Java

Programming Research

Chayton Morris
CS-410

cmorris8@una.edu

Introduction

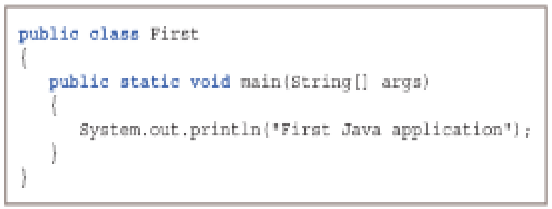
This paper demonstrates the capabilities and structure of Java. Java is a widely used object-oriented programming language that can be compiled one time and used on any platform that supports Java without the need of being compiled again. This write once, run anywhere tactic is implemented with the help of the Java Virtual Machine. We will be placing a focus on the Java Virtual Machine(JVM), and how it compares to compilation in natively compiled languages such as C++. With the rise of new technology and advancements in the tech world, it will be interesting to see how Java evolves and what kind of changes we will see within the JVM specifically. In this paper we will investigate the usage of the JVM and how it affects performance and memory usage on Java applications. We will hypothesize that the JVM will have a significant positive effect on the performance and memory usage of Java applications. We will make the assumption that the JVM will be using large sums of memory to execute these programs, but the performance will be able to compete with that of C++. We will use this investigation to compare our findings to natively compiled languages to demonstrate the relative effect that the JVM has. These results will aid in a better understanding of the JVM and the Java programming language as a whole.

1 Background

Formerly known as Oak, Java is a highly popular object-oriented programming language. Java has a similar syntax to that of C++, but with fewer low-level facilities. Java was released in 1996 by James Gosling at Sun Microsystems[4]. Gosling was said to release Java with a similar syntax to C++ in order for programmers to adjust to a language that they would find familiar. This production of Java introduced the Write Once, Run Anywhere concept, also known as WORA. Soon after its initial release, Java became very popular within web pages with the use of Java applets. Soon after, new releases were made to comply with mobile apps and typical server environments. Sun Microsystems attempted to formalize Java in 1997, but this process did not go through. The company released most of the Java implementations for free, and most of their revenue came from selling specialized products. The Java Virtual Machine had its first official release by Sun Microsystems in late 2006. Nearly all of this software was made open-source besides a few sections due to copyright issues. Java was then obtained by Oracle around 2009. In 2010, Android was using Java to aid in the making of Android applications. This became a legal dispute. The jury stated that Google could only be seen responsible for copyright infringement if API's could be copyrighted. The lawsuit went back and forth for years, and eventually in 2016, Google was found to be favored in the lawsuit, and use of the API was constituted as fair use. Other motives for the creation of this language include: portability, high-level, and object-oriented. Java was later acquired by the Oracle Corporation. Java immediately became popular after its release due to the fact that it supported object-oriented programming, was easy to use, and portable. Today, Java is used in a wide range of niches including web applications and mobile apps.

1.1 Capabilities

Java is an interpreted language. This means that the language is compiled into bytecode, and then executed by the means of an interpreter. These Java programs must still be interpreted at every execution. Most Java applications are executed using a Java Runtime Environment(JRE), but some web browser applications use Java applets. Although the JVM is the most common way to execute a Java program, it is possible to use a micro controller that will run the bytecode within the hardware of a system. While this process makes the language more portable, it also makes the performance suffer. The figure below demonstrates a simple Java program[5].



```
public class First
{
   public static void main(String[] args)
   {
      System.out.println("First Java application");
   }
}
```

Figure 1: Sample Java program[5].

This sample program proves to be familiar. The "First" class is the main class and is a necessary part of any Java program. We can also see the standard output protocol when writing code in Java. "System.out.println" is similar to that out "cout" in C++.

