Lambda Expression in Java

Duc-Anh-Vu Nguyen
College of Engineering and Computer Science, Arkansas State University at Jonesboro, Arkansas, USA
vu.nguyen1@smail.astate.edu

Cory Huffine
College of Engineering and Computer Science, Arkansas State University at Jonesboro, Arkansas, USA
steven.huffine1@smail.astate.edu

ABSTRACT

Lambda expression was first introduced in Java 8. This is an interesting feature that contributes to the change of programming trending in Java. Lambda expressions are used to condense functions, improve productivity and performance. Object-oriented languages have adopted this functional hallmark, a lot of applications of Lambda expression could be seen in many projects. However, there are still debates over how programmers use it. Especially, the possibility of implementing Lambda Expression on the more complex algorithms is still a big concern.

In order to clarify that concern as well as ascertaining the real benefits of Lambda expression in Java programming, we analyzed the usage rate over time of this new language feature in Java open-source projects by utilizing some programming language static analysis tools and MySQL. Our tools detected and counted the total number of Lambda expressions implemented in the projects. With the paths produced out by the tools, which lead to the lines of code that contain Lambda expressions, we checked the actual code to confirm that Lambda expressions were detected accurately and how they were used by programmers. Furthermore, we also wanted to determine when open-source projects in Java began implementing this feature.

Our results indicate that widespread adoption of Lambda expressions occurred in 2017-2018 and that the benefits of lambda expressions outweigh some of its criticism. Lambda expression is a great language feature that was added but to utilize it effectively in the programming, it still depends on the context. Lambda expression might be inappropriate to be implemented in the complex algorithms and still has limitations that need to be improved while being used widely to represent regular method interfaces to shorten the source code.

KEYWORDS

Lambda Expression, Java 8, benefits, complex algorithms

1. INTRODUCTION

Over the past decade, many object-oriented programing languages have added a language feature know as Lambda Expression that quickly define local, unnamed functions. While this feature was originally only seen in functional programming languages, it has been adopted by object-oriented languages in the modern era.

This study will determine the number of lambda expressions being used in various open-source Java projects, determine when this feature started being used, as well as answering two fundamental research questions posed at the start of the research.

To accomplish this, we used various language analysis tools to analyze four projects, then compared their revision history against the occurrences of lambda expressions to create a line graph of total lambda expressions over time.

2. RELATED WORK

Katayama's paper, "Systematic Search for Lambda Expressions", performed similar research, but with the purpose of easing functional programming and identifying the best possible performance of lambda expressions.[3] Their primary focuses were on making improvements to their searching algorithm, while our paper will be focusing on the impact lambda expressions have had on recent projects, specifically with non-functional programming. Our implementation used to identify lambda functions was also completely different from this study.

Mazinanian et al.'s paper, "Understanding the Use of Lambda Expressions in Java", is almost identical to our study, but with a much larger sample size of 241 open-source projects. In addition to this, they also surveyed 97 developers about how they introduced lambdas into their projects.[2] This study implemented a process incredibly similar to our own, with only minor differences in the specific programs used, and is recommended as further reading.

3. BACKGROUND AND OVERVIEW

In their 2010 paper adapting lambda expressions into C++, Jarvi and Freeman concluded that "Lambda functions are very useful, even necessary, for effective use of modern C++ libraries," explaining further that many popular algorithms contained in the STL library are hindered by "The lack of a syntactically lightweight mechanism for defining simple local functions".[1]

However, in their 2017 paper, Mazinanian et al. found that Java programmers are still having a difficult time using lambdas to their full potential and are implementing them in inefficient ways.[2]

[1] Jaakko Jarvi and John Freeman

[2] Mazinanian et al.

[3] Susumu Katayama

To understand the application of lambda expressions in object-oriented languages, especially in Java, we formulated two fundamental research questions:

- **RQ1**: Why do language designers implement lambda expressions into their languages?
- **RQ2**: Is it possible to implement Lambda expressions on complex methods/functions?

The four open-source projects we chose to analyze were:

- PMD, a cross-language static code analyzer
- Hanabi-Java, a card game in java used for research
- Apache Flink, a framework and processing engine for computations over data streams
- Guava, a collection of core Google libraries

Our programming tools provided by Dr. Kim – our Software Engineer class professor - is a research toolchain for analysis of Java language feature usage in open source projects:

- Java: Java works as a local server that receives requests from the client to analyze the projects. There are many different built-in methods available in Java libraries that help us save time in finding ways to detect the Lambda expression pattern.

- Python: Working as a client and a connector between the database and Java local sever. Python client is responsible for sending various requests over a socket to the Java local sever after expanding the open-source projects. When the open-source projects are done with analyzing, Python client plays another role of loading all the data into the database for parsing to produce data output files.

