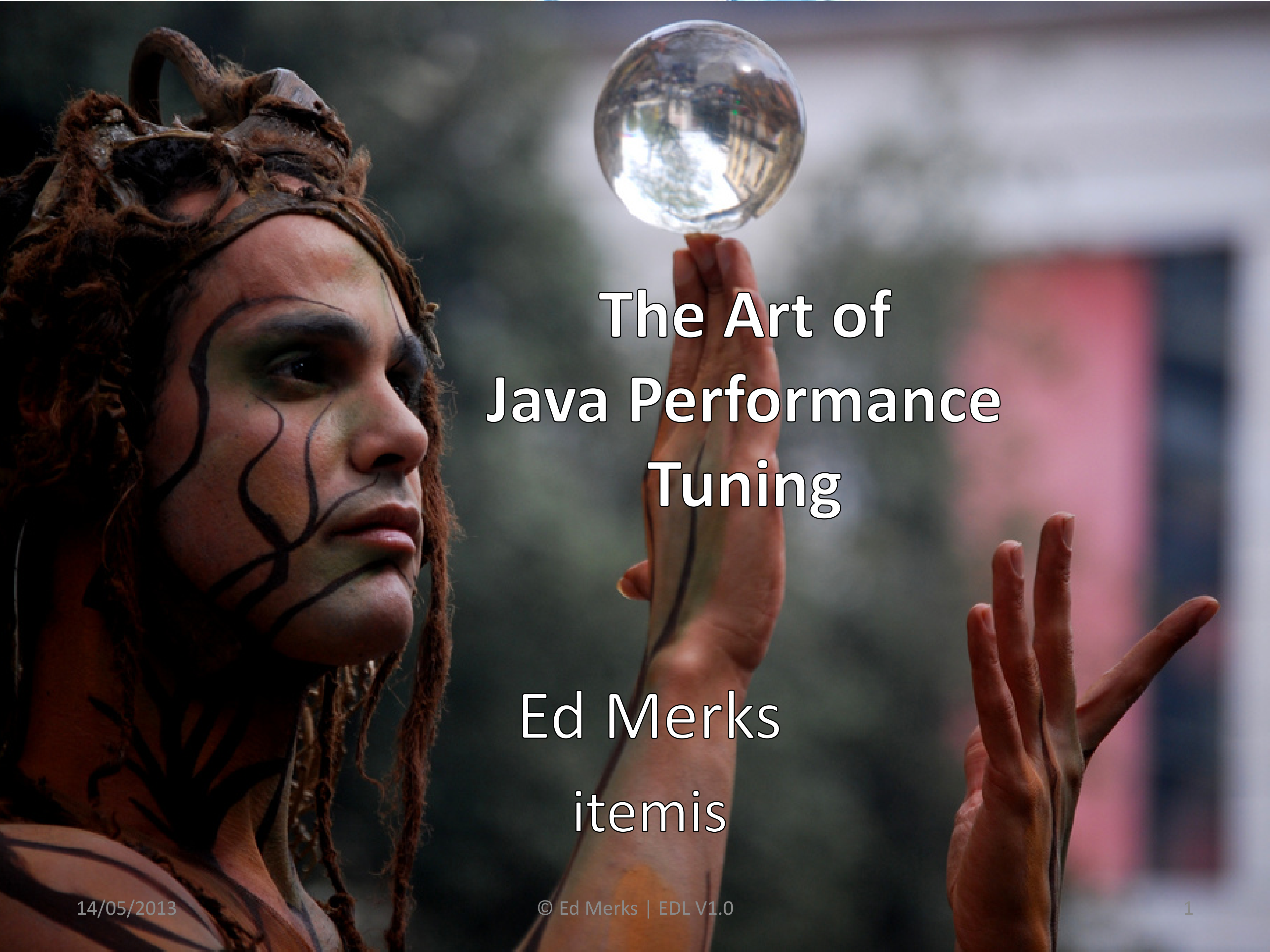


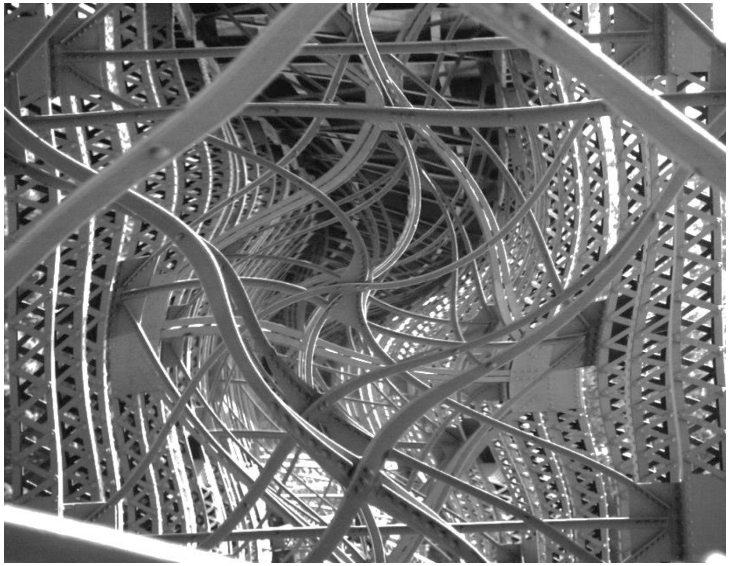
The Art of Java Performance Tuning

Ed Merks

itemis

Java Performance is Complex

- Write once run everywhere
  - Java is slow because it's interpreted
    - No, there are Just In Time (JIT) compilers
  - Different hardware and platforms
  - Different JVMs
    - Different tuning options
  - Different language versions



