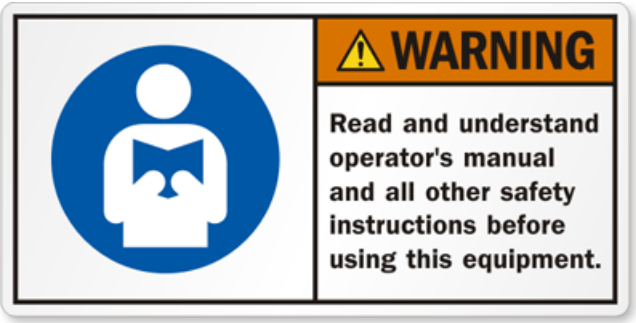
JMH

Jiří Pinkas
@jirkapinkas
https://github.com/jirkapinkas

Warning

- Premature optimization is the root of all evil
  - Donald Knuth



- To neznamená, že je microbenchmarking “evil”. ale předtím, než k němu přistoupíme, tak musíme mít reálný problém, který potřebujeme řešit. Jak zjistit, jestli vůbec problém máme? Pomocí nástrojů jako je JProfiler, VisualVM (obojí na dev / test), nebo JMC & FlightRecorder (na produkci).

Real World Example

- Můj problém:
  - Čtu data ze zdrojových souborů, provádím jejich transformaci a ukládám je do databáze. Data jsou na SSD disku, rychleji přečíst nejdou. Do databáze je ukládám přes jdbc batch, rychleji to už nejde. Celá operace trvá hodinu (a provádí se periodicky, jedná se o batch operaci). Na tuto operaci jsem nahodil JProfiler a jaké bylo mé překvapení, 90% času se trávilo v metodě replace() ze třídy String. Začal jsem googlit a zjistil jsem, že implementace této metody je velice neefektivní a alternativy z různých Java knihoven (Apache Commons Lang, Guava, …) jsou efektivnější, tak jsem nažhavil JMH abych zjistil jaká z nich bude nejlepší. Nakonec jsem použil Apache Commons Lang a operaci, která trvala hodinu jsem tím zkrátil na pouhých 10 minut.
    - Poznámka: V Java 9 byla tato metoda v Javě reimplementována a její výkon je v současné době srovnatelný s Apache Commons Lang implementací.

Nejsem jediný kdo na tento problém narazil, následující příklad jsem obšlehnul (a rozšířil) z:
https://medium.com/javarevisited/micro-optimizations-in-java-string-replaceall-c6d0edf2ef6

JMH

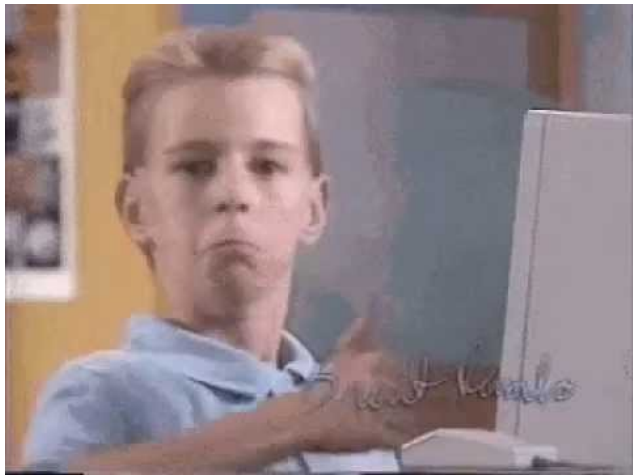
- JMH (Java Microbenchmark Harness) je nástroj, který pomáhá správným způsobem implementovat microbenchmark v Javě.
- Prosím nikdy za žádných okolností nepište něco takového:

```
public static void main(String[] args) {
    long millis = System.currentTimeMillis();
    for (int i = 0; i < 10_000_000; i++) {
        "some.Path.With.Dot".replace(".", "/");
    }
    System.out.println("delka operace: " + (System.currentTimeMillis() - millis));
}
```

