Java (Micro) Performance Measuring

How to measure and test the performance of Java applications

Agenda

What is performance?
Definition | Why Java? | Micro

A simple approach
Timestamps | Pitfalls

The JVM
Warm-Up | JIT | GC

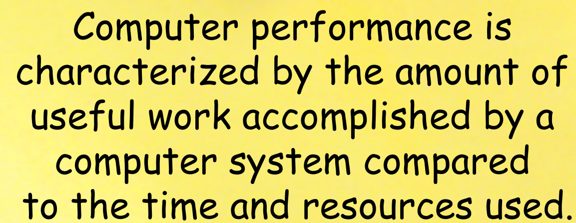
A suitable approach
Framework

Examples
Sorts | Ordering | Repeatable

Some remarks on performance testing
Reports | Statistics

References
Papers | Websites | Tools

What is performance?



- throughput
- response time
- utilization of resources
- memory
- subjective impressions
- locks

Why (just) Java?

- most of you are familiar with Java
- Java isn't just a language, it's a platform (JRuby, Scala, Clojure, Groovy, etc.)
- the general concept of a virtual machine language
- it's complicated to measure right, but easy to do it wrong

Java Performance, Addison-Wesley

What means “micro”?

Micro means small.

- small code blocks (classes or methods)
- everybody can use it, even in small projects
- I'm sure, everybody has tried it already

pennystockalerts.com

A simple approach

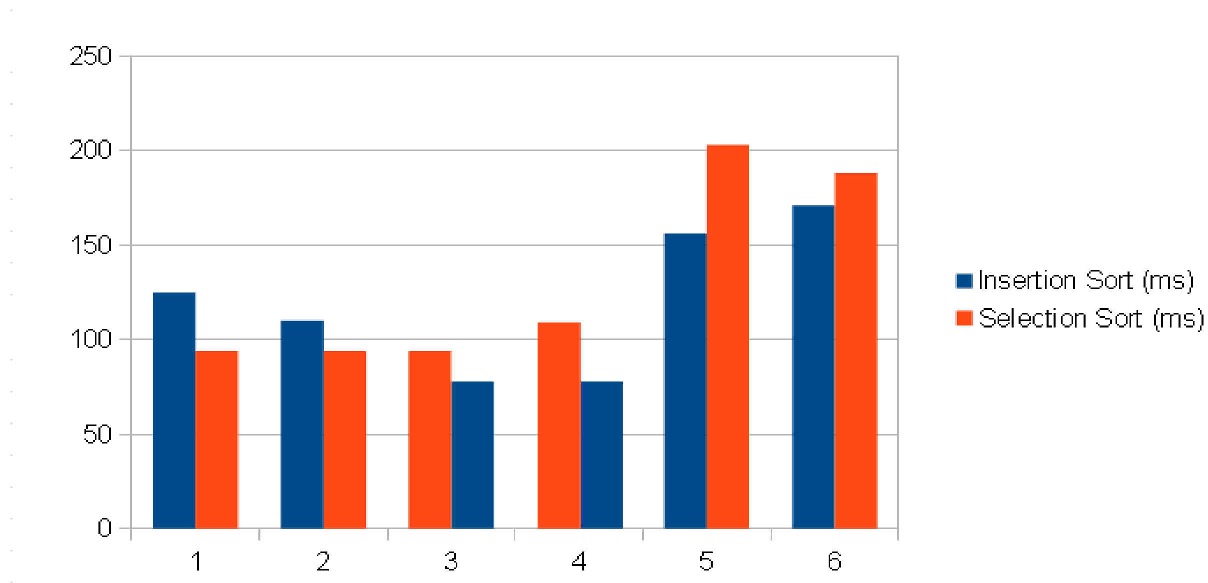
```
long before = System.currentTimeMillis();
methodCall();
long after = System.currentTimeMillis();
long time = after - before;
```

```
for(int i = 0; i < 1000; i++) {
    long before = System.currentTimeMillis();
    methodCall();
    long after = System.currentTimeMillis();
    total = after - before;
}
```

1. record start time
2. run the code
3. record end time
4. calculate the difference

