Abstract — A major feature introduced to developers in Java 8 is language-level support for lambda expressions. Oracle claims that the use of lambda expressions will allow for the writing of more concise and efficient code for processing elements that are stored in collections. We consider the problem of determining if the runtime of lambda expressions in Java 8 are faster than non-lambda expressions. By comparing the speed of a lambda expression with a corresponding non-lambda expression, that renders the same result, we describe where lambda expressions have performance advantages or disadvantages.

Keywords: lambda expression, stream, Java 8

1. Introduction

Lambda expressions in computer programming are byproducts of the mathematical logic system, lambda calculus. It was Alonzo Church who came up with lambda calculus [16] to give structure to the concept of effective computability. Lambda expressions are also known anonymous functions because in lambda calculus all functions are anonymous (not bound to an identifier). Lambda expressions have now been used in computer programming since their introduction in Lisp in 1958. Forty-six years later lambda expressions are getting introduced to Java programmers.

1.1 Problem

The question we are trying to answer is the following: are lambda expressions in Java 8 faster than non-lambda expressions at accomplishing the same tasks.

1.2 Motivation

Our motivation for this work came from Oracle’s claim that the use of lambda expressions would result in more efficient code [1]. We were interested in determining if there were advantages to using lambda expressions beyond their obvious conciseness. Considering Oracle claimed lambda expressions were more efficient, we decided to validate by getting a quantitative speed difference in milliseconds and as a percent.

1.3 Goals

Our main goal is to determine if the newly introduced lambda expressions have a speed advantage over non-lambda expressions for an identical task. We also wanted to make a website that ran our comparisons. The website is also intended to allow users to educate themselves on the performance and uses of lambda expressions.

1.4 Objectives

To meet our goals, we have the following objectives:

• Find example lambda expressions to use for comparisons. These will be found through Java 8 books and Oracle documentation.
• In addition to finding lambda expressions, create our own lambda expressions. The created lambda expressions are to show additional uses for lambda expressions that are not covered in the Java 8 books or in Oracle’s documentation.
• Using Eclipse, calculate how long a lambda expression takes to finish its task. Then test the speed of a non-lambda expression completing the same task. These results are used to determine the difference, between the lambda and non-lambda expression, in milliseconds and as a percentage.
• Complete a website that runs and outputs the comparison results. This would allow speed comparisons to be tested across multiple operating systems and processor speeds.
• Make the website user friendly and provide thorough instructions on how to allow the website to run. The purpose of this is to allow users of the site to run their own performance comparisons and see the speed difference for themselves.
• Display the code used for comparisons on the website. This is meant to allow users to learn some ways that lambda expressions can be used. Additionally, giving users the ability to see the code used will allow further discussion about if there is a better, faster way to code the lambda or non-lambda expressions.

1.5 Outline

In section 2 we give a brief background on lambda expressions. In section 3 we give some examples of lambda expressions and compare them against their equivalent non-lambda expression. In section 4, report on performance comparison of the examples discussed in section 3. Finally, in section 5, we give our conclusions.

2. Background

The use of lambda expressions has long been ubiquitous in various functional programming languages such as Lisp,
Scala, Haskell, and F# [2], among others. Before Java 8, Java programmers have been forced into writing more verbose code than needed and lacked key functionality such as the ability to pass in a function as a parameter.

The following subsections give an overview of lambda expressions. This discussion includes what a lambda expression is, why they are useful, and finally their performance in other languages.

2.1 What is a Lambda Expression?

Lambda expressions are also known as anonymous functions because they are functions without an identifier. These expressions can make use of already programmed functional interfaces, such as a Predicate or Function. With no identifier, a lambda expression isn’t intended to be called many times like a method. They are actually commonly used to avoid coding unnecessary methods. Thus, if the functionality is only needed once or for a short amount of time, lambda expressions help make code clearer and concise [3].

Example:
/*to get the total + tax of a list of prices */
ArrayList<Integer> prices = new
ArrayList<Integer>(Arrays.asList(90,87,34,21));
// using lambda expression
double total = prices.stream()
.mapToDouble(x -> x*1.14)
.sum();
//using non-lambda expression
double total2 = 0.0;
for (Integer i : prices) {
    total2 += i*1.14;
}

For example, the above lambda expression can be written using one line of code. The non-lambda expression involves first initializing a variable and then creating a for-loop to add each item price, with tax, to the total.

2.2 Why Use Lambda Expressions?

As stated in [3], the use of functional interfaces paired with anonymous inner classes is a common theme in Java. To simplify the coding, functional interfaces are taken advantage of for use with lambda expressions, eliminating the need to program inner classes.

