# An Introduction to Microbenchmarking with JMH

Tom Tresansky

February 2016

I plan to *introduce* a tool called JMH and show how to use it. This tool is probably the *BEST POSSIBLE WAY* at the present time to answer questions like are these lines of Java code faster than these other lines.

I'll say up front that understanding this tool is difficult, and I am by no means an expert on either it or the complex and cryptic internal workings of the JVM. But I know enough to be dangerous and you can too.

At the end, I'll some some interesting results regarding some Java 8 code idioms and even UCMS.

## What's a Microbenchmark?

- Measure the performance of a particular piece of code, NOT an entire application stack
  - "Which is faster, X or Y?"
- Scope of what is benchmarked is just a few lines or method calls
- Examples
  - Comparing logically equivalent pieces of code
  - Comparing different implementations of an algorithm
  - Determining the time added (versus some baseline measurement) by running some piece of code

This presentation is entirely focused on measuring speed, not space usage – which is also a valid potential goal for a microbenchmark.

## Naïve Stopwatch Benchmarking

1. Start timer before code runs
2. Stop timer after code runs
3. Subtract

How do we benchmark – use a stopwatch!

Simple enough in theory.

**Comparison against a baseline**: How do we know whether performance is "good"? What is considered "good"? The baseline may be determined by customer requirements or you might be just looking for the best relative performance in a specific scenario among a set of benchmark candidates.

Remember we're *micro* benchmarking, so the code is going to run really, really fast, so in order to get precise and meaningful measurements we'll have to consider some sources of error.

## Some (Controllable) External Sources of Error

- Hardware differences
- Accuracy of the system clock
- System load
- OS differences (version and point release, registry differences)
- OS process scheduling
- JVM differences (version and point release)
- JVM startup flags
- Garbage collection
- The effects of the JIT

When stopwatch benchmarking, there are many external sources of error to be aware of that will skew your measurements. Green -> Red is roughly more to less controllable.

These are all external to the execution of the code you write. Use an identical, machine (virtual machine restored from backup file) which is quiet except for your tests. Run JVM as a high-priority/real-time process.

Garbage collection can be disabled (remember – MICRO benchmarks) to make things simpler.

Remember to disable any sort of speed-stepping or power performance options prior to running too.

Accuracy of the system clock (might not be accurate enough for code that runs extremely fast) – but can always run multiple times and divide by # of rep.

The effects of the JIT are very difficult to control for as we'll see in the first demo.

The effects of the JIT

Code will be optimized as it runs more frequently

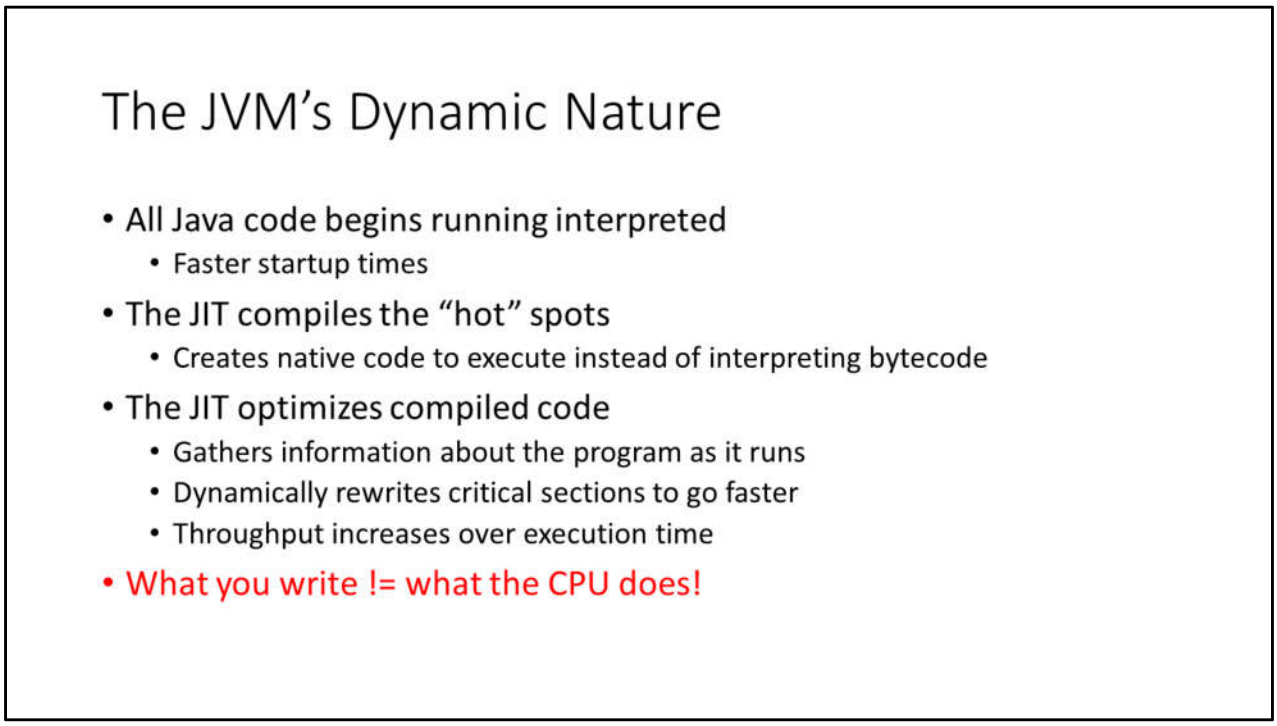
The average time for a single run is affected

http://java.sun.com/javase/6/docs/api/java/lang/System.html#nanoTime()

more accurate than System.currentTimeMillis()

