My code is faster than yours... let me prove it to you!

François Martin

KARAKUN

François Martin

Senior Full Stack Software Engineer

francois.martin@karakun.com

martinfrancois

/in/françoismartin

@fmartin_

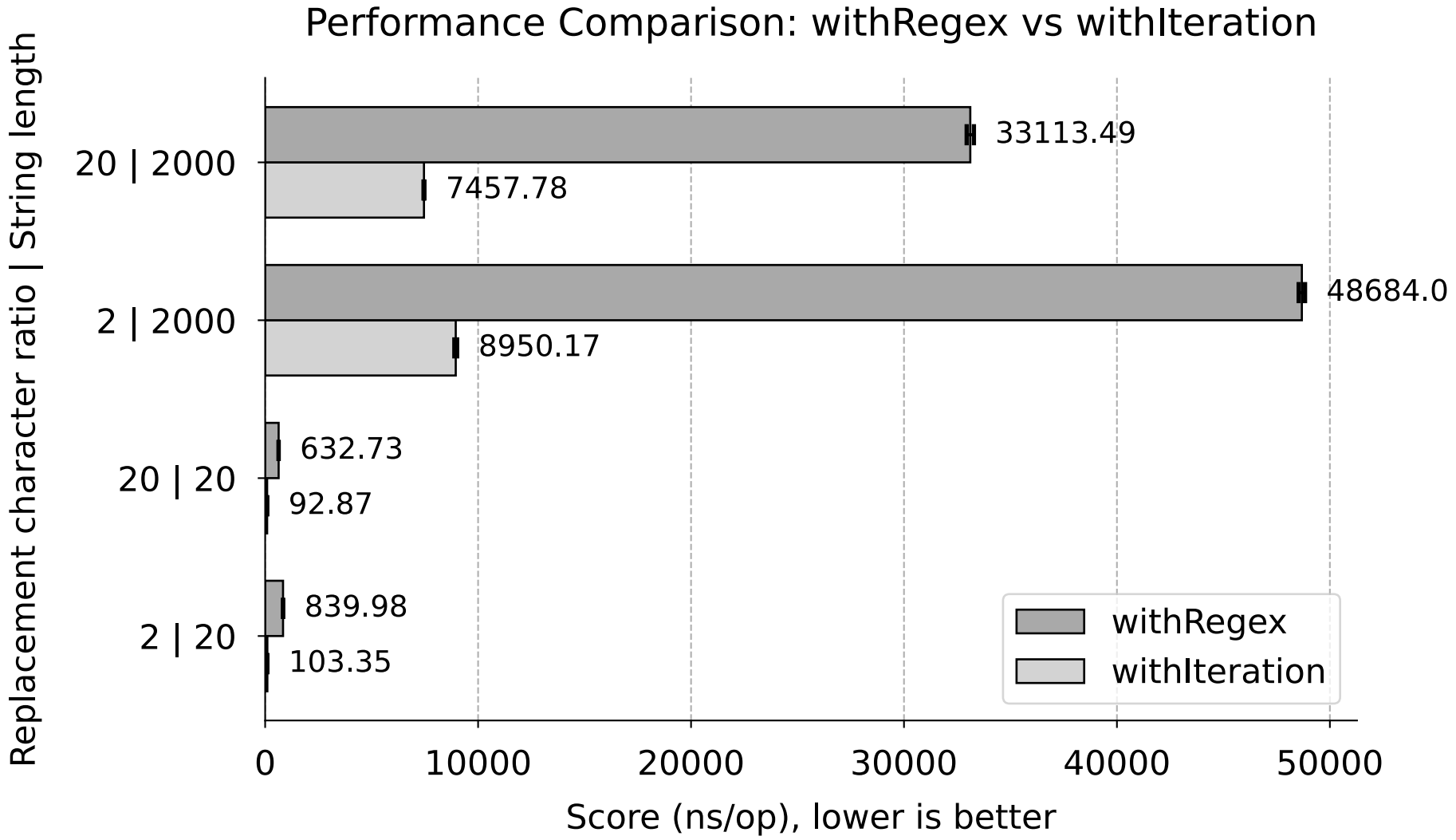
Performance of Regular Expressions

```
String removeIllegalCharacters(String text, String replace) {
  text = text.replaceAll("[\\p{Cntrl}/\\\\:\\*\\?\"<>|]", replace);
  text = text.replaceAll("(^[ \\t]+)|([ \\t]+$)", replace);
  return text;
}
```

- Example input: `"\tproject\\name:final<version>\u007F "`
- Expected output: `"-project-name-final-version---“`
- `(^[ \t]+)|([ \t]+$)` is susceptible to ReDoS
  - Attack string: `'\x00' + '\t'.repeat(54773) + '\t\x00'`