Faster is Better

Smaller is Better

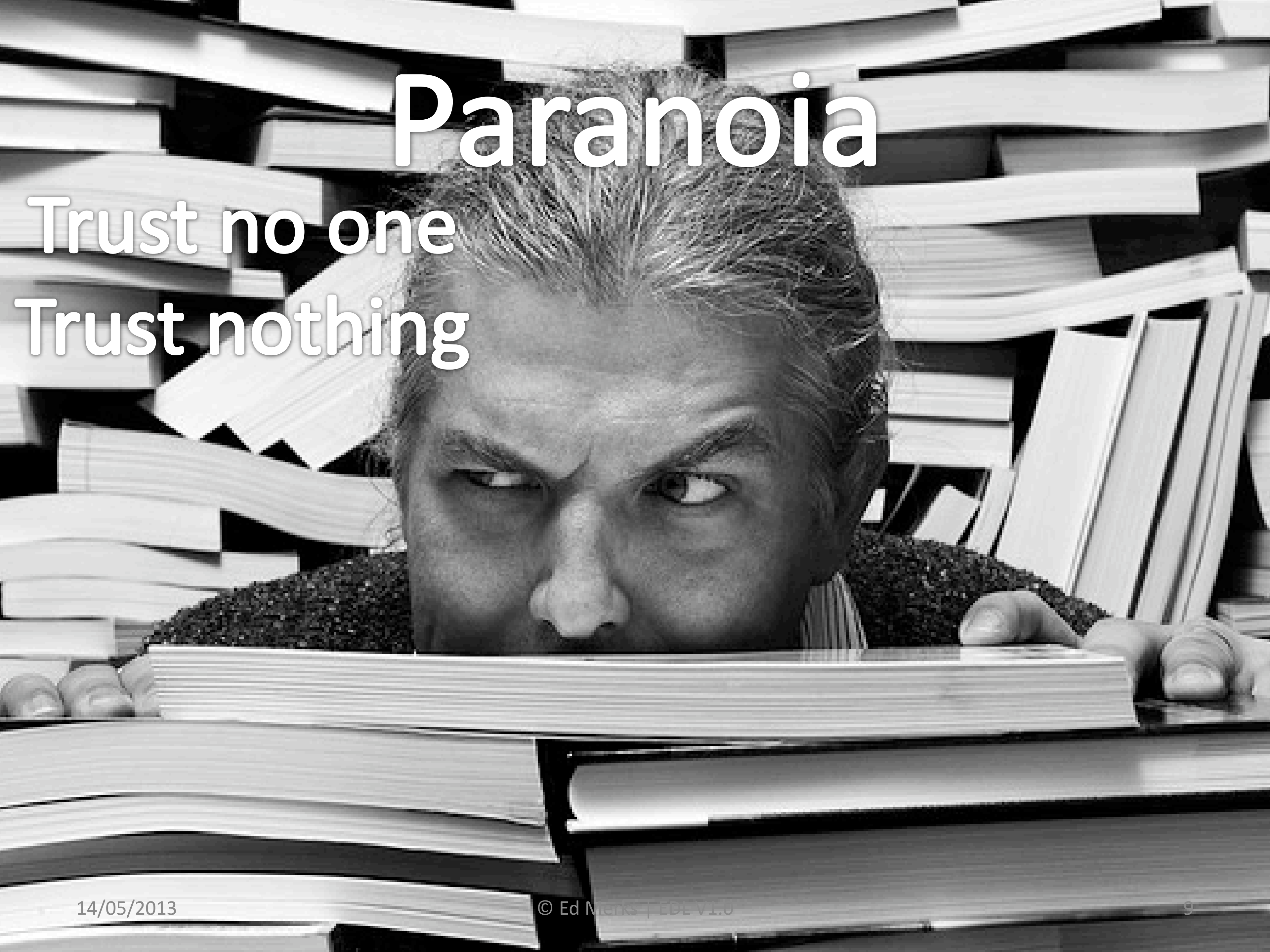
Faster and Smaller is Best

Measuring

Benchmarking

Profiling

Paranoia



Trust no one

Trust nothing

Don’t Trust Your Friends

- Your friends are stupid

Don’t Trust Your Measurements

- Your measurements are unreliable

Don’t Trust Yourself

- You know nothing

Don’t Trust the Experts

- The experts are misguided

Definitely Don't Trust Me!

Don’t Trust Anything

- Everything that’s true today might be false tomorrow
- Whatever you verify is true today is false somewhere else

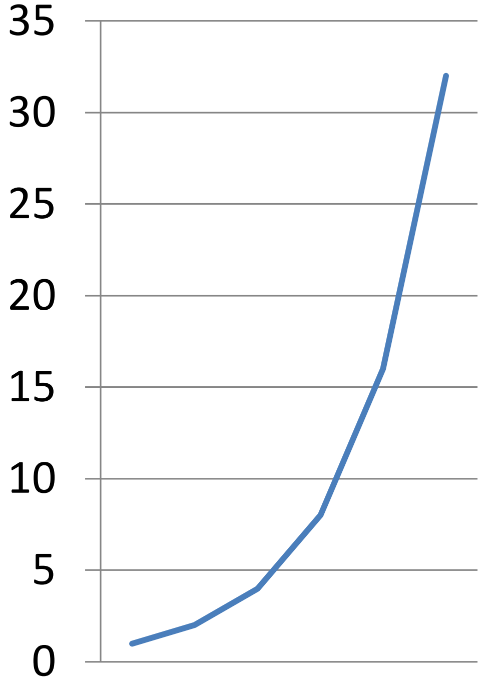
Where Does That Leave You?

- Don’t worry
- Be happy
- Write sloppy code and place blame elsewhere
  - Java
  - The hardware
  - The platform
  - JVM
  - Poor tools

There’s No Excuse for Bad Code

Algorithmic Complexity

- How does the performance scale relative to the growth of the input?
  - O(1) – hashed lookup
  - O(log n) – binary search
  - O(n) – list contains
  - O(n log n) – efficient sorting
  - O(n^2) – bubble sorting
  - O(2^n) – combinatorial explosion
- No measurement is required



Loop Invariants

- Don’t do something in a loop you that can do outside the loop

```
public NamedElement find(NamedElement namedElement){
 for (NamedElement otherNamedElement : getNamedElements()) {
   if (namedElement.getName().equals(otherNamedElement.getName())) {
     return otherNamedElement;
   }
 }
 return null;
}
```