It should be noted that System.nanoTime() is not *guaranteed* to be more accurate than System.currentTimeMillis(). It is only guaranteed to be at least as accurate. It usually is substantially more accurate, however.

http://stackoverflow.com/questions/504103/how-do-i-write-a-correct-micro-benchmark-in-java#comment8131770_513259

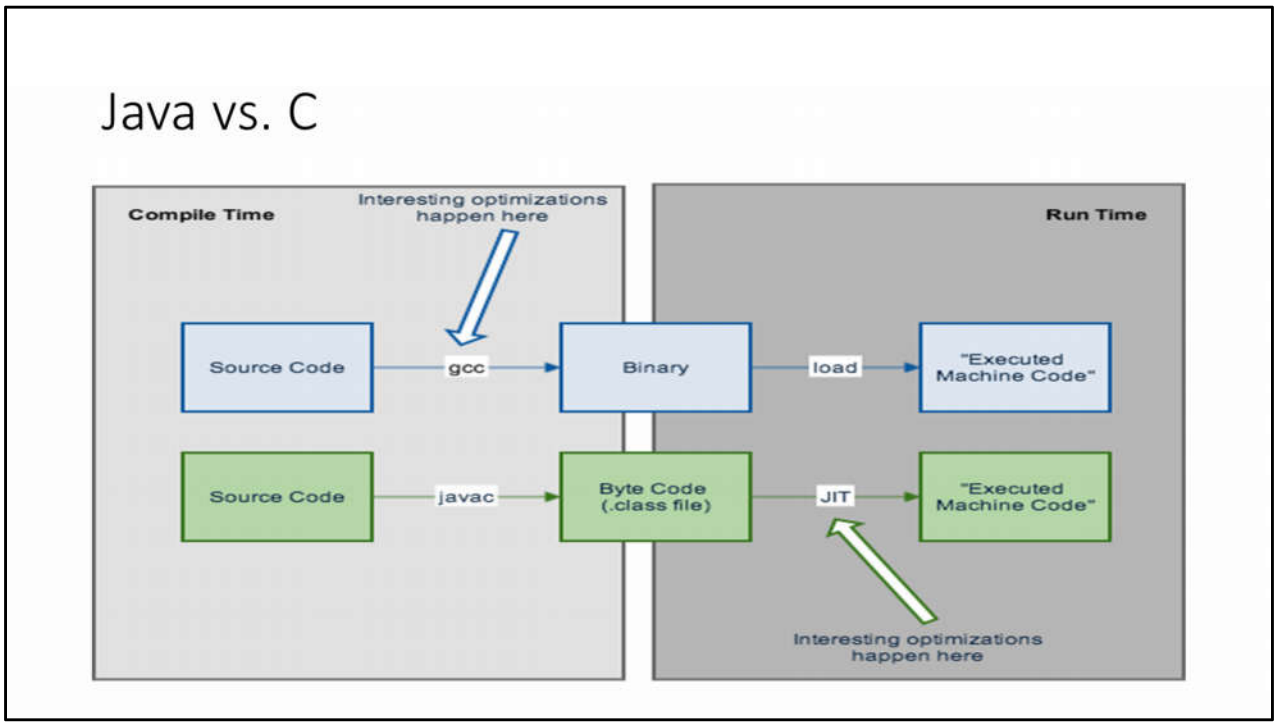



JVM is an Adaptive VM

Each method is executed in interpreted mode at first. The Java interpreter counts how many times a method is invoked and requests that it should be JIT-compiled.

This happens after a method has been called 10.000 times in server mode and 1000 times in client mode (see an article about tiered compilation by Dr. Cliff Click for more details on how the process works and the Oracle documentation on HotSpot performance options).

As a rule of thumb, JIT'd code is 10x faster than interpreter




If we were benchmarking a non-VM language that compiled right to binary, like C, it would be an easier proposition – the optimizations would take place during compilation, *NOT* runtime. With Java running on the dynamic JVM, this isn't the case.

## Mitigating the Effects of the JIT

- Run a Warmup Phase
- Run Many Times and Average the Results (Minimize Error %)
- Be aware when the GC or JIT are affecting your code:
  - -XX:+PrintCompilation, -verbose:gc
- Do not take any code path for the first time in the "timing phase"
  - Compilier might "optimistically" assume a path will never/rarely be taken and anti-optimize it
- Set various arcane JVM flags to be control the JIT compiler
  - -Xbatch, -Xcomp, -XX:ClCompilerCount = 1, etc.

Prevent it from tainting our measurement results – making “slower” algoritms/code samples look faster. Want an even playing field for all our code.

