International Journal of Emerging Trends & Technology in Computer Science (IJETTCS)
Web Site: www.ijettcs.org Email: editor@ijettcs.org
Volume 5, Issue 6, November - December 2016 **ISSN 2278-6856**

Optimizing the usage of SimpleDateFormat in a Java multi-threaded environment

Mr. Mihai Stancu

Faculty of Sciences, Department of Computer Science, University of Craiova, 200585, Craiova, Romania

Abstract

In almost every Java application that uses dates we are required at some point to perform date formatting. The Java core libraries offer a great way to format dates using the DateFormat abstract class, and it's implementing class SimpleDateFormat. However the way SimpleDateFormat is generally used, creating a new SimpleDateFormat object for every formatting block, is very expensive and should not be used this way. In this article we will be looking at why this method is not the way to do it, and how it would be better to get the same results with lower execution times.

Keywords: Java, optimization, multi-thread enviroment

1. INTRODUCTION

Dates can be represented and displayed in many forms. Some are more human readable, other are more machinefriendly. Taking a raw date and transforming it in another form is called *formatting* [1]. Almost every application that works with dates will require some formatting at some point. Formatting can be done to include that date in a query that requires a specific date format, or to represent a date to a user using that user's locale date format. The date formatting is so important that based on the date format we used, two distinct users from two different countries that use different date formats, may interpret the same date as two distinct dates.

Let us assume we have the following date: *2013.02.03*. Without knowing the date's format we can not determine the exact date. Based on our general knowledge we can only assume that the year is *2013*, but we can not tell which is the day and which is the month. For example if the date format used was *yyyy.MM.dd* than the date is *03 February 2013*. But if the date format used was *yyyy.dd.MM* than the date is *02 March 2013*. So the importance of date formatting is major, and is a must in almost every application.

Java provides us in it's core libraries a great way to format dates: the *DateFormat* abstract class, and more specific the implementing class *SimpleDateFormat*. Even if there are some great Java libraries out there that deal with dates, like Joda [4], in many cases developers choose not to depend on an external library only for date formatting, because an external library may bring bugs to our application, bugs on which we don't have any controll because we didn't write and maintain that code.

Unfortunatelly some objects in Java are expensive to create, and one such case in the creation of *SimpleDateFormat* objects. In the next section we will how expensive creating lots of *SimpleDateFormat* objects can be, and why it is better to reuse *SimpleDateFormat* instances, and most important how to reuse them correctlly.

2. METHODS

In order to gather information on the costs of formatting dates using a new *SimpleDateFormat* object every time we will write a small program to determine how long it takes to format dates, and also we will decompile the *SimpleDateFormat* class to see what code goes behind the creation of an object of this type.

Using the code from Figure 1, we will determine the actual time it takes to format for 1.000.000 times a date, in this case the current day, using the format *dd.MM.yyyy HH:mm:ss*, creating a new instance of *SimpleDateFormat* each time.

```
public static void main(String[] args) {
        long startTime = System.currentTimeMillis();
        Date currentDay = new Date();
        String dateFormat = "dd.MM.yyyy HH:mm:ss";
        for (int i = 0; i < 1000000; i++) {
            DateFormat formatter = new SimpleDateFormat(
                  dateFormat);
            formatter.format(currentDay);
        }
        long endTime = System.currentTimeMillis();
        long executionDuration = endTime - startTime;
        System.out.println("Execution took " +
              executionDuration + " milliseconds");
    }
```

Figure 1 Measure the time needed for the standard formatting

In order to test how long it takes to format the same number of dates, but reusing the *SimpleDateFormat* instance we will modify the code from Figure 1 and create the *SimpleDateFormat* instance outside of the for loop as depicted in Figure 2.

```
...
DateFormat formatter = new SimpleDateFormat(
      dateFormat);
for (int i = 0; i < 1000000; i++) {
    formatter.format(currentDay);
}
...
```

Figure 2 Using one formatting instance

For the decompilation of the *SimpleDateFormat* we wil be using *JD-GUI* which can be downloaded from [3].

4 RESULTS

Running the unoptimized code, which creates a new instance of *SimpleDateFormat* every time, for 100.000, 1.000.000 and 10.000.000 dates we get the following results:

- For 100.000 formattings we have an execution time of 700 milliseconds.
- For 1.000.000 formattings we have an execution time of 5594 milliseconds.
- For 10.000.000 formattings we have an execution time of 52688 milliseconds.

Based on the above values we get the chart shown in Figure 3.

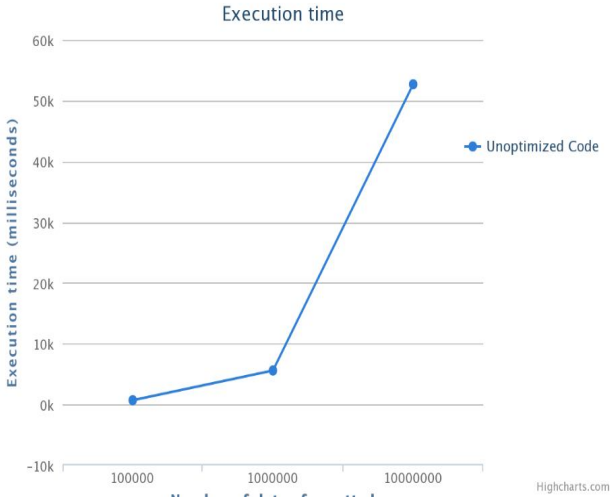

Figure 3 Execution time for unoptimized code

In order to make a comparison between the optimized and unoptimized code, we executed the optimized code getting the following results:

- For 100.000 formattings we have an execution time of 125 milliseconds.
- For 1.000.000 formattings we have an execution time of 657 milliseconds.
- For 10.000.000 formattings we have an execution time of 5984 milliseconds.