- Learn to use Alt-Shift-↑ and Alt-Shift-L

Generics Hide Casting

- Java 5 hides things in the source, but it doesn’t make that free at runtime

```
public NamedElement find(NamedElement namedElement) {
  String name = namedElement.getName();
  for (NamedElement otherNamedElement : getNamedElements()) {
    if (name.equals(otherNamedElement.getName())) {
      return otherNamedElement;
    }
  }
  return null;
}
```

- Not just the casting is hidden but the iterator too

Overriding Generic Methods

- Overriding a generic method often results in calls through a bridge method
  - That bridge method does casting which isn’t free

```
new HashMap<String, Object>() {
  @Override
  public Object put(String key, Object value)  {
    return super.put(key == null ? null : key.intern(), value);
  }
};
```

Accessing Private Fields

- Accessing a private field of another class implies a method call

```
public static class Context {
  private class Point {
    private int x;
    private int y;
  }

  public void compute()
  {
    Point point = new Point();
    point.x = 10;
    point.y = 10;
  }
}
```

External Measurements

- Profiling
  - Tracing
    - Each and every (unfiltered) call in the process is carefully tracked and recorded
    - Detailed counts and times, but is slow, and intrusive, and doesn’t reliably reflect non-profiled performance
  - Sampling
    - The running process is periodically sampled to give a statistical estimate of where the time is being spent
    - Fast and unintrusive, but unreliable beyond hot spot identification

Call It Less Often

- Before you focus on making something faster focus on calling it less often

External Measurements

- Consider using YourKit
  - They support* open source

Internal Measurements

- Clock-based measurements
  - System.currentTimeMillis
  - System.nanoTime (Java 1.5)
- Accuracy verses Precision
  - Nanoseconds are more precise than milliseconds
  - But you can’t trust the accuracy of either

Micro Benchmarks

- Measuring small bits of logic to draw conclusions about which approach performs best
  - These are fraught with problems
  - The same JIT will produce very different results in isolation from what it does in real life
  - The hardware may produce very different results in isolation from what it does in a real application
  - You simply can’t measure threading reliably

Micro Benchmarks

- The JIT will turn your code into a very cheap no-op
  - Your benchmark must compute a result visible to the harness
- Because the clocks are inaccurate you must execute for a long time
  - That typically implies doing something in a loop and then of course you’re measuring the loop overhead too

Micro Benchmarks

- Do as much as possible outside the benchmark and outside the loop
- You want to know the performance of the compiled code, not the interpreted code
  - You need a warmup
    - Use -XX:+PrintCompilation
  - Beware the garbage collector
    - Use -verbose:gc

Micro Measurements

- I wrote a small benchmark harness
  - http://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/tests/org.eclipse.emf.test.core/src/org/eclipse/emf/test/core/BenchmarkHarness.java
  - Write a class that extends Benchmark and implements run
  - The harness runs the benchmark to determine many times it must run to use approximately a minimum of one second
  - Then it runs it repeatedly, gathering statistics

Platform

- Hardware

  Intel Core i7-2920XM CPU @ 2.5Ghz

- OS

  Windows 7 Professional
  Service Pack 1

- JVM

  java version "1.6.0_32"
  Java(TM) SE Runtime Environment (build 1.6.0_32-b05)
  Java HotSpot(TM) 64-Bit Server VM (build 20.7-b02, mixed mode)

The Simplest Micro Measurement

- This is the simplest thing you can measure

```
public static class CountedLoop extends Benchmark {
   public CountedLoop() { super(1000000); }

   @Override
   public int run() {
     int total = 0;
     for (int i = 0; i < count; ++i) {
       total += i;
     }
     return total;
   }

   @Override
   public String getLogic() {
     return "total += i;";
   }
 }
```

- 0.348 < **0.348** < 0.350 CV%: 0.00 CR 95%: 0.348 <- 0.350

Cache Field in Local Variable

- I heard that caching a repeatedly-accessed field in a local variable improves performance

```
public int run() {
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i;
  }
  return total;
}
```