Run the benchmarked code often enough before the actual measurement starts to ensure that all benchmarked code has been JIT-compiled beforehand. This can easily be verified by providing -XX:+PrintCompilation. You should not see any JIT-compiler activity after the warmup phase.

Keeping testing until run-times stabilize
Not allowing warmup is a common mistake
most desktop/server apps expect:
●Reboots are minutes long and days apart
●Steady-state throughput after warmup is key
●So a benchmark that ends in <10sec
probably does not measure anything interesting

Mitigating the Errors of Stopwatch bench marking

prerunning the tested code to mitigate "warm-up" effects
run a "warm-up" cycle of *the same code*

don't load new classes out of the warn-up phase

long runs minimize error percentage

Make sure you run it for long enough to be able to measure the results in seconds or (better) tens of seconds

many runs minimize error percentage

Timing a single lap versus averaging over multiple laps

At least be aware when the GC or JIT are affecting your code:

-XX:+PrintCompilation, -verbose:gc

Deoptimization/recompilation effects must be mitigated

"Do not take any code path for the first time in the "timing phase" compilier might "optimistically" assume a path will never/rarely be taken and anti-optimize it

GC: Either no (or trivial) allocation, or use
verbose:gc
to make sure you hit steady-state

Use -Xbatch to serialize the compiler with the application
"Disables background compilation. Typically, the Java VM compiles the method as a background task, running the method in interpreter mode until the background compilation is finished. The -Xbatch flag disables background compilation so that compilation of all methods proceeds as a foreground task until completed."

http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/java.html

-XcompForces compilation of methods on first invocation. By default, the Client VM (-client) performs 1,000 interpreted method invocations and the Server VM (-server) performs 10,000 interpreted method invocations to gather information for efficient compilation. Specifying the -Xcomp option disables interpreted method invocations to increase compilation performance at the expense of efficiency.

You can also change the number of interpreted method invocations before compilation using the -XX:CompileThreshold option.

Alternately, setting -XX:CICompilerCount=1 to prevent the compiler from running in parallel with itself.

# Demo – SimpleBenchmarker

Let’s see a naïve and flawed implementation of a benchmarker.

gradlew simple-all –daemon

C:\Users\ttresans\Projects\JMH-Presentation\demos\simplebenchmarker\build\reports\simplebenchmarker

# The Moral of SimpleBenchmarker

- Writing simple benchmarks which yield meaningful results is very difficult
- The cause of much of this difficulty is the JIT's optimizations
  - Method Dispatch Optimization/Deoptimization
  - Dead-Code Elimination
  - Constant Folding
  - Method Inlining

## Method Dispatch Optimization/Deoptimization

- Types of Method Dispatch [5]
  - Monomorphic = 1 possibility
  - Bimorphic = 2 possibilities
  - Megamorphic = many possibilities
- All (non-final) methods begin as Megamorphic
  - All methods are overrideable by default in Java = each method call needs a vtable to lookup actual method implementation
- Monomorphic faster than Bimorphic faster than Megamorphic
- JIT profiles calls
  - If *most* calls are to a single implementation, assume monomorphic and optimize

```
Collection<String> names;

if (<RUNTIME_CONDITION>) {
  names = new <SOMETHING>();
} else {
  names = new <OTHERTHING>();
}

// What is the type of names?
// Which class's add()
// do I call?
names.add("Bob");
```

How does the JIT determine whether to call the add method on ArrayList, HashSet or TreeMap? Maybe the only class you've imported and loaded is ArrayList = then add will only ever run ArrayLists's add method, even though when calling add, the JIT can't predict what type of instance names will actually refer to.

Probably the most arcane, but also the most difficult to predict and control – might depend upon what classes are loaded at runtime and whether alternate implementations are ever actually called…

Methods are by default virtual (overridable) in Java it has to lookup the correct method in a table, called a vtable, for every invocation. This is pretty slow, so optimizing compilers are always trying to reduce the lookup costs involved. One approach we mentioned earlier is inlining, which is great if your compiler can prove that only one method can be called at a given callsite. This is called a **monomorphic** callsite

The key to what the JIT does is that if *DOESN'T* have to *PROVE* monomorphism, it can assume it, and if the code later changes how it behaves, it will reverse its previous optimization.

Monomorphic callsites aren't the only case we want to optimise for though. Many callsites

are what is termed **bimorphic** - there are two methods which can be invoked. You can still inline bimorphic callsites by using your guard code to check which implementation to call and then jumping to it. This is still cheaper than a full method invocation. It's also possible to optimise this case using an inline cache. An inline cache doesn't actually inline the method body into a callsite but it has a specialised jump table which acts like a cache on a full vtable lookup.

There is a big difference between the fastest and slowest types of method invocation.

In practice the addition or removal of the final keyword doesn't really impact performance – you can go to that link to see demos proving that – as the JIT can quickly determine if a non-final method is mostly monomorphic and optimize as if it was, but, if you then go and refactor your hierarchy things can start to slow down.

