current IDE.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement

Keywords Java 8, refactoring, lambda expressions

1. Introduction

Java 8 is one of the most significant upgrades to Java programming language and framework in over a decade. It provides supports for functional programming, a new JavaScript engine, new APIs for date time manipulation, a new stream API, and more [4]. Such features can help make programs easier to read, write, and maintain [2].

Among the new Java 8 features, lambda expressions are touted to be most significant. Lambda expression simplify the development process by facilitating functional programming. They also provide a concise way to write anonymous inner classes and make it easier to iterate through, filter, and exact data from a `Collection` [3].

Eclipse (`http://eclipse.org`) is one of the most popular IDEs for Java. While Eclipse has incorporated several Java 8 feature quick-fixes and refactorings, there are still many features left to be done. For example, the NetBeans IDE (`http://netbeans.org`) has a refactoring (originally proposed by Gyori et al. [2]) that converts enhanced **for** loops to a lambda expression. This paper discusses our ongoing work in exploring the porting of such conversion mechanisms from NetBeans to the Eclipse IDE.

Making such changes manually would require changing approx. 1,700 line of non-commented, non-blank lines of code across approx. 100 files per project, on average [2]. With our plug-in, Eclipse developers would not need to make these changes manually. Our tool will help Eclipse developers adopt the functional-like programming model offered by Java 8. Furthermore, we are in the process of exploring other Java 8 refactorings that do not currently exist in any IDE.

2. Examples

In this section, we will present several examples of how the final version of our plug-in will work on pre-Java 8 code. Assume we have a `List`, and suppose we wished to print each element of the list. In pre-Java 8, we can use `for (String s : list) System.out.println(s)` to do so.

One possible refactoring to use lambda expressions would be `list.forEach(s → System.out.println(s))`. Here, we call the new `forEach()` method on the list, passing the lambda expression `s → System.out.println(s)`, meaning that each `String` in the list (each of which gets bound to `s`) is passed to the `println()` method.

Another possible lambda expression refactoring would be `list.forEach(System.out::println)`. In this case, we again call the `forEach()` on the list but this time, instead of passing a lambda expression, we pass a method reference, a new concept in Java 8 as well. Here, we refer to the `println()` method of the `PrintStream` class. When the lambda expression is processed, the `println()` method is called on the object referred to by the `out` variable.

Yet, another possible refactoring is one where the new stream API, which allows for parallel processing, would be `list.stream().forEach(System.out::println)`. This code sequentially processes the list, while `list.parallelStream().forEach(System.out::println)` processes it in parallel.

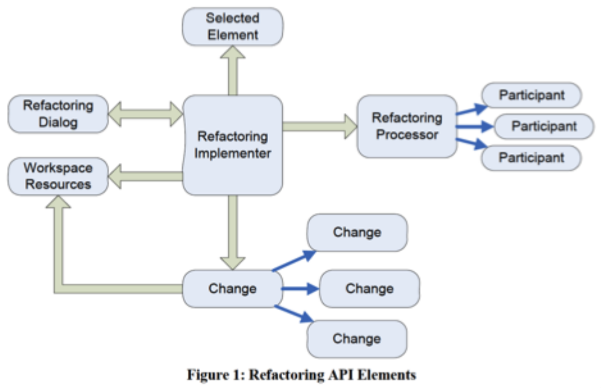

Figure 1. Refactoring API Elements

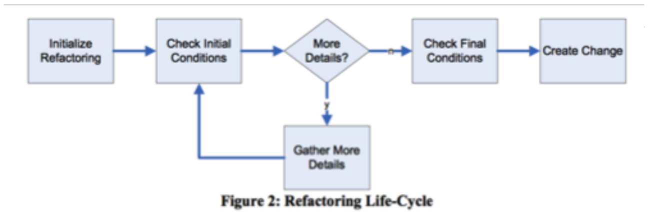

Figure 2. Refactoring Life-Cycle

3. Eclipse Refactoring Model

This project involves developing a refactoring plug-in for Eclipse, which work in the following way:

- The API for refactoring provides a process-level abstraction upon which specific refactorings may be built.
- Figure 1 shows elements of this abstraction at a high level. Arrows between elements represent dependencies.
- Once a refactoring has been initiated, an implementer of that refactoring is used to coordinate condition checking, gathering details about the refactoring, and ultimately produce a series of changes that may be applied to the Eclipse workspace to accomplish the refactoring.
- This implementer can extend the abstract class org . eclipse . ltk . core . refactoring . Refactoring. The life-cycle for this class is shown in Figure 2.

4. Implication and Preconditions

Although many enhanced **for** loops can be converted to lambda expressions, there are some precondition to check. For example, lambda expression bodies cannot reference variables that are not **final** or *effectively* final (i.e., the variable *could have* been marked as final) [1]. Below are some preconditions for the lambda conversion [2].

1. The conversion must be semantics-preserving. That is, the behavior of the program prior to the refactoring must match that of after the refactoring.
2. The **for** loop must iterate over an instance of a Collection as this is where stream() is declared.
3. The body of the initial **for** loop must not throw a checked exception.
4. The body of the initial **for** loop must not have more than one reference to a local, non-effectively final variable defined outside the loop.
5. The loop body must not contain a **break**, **continue** statement as these semantics cannot be expressed via a lambda expression.

From our experience, we believe that these preconditions are sufficient for the kinds of loops being refactored. It would also be possible to refactor loops over Iterables as it declares forEach(). Moreover, it would be helpful to prove the soundness. We designate these tasks for future work.

5. Current State of Progress

As we are making progress toward the implementation, one of the technical challenges we face is the difference between the Eclipse and NetBeans refactoring and source code representation models. The IDEs use different libraries for these purposes, e.g., NetBeans uses primarily javac libraries, while Eclipse uses the Java Development Tools (JDT). For instance, NetBeans has API for retrieving uncaught exceptions in an AST subtree, while the JDT does not expose such external API. To check the corresponding precondition, we implemented this functionality ourselves. The plug-in is open source and publicly available at `http://git.io/lambdarefact`.

6. Conclusion and Future Work

Java is a highly used programming language in industry, and Java 8 is one of the most significant upgrades to the language. Eclipse is a popular IDE for Java. The purpose of our project is to allow Eclipse developers to automatically refactor their legacy Java software to use new Java 8 features. We are currently in the process of porting an enhanced **for** loop refactoring from NetBeans to Eclipse. In future, we plan to develop refactorings for Java 8 that do not currently exist in any IDE.

References

[1] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification.* Pearson Education, 2014.

[2] A. Gyori, L. Franklin, D. Dig, and J. Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *Foundations of Software Engineering*, 2013.

[3] Oracle Corporation. Java SE8: Lambda quick start, 2015. URL `http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html`.

[4] Oracle Corporation. What's new in jdk 8, 2015. URL `http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html`.