- 0.328 < **0.329** < 0.330 CV%: 0.00 CR 95%: 0.328 <- 0.330
- **10%** faster

Questionable Conclusions

- Depending on the order in which I run the benchmarks together, I get different results

```
public static void main(String[] args) {
  Benchmark[] benchmarks = {
        new CountedLoop(),
        new CountedLoopWithLocalCounter(),
  };
  new BenchmarkHarness(1).run(20, benchmarks);
}
```

- In isolation they perform the same
- In combination, whichever is first is faster

Array Access

- Let’s measure the cost of accessing an array

```
public int run() {
  int[] array = this.array;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += array[i];
  }
  return total;
}
```

- 0.315 < **0.317** < 0.325 CV%: 0.63 CR 90%: 0.316 <- 0.325
- Hmmm, it takes negative time to access an array

Array Access Revised

- Let’s try again

```
public int run() {
  int[] array = this.array;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + array[i];
  }
  return total;
}
```

- 0.498 < **0.499** < 0.504 CV%: 0.20 CR 90%: 0.498 <- 0.504
- Subtracting out the cost of the scaffolding, we could conclude that array access takes **0.151** nanoseconds

Array Assignment

- Let’s measure array assignment

```
public int run() {
  int[] array = this.array;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    array[i] = total += i + array[i];
  }
  return total;
}
```

- 0.793 < **0.795** < 0.798 CV%: 0.13 CR 90%: 0.793 <- 0.798
- We could conclude that array assignment takes **0.296** nanoseconds

Method Call

- How expensive is calling a method?

```
public int run() {
  String[] array = this.array;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + array[i].hashCode();
  }
  return total;
}
```

- 5.308 < **5.328** < 5.362 CV%: 0.24 CR 90%: 5.315 <- 5.362
- We could conclude that **this** method call takes **4.829** nanoseconds

Method Call

- How expensive is calling a native method?

```
public int run() {
  Object[] array = this.array;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + array[i].hashCode();
  }
  return total;
}
```

- 2.442 < **2.456** < 2.480 CV%: 0.45 CR 90%: 2.443 <- 2.480
- We could conclude that **this** native method call takes **1.975** nanoseconds

Array Verses List

- How fast is an array list compare to an array

```
public int run() {
  ArrayList<String> list = this.list;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + list.get(i).hashCode();
  }
  return total;
}
```

- 5.565 < **5.617** < 5.703 CV%: 0.69 CR 90%: 5.568 <- 5.703
- We could conclude that calling get(i) takes **0.289** nanoseconds

JIT Inlining

- How can calling String.hashCode take 4.829 nanoseconds while calling ArrayList.get takes 0.289 nanoseconds?
  - That’s 95% faster, and hashCode doesn’t do much
  - Inlining
    - java.util.ArrayList::RangeCheck (48 bytes)
    - java.util.ArrayList::get (12 bytes)
- You never know whether the JIT will inline your calls but the difference is dramatic

What Can the JIT Inline?

- Calls to relatively small methods which is influenced by server mode and by JVM options
- Calls to static methods which are always final
- Calls to methods implicitly or explicitly via this or super when the JIT can infer final
- Calls to methods declared in other classes, if final can be inferred
- Calls to methods on interfaces
  - That depends on how many classes implement the interface, i.e., how well final can be inferred

When Does the JIT Inline?

- Only after many calls to a method, i.e., on the order of 10,000
- The JIT focuses on methods whose improvement will have a significant overall impact
- Loading of classes can impact the determination of finalness of methods such that optimizations may need to be reverted

How Does BasicEList Compare?

- How fast is EMF’s BasicEList relative to ArrayList

```java
public int run() {
  BasicEList<String> eList = this.list;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + eList.get(i).hashCode();
  }
  return total;
}
```

- 5.567 < **5.580** < 5.599 CV%: 0.14 CR 90%: 5.572 <- 5.599
- Quite well, but there are many subclasses!