- MySql: The created database in MySql contains multiple tables for each language feature that will be analyzed. Each table stores the relevant data of the open-source projects before and after the analyzing process. In our project, we created a *lambda_expressions* table in the *generic* database.

- Octave: This is a software featuring high-level programming language that provided us the ability to produce graphs from our output data files.

4. IMPLEMENTATION

Finding Lambda Expression Usage over Time

To get the number of Lambda expressions used in the open-source projects, the local server must be able to recognize the pattern. The "LambdaExpressionVisitor" class (Figure 1) was invoked to scan each line of code. When a lambda expression was detected, the "visit" method recorded the line of code's number for later verification.

```
package patterns;

import java.io.Writer;

public class LambdaExpressionVisitor extends AbstractVisitor {

    public LambdaExpressionVisitor (Writer writer) {
        super(writer);
        System.out.println("(In LambdaVisitor) Debug line #: " +
                new Exception().getStackTrace()[0].getLineNumber());
    }


    @Override
    public boolean visit(LambdaExpression lambda)
    {
        System.out.println("~ Lambda Visited, Debug Line #: " +
                new Exception().getStackTrace()[0].getLineNumber());

        int lineNumber = Unit.getLineNumber(lambda.getStartPosition()); // get the position of lambda expression

        String parent = GetEnclosingContainerName(lambda);
        if( parent != null ) // print the output to the file
        {
            WriteLine(lineNumber + ":" + parent);
            System.out.println(lineNumber + ":" + parent);
        }
        return true;
    }
}
```

Figure 1: LambdaExpressionVisitor class in Java Server

On the client end, the file run.py made requests to the server to analyze the open-source project currently stored in the database. The "LambdaPattern" class (Figure 2) in run.py was responsible for exporting data into *.data* files. A *.sql* file is also generated, which is used to load all the data from the *.data* files into the database.

```
class LambdaPattern(PatternInfo):
    def __init__(self):
        self.patternName = "lambda"
        self.table = "lambda_expressions"
        self.fields = "container container_granularity count".split()
        self.changedFields = "added container container_granularity type".split()
        self.doChanged = True

    def outputData(self, project, module, revision, tokens, containerGranularity, output):
        # tokens for this pattern are of the form:
        # line:container:type
        for filename in tokens.keys():
            for (container, type), count in self.getCountSet(tokens[filename]).items():
                print >> output, tabSeparatedData(project, module, filename, revision, container, containerGranularity, count)

    def outputDiffData(self, project, module, revision, diffFiles, containerGranularity, output):
        for kind, filename, info in diffFiles:
            if kind == "M" or kind == "A":
                for container, addedOrDeleted, type in info:
                    if addedOrDeleted[0] == "-":
                        added = "0"
                    elif addedOrDeleted[0] == "+":
                        added = "1"
                    else:
                        raise Exception("don't know how to handle addedOrDeleted = '" + addedOrDeleted + "'")
                    print >> output, tabSeparatedData(project, module, filename, revision, added, container, containerGranularity,
                        type)


    def getCountSet(self, lines):
        set = {}
        for container, type in [line.strip().split(":") for line in lines]:
            set[container, type] = set.get((container,type), 0) + 1
        return set

    def getSetForDiff(self, lines):
        set = {}
        for container, count in [line.strip().split(":") for line in lines]:
            set[container] = set.get(container, {})
            set[container][count] = set[container].get(count, 0) + 1
        return set
```

Figure 2: LambdaPattern class in run.py

Finally, another program in the client, "lambdas_over_time.py" (Figure 3), parsed the data into a *.tsv* output file by combining the data loaded into the database by run.py with a table named "revisions" that contains the history of changes of the open-source project.

This *.tsv* file contains the dates that lambda expressions were added or removed from the open-source project and the total number of lambda expressions at that time. We used Octave to produce line graphs from the output files, showing the trend of lambda expressions over time.

```
def main(project, conn):
    #foreach =>rawtype, item_types=>fileid
    sql = """ select datetime, r.filename, count, state, container_granularity
        from revisions r, lambda_expressions t where r.project = t.project and r.project = '%(project)s'
        and r.transactionid = t.revision""" % locals()
    print sql

    cursor = conn.cursor()
    cursor.execute(sql)
    timeHash = {}
    for time, filename, state, count, container_granularity in cursor:
        #if container_granularity == "full":
            #continue
        if not timeHash.has_key(time):
            timeHash[time] = []
        timeHash[time].append( (filename, state, count,  "lambda") )

    print sql

    times = timeHash.keys()
    times.sort()

    print times

    fd = open(project + "_lambdas_2019_1121.tsv", "w")
    print >> fd, "datetime\tCount"
    files = {}
    for time in times:
        print time
        for file in [x[0] for x in timeHash[time]]:
            files[file] = []
        for filename, state, count, kind  in timeHash[time]:
            if state == 'deleted':
                #print >> fd, "{deleted}state="+state
                pass
            else:
                #print >> fd, "{except}state="+state
                files[filename].append( (count, kind) )

        #Raw -> foreach
        #para -> forloop
        totalLambdas = 0

        print files.items();


        for file, types in files.items():
            for count, kind in types:
                totalLambdas += 1;
```

