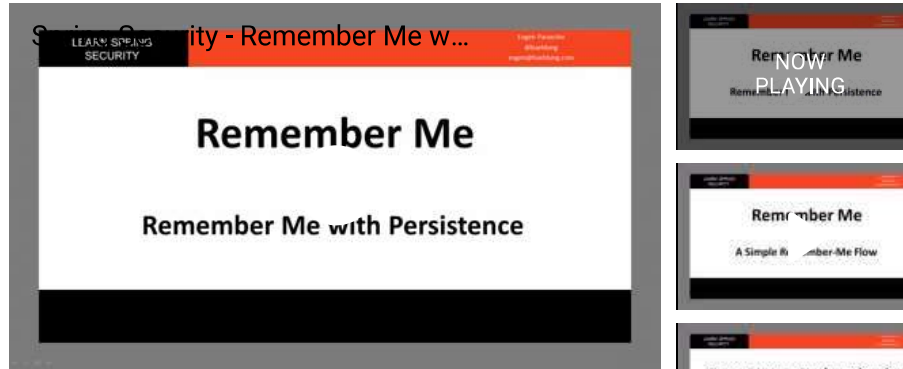




Microbenchmarking with Java

FEATURED VIDEOS



Last updated: June 26, 2024



Written by: [baeldung \(https://www.baeldung.com/author/baeldung\)](https://www.baeldung.com/author/baeldung)



Reviewed by: [Zeger Hendrikse \(https://www.baeldung.com/editor/zeger-author\)](https://www.baeldung.com/editor/zeger-author)

[Java \(https://www.baeldung.com/category/java\)](https://www.baeldung.com/category/java) +

[JMH \(https://www.baeldung.com/tag/jmh\)](https://www.baeldung.com/tag/jmh) [reference](#) >

1. Introduction

This quick article is focused on JMH (the Java Microbenchmark Harness). First, we get familiar with the API and learn its basics. Then we would see a few best practices that we should consider when writing microbenchmarks.

Simply put, JMH takes care of the things like JVM warm-up and code-optimization paths, making benchmarking as simple as possible.

2. Getting Started

To get started with Maven, **we need to declare dependencies in *pom.xml***:

```

</dependency>
<groupId>org.openjdk.jmh</groupId>
<artifactId>jmh-core</artifactId>
<version>1.37</version>
</dependency>
<dependency>
<groupId>org.openjdk.jmh</groupId>
<artifactId>jmh-generator-annprocess</artifactId>
<version>1.37</version>
</dependency>

```

[Start Here \(/start-here\)](#)

[Courses ▾](#)

[Guides ▾](#)

[About ▾](#)

[\(/feed\)](#)



[\(/members/\)](#)

The latest versions of the JMH Core (<https://mvnrepository.com/artifact/org.openjdk.jmh/jmh-core>) and JMH Annotation Processor (<https://mvnrepository.com/artifact/org.openjdk.jmh/jmh-generator-annprocess>) can be found in Maven Central.

Next, **we need to add the JMH Annotation Processor to the Maven Compiler Plugin configuration**. This step is critical because, without it, we may get the error *Unable to find the resource: /META-INF/BenchmarkList*.

```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.13.0</version>
      <configuration>
        <source>17</source>
        <target>17</target>
        <annotationProcessorPaths>
          <path>
            <groupId>org.openjdk.jmh</groupId>
            <artifactId>jmh-generator-annprocess</artifactId>
            <version>1.37</version>
          </path>
        </annotationProcessorPaths>
      </configuration>
    </plugin>
  </plugins>
</build>

```

Next, let's create a simple benchmark using **the `@Benchmark` annotation, which requires that the method it is applied to have the `public` modifier**. The class containing the benchmark methods must also be declared *public*:

```

@Benchmark
public void init() {
    // Do nothing
}

```

Then we add the *main* class that starts the benchmarking process:

```

public class BenchmarkRunner {
    public static void main(String[] args) throws Exception {
        org.openjdk.jmh.Main.main(args);
    }
}

```

[Start Here \(/start-here\)](#)

[Courses ▾](#)

[Guides ▾](#)

[About ▾](#)

[\(/feed\)](#)



[\(/members/\)](#)

However, this *main* will only work if we run our Maven project using *exec:exec*. If we use *exec:java* instead and the *@fork* annotation is missing or different from *@fork(0)*, we may get the error *Could not find or load main class org.openjdk.jmh.runner.ForkedMain*. There are two ways to work around this problem. The first is to **add this code to the *main* method before running *org.openjdk.jmh.Main.main(args)***.

```

URLClassLoader classLoader = (URLClassLoader) BenchmarkRunner.class.getClassLoader();
StringBuilder classpath = new StringBuilder();
for (URL url : classLoader.getURLs()) {
    classpath.append(url.getPath()).append(File.pathSeparator);
}
System.setProperty("java.class.path", classpath.toString());