Performance of Regular Expressions

```java
Set<Character> ILLEGAL_CHARS = Set.of('/', '\\', '<', '>', ':', '\"', '|', '?', '*', '\u007F');
String removeIllegalCharacters(String text, String replaceWith) {
  int startIndex = 0;
  int endIndex = text.length();
  StringBuilder sb = new StringBuilder();
  // find the index of when the string (excluding whitespaces) starts
  while ((startIndex < endIndex) && (text.charAt(startIndex) <= ' '))
    startIndex++;
  // find the index of when the string (excluding whitespaces) ends
  while ((startIndex < endIndex) && (text.charAt(endIndex - 1) <= ' '))
    endIndex--;
  // if there are whitespaces at the of the string, replace all of them with ONE `replaceWith`
  if (startIndex != 0)
    sb.append(replaceWith);
  // replace control and illegal characters in the string itself
  for (int i = startIndex; i < endIndex; i++) {
    char current = text.charAt(i);
    boolean isControlCharacter = current < ' ';
    if (isControlCharacter || ILLEGAL_CHARS.contains(current)) {
        sb.append(replaceWith);
    } else {
        sb.append(current);
    }
  }
  // if there are whitespaces at the end of the string, replace all of them with ONE `replaceWith`
  if (text.length() != endIndex)
    sb.append(replaceWith);
  return sb.toString();
}
```


Performance Comparison: withRegex vs withIteration
Replacement character ratio | String length
20 | 2000
2 | 2000
20 | 20
2 | 20
33113.49
7457.78
48684.0
8950.17
632.73
92.87
839.98
103.35
withRegex
withIteration
0
10000
20000
30000
40000
50000
Score (ns/op), lower is better

Microbenchmarking

- Definition: „Microbenchmarking is about **measuring the time** or performance of **small** to very small building blocks of **real programs**. This can be a common data access pattern, a sequence of operations or **even a single instruction**.“ – Source: HPC Wiki
- State-of-the-art tool for Java (and other JVM languages): **JMH (Java Microbenchmarking Harness)**
- Microbenchmarking in JavaScript
  - jsbenchmark.com (Example: Sorting an array of integers)
  - perf.link (Example Regex Microbenchmark)
  - Benchmark.js (from 2016 and unmaintained, but still working well)
- **Microbenchmarking** is best suited for **comparing** performance of different implementations under controlled conditions
  - To **understand** and **improve** the performance of applications in **real-world scenarios**, use **profiling** instead!

Step 1 – Setting up JMH

- Follow the README.md in the JMH repository **closely**
- Run the following command (adjust groupId and artifactId):

```
mvn archetype:generate \
  -DinteractiveMode=false \
  -DarchetypeGroupId=org.openjdk.jmh \
  -DarchetypeArtifactId=jmh-java-benchmark-archetype \
  -DgroupId=org.sample \
  -DartifactId=test \
  -Dversion=1.0
```

- Creates a folder with the same name as the artifactId
- Open the folder in your IDE

Step 2 – What to compare?

- *I can't believe I have to pay £50 & wait 5 days—just fix it already!*
  - HTML encoded string
- Decoded:
  - *I can't believe I have to pay £50 & wait 5 days—just fix it already!*
- Many libraries available for decoding:
  - Apache Commons Text: `StringEscapeUtils.unescapeHtml4(htmlEncodedString)`
  - jsoup: `Jsoup.parse(htmlEncodedString).text()`
  - unbescape: `HtmlEscape.unescapeHtml(htmlEncodedString)`
  - Spring Web: `htmlUnescape(htmlEncodedString)`
- Which one is the **fastest when decoding lots of different strings?**
- Optional, but recommended if possible: **what is the expected outcome and why?**

„Fastest“ depends on the context

- Knowing the **context** in which code is used is **crucial** for accurate microbenchmarking
- Microbenchmark should replicate production conditions as closely as possible
- Depending on the data being used, the performance can vary significantly
  - Example: Quicksort generally operates with O(n log n) complexity, making it one of the fastest sorting algorithms for most cases
  - However, for sorted data, Quicksort degrades to $O(n^2)$ → other sorting algorithms are better
  - If sorting lots of (almost) sorted data in production, but only microbenchmarking randomized ordered datasets leads to a wrong decision!
- Ideally use data observed in production, benchmarking on production hardware with same JRE

Step 2 – Adding benchmarks

- The command from step 1 generates a basic `MyBenchmark.java`:

```
public class MyBenchmark {
    @Benchmark
    public void testMethod() {
        // This is a demo/sample...
        // Put your benchmark code here.
    }
}
```

- Each comparison we want to make is its own method annotated with **`@Benchmark`**