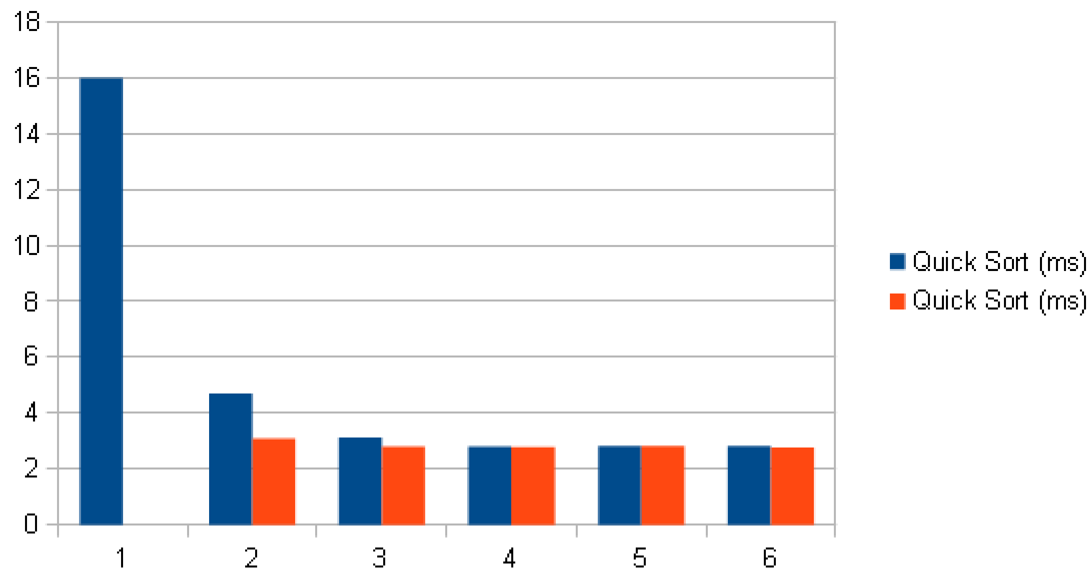
Something went wrong...

The execution time changes when we rearrange the tests or change the input data the test is operating on!



Something went wrong...

The execution time changes converge when we increase the iterations!



But what went wrong?

We missed some pitfalls around benchmarking in general and especially around Java micro benchmarking!

zephram.de

Pitfalls 1/4

We missed some important facts of the Java platform:

- `System.currentTimeMillis()` has a pretty bad resolution
  - if the measurement error should be < ±1 ms, it fails for all tasks < 200 ms
  - in reality, the resolution varies between 10 and 100 ms
  - it's a wall-clock time!
- the JVM has a “warm-up phase”
  - classes are loaded when they're first used (I/O, search, parsing, verification, etc.)
  - code can be JIT compiled (after 1.500 calls on a client or 10.000 on a server)
- Garbage Collection & Object Finalization
  - GC & OF is nearly out of control for the programmer

Pitfalls 2/4

We missed some important facts of the Java platform:

- dead code elimination
  - the JVM can remove “dead code” during the execution
  - if you don't use the return value, the whole method-call *can* be removed!
- code optimization
  - e.g. loop-enrollment, removing duplicate assignments or type checks
- the JVM can also de-optimize the code!
  - if a new class is loaded which can be used in a generic method
  - if a new path (e.g. an catch-block) becomes a “hotspot”

Pitfalls 3/4

We missed some important facts of the Java platform:

- a lot of JVM options can influence the performance (of course):
  - `-server` or `-client` are influencing the JIT compilation threshold
  - `-enableassertions` or `-disableassertions` can influence checks
  - the type of garbage collector
  - the memory size (`-Xmx`)

Pitfalls 4/4

We also missed some important facts of modern computers:

- caching
  - especially I/O caching, e.g. by reading files
  - but also CPU caching
- Power Management
- other programs and processes
  - virus scanners, screen savers, updates
  - Eclipse itself

Suggestions for improvements

Some simple tips to improve the measurement:

- `System.nanoTime()` has a better resolution
    - it should never be worse than `System.currentTimeMillis()`
    - depending on the hardware and OS it can deliver accuracy in the microsecond range
- load all classes before the test
    - manually
    - by executing the task several time before measuring
    - use `ClassLoadingMXBean` to control class loading during the measurement

Suggestions for improvements

Some simple tips to improve the measurement:

- use the return values of your method to avoid “dead code” elimination
- avoid JIT compilation
    - run the code repeatedly until it execution time doesn't varies anymore
    - use `CompilationMXBean` to keep track of the compilation

GC/OF

GC/OF is nearly out of control for the programmer, so what we can do?

