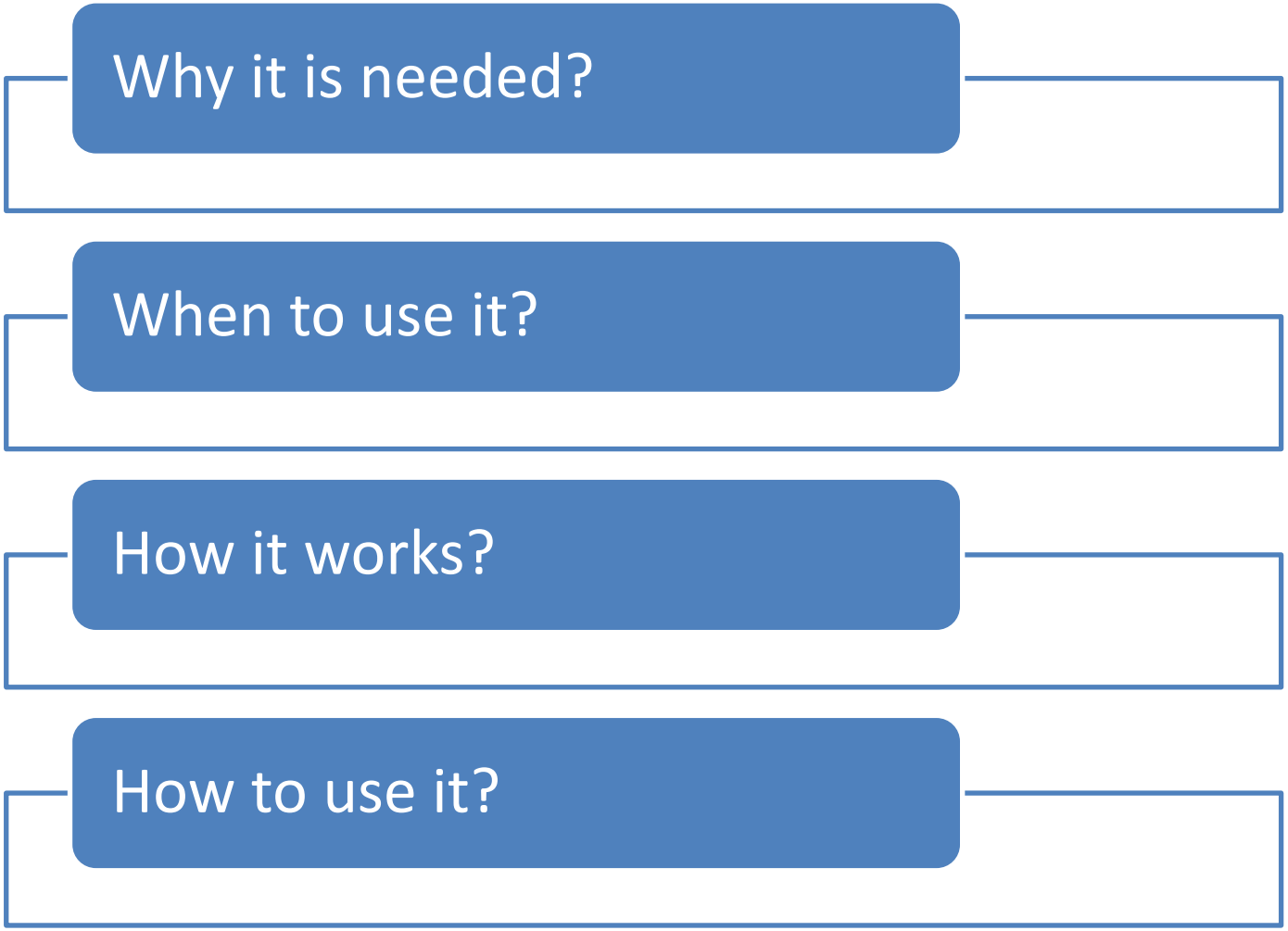
Why GC is eating all my CPU?