Step 2 – Adding benchmarks

- So we can simply call the method we want to benchmark… right?

```
@Benchmark
public void apacheCommonsText() {
  StringEscapeUtils.unescapeHtml4("Wow, that&#x27;s easy!");
}
```

- Wrong! The Java compiler is too smart and **removes the line of code**, as it is redundant, so we would measure **nothing** being executed! (dead code elimination)
- If only performing one operation, can return the result instead:

```
@Benchmark
public String apacheCommonsText() {
  return StringEscapeUtils.unescapeHtml4("That&#x27;s still quite easy!");
}
```

Step 2 – Adding benchmarks

- When we need to „return multiple results“, we can use a **blackhole** instead:

```
String[] strings = new String[]{"That&#x27;s String 1",
                                "That&#x27;s String 2"};

@Benchmark
public void apacheCommonsText(Blackhole bh) {
  for (int i = 0; i < strings.length; i++) {
    bh.consume(StringEscapeUtils.unescapeHtml4(strings[i]));
  }
}
```

- **Caution: only** use **loops** in benchmarks if you use them in your production context as well!
  - The JIT is very good at optimizing loops, which you **want** to **include** in your benchmark **if** it is part of your **production use case**
  - Do **NOT** use loops to test different data!

Step 2 – Benchmarking with State

- Microbenchmarks are usually executed in multiple measurement iterations
- When using state like this:

```
String[] strings = new String[]{"That&#x27;s String 1",
                                "That&#x27;s String 2"};
@Benchmark
public void apacheCommonsText(Blackhole bh) {
  for (int i = 0; i < strings.length; i++) {
    bh.consume(StringEscapeUtils.unescapeHtml4(strings[i]));
  }
}
```

- The JVM could perform an optimization called “constant folding” and even with blackhole **reuse** the **same result in each iteration without executing the code again**, leading to incorrect measurements

Step 2 – Benchmarking with State

- Use @State instead:

```
@State(Scope.Thread)
public static class ThreadState {
  public String[] strings = new String[]{
    "That&#x27;s String 1", "That&#x27;s String 2"};
}
@Benchmark
public void apacheCommonsText(ThreadState state, Blackhole bh) {
  for (int i = 0; i < state.strings.length; i++) {
    bh.consume(StringEscapeUtils.unescapeHtml4(state.strings[i]));
  }
}
```

Step 3 – Adding Annotations

- Benchmark Modes (excerpt)
  - `@BenchmarkMode(Mode.Throughput)`
    - Measures how many executions it can perform within a given time unit (**the only mode** where higher score is better)
  - `@BenchmarkMode(Mode.AverageTime)`
    - Measures the average time it takes to execute (lower score is better)
- Other annotations to use as a starting point

```
@Warmup(iterations = 10, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 20, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(1)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
```

