Microbenchmarking Using JMH

Josh Cummings
PRINCIPAL SOFTWARE ENGINEER

@jzheaux tech.joshuacummings.com

JVM

very complex

A piece of software that implements the Java Virtual Machine Specification and is mainly intended to convert Java bytecode into machine code and execute it.

very quickly

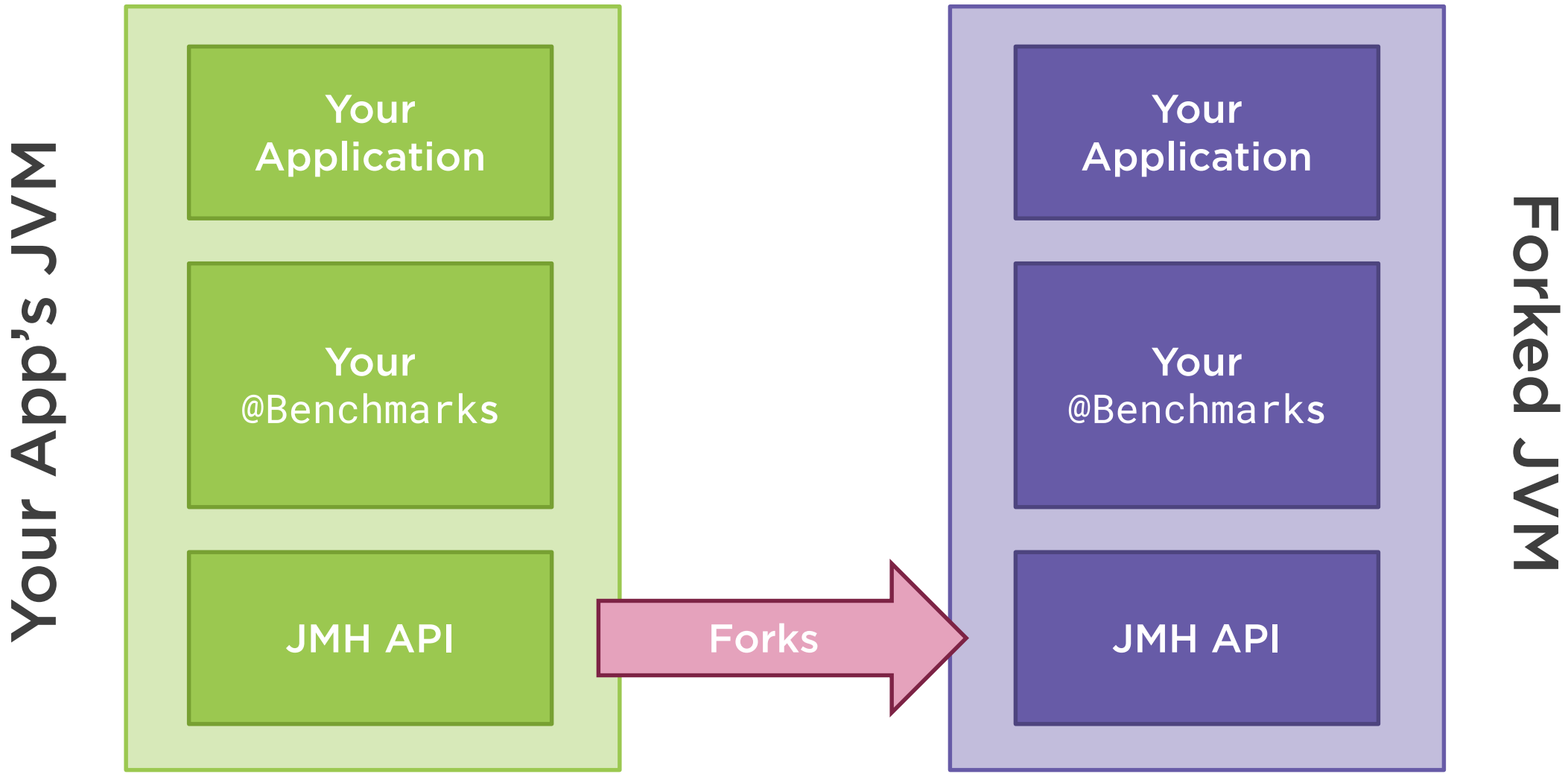
The Java Microbenchmark Harness

JMH is an OpenJDK project for writing and running benchmarks

First shipment of code was in 2013

JEP 230 (not yet complete) is to roll Java 9's own benchmarks into the Java 9 project

JMH Architectural Overview



```
Running com.joshcummings.cats.StartupTest
# JMH version: 1.19
# VM version: JDK 1.8.0_131, VM 25.131-b11
# VM invoker: /usr/lib/jvm/java-8-openjdk-
amd64/jre/bin/java
# VM options: -Xmx256M
…

# Run progress: 0.00% complete, ETA 00:01:20
# Fork: 1 of 1
# Warmup Iteration   1: 0.021 us/op
# Warmup Iteration   2: 0.018 us/op
# Warmup Iteration   3: 0.012 us/op
…

Iteration   1: 0.033 us/op
Iteration   2: 0.041 us/op
Iteration   3: 0.049 us/op
…

# Run complete. Total time: 00:01:22
Benchmark  Mode  Cnt  Score   Error  Units
baseline   avgt   20  0.013 ± 0.001  us/op
startup    avgt   20  0.037 ± 0.004  us/op
```

- JMH Begins
- New forked JVM beings a Trial
- Warmup Iterations
- Measurement Iterations
- Report

“In God we trust; everyone else bring data.”

