Time Report

By Omer Irfan

Set-up:

All experiments were done on a MacBook Pro with an Intel Core i5 processor and 8GB of memory. Additionally, all other applications apart from the Java Virtual Machine were closed during the following experiments. Before running each program, I did 3 warmup runs to ensure the optimization of the JVM and accurate results. Each experiment is run 5 times and then a final average of those runs is also provided.

Exercise 6:

For this exercise I was tasked to concatenate and append a long and short string 'a' for 1 second. I was instructed to do this using a String and a StringBuilder. I calculated the time taken with milliseconds using the Java command 'System.currentTimeMillis()'. The concatenations of both long and short string were done inside a while loop, with the condition of the milliseconds reaching 1000 i.e. 1 second. Below are the results of all five runs and an average of those results, after my first three test runs.

Number of Concatenations	*Run 1*	*Run 2*	*Run 3*	*Run 4*	*Run 5*	***Average***
Short String	41920	50806	46490	38931	45938	**44817**
Short StringBuilder	103768	102962	108583	106220	109087	**106124**
Long String	5975	4923	5882	5747	5859	**5677.2**
Long StringBuilder	10488	10318	10822	10575	10607	**10562**

Table - 1: Number of Concatenations

Length of String	*Run 1*	*Run 2*	*Run 3*	*Run 4*	*Run 5*	***Average***
Short String	41920	50806	46490	38931	45938	**44817**
Short StringBuilder	103768	102962	108583	106220	109087	**106124**
Long String	478000	393840	470560	459760	468720	**454176**
Long StringBuilder	839040	825440	865760	846000	848560	**844960**

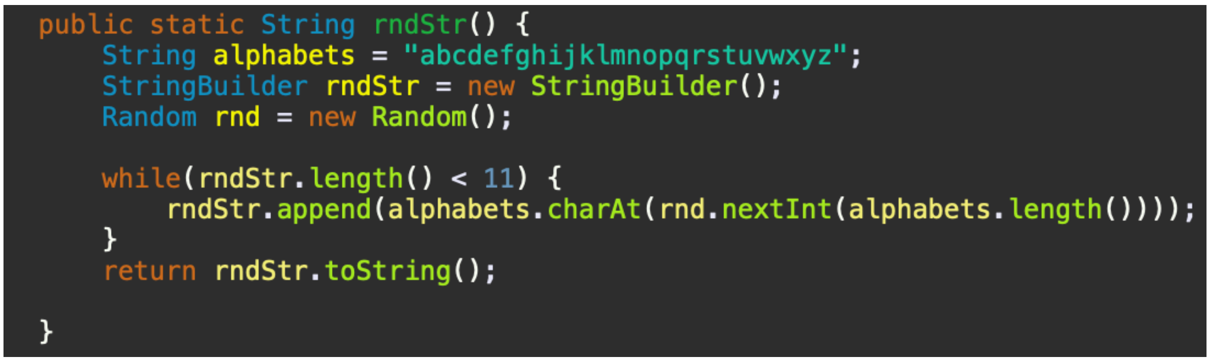
Table - 2: Length of Concatenated Strings

Both tables 1 and 2 show how the length of short strings is not affected by concatenations through regular Strings or through a StringBuilder. The difference in the concatenations within 1 second only shows when concatenating a long String, in this case being the letter 'a' 80 times. The results above show that both String Builders goes through a greater number of concatenations than the Strings. However, the long StringBuilder also produces a longer String in the end compared to the Long String. Hence, showing that the

StringBuilder does much more in 1 second than a String making it faster out of the two. This is because whilst appending a String the memory of both the original string is copied and the memory of the appended string is copied, both are then entered in a new String that has been created to give the final appended String. On the other hand, a StringBuilder only copies the memory of the append itself, not wasting memory on unnecessary copies. Thus, since StringBuilder uses less memory it is faster.

Exercise 7:

The task here was to use the Insertion Sort method previously created to sort an array of integers and strings in 1 second. Both arrays in this case had a size of 50, for the int array I inserted random integers ranging from 1 to 100. On the other hand, for the String array I was to insert randomly generated Strings of ten characters. I did this by creating a separate method that can be seen in the figure below.



```java
public static String rndStr() {
    String alphabets = "abcdefghijklmnopqrstuvwxyz";
    StringBuilder rndStr = new StringBuilder();
    Random rnd = new Random();

    while(rndStr.length() < 11) {
        rndStr.append(alphabets.charAt(rnd.nextInt(alphabets.length())));
    }
    return rndStr.toString();

}
```

Figure – 1: Random String Generator

Using the method in figure 1, I added random strings and random integers to their respective arrays in a while loop similar to the one in exercise 6 and then applied the insertion sort algorithm to each array within each loop. Below are the results after the three test runs, as well as an average of those runs.

Number of Sorted int or string	*Run 1*	*Run 2*	*Run 3*	*Run 4*	*Run 5*	***Average***
Int Array	1469111	984161	1197054	1550171	1564793	**1353058**
String Array	237590	229094	232625	218517	221403	**227845.8**

Table – 3: Amount of Sorted Strings and Integers

The data from table 3 clearly shows how using the insertion sort algorithm is much faster for integers than strings. This can be seen as the number of integers sorted is nearly 4 times the amount of string sorted. This could be due to the extra method for sorting strings in figure 1. Since java does not have a predefined random generator for strings it takes longer to call the separate method inside a while loop and then sort it with a separate algorithm. Whereas, in the case of the integers the random generator is predefined by java and inside the while loop, so no additional calls are made by the program. Hence, resulting in a faster sort of integers.