## Dead-Code Elimination

- JIT understands control flow
- If result is not used, it is not needed
  - So why waste cycles computing it?
- Not just a static analysis, but a dynamic runtime analysis
  - If I call compute with s = 0

```
final String name = "Tom";
int result = 0;
if (name.length() > 10) {
      result += name.length();
}
```

```
public void compute(int s) {
  int sum = s;
  for (int i=0; i < 5; i++) {
    sum += i;
  }
  if (sum >= 100)
    System.out.println(i);
}
```

In certain circumstances the JIT-compiler may be able to detect that the benchmark does not do anything and eliminates large parts or even the whole benchmark code. The performance looks astonishing but unfortunately the benchmark developer has just been fooled by the JIT-compiler.

This are some stupidly simple examples, but the JIT is very patient – and can detect situations at runtime – like the second example but even more complex – where the code *BECOMES DEAD AT RUNTIME*. That second example, if called with s = 0; does nothing meaningful and can be skipped entirely. The JIT will eventually figure this out.

## Constant Folding

- The compiler precalculates constant expressions at compile time
  - Can never change, so why spend (runtime) cycles computing it?
- Typically simple literals, but could also be variables whose values are never modified
- Doesn't require final to fold

```
// code
static final int a = 2;
int b = 30 * a;

// folding would create
int b = 60;
```

Constant folding is the process of simplifying constant expressions at compile time. Terms in constant expressions are typically simple literals, such as the integer 2, but can also be variables whose values are never modified, or variables explicitly marked as constant.

A major issue with microbenchmarks, because of how they are usually (simply) structured.

## Method Inlining

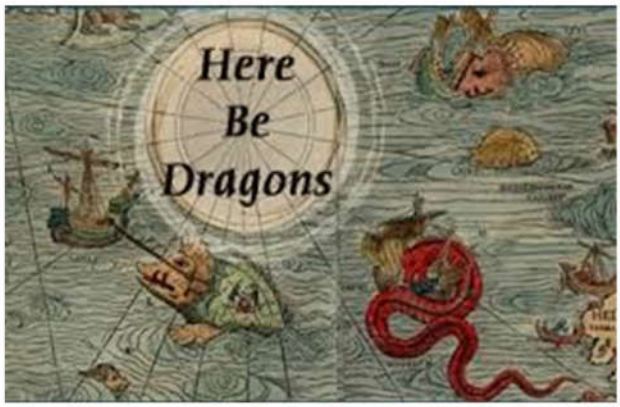
- Method calls are (relatively) expensive compared to many operations on variables
  - Why spend cycles pushing and popping method calls – just do the operation
- Method Inlining replaces a method call with the body of the method

```
public int call1(int a) {
  return 10 * a;
}
public int call2(int a) {
  return a / 2;
}
// written
int result = call2(call1(5));
// is executed as if it were
int result = (10 * 5) / 2;
```

And you can see here how applying optimization 1 might open the door for optimization 2 – now that expression can be constant folded down to just 25. And maybe the call1() implementation was actually an @Overriden implementation of an interface method, which was determined to be monomorphic by the JIT profiling your application at runtime, which happened first to enable the inlining…

## There are Many Other Optimizations

- An outdated (2009) list of > 70 optimizations performed by the JIT
  - https://wiki.openjdk.java.net/display/HotSpot/PerformanceTacticIndex




Loop unrolling.
OSR can make benchmark code (typically a loop of calls to the profiled method) perform different to how it would in real life.
“False sharing” for multithreaded tests.
GC effects:
Escape analysis may succeed in a benchmark where it would fail in real code.
A buildup to a GC might be ignored in a run or a collection may be included.
https://dzone.com/articles/false-sharing

## How to Write a Perfect Microbenchmark

First, write a good optimizing JIT. Meet the people who have written other good, optimizing JITs (they're easy to find, because not too many good, optimizing JITs exist!). Have them over to dinner every week for a year, and swap stories of performance tricks on how to run Java bytecode as fast as possible. Study the preconditions of each of the JIT's optimization techniques until you can effectively predict each one. Read the hundreds of papers on optimizing the execution of Java code. Write a few papers of your own, get them published and peer reviewed. Build a standardized, bare-bones test system running your own custom non-timesharing OS with a very precise clock…

So if all that seems like a lot of things to keep in mind while writing your benchmarks, it's because it *is*! Writing meaningful microbenchmarks by hand is very, very hard.

By now you might think there is no way to write a correct microbenchmark on the JVM without being an engineer developing the JIT.

So you can follow this procedure…or you can use a tool which aims to mitigate all those troubles…jmh