- first, call `System.gc()` and `System.runFinalization()` until memory usage is stabilized
- do your measurement
- repeat step one after your measurement
    - does it take long?
    - have many objects been created during the test?
- log the results/time of the GC/OF, because it's part of the performance

But:

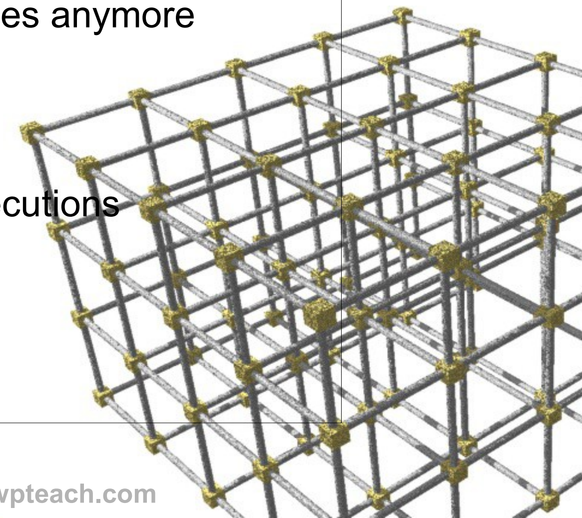
- There's no guarantee for `System.gc()` or system.runFinalization()
- Don't call `System.gc()` or `System.runFinalization()` during your measurement, because the can influence your results.

A suitable approach

Use a framework!

e.g. JBenchmark by Brent Boyer (see references)

1. load all classes (manually or by executing the task once)
2. execute the task repeatedly until its execution time doesn't varies anymore
3. get a first estimate of the execution time
4. try to clean the JVM (GC/OF)
5. use this estimate to measure a sufficiently large number of executions
6. keep track of JIT compilation
7. calculate the elapsed time of multiple executions
8. clean the JVM again and measure the needed time



wpteach.com

A suitable approach

...has to give reliable answers to the most crucial questions:

How long takes task A?

Does it fluctuate?

Is task A faster then task B?

weddingbycolor.com

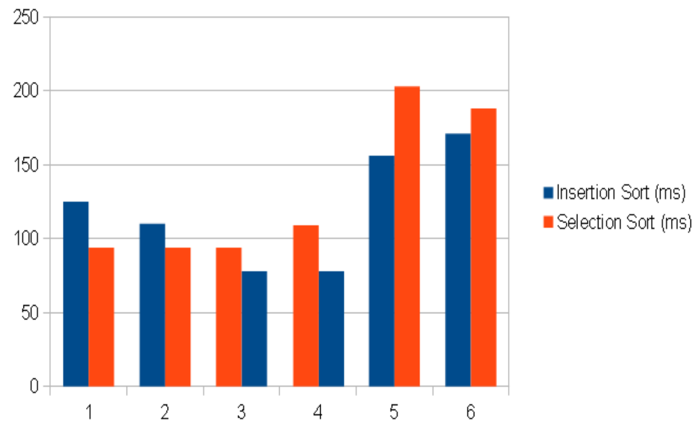
A suitable approach

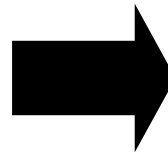
...hast to give use some statistics!

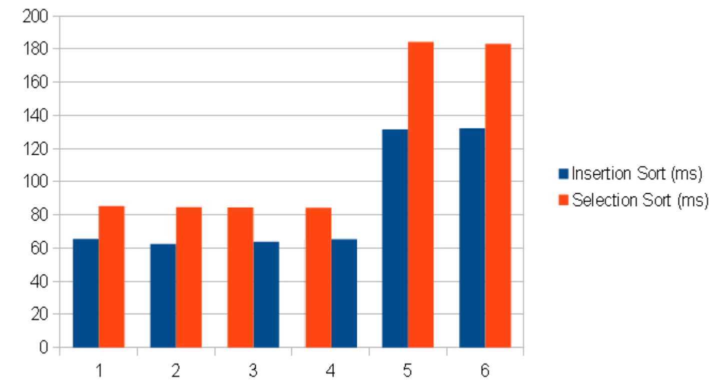
The most important metrics are:

- The **mean** can tell you how long the task is running over a (huge) number of executions.
- The **standard deviation** can tell whether the execution time fluctuates
- A **historical log** can tell how the performance has evolved over time

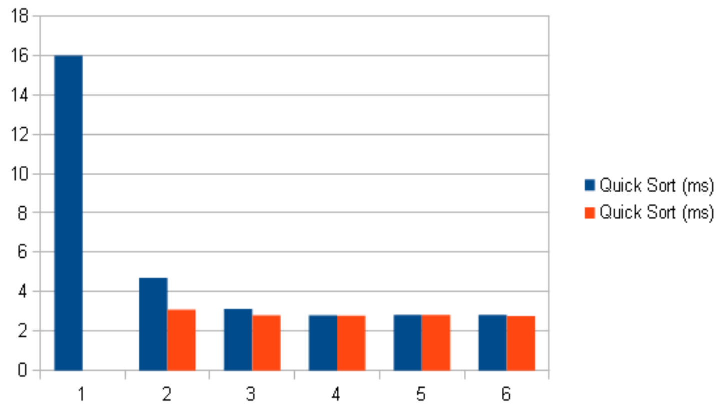
Doing it right!