JMH

- Místo toho použijte JMH:

```
@BenchmarkMode(Mode.AverageTime)
@Fork(1)
@State(Scope.Thread)
@Warmup(iterations = 5, time = 1)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@Measurement(iterations = 10, time = 1)
public class MyBenchmark {

    @Param({"", "somePathNoDoT", "some.Path.With.Dot"})
    String value;

    @Benchmark
    public String stringReplace() {
        return value.replace(".", "/");
    }

    @Benchmark
    public String replaceApache() {
        return StringUtils.replace(value, ".", "/");
    }

}
```



Výsledky běhů Java 8, 11 & 17 a Apache Commons

java 8:

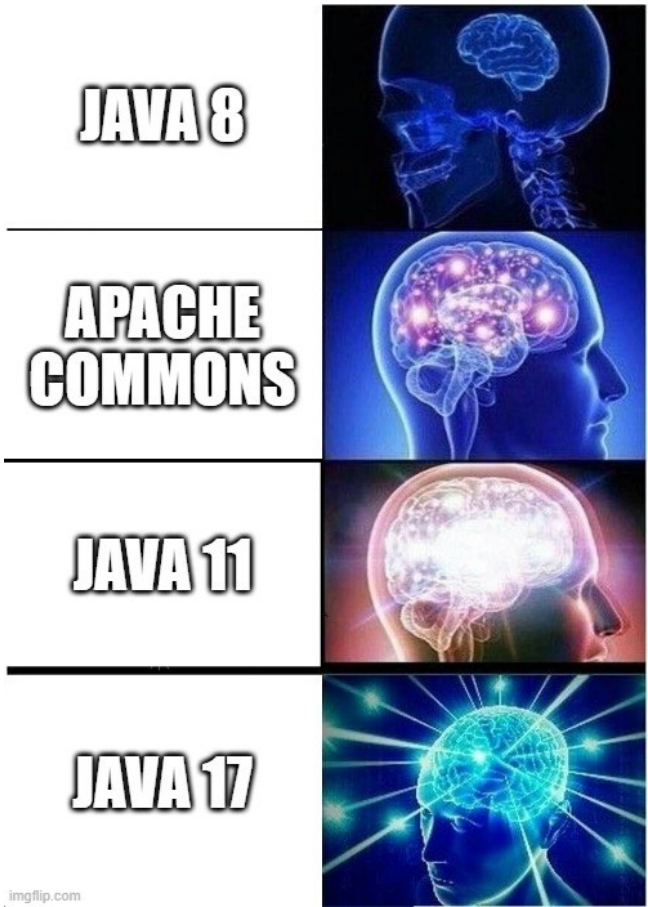
Benchmark	(value)	Mode	Cnt	Score	Error	Units
MyBenchmark.replaceApache		avgt	10	2.736	± 0.157	ns/op
MyBenchmark.replaceApache	somePathNoDoT	avgt	10	10.312	± 0.330	ns/op
MyBenchmark.replaceApache	some.Path.With.Dot	avgt	10	121.485	± 2.981	ns/op
MyBenchmark.stringReplace		avgt	10	129.949	± 58.219	ns/op
MyBenchmark.stringReplace	somePathNoDoT	avgt	10	125.007	± 2.180	ns/op
MyBenchmark.stringReplace	some.Path.With.Dot	avgt	10	369.159	± 7.372	ns/op

java 11:

Benchmark	(value)	Mode	Cnt	Score	Error	Units
MyBenchmark.replaceApache		avgt	10	3.544	± 0.171	ns/op
MyBenchmark.replaceApache	somePathNoDoT	avgt	10	7.865	± 0.227	ns/op
MyBenchmark.replaceApache	some.Path.With.Dot	avgt	10	118.892	± 5.880	ns/op
MyBenchmark.stringReplace		avgt	10	3.526	± 0.216	ns/op
MyBenchmark.stringReplace	somePathNoDoT	avgt	10	6.946	± 0.239	ns/op
MyBenchmark.stringReplace	some.Path.With.Dot	avgt	10	111.893	± 8.758	ns/op

java 17:

Benchmark	(value)	Mode	Cnt	Score	Error	Units
MyBenchmark.replaceApache		avgt	10	2.832	± 0.061	ns/op
MyBenchmark.replaceApache	somePathNoDoT	avgt	10	6.847	± 0.201	ns/op
MyBenchmark.replaceApache	some.Path.With.Dot	avgt	10	111.250	± 2.963	ns/op
MyBenchmark.stringReplace		avgt	10	3.362	± 0.163	ns/op
MyBenchmark.stringReplace	somePathNoDoT	avgt	10	7.759	± 0.125	ns/op
MyBenchmark.stringReplace	some.Path.With.Dot	avgt	10	19.232	± 0.506	ns/op



Jak na to? I. (správný způsob)

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

Poté přidat svůj benchmark a zavolat:

```
cd first-benchmark
mvn clean package
java -jar target/benchmarks.jar
```

Jak na to? II. (jednodušší způsob)

1. Dependency:

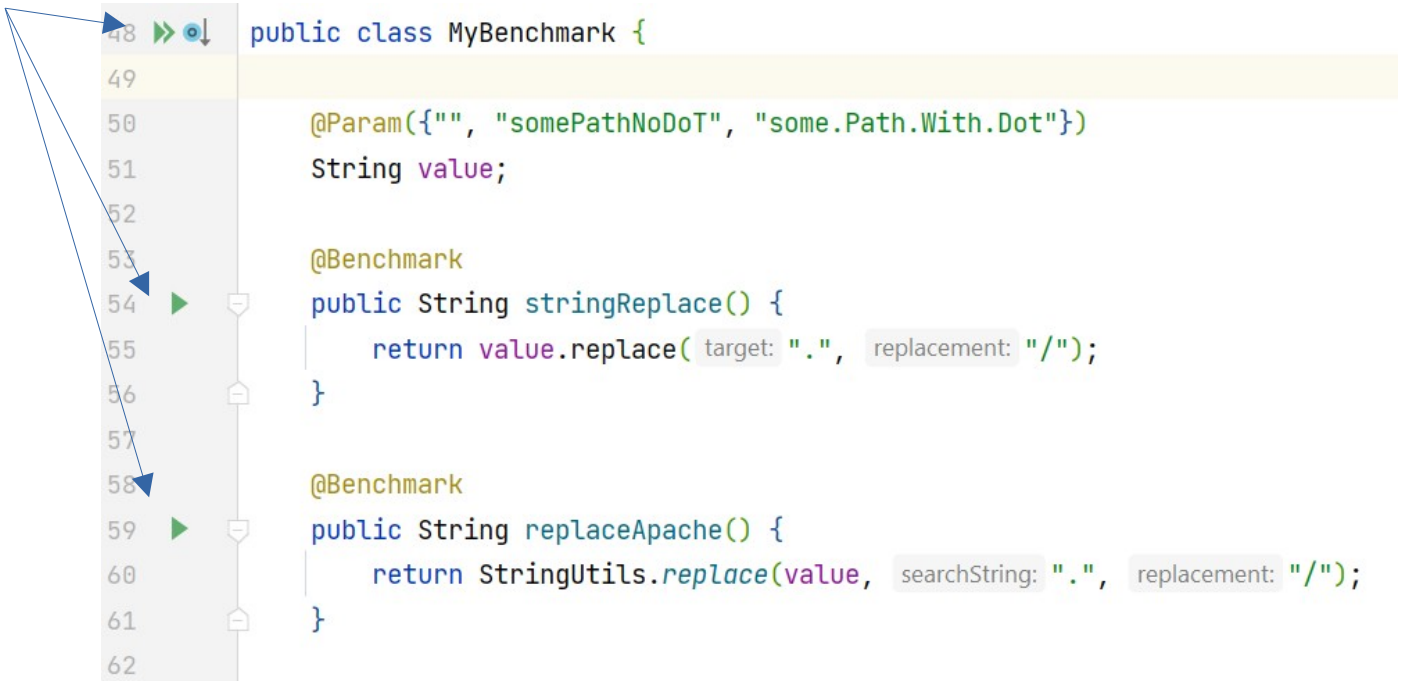
```xml
<properties>
    <jmh.version>1.33</jmh.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.openjdk.jmh</groupId>
        <artifactId>jmh-core</artifactId>
        <version>${jmh.version}</version>
    </dependency>
    <dependency>
        <groupId>org.openjdk.jmh</groupId>
        <artifactId>jmh-generator-annprocess</artifactId>
        <version>${jmh.version}</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

2. Přidat svůj benchmark a vytvořit následující třídu s metodou main, která ho spustí:

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

Jak na to? III.

- Pro spuštění benchmarku není nutné vytvářet třídu s main metodou, alternativně je možné nainstalovat do IntelliJ Idea plugin “JMH Java Microbenchmark Harness” a pak bude u každého benchmarku tlačítko pro spuštění:



Benchmark