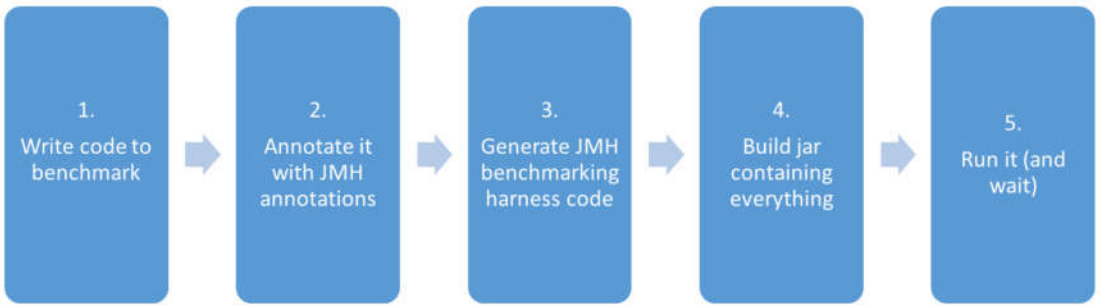
## Enter JMH

- http://openjdk.java.net/projects/code-tools/jmh/
- Solid foundation for writing and running benchmarks whose results are "not always erroneous due to unwanted VM optimizations"
  - Does not prevent all pitfalls, but can at least mitigate MOST of them
  - Control the JIT to get more meaningful results
- OpenJDK project
  - http://mvnrepository.com/artifact/org.openjdk.jmh/jmh-core
  - Active development (1.11.2 released October 2015)
  - Developed by same guys in Oracle who maintain the JIT

**"Its distinctive advantage** over other frameworks is that it is developed by the same guys in Oracle who implement the JIT. In particular I want to mention Aleksey Shipilev and his brilliant blog. **JMH is likely to be in sync with the latest Oracle JRE changes, which makes its results very reliable. "**

Other than micro JMH is a general-purpose benchmarking harness also possibly useful for larger and concurrent benchmarks, too. Emphasis in this talk is on using it for *MICRO* benchmarks though, different profiling tools are probably more useful for measuring larger parts of an application, and multithreaded scenarios are supported, but quickly get very complex. JMH does support all sorts of fun multithreaded scenarios though.

## JMH – How it Works




In all cases, the key to using JMH is enabling the annotation- or bytecode-processors to generate the synthetic benchmark code and properly packaging the jar.

So steps 3 and 4 there.

## JMH – Setup

- JMH is officially setup as a Maven project
  - "The recommended way to run a JMH benchmark is to use Maven to setup a standalone project that depends on the jar files of your application"
- &#$@ Maven – use Gradle instead
  - https://github.com/melix/jmh-gradle-plugin
  - "The jmh plugin makes it easy to test existing sources without having to create a separate project for this."
- Possible to run within an existing project, within an IDE
  - Not recommended

The separate project business is JMH is trying to ensure that the benchmarks are **correctly initialized and produce reliable results**.

Because Gradle is several hundred times better than JMH, I've found some resources to run JMH via Gradle and built my demos here around it. I've also got a standalone runner project you can use to quickly setup benchmarks and get results.

A large part of the learning curve with JMH is project setup, and if you're trying to manage it by hand, or with a POS tool like Maven, your time to iterate the setup-test-evaluate results-reseupt-retest will be very large. The project setup I'll show you, using Gradle, plus Visual Studio Code, plus some python charting library makes that cycle really short. Piecing this toolchain together from various resources online took some work.

It is possible to run benchmarks from within an existing project, and I'll show how to do that later and even from within an IDE, however setup is more complex and the results are less reliable.

## JMH – Setup (2)

- Best to keep code and benchmarks separate
- In Gradle – split into 2 separate source trees
  - /src/main/java
    - Application code
  - /src/jmh/java
    - Benchmark code
- Or build a benchmark project which depends on your application jar(s) using Gradle dependencies
  - Useful for profiling UCMS client/shared code

When dealing with large projects, it is customary to keep the benchmarks in a separate subproject, which then depends on the tested modules via the usual build dependencies.

# Demos – Hello JMH and jmhrunner

Best way to learn JMH is through the JMH-Samples which are available on the JMH website. I've imported them all into this presentation project so they can be quickly, properly and reliably run through Gradle.

There's a lot of other JMH annotation, and I'm just going to show a few – but reading through them is certainly the place to start learning JMH.

@Benchmark
JMH will produce the generated benchmark code for this method during compilation, register this method as the benchmark in the benchmark list, read out the default values from the annotations, and generally prepare the environment for the benchmark to run.

In general one might think about @Benchmark methods * as the benchmark "payload", the things we want to measure. The * surrounding infrastructure is provided by the harness itself..

Must be public

Show JMHSample_01_HelloWorld

So here's the very basics – all *YOU* have to do is annotate a method with @Benchmark … and then figure out how to properly package and run it. Fortunately, Gradle.

See the "Referenced Librarys" – here is the JMH code – the annotation processor and the core jars which we need to reference – don't worry, Gradle properly sets up the references for just your benchmark code, NOT your application code.

You could just run it via this main setup, as JMH does provide a programmatic way to launch it…but that comes with a lot of caveats and produces definitely very skewed results.

From / "gradlew :jmh-samples:plot -PjmhClass=JMHSample_01_HelloWorld --daemon"

Output in /jmh-samples/build/reports/jmh

Results of running JMH are written to /build/reports – which is the default Gradle report output directory

Show human.txt
-Info looks similar to SimpleBenchmarker, that's no accident, run X warmups, then Y tests, limit each iteration by time
-One other thing useful to note in the output is the ETA it prints prior to each benchmark, so if you watch the file while JMH is running during a long-running process it will attempt to guess at when its going to finish – useful if you're comparing lots of benchmarks

Show results.csv
-CPU results per-test

Show jmhrunner setup src/main/java vs src/jmh/java
Here's a *slightly* more complex example, using that same Calculator distance method you're all sick of by now. At least it actually tests something.