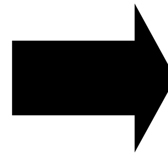


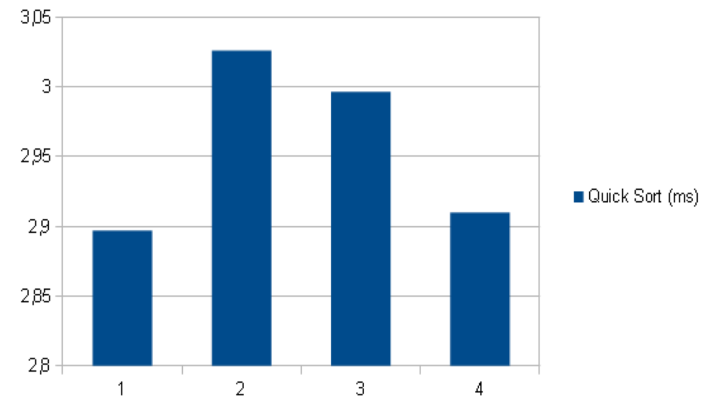




Doing it right!







Keep in mind

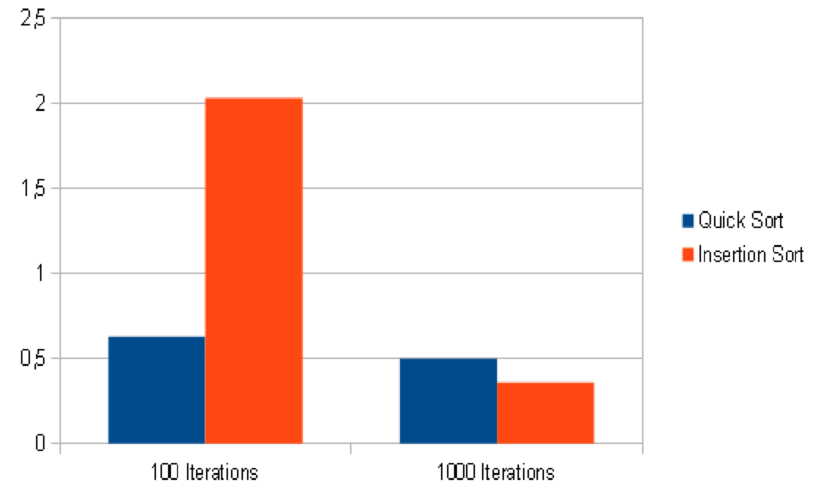
To keep our measurement reliable and repeatable, we have to keep some general things in mind:

- the results depend on the particular machine
- the results depend on the particular JVM version/ vendor
- the results depend on the particular JVM options
- the results depend on the particular running processes of the system
- the results depend on the particular operating system
- the results can depend on “random” events like a network or power management

And last but not least: The results can depend on you! Shit in, shit out!

Shit in, shit out!

```
for (int i = 0; i < loop; i++) {
    before = System.currentTimeMillis();
    new QuickSort().sort(list1);
    after = System.currentTimeMillis();
    total += after - before;
}
```



Shit in, shit out!

- **Insertion Sort**

  Best: O(n)

  Average: O(n + d)

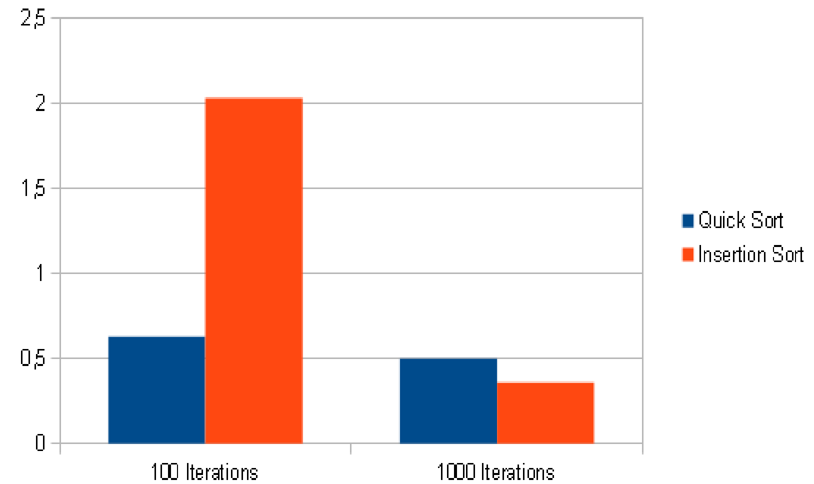
  Worst: O(n2)