How Expensive is Casting?

- First let’s measure this as a baseline

```
public int run() {
  String[] array = this.array;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + array[i].charAt(0);
  }
  return total;
}
```

- 5.946 < **5.967** < 6.001 CV%: 0.22 CR 90%: 5.953 <- 6.001
- Note that calling charAt is **0.639** nanoseconds slower than calling hashCode

How Expensive is Actual Casting?

- Here the call to get really must cast to a String

```
public int run() {
  ArrayList<String> list = this.list;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + list.get(i).charAt(0);
  }
  return total;
}
```

- 6.004 < **6.037** < 6.127 CV%: 0.50 CR 90%: 6.006 <- 6.127
- That’s just a **0.07** nanosecond difference, i.e., smaller than we’d expect for array verses list, so casting is very cheap

Method Call Revisited

- Let’s measure method calls again

```
public int run() {
  ENamedElement[] array = this.array;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + array[i].getName().hashCode();
  }
  return total;
}
```

- 20.154 < **20.181** < 20.266 CV%: 0.12 CR 90%: 20.158 <- 20.266
- Wow, that took long! Calling getName takes **14.853** nanoseconds

So How Expensive is Casting Really?

- Let’s measure that using a list

```
public int run() {
  List<ENamedElement> list = this.list;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + list.get(i).getName().hashCode();
  }
  return total;
}
```

- 19.549 < **19.613** < 19.841 CV%: 0.30 CR 90%: 19.566 <- 19.841
- It’s faster, until my machine nearly catches fire, and then it’s the same, so casting is apparently free. Hmmm….

Casting is Hard to Measure!

- I heard from experts that the cost of casting depends on…
  - The complexity of the runtime hierarchy
- I’ve been told that an object remembers what it was cast to recently and can be cast again more quickly so one should avoid “ping pong” casting
- In any case, casting is **much** faster today than it was 10 years ago, when it was shockingly slow

O(n) With a Large Constant

- Contains testing on a list is O(n), for n 1000

```
public int run() {
  List<ENamedElement> list = this.list;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + (list.contains(lastENamedElement) ? 1 : 0);
  }
  return total;
}
```

- 3,544.660 < **3,562.194** < 3,692.060 CV%: 0.90 CR 90%: 3,545.132 <- 3,692.060

O(n) With a Small Constant

- Contains testing on a list is O(n), for n 1000

```
public int run() {
  BasicEList.FastCompare<ENamedElement> eList = this.list;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + (eList.contains(lastENamedElement) ? 1 : 0);
  }
  return total;
}
```

- 365.123 < **365.948** < 367.809 CV%: 0.18 CR 90%: 365.194 <- 367.809
- It’s ~10 times faster because it uses == rather than Object.equals!
- And that’s why you can’t override EObject.equals

O(1) List Contains

- Contains testing on a *containment* list is O(1), for any value of n, here 1000

```
public int run() {
  EObjectContainmentEList<ENamedElement> eList = this.list;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + (eList.contains(lastENamedElement) ? 1 : 0);
  }
  return total;
}
```

- 4.733 < **4.750** < 4.820 CV%: 0.38 CR 90%: 4.740 <- 4.820
- It's another ~75 times faster because an EObject **knows** whether or not it's in a containment list

O(1) HashSet Contains

- Contains testing on a HashSet is always O(1)

```
public int run() {
  HashSet<ENamedElement> set = this.set;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + (set.contains(lastENamedElement) ? 1 : 0);
  }
  return total;
}
```

- 5.758 < **5.775** < 5.797 CV%: 0.16 CR 90%: 5.765 <- 5.797
- It takes **5.276** nanoseconds to do a contains test; it’s still slower than a containment list’s contains testing…

Synchronize: Thread Safety

- Suppose we used Collections.*synchronizedSet*

```
public int run() {
  Set<ENamedElement> set = this.set;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + (set.contains(lastENamedElement) ? 1 : 0);
  }
  return total;
}
```

- 26.309 < **26.400** < 26.592 CV%: 0.24 CR 90%: 26.336 <- 26.592
- It takes ~**20** nanoseconds to do the synchronize, even with only a single thread using this set
- Even with a derived class that simply overrides contains, rather than a wrapper, I get the same result

Object Allocation

- Creating just a plain old Object

```
public int run() {
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + new Object().hashCode();
  }
  return total;
}
```

- 46.684 < **47.113** < 49.081 CV%: 1.32 CR 90%: 46.738 <- 49.081
- It's hard to avoid measuring GC impact
- Allocation is relatively expensive!