```
@BenchmarkMode(Mode.AverageTime)
@Fork(1)
@State(Scope.Thread)
@Warmup(iterations = 5, time = 1)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@Measurement(iterations = 10, time = 1)
public class MyBenchmark {

    @Param({"", "somePathNoDoT", "some.Path.With.Dot"})
    String value;

    @Benchmark
    public String stringReplace() {
        return value.replace(".", "/");
    }

    @Benchmark
    public String replaceApache() {
        return StringUtils.replace(value, ".", "/");
    }

}
```

Výchozí hodnota je Mode.Throughput, u které ve výsledném reportu bude počet operací (volání) benchmark operace za časovou jednotku (ns, ms, s). S AverageTime tam bude jak dlouho v průměru trvalo volání benchmark operace.

Parametry, se kterými se bude volat benchmark operace. Ideální když chceme otestovat chování nějaké metody s různými typy vstupů

Jediná povinná anotace

Pokud je definována @Setup metoda, pak se tato metoda zavolá před @Benchmark.
Pokud je definována @Teardown metoda, pak se zavolá po @Benchmark.
Poznámka: Délka běhu těchto metod není započítána do běhu @Benchmark metody.
http://tutorials.jenkov.com/java-performance/jmh.html#state-setup-and-teardown

Benchmark

```java
@BenchmarkMode(Mode.AverageTime)
@Fork(1)
@State(Scope.Thread)
@Warmup(iterations = 5, time = 1)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@Measurement(iterations = 10, time = 1)
public class MyBenchmark {

    @Param({"", "somePathNoDoT", "some.Path.With.Dot"})
    String value;

    @Benchmark
    public String stringReplace() {
        return value.replace(".", "/");
    }

    @Benchmark
    public String replaceApache() {
        return StringUtils.replace(value, ".", "/");
    }

}
```

Kolikrát bude celá operace běžet je definováno:

```
fork * (warmupIterations + measurementIterations)
```

OutputTimeUnit definuje časovou jednotku v jaké bude výstup v reportu. Pokud se neuvede, pak JMH použije vhodnou časovou jednotku podle délky operace.

Easy, right? … not so fast!

- JVM (konkrétně JIT kompilátor) je velice efektivní a velice lehce se můžete dostat do situace, kdy benchmark napíšete špatně:

```
@Benchmark
public void doNothing() {
}

@Benchmark
public void objectCreation() {
    new Object();
}
```

Hmmm, opravdu vytváření objektů v Javě nic nestojí?

```
Benchmark                        Mode  Cnt  Score   Error  Units
MyBenchmark.doNothing            avgt   10  0.434 ± 0.013  ns/op
MyBenchmark.objectCreation       avgt   10  0.432 ± 0.012  ns/op
```

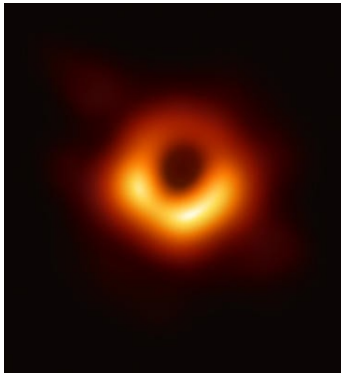
Enter Blackhole