Michael Bloomberg

```
mvn archetype:generate \
      -DinteractiveMode=false \
      -DarchetypeGroupId=org.openjdk.jmh \
      -DarchetypeArtifactId=jmh-java-benchmark-archetype \
      -DgroupId=your.group \
      -DartifactId=your-apps-benchmarks \
      -Dversion=0.0.1-SNAPSHOT
```

Our First Benchmark

Assumes maven

Creates a project separate from the code we are testing

Easy for first-time setup

Baseline Benchmarks

Provide a sanity check and a point of reference

Provide a way to isolate the work we care about from boilerplate

@BenchmarkMode

`Mode.Throughput`	**Measure ops/sec**
`Mode.AverageTime`	**Measure secs/op**
`Mode.SampleTime`	**Verbose** `AverageTime`
`Mode.SingleShotTime`	**Run once** `AverageTime`
`Mode.All`	**Run all modes**

```
@BenchmarkMode
    (Mode.AverageTime)
public class CatBenchmarks {

    @Benchmark
    @BenchmarkMode
        (Mode.Throughput)
    public int firstTest() {
        // will measure ops/sec
    }

    @Benchmark
    public int secondTest() {
        // will measure secs/op
    }
}
```

- Specify class-level setting that overrides JMH-level setting
- Specify method-level setting that overrides class-level setting
- Methods inherit class-level setting

```
@OutputTimeUnit(TimeUnit.NANOSECONDS)  // very tricky
@OutputTimeUnit(TimeUnit.MICROSECONDS) // still very tricky
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@OutputTimeUnit(TimeUnit.SECONDS)
@OutputTimeUnit(TimeUnit.DAYS)         // I'm sorry for you
```

Use Units with the Appropriate Precision

Lack of precision => Information loss

@State

Scope.Benchmark	Isolate to benchmark
Scope.Thread	Isolate to thread
Scope.Group	Isolate to thread group

```
@State(Scope.Benchmark)
public static class MyState {
    @Param({“1”, “2”, “3”, “4”}) Integer numberOfCats;

    @Param({“Fluffkins”, “Sgt. Snuggles”}) String catNames;
}
```

Use `@Param` to prevent test duplication

Coerces to appropriate primitive type or enum

Executes on the product of all `@Param` values

Helps isolate and reuse state across tests (D.R.Y.)

@Setup/@Teardown

Level.Trial	Invoke per trial
Level.Iteration	Invoke per iteration
Level.Invocation	Invoke per invocation

```
@Benchmark
public void myBenchmark(CatServiceState state,
                        Blackhole bh) {

   // can't return both, so send them to a black hole

   bh.consume(state.service.add(cat));
   bh.consume(state.service.remove(cat));

}
```

Use `Blackhole` to prevent dead code elimination

Avoid boilerplate of returning a wrapper object

Blackhole well-defended against JVM optimizations

```
private static final String
GROUP_NAME = “myGroup”;

@Benchmark
@Group(GROUP_NAME)
@GroupThreads(3)
public void reads() {

}

@Benchmark
@Group(GROUP_NAME)
@GroupThreads(1)
public void writes() {

}
```

- Use a constant to configure the group name
- Annotate each method that should be part of the same trial
- Annotate each method with the number of threads from the group it should use

```java
@Fork(5)
public class MyBenchmarks {
    // fork the JVM 5 times, running the trials once in each

}
```

Use `@Fork` to run the tests multiple times

Further iron out OS jitter and anomalies

Use 1 to run the trials once

Use 0 to not fork the JVM at all (not recommended)

```
@Measurement(iterations = 10, time = 10)
@Warmup(iterations = 10, time = 10)
public class MyBenchmarks {
    // for each trial, do 10 warmup and measurement
iterations instead of the default 20

    // for each iteration, run for 10 seconds instead of the
default 1

}
```

Use `@Measurement` to adjust sample collection

Longer iterations for longer invocations; should have many invocations per iteration

Shorter iterations for shorter invocations; same accuracy in less time

```
@Benchmark
@OperationsPerInvocation(numberOfCats)
public int countCats() {
    for ( int i = 0; i < numberOfCats; i++ ) {
        // we want to measure each of these as invocations
    }
}
```

Use `@OperationsPerInvocation` with loops

Avoid loops where possible in benchmarks

Measure inside of for loop as separate invocations

```java
@Benchmark
@Threads(4) // defaults to # of available processors
public int countCats() {
}
```

Use `@Threads` to adjust concurrency level

Defaults to `Runtime.getRuntime.availableProcessors();`

Use 1 for single-threaded scenarios

Use more than number of processors when testing I/O wait and the like

JMH

Is Java's framework for building reliable benchmarks

Assists in quieting certain JVM optimizations

- Though you still need to write your benchmarks with the JVM in mind

Enables multi-dimensional exploration of benchmark performance

- Sample size
- Measurement units
- Thread counts
- State settings