Aprof - Java Memory Allocation Profiler

Roman Elizarov, Devexperts

Joker Conference, St. Petersburg, 2014

Java Memory Allocation Profiler



Java

- Does not have stack-allocation, does not have structs, does not have tuples…
- Promotes the “Object Oriented” style of programming with lots of object allocations
  - Most of which are only temporary → garbage

Garbage collection

Enjoy Java and GC

Until it becomes performance bottleneck

PREMATURE OPTIMIZATION

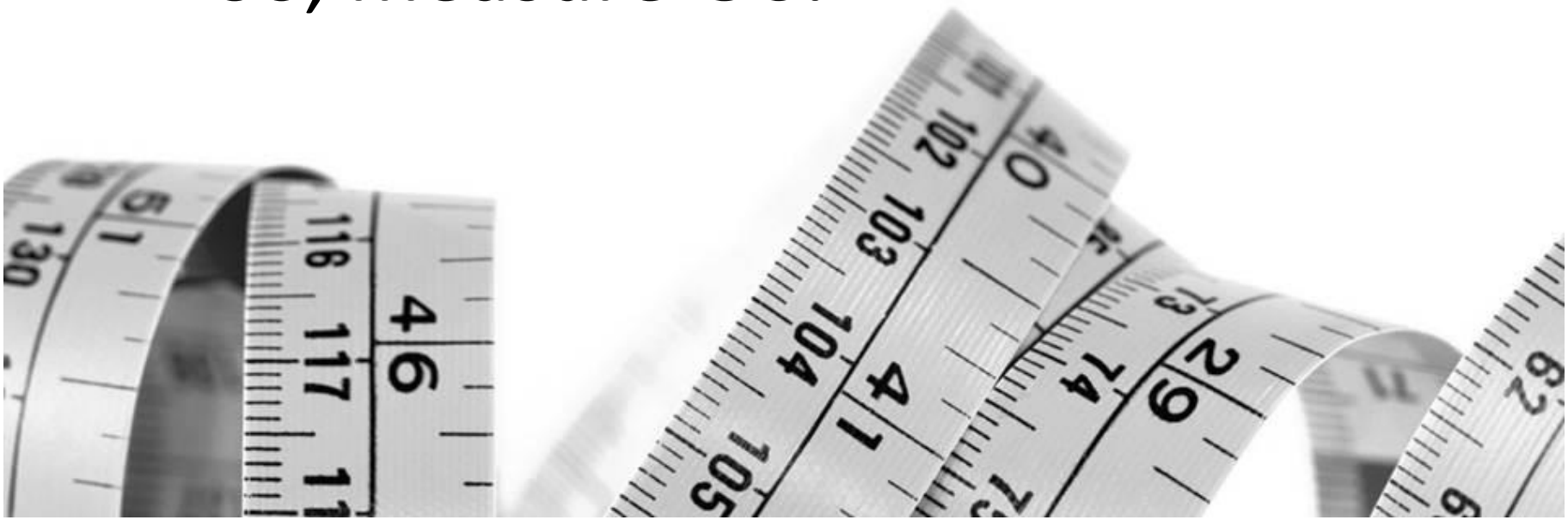
Come on, do it! Do it now! It feels soooo good.