Figure 3: lambdas_over_time.py

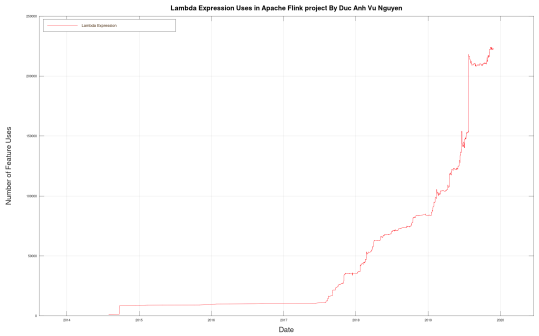

Figure 4: Lambda Expression Usage in Apache Flink

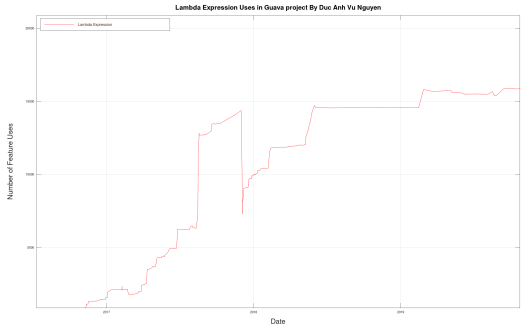

Figure 5: Lambda Expression Usage in Guava

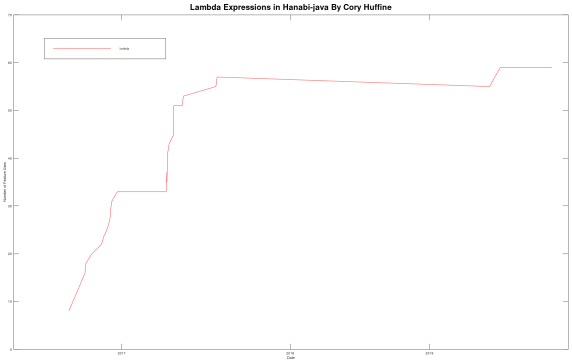

Figure 6: Lambda Expression Usage in Hanabi-java

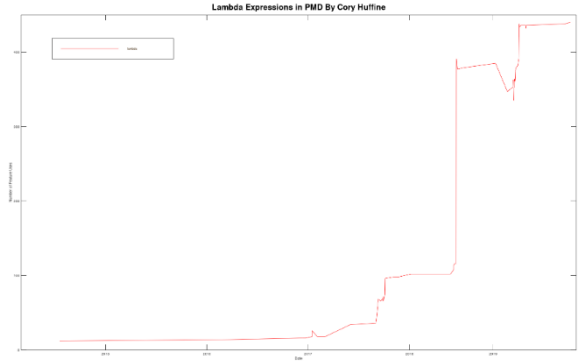

Figure 7: Lambda Expression Usage in PMD

5. EVALUATION

5.1 Table

We analyzed from small projects such as Hanabi-java to a big project like Apache Flink with more than a million lines of code. Below is the table of basic information about the open-source projects that we used in this project:

Name	Start	End	LOC	Lambda
Guava	September, 2009	November , 2019	517,054	15888
Apache Flink	April, 2013	November , 2019	1,307,457	222,482
PMD	June, 2002	December , 2019	259,566	440
Hanabi-java	May, 2016	November , 2019	26,360	59

5.2 Graph Inference

The graphs show that lambda expressions were clearly adopted in 2017 by all four open-source projects, with a sharp spike in their usage in 2018. A notable outlier from this trend occurs in Hanabi-Java, which remained stagnant after an initial jump in 2017. This is understandable since it is by far the smallest project in our sample and is updated less often.

5.3 Responses to Research Questions

RQ1: Why do language designers implement lambda expressions into their languages?

The first and most obvious, reason to implement lambda functions is to reduce total lines of code. With enough use of this feature, programming language static analysis tools, such as the ones we used in this paper, will have an easier time analyzing large projects.