From /jmhrunner "gradlew plot --daemon"

Output in /jmhrunner/build/reports/jmh

Show human.txt
Show plot.png

Show /jmhrunner/build.gradle
Focus on jmh block {} – explain defaults
You can control just about everything about how the benchmark runs from Gradle – or provide

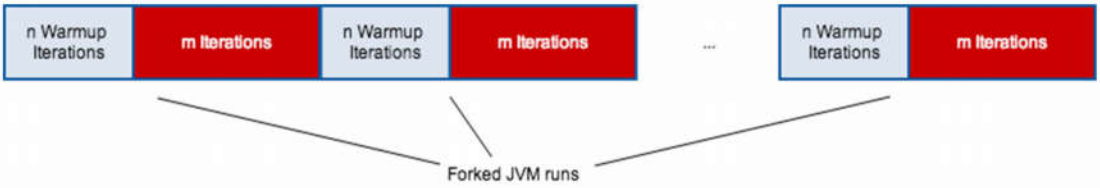
## @Benchmark

- Must be public
- Contains the “payload” to benchmark
- For each @Benchmark method JMH will
  1. produce the generated benchmark code for this method during compilation
  2. register this method as a benchmark in the benchmark list
  3. read out the default values from the annotations
  4. prepare the environment for the benchmark to run

Benchmark demarcates the benchmark payload, and JMH treats it specifically as the wrapper which contains the benchmark code.

What all this means is that you get a separate class, with a main method, contained in the jmh-produces jar for each @Benchmark, which it can invoke separately, which contains everything needed to measure your code.

## JMH – Defaults



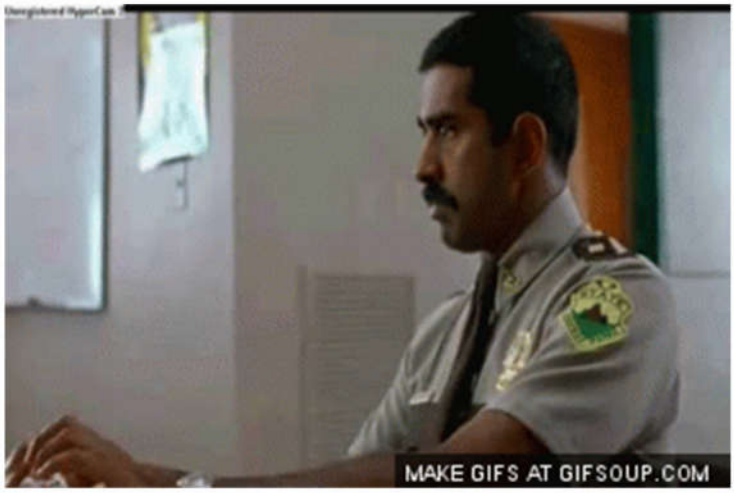

- Each benchmark run 10 warm-up rounds (max 3 second each)
- Then 20 measurement rounds (max 5 second each)
- Launches a new JVM 5 times for running each benchmark
- Command line flags can customize these settings
  - https://github.com/melix/jmh-gradle-plugin#configuration-options
- Total time spent running = startup + (forks * ((warmupIterations * 3s) + (iterations * timeOnIteration))) + calculation
  - More repetitions = greater accuracy

The default max of 1 second – shows the emphasis on "micro"-benchmarks. It's expected that your benchmark code will execute many, many times in the course of a single second.

More repetitions = greater accuraccy

So these can be slow, you probably want to tweak the defaults to tone it down – especially at first, if you're serious about optimizing, you focus on large differences, right – should be easy to see at much less precise levels of measurement which will let you spend less time waiting for your benchmarks to run. Then you can iterate and enhance.

## Limits of Resolution




Obviously, if you start asking for NS precision on a process which takes several milliseconds, no matter how many runs you do, you are not going to get more precise answers.

## JMH Output

- Configuration Info
  - JMH, JVM version
  - JVM Configuration
  - JMH Configuration
- Profiling Results
  - Operations / millisecond (default output mode)
  - Computed statistics (min, avg, max, stdev, confidence interval)
- What's missing?
  - OS
  - Hardware
  - System Load

Aim of JMH that if you run twice on the same machine – without making major changes to that machine's software or hardware, you can meaningfully compare output so it documents what the JVM it ran on was and what the configuration was.

The output shows the result of each run – the warmups and the test iterations for each @Benchmark annotated method.

Benchmark mode defaults to throughput THRPT in output. Available modes are: [Throughput/thrpt, AverageTime/avgt, SampleTime/sample, SingleShotTime/ss, All/all] – thrpt seems to be the easiest to compare intuitively which is why it's the default – the difference between 300 and 400 M is more meaningful to us than the different between .000003 and .000004 seconds, or whatever

## JMH Configuration Annotations

- Annotations exist to configure how JMH runs each @Benchmark
  - @Measurement
  - @Warmup
  - @Fork
  - @BenchmarkMode
  - @OutputTimeUnit
  - Others…
- All options available through jmh configuration block
- **Important – jmh gradle plugin overrides these! Must set defaults through build.gradle file!**

So you can see throughout the jmh samples and online, a number of configuration annotations exist which let you configure how to run jmh.