Counted Loop

- Iterating over an empty array list via a counter

```
public int run() {
  List<Object> list = this.list;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    for (int j = 0, size = list.size(); j < size; ++j) {
      total += i + list.get(j).hashCode();
    }
  }
  return total;
}
```

- 0.937 < **0.939** < 0.943 CV%: 0.11 CR 90%: 0.937 <- 0.943
- This is essentially the cost of getting the size and noticing it's 0

For-each Loop

- Iterating over an empty array list via a counter

```
public int run() {
  List<Object> list = this.list;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    for (Object object : list) {
      total += i + object.hashCode();
    }
  }
  return total;
}
```

- 5.937 < **5.992** < 6.059 CV%: 0.42 CR 90%: 5.967 <- 6.059
- This 6 times slower, reflects the high cost of allocating the iterator, though that’s much cheap than creating an object

Non-empty Loops

- We can repeat these tests with a list of size 10
  - 46.579 < **46.932** < 47.340 CV%: 0.48 CR 90%: 46.669 <- 47.340
  - 54.898 < **55.104** < 55.442 CV%: 0.32 CR 90%: 54.917 <- 55.442
- Given we know Object.hashCode takes 1.975 nanoseconds we can subtract the 10 calls and the empty loop overhead
  - 46.932 – 10 * 1.975 – 0.939 = **26.243**
  - 55.104 – 10 * 1.975 – 5.992 = **29.362**
- The difference between those divided 10, i.e., **0.331** nanoseconds, is the per-iteration overhead of the iterator

Old URI Implementation

- I recently revised EMF’s URI implementation

```
public int run() {
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i)  {
    total += i +
     (uris[repetition][i] =
        URI2.createURI(strings[repetition][i])).hashCode();
  }
  ++repetition;
  return total;
}
```

- 946.633 < **988.341** < 1,036.170 CV%: 2.25 CR 90%: 956.324 <- 1,036.170
- With forced System.gc outside the measurement runs

New URI Implementation

- New URI implementation

```
public int run() {
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i)  {
    total += i +
     (uris[repetition][i] =
        URI.createURI(strings[repetition][i])).hashCode();
  }
  ++repetition;
  return total;
}
```

- 720.208 < **746.296** < 783.516 CV%: 2.29 CR 90%: 722.827 <- 783.516
- It’s 25% faster than before (in this scenario/configuration)

New URI has Faster Equality

- URIs are often used as keys where equals is used

```
public int run() {
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + (uri1.equals(choose[i & 3]) ? 1 : 0);
  }
  return total;
}
```

- 4.628 < **4.638** < 4.659 CV%: 0.15 CR 90%: 4.629 <- 4.659
- 1.547 < **1.550** < 1.556 CV%: 0.13 CR 90%: 1.547 <- 1.556
- Factoring out the scaffolding, it's 4 times faster.