```

Basically, this code dynamically retrieves and sets the classpath for the Java Virtual Machine at runtime. This helps to ensure that all required classes and resources are available during execution. **Alternatively, we can use a solution based solely on *pom.xml*:**

```

<plugin>
  <artifactId>maven-dependency-plugin</artifactId>
  <executions>
    <execution>
      <id>build-classpath</id>
      <goals>
        <goal>build-classpath</goal>
      </goals>
      <configuration>
        <includeScope>runtime</includeScope>
        <outputProperty>depClasspath</outputProperty>
      </configuration>
    </execution>
  </executions>
</plugin>
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <configuration>
    <mainClass>BenchmarkRunner</mainClass>
    <systemProperties>
      <systemProperty>
        <key>java.class.path</key>
        <value>${project.build.outputDirectory}${path.separator}${depClasspath}</value>
      </systemProperty>
    </systemProperties>
  </configuration>
</plugin>

```

This *pom.xml* configuration accomplishes a similar goal to the previous Java code. Both solutions are equivalent in that they ensure that all required runtime dependencies are included in the classpath, but the *pom.xml* approach uses Maven's build and execution capabilities to automate this process within the build lifecycle, while the Java code does it programmatically at runtime.



Now running *BenchmarkRunner* will execute our arguably somewhat useless benchmark. Once the run is complete, a summary table is presented:

```
# Run complete. Total time: 00:06:45
Benchmark      Mode  Cnt Score          Error       Units
BenchMark.init thrpt 200 3099210741.962 ± 17510507.589 ops/s
```

3. Types of Benchmarks

JMH supports some possible benchmarks: *Throughput*, *AverageTime*, *SampleTime*, and *SingleShotTime*. These can be configured via *@BenchmarkMode* annotation:

```
@Benchmark
@BenchmarkMode(Mode.AverageTime)
public void init() {
    // Do nothing
}
```

The resulting table will have an average time metric (instead of throughput):

```
# Run complete. Total time: 00:00:40
Benchmark Mode  Cnt   Score Error Units
BenchMark.init avgt 20 ~ 10-9 s/op
```

4. Configuring Warmup and Execution

By using the *@Fork* annotation, we can set up how the benchmark execution happens: the *value* parameter controls how many times the benchmark will be executed, and the *warmup* parameter controls how many times a benchmark will dry run before results are collected, for example:

```
@Benchmark
@Fork(value = 1, warmups = 2)
@BenchmarkMode(Mode.Throughput)
public void init() {
    // Do nothing
}
```

This instructs JMH to run two warm-up forks and discard results before moving onto real timed benchmarking.



Also, the `@Warmup` annotation can be used to control the number of warmup iterations. For example, `@Warmup(iterations = 5)` tells JMH that five warm-up iterations will suffice, as opposed to the default 20.

5. State

Let's now examine how a less trivial and more indicative task of benchmarking a hashing algorithm can be performed by utilizing *State*. Suppose we decide to add extra protection from dictionary attacks on a password database by hashing the password a few hundred times.

We can explore the performance impact by using a *State* object:

```
@State(Scope.Benchmark)
public class ExecutionPlan {

    @Param({ "100", "200", "300", "500", "1000" })
    public int iterations;

    public Hasher murmur3;

    public String password = "4v3rys3kur3p455w0rd";

    @Setup(Level.Invocation)
    public void setUp() {
        murmur3 = Hashing.murmur3_128().newHasher();
    }
}
```

Our benchmark method then will look like:

```
@Fork(value = 1, warmups = 1)
@Benchmark
@BenchmarkMode(Mode.Throughput)
public void benchMurmur3_128(ExecutionPlan plan) {

    for (int i = plan.iterations; i > 0; i--) {
        plan.murmur3.putString(plan.password, Charset.defaultCharset());
    }

    plan.murmur3.hash();
}
```

Here, the field *iterations* will be populated with appropriate values from the `@Param` annotation by the JMH when it is passed to the benchmark method. The `@Setup` annotated method is invoked before each invocation of the benchmark and creates a new *Hasher* ensuring isolation.

When the execution is finished, we'll get a result similar to the one below:

Run complete. Total time: 00:06:47

Start Here (/start-here) Courses Guides About (/feed) (/members/)

Benchmark	(iterations)	Mode	Cnt	Score	Error	Units
BenchMark.benchMurmur3_128	100	thrpt	20	97463.622 ± 1672.227		ops/s
BenchMark.benchMurmur3_128	200	thrpt	20	39737.532 ± 5294.200		ops/s
BenchMark.benchMurmur3_128	300	thrpt	20	30381.144 ± 614.500		ops/s
BenchMark.benchMurmur3_128	500	thrpt	20	18315.211 ± 222.534		ops/s
BenchMark.benchMurmur3_128	1000	thrpt	20	8960.008 ± 658.524		ops/s

6. Dead Code Elimination

When running microbenchmarks, it's very important to be aware of optimizations. Otherwise, they may affect the benchmark results in a very misleading way.