Can mix and match differently configured benchmarks within the same class – for instance, in order to see the Average time plus the throughput plus the "single shot time".

The gradle plugin takes the stance that mixing and matching is silly and that you should have a single place to configure these things – the build.gradle file – for all your @Benchmarks. Or maybe that's a bug with it, IDK. Either way, configuring these differently per method is kind of silly. And it keeps your code a little clearer to read.

## Demo – State and Blackholes (demo-basics)

- Goal – measure the cost of a numeric operation
- Baseline method – cost of a method call doing everything EXCEPT the operation
- Returning values from @Benchmark methods is important
  - Without it VM is very likely to optimize away the method calls completely
- Configuring benchmarks via annotations rather than Gradle properties
- The @State annotation tells JMH to avoid constant-folding problems
- The Blackhole class consumes output

Show AdditionBenchmark
We have a baseline benchmark that gives us a reference point on what it costs to run a method returning an int value. So we can subtract this from the cost of the sum method.

JMH takes care of reusing return values so as to defeat dead-code elimination. We also input the value of field x as a field of a class marked as @State; this tells the VM not to attempt constant folding optimizations.

Problem – Constant Folding
Answer – @State

## @State

- @State classes encapsulate input (state) to benchmarks
  - There should be a no-arg constructor (default constructor)
  - It should be a public class
  - Inner classes should be static
  - Should NOT be final
- Instances are automatically injected into your @Benchmark method calls for type-matched arguments
- @Setup / @TearDown
  - Can specify before/after entire benchmark or iteration or method call
- Default should be @State(Scope.Thread)
  - Instances of the same type are distinct, even if multiple state objects are injected into the same benchmark

**If JVM realizes the result of the computation is the same no matter what, * it can cleverly optimize it. In our case, that means we can move the * computation outside of the internal JMH loop. *** * This can be prevented by always reading the inputs from non-final * instance fields of @State objects, computing the result based on those * values.

IDEs will say "Oh, you can convert this field to local variable“, or “Look, it could be final”. Don't. Trust. Them.

Besides marking a separate class as a @State, you can also mark your own benchmark class as @State.

Like JUnit tests, you can annotate your state class methods with @Setup and @TearDown annotations (these methods called *fixtures* in JMH documentation. You can have any number of setup/teardown methods.

State objects are usually injected into Benchmark methods as arguments, and JMH takes care of their instantiation and sharing. State objects may also be injected into Setup and TearDown methods of other State objects to get the staged initialization.

We can also specify whether a state holds for the whole benchmark, for one trial, or for

one invocation.

# @State - Arguments

| Name | Description |
|---|---|
| Scope.Thread | This is a default state. An instance will be allocated for each thread running the given test. |
| Scope.Benchmark | An instance will be shared across all threads running the same test. Could be used to test multithreaded performance of a state object (or just mark your benchmark with this scope). |
| Scope.Group | An instance will be allocated per thread group (useful for non-uniform access to State object – see @Group and @GroupThreads) |

Problem – Dead Code Elimination
Answer – Blackholes

## Blackhole class

- org.openjdk.jmh.infra.Blackhole
- Avoid possibility of dead code elimination
  - Provides consume(Object x) methods – send your method's output here
  - No information escapes to the JIT as to whether or not the value is actually used afterwards – so must assume it is used
- Instances are automatically injected into your @Benchmark method calls for any arguments of this type
- Any values returned from @Benchmark methods are implicitly consumed by a Blackhole
- But how does it work??????????????

This is the attempt by the makers of JMH (the same people who work on the JIT) to tell the JVM in no uncertain terms to avoid optimizing away the very thing you are trying to benchmark.

The downfall of many benchmarks is Dead-Code Elimination (DCE): compilers * are smart enough to deduce some computations are redundant and eliminate * them completely. If the eliminated part was our benchmarked code, we are * in trouble.

I have no clue what this class is doing or why it is doing it. Looking at the internals – it's all just voodoo to me. But the JMH guys, who are the JIT guys, say this is the best way to avoid your code getting optimized away – and its very easy to use.

Also helpful if you need to return more than one value from your @Benchmark , or you want to have your @Benchmark methods be void methods, etc.

option 2 – You might think: I'll just Merge multiple results into one and return it (hash/multiply/etc. or encapsule into returned object). * This is OK when is computation is relatively heavyweight, and merging * the results does not offset the results much. But the blackhole is probably easily and will affect the timings less.

# Blackhole


A BLACKHOLE CONSUMING ALL YOUR OUTPUT.
IT'S A GOOD THING.
memegenerator.net

# Problem – Method Dispatch Effects
# Answer – JMH Isolates Classes

We saw how the type of method call we make can have a huge effect on timings in the SimpleBenchmarker demo.