One of the biggest strengths of Java is its portability on top of supporting object-oriented programming. This being said, developers are able to create reusable code that can be modified or edited over time. Java also supports multithreading, and this proves to be a necessary feature. Multithreading means that developers can complete several activities at once. This feature enhances the performance and response time of applications. Multithreading is especially helpful for applications that must manage large volumes of data. Multithreading is essential for scalability and building high performing applications. Java comes with built-in classes that can make building certain applications simple. This language is also a great option for building web apps as its GUI provides the user with many different necessary elements such as buttons and menus. Java includes a few security features as well, such as sandboxing in order to protect from unauthorized access. Although sandboxing is a relatively known concept among the Web, Java applications do not run within the context of a web browser. This means that the invocation of a Java application is typically not directly over the internet. This is where the JVM comes into play, as it is a separate entity in which Java applications are executed over. Java comes with a large set of classes that can be implemented into applications. One of these classes is the "SecurityManager" class. This class determines the accessible characteristics of a unit that is being invoked over a Java application.

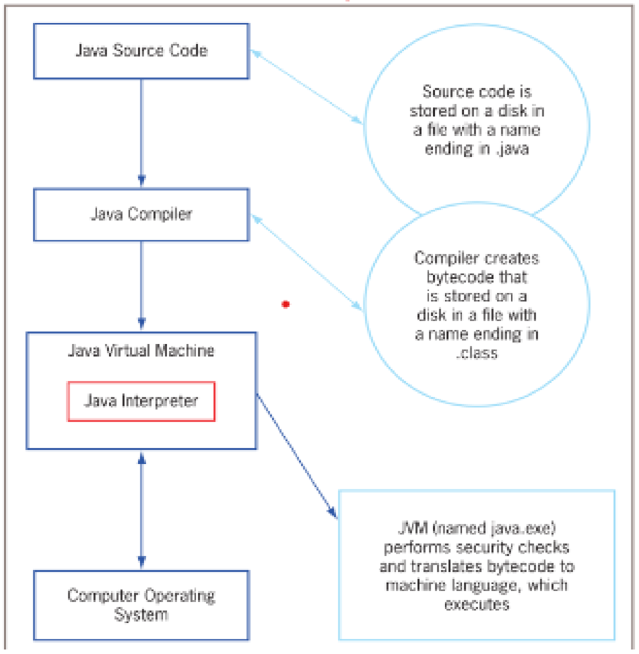

Java Source Code
Source code is stored on a disk in a file with a name ending in .java
Java Compiler
Compiler creates bytecode that is stored on a disk in a file with a name ending in .class
Java Virtual Machine
Java Interpreter
JVM (named java.exe) performs security checks and translates bytecode to machine language, which executes
Computer Operating System

Figure 2: The Java environment and how programs are interpreted[5].

Another crucial aspect of Java as a programming language is its garbage collector. Garbage collection has proven to be an essential part of many languages, and this is no different for Java. The garbage collector is a feature of the JVM, and it manages the allocation and deallocation processes within the stored memory. The main purpose of the garbage collector is to free up unused memory. This memory usually comes from objects that are not being used or are not being referenced by an application. Within the JVM are hundreds of garbage collection algorithms. The most common of these algorithms is called the mark-and-sweep garbage collection. This algorithm uses a heap where each object in the program has a tag that identifies the class of object that it is in. The garbage collector initially creates an empty heap. Once the heap is created, the garbage collector finds unmarked objects and copies them to the heap. This algorithm is recursive and continues until only live objects are in the heap, and the heap with unmarked objects are discarded. The only drawback to not only the mark-and-sweep algorithm, but many of the garbage collection algorithms, is the memory that is used to complete these processes. With larger programs, obviously the heap is going to be larger. The garbage collector will take a substantial amount of time to run, and this uses a large amount of memory to complete. Garbage collection is often confused with the prevention of memory leaks, but this is not the case. This garbage collection typically happens during the idle stage of a program. This garbage collector prevents the use of pointer arithmetic.