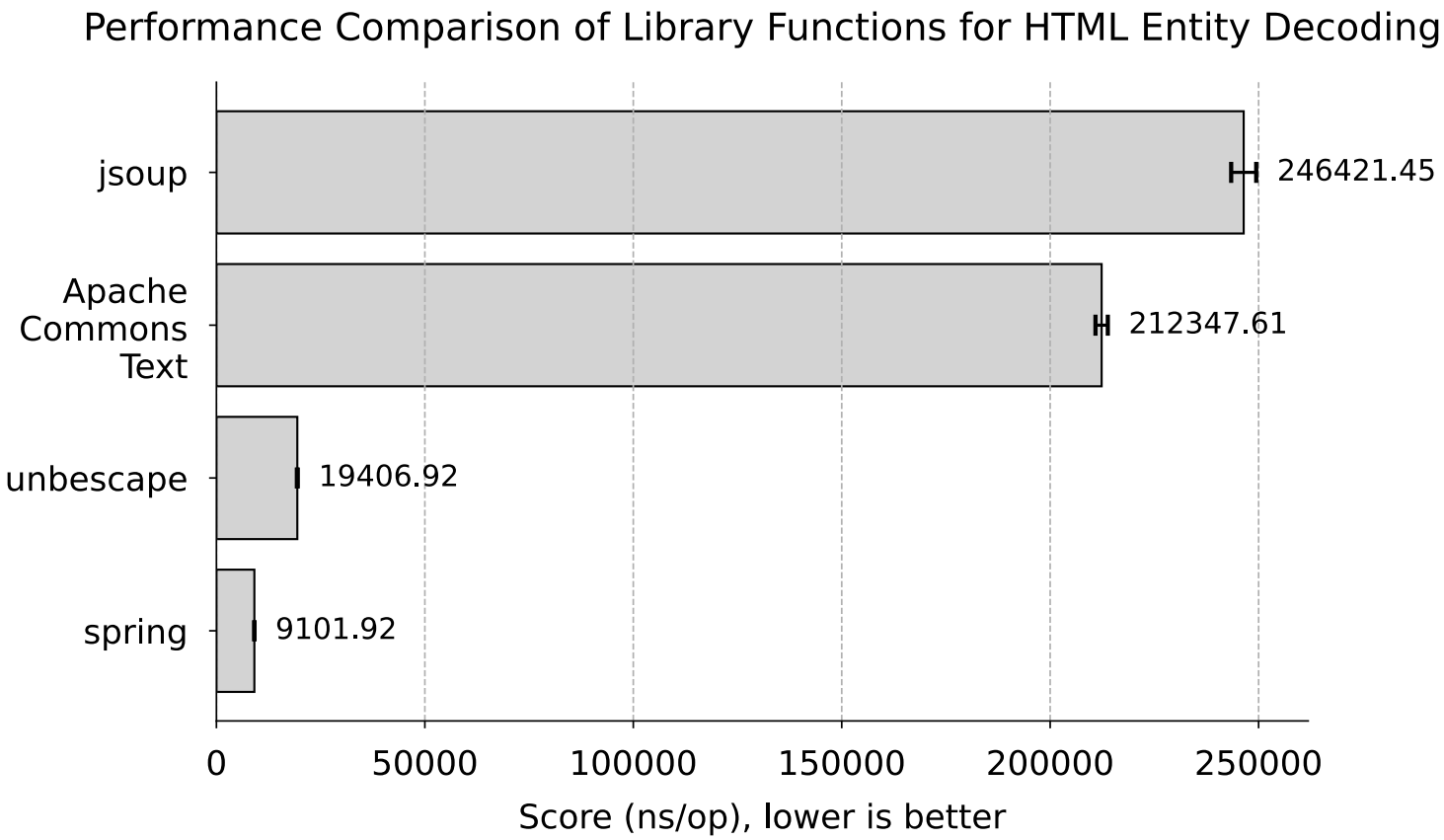
- Increase the amount of iterations if the error is too high
- For long-running operations (I/O, complex computations), increase time of iterations

Step 4 – Executing the benchmark

- For the most accurate results
  - If possible, turn off dynamic CPU frequency scaling in your BIOS (power saving, turbo boost)
  - Restart your computer
  - Close all other applications, background processes, browser, IDE, etc. except for a terminal
- Run the following commands in the terminal window:
  - `mvn clean verify`
  - `java -jar target/benchmarks.jar`
- The results will be printed at the end:

```
Benchmark              Mode  Cnt       Score      Error  Units
apacheCommonsText      avgt   20  212347.609 ± 1479.364  ns/op
jsoup                  avgt   20  246421.453 ± 3004.214  ns/op
spring                 avgt   20    9101.915 ±   40.974  ns/op
unbescape              avgt   20   19406.916 ±  108.725  ns/op
```

Step 5 – Interpretation of Results



Step 5 – Interpretation of Results

- Advice printed at the end of each JMH run:
  - REMEMBER: The numbers below are just data. **To gain reusable insights, you need to follow up on why the numbers are the way they are**. Use profilers (see -prof, -lprof), design factorial experiments, perform baseline and negative tests that provide experimental control, make sure the benchmarking environment is safe on JVM/OS/HW level, **ask for reviews from the domain experts**. **Do not assume the numbers tell you what you want them to tell**.
- Does **NOT** mean that it will perform the best in every scenario, only in this specific context
- Verify to ensure the functions you benchmark also work correctly (for example, with unit tests)
- Microbenchmarking results will be different depending on the hardware and other factors
- Recommendation: work through the samples provided by JMH to learn about pitfalls

Learn more

- https://github.com/openjdk/jmh
- https://github.com/openjdk/jmh/tree/master/jmh-samples/src/main/java/org/openjdk/jmh/samples
- https://github.com/Valloric/jmh-playground/blob/master/README.md#learning-to-use-jmh
- https://blog.avenuecode.com/java-microbenchmarks-with-jmh-part-1
- https://blog.avenuecode.com/java-microbenchmarks-with-jmh-part-2
- https://blog.avenuecode.com/java-microbenchmarks-with-jmh-part-3
- https://jenkov.com/tutorials/java-performance/jmh.html
- https://www.oracle.com/technical-resources/articles/java/architect-benchmarking.html
- https://coderstower.com/2019/08/06/arraylist-vs-linkedlist-sort-get-iteration/
- https://medium.com/@AlexanderObregon/introduction-to-java-microbenchmarking-with-jmh-java-microbenchmark-harness-55af74b2fd38

Slides & Code:

https://fm.ht/baselone2024