Example:
// Using inner class
btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
});

The “horizontal solution” of using lambda expressions, solves the “vertical problem” presented by using inner classes [3]. A lambda expression addresses the bulkiness of an inner class by converting 5, 6, or even more lines of code into a single statement.

2.3 Performance in Other Programming Languages

Performance of lambda expressions differs from one programming language to another. In some languages, the use of a lambda expression not only gives the code a more clear and concise look, but also has a faster execution time. However, in other programming languages, code runs faster using the more verbose approach [6]. As stated in [6], the “lazy” approach can have its costs when it comes to efficiency; using lambda expressions when needed is slower than calling a function by name. The following subsections review the use of lambda expressions in other popular programming languages.

2.3.1 In Haskell

Haskell is a purely functional programming language, based on lambda calculus. In the release of Haskell version 1.0, in 1990, it was well known that the use of lambda expressions caused a significant and constant performance loss [6]. Haskell 1.0 was also inefficient when it came to defining streams [6], making it more inefficient to use lambda expressions on streams of data.

Haskell Prime was released in 2006, where much of the development focus was on performance [6]. Now competitive performance is available with Haskell [6]. Haskell programmers can now use the functionality of lambda expressions without the inferiority of performance.

2.3.2 In Python

According to Python’s official documentation, lambda expressions are equivalent to regular function objects [8]. In Python, lambda expressions are just a better syntactic way to write a normal function [9]. Thus, the implication is that lambda expressions have equivalent performance as non-lambda expressions in Python. However, as stated in [10], lambda expressions in Python can be more efficient to use for common programming idioms such as mapping, filtering, and list comprehension.

2.3.3 In C++

C++11 was released in 2011 and saw major revisions including the use of lambda expressions [11]. According to the ISO (International Organization for Standardization), the addition of lambda expressions to C++ has added much strength flexibility and efficiency [11]. C++ programmers are
now enabled to use powerful expressiveness, and write efficient, high performance code [11].

3. Lambda Comparisons

Lambda expressions allow for a much more concise way of iterating over a collection of data such as a list. Lambda expressions can use multiple functions and interfaces to accomplish a task. The following subsections go over some of the varying ways that lambda expressions can be used.

3.1 Reduction

The stream.reduce() method is a general reduction operation. It is comprised of an identity element that is the starting value of the reduction and default value if there are no elements in the stream. The method also consists of an accumulator that takes as parameters, the partial result so far of the reduction, and the next element in the stream. A new partial result gets returned.

Example: int total = nums.stream()
   .reduce(0, (a, b) -> a+b);

The non-lambda way of accomplishing the same thing as in the above example would be creating a variable to store the sum, and then running a for-each loop where each number in the list was added to the total.

Example: int total = 0;
   for (int i : nums) {
      total += i;
   }

What would have taken 3 lines of code in previous Java versions, can now be done in 1 line using lambda expressions.

3.2 Filtering

The stream.filter() method takes a predicate as an argument and returns a new stream containing the elements that matched the conditions of predicate. Each predicate can have multiple conditions that need to be satisfied. A lambda expression can be passed into the stream.filter() method instead.

Example: List<String> filtered = strList.stream()
   .filter(x -> x.length()> 3)
   .collect(Collectors.toList());

The above example filters out all strings with a length less than 3. This creates a new stream with only the remaining strings. The non-lambda expression once again uses a for-each loop. Instead of filtering, the non-lambda expression uses an if-statement.

Example: List<String> filtered = new
   ArrayList<String>();

3.3 Collecting

The stream.Collectors class has a variety of methods that are of great use to streams and lambda expressions. These methods from the Collectors class can be used inside the stream.collect method of the lambda expression.

3.3.1 To List

One method of the Collectors class is the Collector.toList() method. The method takes all the elements that are left in a stream, and stores them in a list. This makes it quick and easy to create a new list, filtering out unwanted elements from the old list. For example, extracting all the numbers in a list that are greater than 5.

Example: List<Integer> above5 = numberList
   .stream()
   .filter(x -> x > 5)
   .collect(Collectors.toList());

In the above example, with 1 line of code a new list is created, containing only the desired numbers. A non-Lambda expression to accomplish the same feat requires creating a new list, using an if-statement to check the value of each number, and then adding the wanted numbers to the List.