1.2 Syntax

As said before, the syntax of Java is similar to that of C++ and C#. Every bit of code that is written within a Java program must be written within a class. The only exception to this is the primitive data types which will be mentioned shortly. Java also supports inheritance, but the procedure of operator overloading is not a concept that is supported in Java. Java also includes type checking for assignment statements, expression operands, and method parameters. Every Java program must begin with a class. In regard to Figure 1, the "public class First" is the class that is defined for the specific program. Also, the name of the save file is the name of the public class and the extension is ".java." This is similar to the main function within a C++ program. However, as we can see, the class names in Java must always begin with a capital letter. Java can also contain multiple classes, but only one public class. A .java file can store any class that is not declared public. When declaring a public class in Java, the keyword represents a method that has the ability of being called from other classes. Java also includes "private" and "protected" access modifiers. Classes stated to be public can be accessed with any class. Classes that are protected can be accessed by subclasses and classes in the same package. Private classes can only be accessed within their own class. "Static" is also another keyword. This keyword is not associated with a specific instance of the class, but the class itself. An array of strings will be accepted in every main method of Java. Java also contains a number of libraries. A few of these libraries are as follows: IO, Networking, Reflection, Concurrency, Scripting, XML processing, Functional programming, and Security. The terminating character in Java is the semicolon(;).Just like any other language, Java has specific rules and regulations that must be followed when building a program. Some of the features of these rules/syntax are as follows:

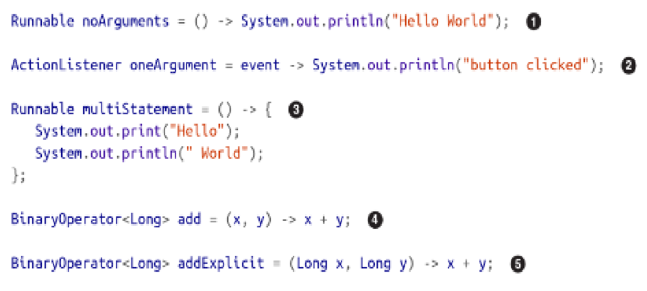
- **Comments:** In Java, single line comments are structured "//." Comments that span multiple lines start with an opening tag /* and end with a closing tag */. Java also offers a "Javadoc" style of comments. These comments begin with "/**" and end with "*/." These comments allow developers to access the documentation with certain IDE's that are able to perform this action.

- **Variables:** Java variables must be explicitly declared. For example, to declare a floating-point variable, the syntax is as such: “float x;.”
- **Operators:** The operators in Java are similar to those of other languages and are as follows: (+, -, *, /, %, ==, !=, <, >, <=, >=, &&, ||, !).

In regard to the operators above, “+” is the addition operator as “-“ is the subtraction operator. The “*” operator is meant for multiplication, and the “/” operator handles division. There are also many comparison operators. The “==” operator is the “is equal to” operator, and its counterpart “!=” serves as the “not equal to” operator. Also, “<” is the less than operator as “>” is the greater than operator. We can also see the greater than or equal to operator, as well as the less than or equal to operator. The “&&” serves as the “and” operator. Finally, the “||” is the “or” operator and “!” is the “not” operator. Another useful tool in Java is the ability to use shortcuts with arithmetic operators. For example, “x = x +1” is the same as saying “x += 1” These shortcuts can be used with subtraction, multiplication, and division/remainder as well. On top of these shortcuts, Java also offers prefix and postfix increment operators. These operators are used to increment and decrement values by one. For example, “x += 1” can be written like “++x” or “x++”. However, a prefix operator the value is returned after the calculation, likewise, values are returned before calculation with postfix operators. Java also supports type casting, the act of converting a value from one specific type to another. These conversions can either be widening conversion or narrowing conversions(implicit v. explicit). In addition to the various other data types explained, Java also includes literals. These literals give a value to a specific type of variable. Java also allows for floating point variables to be set using scientific notation by using the symbol “e” at the end of declaration. On top of numbers, there also exists character literals. These values must be within single quotes and can contain any character.

1.3 Expressions

Expressions in Java are similar to those of other languages. Arithmetic expressions are written as “float sum = 5.5 + 2.0;.” Assignments should also seem familiar. For example, “int y = 30;” assigns the value 30 to the variable y. Java 8.0 also introduced a new concept to the Java programming language, lambda expressions. Lambda expressions allow developers to create functions without names. This change came into light because of the need to write code that could be easily modified to run on multicore CPU’s[9]. A few lambda expression examples are shown below.