- **Quick Sort**

  Best: O(nlog n)

  Average: O(nlog n)

  Worst:: O(n2)

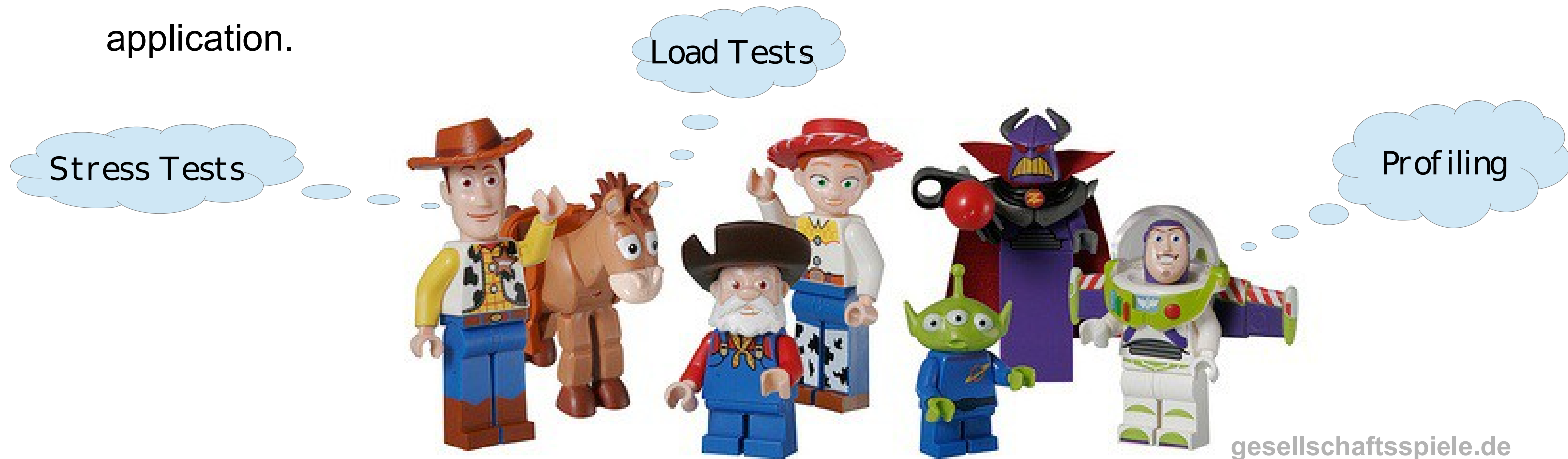


Keep in mind

Micro Performance Measuring is just *one* test,

but not the only one!

- Because only single classes or methods are tested, the overhead of a complex class-/call-hierarchy is ignored.
- A single measurement of an algorithm can't tell you much about the performance of a whole application.



References

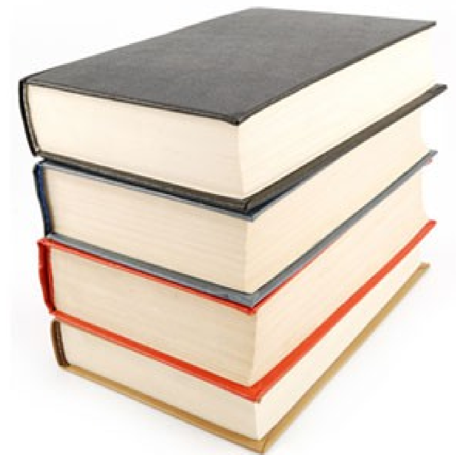
Brent Boyer, Robust Java benchmarking, 2008,
https://www.ibm.com/developerworks/java/library/j-benchmark1

Brian Goetz, Java theory and practice: Anatomy of a flawed microbenchmark, 2005,
https://www.ibm.com/developerworks/java/library/j-jtp02225

Daan van Etten, Why Many Java Performance Tests are Wrong, 2009,
http://java.dzone.com/articles/why-many-java-performance-test

Orcacle, Monitoring and Management of the Java Platform, 2012,
http://docs.oracle.com/javase/7/docs/technotes/guides/management

R. Henry, Sorting Algorithms Demonstration in Java,
http://home.westman.wave.ca/~rhenry/sort



thesqueezepage.org

Questions?