Below is a program is written in the way before Java 8 was released:

```
class ExampleLambda {
  public static void main(String[] args) {
    List<String> languages = Arrays.asList("Java", "C#", "C++");
    Collections.sort(languages, new Comparator<String>() {
      @Override
      public int compare (String s1, String s2) {
        return s1.compareTo(s2);
      }
    });

    for (String language : languages)
    {
      System.out.println(language);
    }
  }
}
```

Figure 8.1: Example of Lambda expression can reduce LOC

With the Lambda expression in Java 8, the above program could be rewritten:

```
class ExampleLambda {
  public static void main(String[] args) {
    List<String> languages = Arrays.asList("Java", "C#", "C++");

    Collection.sort(languages, (String s1, String s2)) -> {
      return s1.compareTo(s2);
    });
    for (String language : languages)
    {
      System.out.println(language);
    }
  }
}
```

Figure 8.2: Example of Lambda expression can reduce LOC

Another benefit of Lambda expression that makes it a right addition to Java is supporting parallel and sequential processing by passing behavior in methods. In the previous version of Java, to process the elements of any collection, we need to run an iterator to go over all elements and then process each of them. With the support of Stream API in Java 8, functions can be passed to collection methods and elements then are processed in a sequential or parallel manner by the collection. Lambda expression also brings to us the higher efficiency programming concept by utilizing multicore CPUs. Implementing lambda expression and using Stream API can achieve efficiency in case of bulk operations on collections. It is also easier to achieve internal iteration of collections rather than external iteration. With parallel processing using lambda, we can take advantage of the multicore CPU.

[4] Oracle

According to Oracle's documentation, "Lambda expressions also improve the [Java] Collection libraries making it easier to iterate through, filter, and extract data from a Collection. In addition, new concurrency features improve performance in multicore environments."[4]

```
// Using for loop for iterating a list
public static void printUsingForLoop(){
   List<String> stringList = new ArrayList<String>();
   stringList.add("Hello");
   for (String string : stringList) {
        System.out.println("Content of List is: " + string);
   }

}

// Using forEach and passing lambda expression for iteration
public static void printUsingForEach() {
   List<String> stringList = new ArrayList<String>();
   stringList.add("Hello");
   stringList.stream().forEach((string) -> {
     System.out.println("Content of List is: " + string);
   });
}
```

Figure 9: Use of Stream API in Java 8 with Lambda expression

RQ2: Is it possible to implement Lambda expressions on complex methods/functions?

Lambda expressions reduce the readability of code, and a programmer who is not experienced with them will have a difficult time understanding a function that could have been more eloquently described with a function header and descriptive parameters. Even those who feel experienced will have a hard time determining the purpose of lambda expressions they themselves did not write. Even though lambdas simplify the Java syntax, the abstraction they bring is not suitable for complex methods and functions.

```
Stream<Pair> allFactorials = Stream.iterate(
  new Pair(BigInteger.ONE, BigInteger.ONE),
  x -> new Pair(
    x.num.add(BigInteger.ONE),
    x.value.multiply(x.num.add(BigInteger.ONE))));
return allFactorials.filter(
  (x) -> x.num.equals(num)).findAny().get().value;
```

Figure 10.1: Lambda expression reduces the readability

Let's take a look at the example above (Figure 10). The code is such a pain to read and it will take a while to determine what the program is written for. For the same purpose but the code is rewritten by classic Java code with the traditional loop is much more readable since everyone knows the loop, but not everyone knows the Stream API.

```
BigInteger cur = BigInteger.ONE, acc = BigInteger.ONE;
while(cur.compareTo(num) <= 0) {
  cur = cur.add(BigInteger.ONE); // Unfortunately, BigInteger is immutable!
  acc = acc.multiply(cur);
}
return acc;
```

Figure 10.2: Lambda expression reduces the readability

6. CONCLUSION

C++ 11 and Java 8's inclusion of lambda expressions, a powerful feature of functional languages, were important events in the history of programming, marking a time when programming languages were beginning to adopt various styles of programming together. Furthermore, their widespread adoption in 2017-2018 shows the community's willingness to adapt functional language features into their programs in an effort to improve performance.

Despite this, it is clear that most object-oriented programmers are still not comfortable in their use. The supposed increase in productivity can be stalled by the reduction in readability, especially in large, complex functions. Even with the drawbacks, lambda expressions have great potential that will flourish as programmers become more familiar with them over time.

7. REFERENCES

1. Jaakko Jarvi and John Freeman. 2010. C++ lambda expressions and closures. *Science of Computer Programming*. 75, 9 (September 2010), 762-772. DOI https://www.sciencedirect.com/science/article/pii/S0167642309000720

2. Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2017. Understanding the Use of Lambda Expressions in Java. *Proceedings of the ACM on Programming Languages (PACMPL)*. 1, Article 85 (October 2017), 31 pages. DOI: https://users.encs.concordia.ca/~nikolaos/publications/OOPSLA_2017.pdf

3. Susumu Katayama. 2005. Systematic search for lambda expressions. *In Proceedings Sixth Symposium on Trends in Functional Programming (TFP 2005)*. 195-205. DOI: http://www.cs.ioc.ee/tfp-icfp-gpce05/tfp-proc/14num.pdf

4. Oracle. 2013. Java SE 8: Lambda Quick Start. (December 2013). http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html