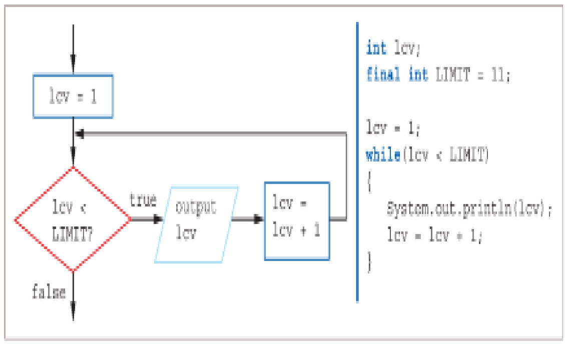
```
Runnable noArguments = () -> System.out.println("Hello World");  ❶

ActionListener oneArgument = event -> System.out.println("button clicked");  ❷

Runnable multiStatement = () -> {  ❸
   System.out.print("Hello");
   System.out.println(" World");
};

BinaryOperator<Long> add = (x, y) -> x + y;  ❹

BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;  ❺
```

Figure 3: Lamba expressions common in Java[9].

The first example allows a lambda expression with no arguments. The second example is meant to neglect the parenthesis in the arguments. The third example represents a lambda expression being used within a block of code. The fourth example represents a lambda expression taking multiple arguments. This expression will make a function that can add the two numbers together. Finally, the last example showcases how we are able to explicitly define the type of a variable within a lambda expression.

1.4 Control Structures

Control structures in Java are also similar to those in other languages. These structures include if-else statements, for, while, and do while loops, and switch statements.



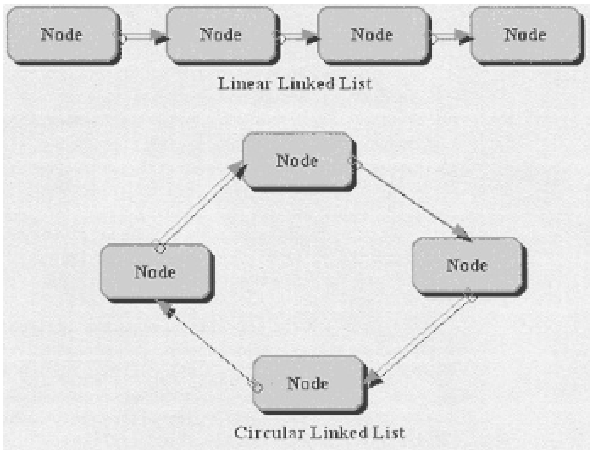
```
int lcv;
final int LIMIT = 11;

lcv = 1;
while(lcv < LIMIT)
{
   System.out.println(lcv);
   lcv = lcv + 1;
}
```

Figure 4: An example of a while loop and how they are executed [5].

The structure of these loops are similar to those we may have seen before in languages such as C and C++. The condition of a while loop is enclosed in parenthesis and the contents of the loop are enclosed within brackets.

1.5 Abstract Data Types

Java also contains many different abstract data types. These ADT's include stacks, queues, trees, heaps, arrays, vectors,



hash tables, linked lists, etc. Another instance of these data types is circular and doubly-linked lists. The doubly-linked list has much more capabilities than a normal singly linked list. In doubly-linked lists, it is possible to traverse the lists in both directions. This is in contrast to a singly linked list that must start at the head and move towards the end. Also, in both singly linked and doubly-linked lists, end nodes are references to be null. However, this is not the case in circular linked lists. In this form of linked lists, each node will reference each other, hence the term circular. Figure 5 shows a visual representation of the difference between a linear linked list and a circular linked list.

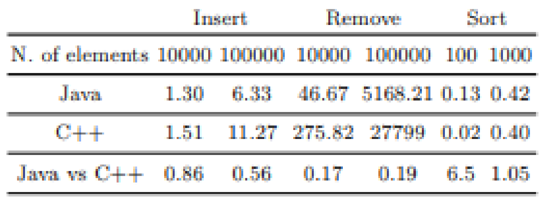
Figure 5: Linear Linked List vs. Circular Linked List [2]

Java also provides eight different primitive data types. These data types must be initialized, or the compiler assign a default value to each. The eight data types are listed below.