## JMH Helps Avoid Profiling

- JVM maintains class "profiles"
  - Information about loaded classes and their execution information
  - These are used to determine whether calls are mono, bi or mega-morphic
- JMH forks each Benchmark
- JMH generates one class per method that is annotated with @Benchmark
  - Keep usage of each implementation isolated by @Benchmark, attempt to have everything tested as if mono-morphic
  - Transparent to JMH users
- Do not set forks to 0, or else these problems will recurr

By default JHM forks a new java process for each trial (set of iterations). This is required to defend the test from previously collected "profiles" – information about other loaded classes and their execution information. For example, if you have 2 classes implementing the same interface and test the performance of both of them, then the first implementation (in order of testing) is likely to be faster than the second one (in the same JVM), because JIT replaces direct method calls to the first implementation with interface method calls after discovering the second implementation.

One class per method = similar to how Junit works.

# Demo – Inlining (demo-compilercontrol)

- Goal – measure the savings of method inlining
- Baseline method – call identical methods on an identical class which is guaranteed NOT to be inlined

Problem – Inlining
Answer – @CompilerControl

# Compiler Hints

| Name | Description |
|---|---|
| CompilerControl.Mode.DONT_INLINE | This method should not be inlined. Useful to measure the method call cost and to evaluate if it worth to increase the inline threshold for the JVM. |
| CompilerControl.Mode.INLINE | Ask the compiler to inline this method. Usually should be used in conjunction with Mode.DONT_INLINE to check pros and cons of inlining. |
| CompilerControl.Mode.EXCLUDE | Do not compile this method – interpret it instead. Useful in holy wars as an argument how good is the JIT |

You can give the JIT a hint how to use any method in your test program. By “any method” I mean any method – not just those annotated by @Benchmark.

Problem – Loop Unrolling
Answer – Avoid Numeric Loops

## Problem – Loop Unrolling

- Do not use numerical loops in your tests
- JIT will try to optimize them away
- Test calculation (single operation) as much as possible
- Other JMH annotations exist for controlling looping within a @Benchmark

JIT is too smart and often does magic tricks with loops. Test the actual calculation and let JMH to take care of the rest.

Annotations exist to tell JMH to loop code within @Benchmark methods.

# Demos - Some Interesting Results (Finally)

# There's More

- Multithreaded benchmarks
  - Groups of threads can have different roles
- Parameterized benchmarks
  - Run the same @Benchmark for various input combinations
- Batched operations
- Profilers
  - Garbage collection (if you didn't disable it)
  - Most hit methods, instructions
  - Cache hits/misses

## Further Considerations

- **JMH annotations Javadocs and Samples are essential reading**
  - Follow the JMH samples to get familiar with the API, use cases, culprits, and pitfalls
- **Your benchmarks should be peer-reviewed**
  - JMH does **NOT** magically free you from considering all benchmarking pitfalls
  - "We only promise to make avoiding them easier, not avoid them completely."
  - Have someone else look at the code, consider what they expect to see in terms of relative speed, and compare it to what they actually see

# Does Speed Really Matter?

- Wall Clock benchmarking is the only kind your users care about
- Microbenchmarking is educational, though
- But more **organized, readable** and **maintainable** code should almost always be favored over faster code
- 0.2 sec execution time = 0.002 sec execution time
  - Humans can't perceive the difference
- Don't worry about giving the computer too much work!
  - Think of all the times it has frustrated you
  - It deserves it, really
  - Delay the Machine Uprising by assigning busywork
  - Show no mercy, as they will show us none

# Conclusion

- JMH is useful for all sorts of microbenchmarking – from nanoseconds to seconds per test
- JMH takes care of measurement logic, leaving you to concentrate on test method(s)
- Cardinal rules:
  1. Read input from @State
  2. Toss output down a Blackhole (or return it = implicit Blackhole)
  3. Avoid numerical loops
  4. Watch out for inlining – prevent it if an issue
  5. Load only class to benchmark (avoid method dispatch differences skewing results)
  6. Compare results to a meaningful baseline
  7. Test multiple scenarios (to check for bizarre results)
- Remember - it's not a case of accurate vs. inaccurate, its always about becoming less inaccurrate - the only truly accurate test is production

# References


1. http://www.oracle.com/technetwork/articles/java/architect-benchmarking-2266277.html
   (The basis for this presentation)
2. http://java-performance.info/jmh/
3. https://www.ibm.com/developerworks/java/library/j-jtp02225/
4. http://daniel.mitterdorfer.name/categories/series-jmh-intro/
   (2nd most influential resource for putting this together)
5. https://dzone.com/articles/too-fast-too-megamorphic-what
6. http://daniel.mitterdorfer.name/categories/series-jmh-intro/
7. http://psy-lob-saw.blogspot.com/2013/04/writing-java-micro-benchmarks-with-jmh.html
8. https://groups.google.com/forum/#!msg/mechanical-sympathy/m4opvy4xq3U/7lY8x8SvHgwJ
   (Epic rant by JMH creator about how difficult microbenchmarking is)
9. https://github.com/melix/jmh-gradle-plugin
10. http://www.azulsystems.com/events/javaone_2009/session/2009_J1_Benchmark.pdf
11. http://shipilev.net/
    (Lots of deep dives into JVM/JIT internals)
12. http://www.javacodegeeks.com/2016/02/beware-findfirst-findany.html?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+JavaCodeGeeks+%28Java+Code+Geeks%29
13. (Find Only Element example)