Example: List<Integer> above5 = new
   ArrayList<Integer>();
   for (Integer i : numberList) {
      if (i > 5)
         above5.add(i);
   }

3.3.2 Joining

Another method in the Collectors class is the joining() method. The method is a terminal operation that creates a non-stream result. Inside the stream.collect method, the joining() method returns a Collector that concatenates all the elements in the stream. The joining() method can take a CharSequence as a parameter. In that case a Collector is returned that concatenates the stream elements with the CharSequence separating each element.

Example: String con = names.stream()
   .collect(
      Collectors.joining("", "));

In the above example, in 1 line of code, the joining() method creates a concatenated string with a comma separating
each element. Using a non-lambda expressions involves first creating a blank string and then using a for-loop to add each list element to the string.

Example: String con = "";
    for (int i = 0; i < names.size(); i++) {
        con += names.get(i) + ", " ;
    }

3.4 Mapping

Mapping involves taking an object and assigning it to a new value. If for example there was a stream filled with Person objects, mapping could create a stream filled with numbers, such as the Person object’s age. Mapping can be accomplished with the stream.map() method. The method can take a lambda expression as a parameter.

Example: List<String> upperNames = names.stream()
           .map(name -> name.toUpperCase())
           .collect(Collectors.toList());

The above example streams a list of strings and maps each string to a string with all uppercase letters. The equivalent non-lambda expression creates a new list, and then iterates through a for-loop of the original list. Each string from the original list is converted to all uppercase letters before being added to the new list.

Example: ArrayList<String> upperNames = new ArrayList<String>();
          for (String name : names) {
             upperNames.add(name.toUpperCase());
         }

3.5 Passing In Functions and Predicates

As previously mentioned, a feature lacking in previous Java versions was the ability to pass in functions. The java.util.function package can be used to pass in a Function or Predicate into a stream’s intermediate operation(s) to replace a lambda expression. Both a Function and Predicate can be used as a target for a lambda expression or method reference. The syntax for defining a function is Function<T, R> where T is the type of argument being passed in and R is the type of result for the function. The Function<T, R> takes in a single argument and returns some result. Unlike the predicate the result isn’t necessarily a Boolean.

Example: Function<String, Predicate<String>>
          startsWithLetter = letter -> name ->
          name.startsWith(letter);

In the above example, a predicate is returned by the function. What makes the function different from the predicate example in section 3.4.1 is the ability to check if the string started with any letter. Whereas the predicate in section 3.4.1 was hard coded to only check if the string started with the letter “A”. An equivalent non-lambda expression involves creating a separate method inside the class file.

Example:

```java
public boolean startsWith(String a, String b) {
    return b.startsWith(a);
}
```

3.6 Calling Class Methods

Lambda expressions can be used to call methods written elsewhere in the class or superclass. The method could return
a boolean for filtering, be used for mapping, or be part of the terminal operation. For the comparison on the website we used the following method:

```java
static boolean isPrime(int n) {
    for(int i=2;i<n;i++) {
        if(n%i==0)
            return false;
    }
    return true;
}
```

Since a Boolean is returned, the isPrime() method is used in the body of a lambda expression. The lambda expression is inside the stream.filter() method to filter the stream of numbers. The stream.count() method is then used to add up how many elements are left in the stream.

Example: `int counter = (int) nums.stream().filter(p -> isPrime(p)).count();`

Yet again the non-lambda expression involves the use of a for-loop. An integer is created that keeps track of the total number of prime numbers. The for-loop iterates through each number in a list. The isPrime() method is used inside an if-statement. If the number is prime, 1 is added to the total.