- **Byte –** 8-bit signed integer
- **Short –** 16-bit signed integer
- **Int –** 32-bit signed integer
- **Long –** 64-bit signed integer
- **Float -** 32-bit floating point number
- **Double –** 64-bit floating point number
- **Boolean –** 1 bit, used with flag indicator
- **Char –** 16-bit character

1.6 Recent Literature

Before explaining the methods that will be taken to solve the question relating to the performance of the JVM as compared to other languages, we must first understand what the JVM is and how it works. Whenever a Java program is run, the instructions are not executed directly by your systems hardware. In other words, programs are not run by the CPU, but they are instead run by this "virtual" processer, simply a piece of software that is operating on the computer. The JVM uses a set of about 20 instructions to manipulate objects in Java[3]. Figure 2 above also gives a great example of how this process takes place. The JVM was originally much slower than other languages when it came to its execution. In 1998 the Just-In-Time compiler was released. JIT is integrated within the JVM. The JIT compiler is responsible for translating byte code into binary code. Over the years, many tests have been done to determine the performance of the JVM in comparison to natively compiled languages. In [6], a study was done to compare the amount of time it took Java to perform insertion, removal, and sorting in comparison to C++. The sorting algorithm used in the Java program is a version of merge sort. The sorting algorithm used for C++ is called the introsort algorithm. The Java algorithm is O(nlog(n)), while the worst case for the C++ algorithm is O(nlog(n)). The results of this study are shown in Figure 6.



	Insert		Remove		Sort	
N. of elements	10000	100000	10000	100000	100	1000
Java	1.30	6.33	46.67	5168.21	0.13	0.42
C++	1.51	11.27	275.82	27799	0.02	0.40
Java vs C++	0.86	0.56	0.17	0.19	6.5	1.05

Table 2. Times report - Collection comparison

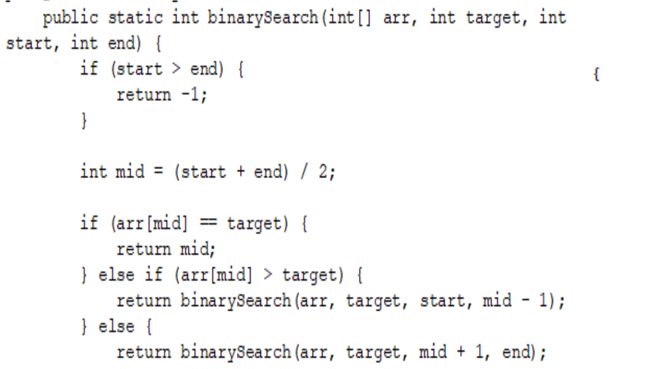
Figure 6: Time in milliseconds to execute the following algorithms[6].

These tests show that Java is slightly faster than C++ when it comes to inserting. Java is significantly faster than C++ when it comes to removing. Finally, it is shown that C++ is slightly faster than Java when it comes to sorting. At the time of this study, Java was in version an earlier version. Now, many updates have been made to the JVM and Java itself. These updates have aided in making execution much faster while still holding its convenience as an interpreted language. Also, these tests were executed on a 3.2 GHz Intel Pentium 4 processor with 1GB of RAM under Ubuntu 10.4. These programs were also executed with OpenJDK Runtime Environment v 1.6.0_20 and GCC v 4.3.3. This will obviously have an effect on how quickly the programs/algorithms will execute and this must be taken into account once we move into our own tests and conclusions.

After the creation of the JIT compiler, JVM offered the HotSpot Compiler. This compiler runs each program while first using an interpreter. The compiler will detect the most critical methods and analyze them. This allows the compiler to optimize at execution and compile only the methods which it deems critical.