Based on the above values we get the chart from Figure 4.

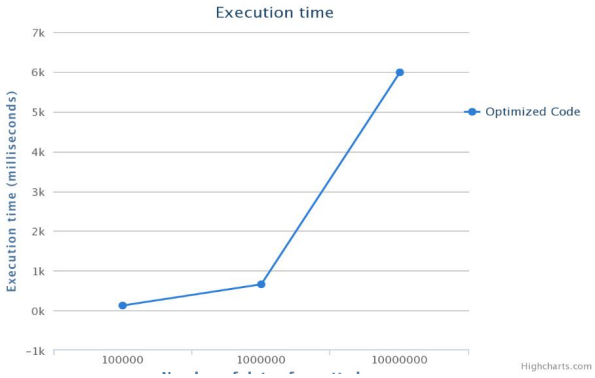

Figure 4 Execution time for optimized code

To get a better understanding of the times involved we will overlap the two charts, and by doing this we obtain the chart from Figure 5.

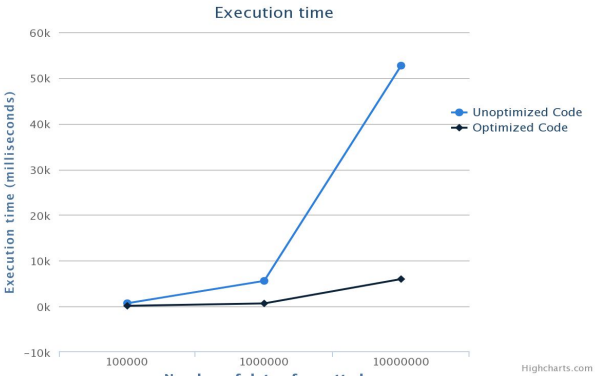

Figure 5 Overlaping the results

In all of the three charts on the X axys we have the number of dates formatted, and on the Y axis we have the execution times. Note that the results were obtained running the code on a virtual machine with 1 processor with a single core at 2.20GHz and a total memory of 2GB RAM.

5 DISCUSSION

Based on the results it is clear that a new instance of the *SimpleDateFormat* is expensive to create. Decompiling the code for the *SimpleDateFormat* we can see that this is due to the fact that the constructor performs a lot of operatios, of which many are time consumming, like the loading of the resource bundles for the locales.

The conclusion is that we must reuse as much as possible the instances of *SimpleDateFormat*. At a first look this can be done by making the *SimpleDateFormat* instance *static*, which means we only need to create one instance. This approach is good except it has one great drawback: it only works well in single threaded enviroments, because

SimpleDateFormat is not thread-safe.

We could try to solve this problem by putting the actual formatting code in a *synchronized* block, but that would only create a bottle-neck in the application, which is unacceptable.

The best way to reuse the instances is using the *ThreadLocal<T>* class. This class allows us to create a static variable, but every thread has it's own instance. It's usage is shown in Figure 6.

```
private static final ThreadLocal<SimpleDateFormat>
      formatter = new ThreadLocal<
      SimpleDateFormat>(){
@Override
protected SimpleDateFormat initialValue()
{
    return new SimpleDateFormat("dd.MM.yyyy");
}
};
```

Figure 6 Using ThreadLocal<T> in formatting process

This allows us to reuse the formatter instance, in a multi-treaded enviroment, by allowing a new instance to be created for every thread. In fact if we have 5 threads, we will have 5 instances of the *SimpleDateFormat* class, solving the problem that the *SimpleDateFormat* class is not thread-safe, and also allowing us to format all the dates in a single thread using the same instance.

6 CONCLUSION AND FUTURE WORK

We highlighted the importance of the date formatting and the high cost of these operation in Java based applications. We discussed the common optimization method and it's drawback into the multi-threaded environment. After some experimental results, we propose a different way of optimization that can be reliable in a Java multi-threaded environment.

In future works, the methods of optimizing the creation and use of objects that are expensive to create can be extended to multiple Java core classes like *NumberFormat*.

References

[1] J. Bloch, Effective Java, Addison-Wesley Professional, ISBN 0321356683, 2008.
[2] B. Goetz, et al., Java Concurrency in Practice. Addison-Wesley Professional, ISBN 0321349601, 2006.
[3] JD-GUI, Standalone graphical Java decompiler. [Online]. Available: http://jd.benow.ca/. [Accessed: November. 26, 2016].
[4] Joda-Time. Date-Time Java library. [Online]. Available: http://www.joda.org/joda-time/. [Accessed: November. 26, 2016].