Example: `for (int n : nums) {
    if (isPrime(n)) {
        counter++;
    }
}
```

### 4. Results

In this section we provide comparisons of the execution times of the examples noted in the previous section.

#### 4.1 Comparison Results

The data presented in the Tables 1 and 2 are the result of executing each lambda expression and non-lambda expression for problems of size 10,000, repeating each experiment 1000 times and then averaging the results using a Mac Pro laptop, 2.9 GHz Intel Core i7, 8 GB 1600 MHz, DDR3 memory, with Java 8. The average execution time in milliseconds and the ratio of improvement between the lambda and non-lambda expressions are noted in Table 1 and 2. The Lambda improvement is calculated as follows: (Lambda - Non-Lambda) / (Non-Lambda) * -100.00. To remove any startup or Just In Time (JIT) effects [17], the results report in Table 1 where from the fifth iteration of running the above experiments. We found that by the third iteration the results were consistent with iterations four and five. Table 2 shows the results of the first iteration, which are considerably different from the results reported in Table 1. Table 3 shows how drastically a small problem size and only one iteration can affect the results. The results in this table show how using a problem size of 1000, only running each experiment 100 times, and then looking at the first iteration of this impacts the performance of Lambdas. To run your own comparisons, visit http://people.scs.carleton.ca/~deugo/java8

**Table 1: Lambda Performance Comparisons (5’th Iteration)**

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Lambda (ms)</th>
<th>Non-Lambda (ms)</th>
<th>Lambda Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counting Primes</td>
<td>16.81</td>
<td>16.42</td>
<td>-2.40</td>
</tr>
<tr>
<td>Adding Up Numbers</td>
<td>16.81</td>
<td>16.42</td>
<td>-2.40</td>
</tr>
<tr>
<td>Concatenating Strings</td>
<td>32.78</td>
<td>73.31</td>
<td>55.29</td>
</tr>
<tr>
<td>Mapping</td>
<td>70.58</td>
<td>105.80</td>
<td>33.29</td>
</tr>
<tr>
<td>Filter List</td>
<td>72.91</td>
<td>106.18</td>
<td>31.23</td>
</tr>
<tr>
<td>Filter List with Predicate</td>
<td>79.47</td>
<td>107.25</td>
<td>25.90</td>
</tr>
<tr>
<td>Filter In List Function</td>
<td>87.13</td>
<td>108.32</td>
<td>19.57</td>
</tr>
</tbody>
</table>

**Table 2: Lambda Performance Comparisons (1’st Iteration)**

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Lambda (ms)</th>
<th>Non-Lambda (ms)</th>
<th>Lambda Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counting Primes</td>
<td>15.96</td>
<td>15.32</td>
<td>2.25</td>
</tr>
<tr>
<td>Adding Up Numbers</td>
<td>15.96</td>
<td>16.33</td>
<td>2.25</td>
</tr>
<tr>
<td>Concatenating Strings</td>
<td>30.44</td>
<td>72.82</td>
<td>58.20</td>
</tr>
<tr>
<td>Mapping</td>
<td>66.9</td>
<td>105.19</td>
<td>36.40</td>
</tr>
<tr>
<td>Filter List</td>
<td>69.21</td>
<td>106.18</td>
<td>31.23</td>
</tr>
<tr>
<td>Filter List with Predicate</td>
<td>74.71</td>
<td>107.25</td>
<td>29.93</td>
</tr>
<tr>
<td>Filter In List Function</td>
<td>81.15</td>
<td>107.71</td>
<td>24.66</td>
</tr>
</tbody>
</table>

**Table 3: Lambda Performance Comparisons (small problem size and repetitions)**

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Lambda (ms)</th>
<th>Non-Lambda (ms)</th>
<th>Lambda Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counting Primes</td>
<td>0.72</td>
<td>0.01</td>
<td>-7100</td>
</tr>
<tr>
<td>Adding Up Numbers</td>
<td>0.74</td>
<td>0.01</td>
<td>-7300</td>
</tr>
<tr>
<td>Concatenating Strings</td>
<td>2.51</td>
<td>9.34</td>
<td>73.13</td>
</tr>
<tr>
<td>Mapping</td>
<td>6.28</td>
<td>12.24</td>
<td>48.70</td>
</tr>
<tr>
<td>Filter List</td>
<td>6.47</td>
<td>12.37</td>
<td>47.70</td>
</tr>
<tr>
<td>Filter List with Predicate</td>
<td>6.51</td>
<td>12.5</td>
<td>47.92</td>
</tr>
<tr>
<td>Filter In List Function</td>
<td>6.69</td>
<td>12.63</td>
<td>47.03</td>
</tr>
</tbody>
</table>
5. Conclusions

The addition of lambda expressions to Java 8 provide for more functional, concise, and readable coding. In addition, given enough execution time, the new lambda expressions can provide a performance advantage. The report entitled ‘Clash of the Lambdas’ also shows that Java’s lambda expressions not only held their own, but in many cases outperformed the lambda expressions in Scala, C#, and F# [15]. These results held true for Windows and Linux, and varying processor speeds. This is impressive on many levels considering lambda expressions have been present in Scala and F# since their introductions, and in C# since C# 3.0. With this being Java’s first attempt at lambda expressions, the results are impressive.

5.1 The Future

With Java 8 being the first version of Java with support for lambda expressions, the future looks promising. Looking at the performance of lambdas in other programming languages, Java’s performance not only competes, but also leads over other languages. More impressively, some of those languages have supported lambda expressions for years. Java 9 was announced for release in 2016. This provides another opportunity for Oracle to continue to increase the performance advantages of lambda.

6. References