2 Methodology

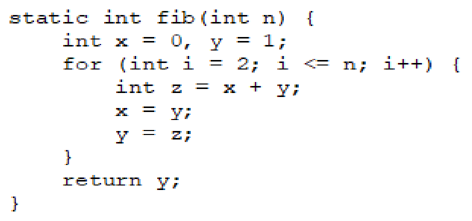
To test our hypothesis, we will be running various tests with different algorithms to determine the difference in performance and memory usage between Java and C++. For clarity, all programs will be executed on a 3.59 GHz AMD Ryzen 5 3600 6-Core Processor with 16GB of RAM under Microsoft Windows Version 10.0.22621.1555. The version of Java that will be used is OpenJDK 17.0.4.10. For C++, all programs will be executed on GCC v 8.1. Also, the benchmarks to determine execution time in each of these programs are made simple. First, we will be testing the execution time and memory usage of implementing an insertion sort algorithm while using two different test sizes. Both programs will be written using an array with O(n) being the best-case complexity. First, we will fill the array with 1000 elements, and test the execution time and memory usage of both languages. Next, we will fill the array with 10000 elements and repeat the process. Both cases(1000 and 10000 elements) will be executed twenty times per language. For both execution time and memory usage, we will take the average. We will calculate time in nanoseconds and find the average in converted milliseconds. These averages will be compared to find the difference in Java and C++. The same



```
    public static int binarySearch(int[] arr, int target, int
start, int end) {
        if (start > end) {
            return -1;
        }

        int mid = (start + end) / 2;

        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] > target) {
            return binarySearch(arr, target, start, mid - 1);
        } else {
            return binarySearch(arr, target, mid + 1, end);
```

procedure will take place when testing the memory usage. We must keep in mind that these programs will be executed on a home network with multiple hosts running at one time. Also, it is obvious that each iteration of the programs will come with a new set of random numbers, and the chance of both the Java and C++ program having the same exact numbers is near impossible. The result of these tests will surely be impacted by all of these factors. Each code snippet below is simply in place to show the algorithms used, and to display a few simple Java programs. The implementations in C++ also use similar code/functions. The function to compute insertion sort is shown below.

Next, we will be running a few tests using an algorithm to compute the Fibonacci sequence of various numbers. The Fibonacci sequence is a series of numbers in which each number in the sequence is a sum of the two numbers before it. Each language will execute the program twenty times to determine the results for our tests. Also, each number that will be computed will be hard coded into the programs for each iteration. Java and C++ will contain the same numbers to make the comparisons more accurate. All iterations will compute numbers that are greater than 100. The function to compute the algorithm is shown below.



```
static int fib(int n) {
    int x = 0, y = 1;
    for (int i = 2; i <= n; i++) {
        int z = x + y;
        x = y;
        y = z;
    }
    return y;
}
```

The final test that will be done is a binary search using arrays with 1000 elements. Each array will be filled with random numbers 1-2000. The target will be the first element in the array. The programs will be run 20 times each and the average for each language will be computed**.** The function to compute the algorithm is shown below.

3 Conclusion

The results of the first initial tests are shown below:

N of elements	1000	10000
Java	1.8ms	11.7ms
C++	1.9ms	20.3ms

Table 1: Insertion Sort

As shown in the tests above, we can see that the two languages preformed almost exactly to each other when the size of the array was fairly small. However, Java performed twice as fast as C++ when the size of the array grew to 10000 elements. These results are largely affected by the conditions in which each was executed and by the hardware they are running on.

Next, we will showcase the results of the Fibonacci algorithms. As explained above, each program was hard coded with the exact same number for each iteration to better determine results. The results of the second test are shown below:

Range of #'s	>100
Java	6.95ms
C++	1.15ms

Table 2: Fibonacci Sequence

As shown in the results above, C++ significantly outperformed Java in executing the Fibonacci algorithm. The results were intended to be split into two sections to showcase the execution time for numbers less than 100 and numbers greater than 100, but the results were too similar, and this representation is within .0001 accuracy(in milliseconds).