HashMap Get

- Getting a key’s value out of a map is fast

```
public int run() {
  Map<Object, String> map = this.map;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i) {
    total += i + map.get(choose[i & 3]).hashCode();
  }
  return total;
}
```

- 8.487 < **8.509** < 8.539 CV%: 0.16 CR 90%: 8.489 <- 8.539
- Factoring out scaffolding, **3.81** nanoseconds, as we’d expect from Set.contains and String.hashCode measurements

EObject eGet

- Getting a feature’s value out of an EObject is faster

```
public int run() {
  EObject eObject = this.eObject;
  int total = 0;
  for (int i = 0, count = this.count; i < count; ++i)  {
    total += i + eObject.eGet(choose[i & 3]).hashCode();
  }
  return total;
}
```

- 7.992 < **8.013** < 8.034 CV%: 0.15 CR 90%: 7.994 <- 8.034
- I.e., **2.685** nanoseconds without scaffolding, so ~30% faster than a hash map lookup

Java Reflection

- Compare EMF reflection with Java reflection

```
public int run() {
  try {
    Object object = this.object;
    int total = 0;
    for (int i = 0, count = this.count; i < count; ++i) {
      total += i + choose[i & 3].get(object).hashCode();
    }
    return total;
  } catch (Exception exception) {
    throw new RuntimeException(exception);
  }
}
```

- 11.813 < **11.849** < 11.897 CV%: 0.17 CR 90%: 11.825 <- 11.897

Don’t Be Fooled

- Suppose you noticed that 5% of a 2 minute running application was spent in this method

```
public Element getElement(String name) {
  for (Element element : getElements())  {
    if (name.equals(element.getName())) {
      return element;
    }
  }
  return null;
}
```

- You might conclude you needed a map to make it fast…

Look Closely at the Details

- Upon closer inspection, you’d notice the getter creates the list on demand

```
public List<Element> getElements() {
  if (elements == null) {
    elements = new ArrayList<Element>();
  }
  return elements;
}
```

- You’d also notice that getName is not called all that often, i.e., most lists are empty

It’s Fast Enough with a Map

- So you could rewrite it as follows

```
public Element getElement(String name) {
  if (elements != null) {
    for (int i = 0, size = elements.size(); i < size; ++i) {
      Element element = elements.get(i);
      if (name.equals(element.getName())) {
        return element;
      }
    }
  }
  return null;
}
```

- It would take less than 1% of the time

Focus on What’s Important

- Conceive well-designed algorithms
  - The JVM and the JIT will not turn O(n^2) algorithms into O(n log n) algorithms
- Write clear maintainable code
  - The JVM and the JIT are often smarter than you are and can make your beautiful code fly
- Don’t make excuses
  - The JIT shouldn’t need to determine your loop invariants; don’t assume it will

Measure, Measure, Measure

- You know nothing without measurements
- You cannot trust measurements taken in isolation
- You cannot know what’s happening in detail within a full application without disturbing the very thing you’re measuring
- Despite the fact that you cannot trust your measurements you cannot tune an application without them

Measurement Driven Focus

- Profilers help determine where your energy is best spent
- Benchmarks help assess your progress and your regressions
- Sometimes big things don’t matter at all
- Sometimes small things matter a lot

Attributions: Thanks for the Flicks

- http://www.flickr.com/photos/jcarlosn/4528401347/sizes/l/in/photostream/
- http://www.flickr.com/photos/42000933@N02/6875870412/sizes/l/in/photostream/
- http://www.flickr.com/photos/jorgeguzman/144812237/sizes/l/in/photostream/
- http://www.flickr.com/photos/tomasino/7206225040/sizes/h/in/photostream/
- http://www.flickr.com/photos/veggiefrog/3667948537/sizes/l/in/photostream/
- http://www.flickr.com/photos/freddyfam/2540701577/sizes/l/in/photostream/
- http://www.flickr.com/photos/jeffk/25374399/sizes/l/in/photostream/
- http://www.flickr.com/photos/mikolski/3269906279/sizes/l/in/photostream/
- http://www.flickr.com/photos/katiew/320161805/sizes/z/in/photostream/
- http://www.flickr.com/photos/aaronjacobs/86952847/sizes/l/in/photostream/
- http://www.flickr.com/photos/seeminglee/8286759305/sizes/l/in/photostream/
- http://www.flickr.com/photos/cayusa/1209794692/sizes/l/in/photostream/
- http://www.flickr.com/photos/gurana/4442576424/sizes/l/in/photostream/
- http://www.flickr.com/photos/megangoodchild/6942503305/sizes/l/in/photostream/