```java
@Benchmark
public void doNothing() {
}

@Benchmark
public void objectCreation() {
    new Object();
}

@Benchmark
public Object pillarsOfCreation() {
    return new Object();
}

@Benchmark
public void blackHole(Blackhole blackhole) {
    blackhole.consume(new Object());
}
```

V čem byl problém? Došlo k tzv. “dead code elimination”. Tento problém se dá vyřešit buď tak, že se z benchmark metody takový objekt vrátí, nebo se pošle do metody consume() třídy Blackhole.

KAŽDÝ objekt, který se v benchmarku vytvoří se buď musí použít jako vstup do nějaké metody, nebo se z benchmarku musí vrátit, nebo se musí vložit do blackhole.consume()

```
Benchmark                        Mode  Cnt  Score   Error  Units
MyBenchmark.doNothing            avgt   10  0.434 ± 0.013  ns/op
MyBenchmark.objectCreation       avgt   10  0.432 ± 0.012  ns/op
MyBenchmark.pillarsOfCreation    avgt   10  3.606 ± 0.284  ns/op
MyBenchmark.blackHole            avgt   10  3.574 ± 0.149  ns/op
```



Loop & accumulation

```
private int xsLength = 100;

private int [] xs;

@Setup
public void setup() {
    xs = new int[xsLength];
    for (int i = 0; i < xsLength; i++) {
        xs[i] = i;
    }
}

public int compute(int a) {
    return a * 100;
}

@Benchmark
public int computeWrong() {
    int acc = 0;
    for (int x : xs) {
        acc += compute(x);
    }
    return acc;
}

@Benchmark
public void computeRight(Blackhole blackhole) {
    for (int x : xs) {
        blackhole.consume(compute(x));
    }
}
```

Když testovaná metoda vrací int, tak nás může napadnout, že v benchmarku výsledek jednotlivých volání sečteme a výsledek vrátíme. Ale ouha: dojde k unrolling loop optimalizaci a výsledek bude špatně.

Daleko bezpečnější (a ve finále i jednodušší) je použít Blackhole objekt a nesnažit se přechytračit kompilátor ;-)

https://www.infoworld.com/article/2078635/jvm-performance-optimization-part-2-compilers.html?page=2

```
Benchmark                  Mode  Cnt    Score    Error  Units
MyBenchmark.computeWrong   avgt   10   31.165 ±  1.721  ns/op
MyBenchmark.computeRight   avgt   10  254.321 ± 11.913  ns/op
```

Constant folding

```
private double x = Math.PI;

private final double finalX = Math.PI;

@Benchmark
public double computeWrong() {
    return Math.log(finalX);
}

@Benchmark
public double computeRight() {
    return Math.log(x);
}

@Benchmark
public int computeWrong2() {
    int a = 1;
    int b = 2;
    int sum = a + b;

    return sum;
}
```

Dřív byl v JMH problém s tím, že když je v benchmarku používaná primitivní konstanta, tak JVM provede optimalizaci zvanou “constant folding” a protože je výsledek predikovatelný, tak JVM ani nemusí testovanou metodu zavolat.

V aktuální době se mi tuto situaci nepodařilo zreplikovat :-(… tak nevím jestli si na to stále dávat pozor.

Poznámka: Týká se to nejenom atributů, ale i proměnných uvnitř benchmark metody. Pokud JVM dokáže lehce odhalit, že jejich stav je neměnný, pak může tuto optimalizaci provést:

https://blog.avenuecode.com/java-microbenchmarks-with-jmh-part-3

Microbenchmarking tips

- JMH samples:
  - https://github.com/openjdk/jmh/tree/master/jmh-samples/src/main/java/org/openjdk/jmh/samples
- Microbenchmarking tips:
  - https://jeeconf.com/program/micro-optimizations-in-java/
  - https://raygun.com/blog/java-performance-optimization-tips/
- But beware!!!
  - Premature optimization is the root of all evil
    - Donald Knuth

Děkuji za pozornost