For the final test, we tested the performance of both the Java program and the C++ program in terms of computing a binary search algorithm. The results of this test are shown below:

Range of #'s	1-2000
Java	.003ms
C++	.001ms

Table 3: Binary Search

As the results show, C++ performed significantly better than the Java program. The C++ program averaged a speed 3 times faster than that of Java. It is also important to note that the arrays were filled with a new set of random integers for each iteration.

In conclusion, it is shown that there are many factors that will affect the performance and memory usage of Java applications using the JVM. Hardware requirements, data structures used, the conditions in which each application is executed, and the environmental space in which the programs were executed. These three tests were not specifically picked to determine which language is better suited for which algorithm, they were simply picked to accurately depict the memory usage and performance under the given conditions. When it comes to insertion sort, Java is able to keep up with C++ when the size of the array is fairly small, however this gap seems to widen once the array size reaches a larger number such as 10000 elements. In terms of the Fibonacci sequence, C++ performed slightly faster than Java when each program was executed using the same sample of numbers. Finally, the binary search algorithm proved to have the least effect on each language. C++ just slightly outperformed Java for each test. The results of memory usage were very similar in each program/algorithm. Initially, there were profiling methods in place to determine the amount of memory used with each language. However, it was determined that this may prove to be inaccurate in terms of the capability of each framework(valgrid for C++ and Jprofiler for Java). Using built in libraries for each language, it was concluded that Java used slightly more memory than C++ for every test case. This outcome is likely due to Java's garbage collection and large use of memory when determining live objects.

4 Future Work

After seeing the results of these tests, we can propose many questions about the JVM and Java as a programming language itself. Each algorithm in this test proved to have slightly different results in terms of execution speed and memory usage. This can propose a number of questions about optimizing Java programs to perform the quickest while consuming a small amount of memory. These results can obviously vary depending on a number of factors such as hardware and the specific applications on which these programs are running on. Also, considering there are different ways to implement JVM(this paper using OpenJDK), how would different implementations affect the performance and memory usage on Java applications? The subject of this paper focused mainly on the software side and how it affects the JVM. Another interesting topic would be to consider what kinds of impacts the specific hardware of a system would have on these applications. Another interesting topic of discussion considers comparing Java to other interpreted languages in terms of the questions that are proposed in this paper. As said before, C++ is a compiled language. Compiled languages are typically faster than interpreted languages. It could prove to be interesting to compare Java to other interpreted languages such as Perl, Python, BASIC, and PHP. We also touched on garbage collection earlier in this discussion. This can propose many questions about the best garbage collection algorithm. It was concluded that a large amount of these algorithms are "hungry[3]" for data, and that introduces the idea of creating a garbage collection procedure that is easy on the CPU and the amount of memory it uses to allocate and deallocate the units of a Java program. There are many aspects of Java that can propose many different questions for further research. Java was made to be a lightweight, portable language. This portability makes Java platform independent. The Java libraries are also very extensive and the packages that can be included allow developers to use a vast amount of built in properties and functions. All of these factors including Java's maintainability prove that Java is reliable enough to compete with languages like C++ and may even be a great replacement considering the specific goal and application.

REFERENCES

[1] S. Andersen, Data Structures in Java. Jones & Bartlett Learning, 2002.
[2] M. S. Jenkins, Abstract Data Types in Java. Computing McGraw-Hill, 1998.
[3] J. Meyer, T. Downing, and I. Netlibrary, Java virtual machine. Cambridge, Mass.: O'reilly, 1997.
[4] hmong.wiki, "Java (programming language)." Available: https://hmong.ru/wiki/Java_language
[5] J. Farrell, Java programming. Australia: Cengage Learning, 2019.
[6] L. Gherardi, D. Brugali, and D. Comotti, "A Java vs. C++ performance evaluation: a 3D modeling benchmark."
[7] J. Alves-Foss, Formal Syntax and Semantics of Java. Springer, 2003.
[8] B. Baesens, Aimée Backiel, and Seppe Vanden Broucke, Beginning Java programming : the object-oriented approach. Indianapolis: Wiley, Cop, 2015.
[9]R. Warburton, Java 8 Lambdas. "O'Reilly Media, Inc.," 2014.