To make matters a bit more concrete, let's consider an example:

```
@Benchmark
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@BenchmarkMode(Mode.AverageTime)
public void doNothing() {
}

@Benchmark
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@BenchmarkMode(Mode.AverageTime)
public void objectCreation() {
    new Object();
}
```

We expect object allocation costs more than doing nothing at all. However, if we run the benchmarks:

Benchmark	Mode	Cnt	Score	Error	Units
BenchMark.doNothing	avgt	40	0.609 ± 0.006		ns/op
BenchMark.objectCreation	avgt	40	0.613 ± 0.007		ns/op

Apparently finding a place in the TLAB (<https://alidg.me/blog/2019/6/21/tlab-jvm>), creating and initializing an object is almost free! Just by looking at these numbers, we should know that something does not quite add up here.

Here, we're the victim of dead code elimination. Compilers are very good at optimizing away the redundant code. As a matter of fact, that's exactly what the JIT compiler did here.

In order to prevent this optimization, we should somehow trick the compiler and make it think that the code is used by some other component. One way to achieve this is just to return the created object:




1

```
@Benchmark
public double foldedLog() {
    return 2.0794415416798357;
}
```

This form of partial evaluation is called **constant folding**. In this case, constant folding completely avoids the *Math.log* call, which was the whole point of the benchmark.

In order to prevent constant folding, we can encapsulate the constant state inside a state object:

Start Here (/start-here) Courses ▾ Guides ▾ About ▾ (/feed)  (/members/)

```
@State(Scope.Benchmark)
public static class Log {
    public int x = 8;
}

@Benchmark
public double log(Log input) {
    return Math.log(input.x);
}
```


If we run these benchmarks against each other:

Benchmark	Mode	Cnt	Score	Error	Units
BenchMark.foldedLog	thrpt	20	449313097.433 ± 11850214.900		ops/s
BenchMark.log	thrpt	20	35317997.064 ± 604370.461		ops/s

Apparently, the *log* benchmark is doing some serious work compared to the *foldedLog*, which is sensible.

8. Conclusion

This tutorial focused on and showcased Java's micro benchmarking harness.
As always, code examples can be found on GitHub (<https://github.com/eugenp/tutorials/tree/master/jmh>).



Get started with Spring Boot and with core Spring, through the *Learn Spring* course:

>> CHECK OUT THE COURSE (course-ls-NPI-7Y5v&)

COURSES

- ALL COURSES (/COURSES/ALL-COURSES)
- BAELDUNG ALL ACCESS (/COURSES/ALL-ACCESS)
- BAELDUNG ALL TEAM ACCESS (/COURSES/ALL-ACCESS-TEAM)
- THE COURSES PLATFORM ([HTTPS://COURSES.BAELDUNG.COM](https://courses.baeldung.com))

[JAVA "BACK TO BASICS" TUTORIAL \(/JAVA-TUTORIAL\)](#)
[JACKSON JSON SERIES \(/JACKSON\)](#)
[APACHE HTTPCLIENT SERIES \(/HTTPCLIENT-SERIES\)](#)
[REST WITH SPRING SERIES \(/REST-WITH-SPRING-SERIES\)](#)
[SPRING PERSISTENCE SERIES \(/PERSISTENCE-WITH-SPRING-SERIES\)](#)
[SECURITY WITH SPRING \(/SECURITY-SPRING\)](#)
[SPRING REACTIVE SERIES \(/SPRING-REACTIVE-SERIES\)](#)

ABOUT

[ABOUT BAELDUNG \(/ABOUT\)](#)
[THE FULL ARCHIVE \(/FULL_ARCHIVE\)](#)
[EDITORS \(/EDITORS\)](#)
[OUR PARTNERS \(/PARTNERS/\)](#)
[PARTNER WITH BAELDUNG \(/PARTNERS/WORK-WITH-US\)](#)
[EBOOKS \(/LIBRARY/\)](#)
[FAQ \(HTTPS://WWW.BAELDUNG.COM/LIBRARY/FAQ\)](#)
[BAELDUNG PRO \(/MEMBERS/\)](#)

[TERMS OF SERVICE \(/TERMS-OF-SERVICE\)](#)
[PRIVACY POLICY \(/PRIVACY-POLICY\)](#)
[COMPANY INFO \(/BAELDUNG-COMPANY-INFO\)](#)
[CONTACT \(/CONTACT\)](#)