http://odetocode.com/Blogs/scott/archive/2008/07/15/optimizing-linq-queries.aspx

You can't manage
what you can't **measure**

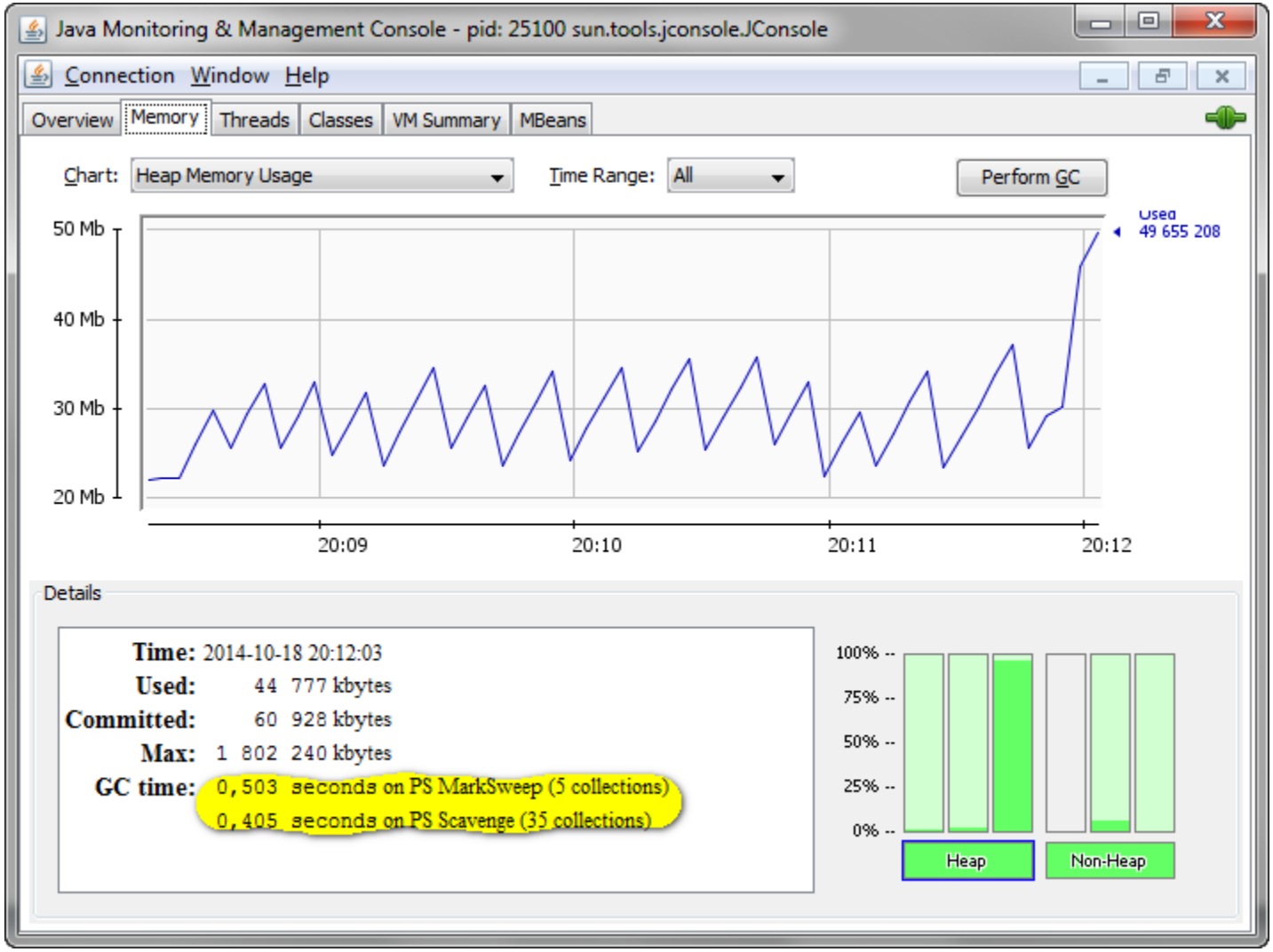
Peter Drucker

So, measure GC!



http://clearlypresentable.files.wordpress.com/2010/10/manage-measure.jpg

Use the tools…



…or use the logs and APIs

- Always use the following settings in prod:

  -XX:+PrintGCDetails

  -XX:+PrintGCTimeStamps

  – So, you can always figure out % time spent in GC by looking at your stdout.

- To figure it out programmatically, see java.lang.management.GarbageCollectorMXBean

  Shameless commercial plug:

  That is the way we do it **MARS** product

How much is much?

- 10%+ time in GC – you start worrying
- 30%+ time in GC – you do something about it

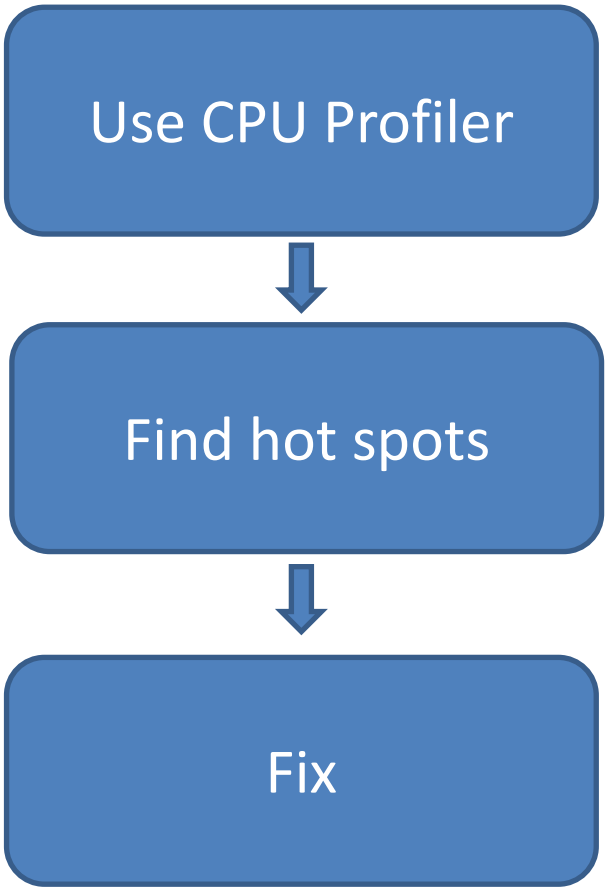
http://upload.wikimedia.org/wikipedia/commons/4/4f/Scheibenbremse-magura.jpg

BLAME

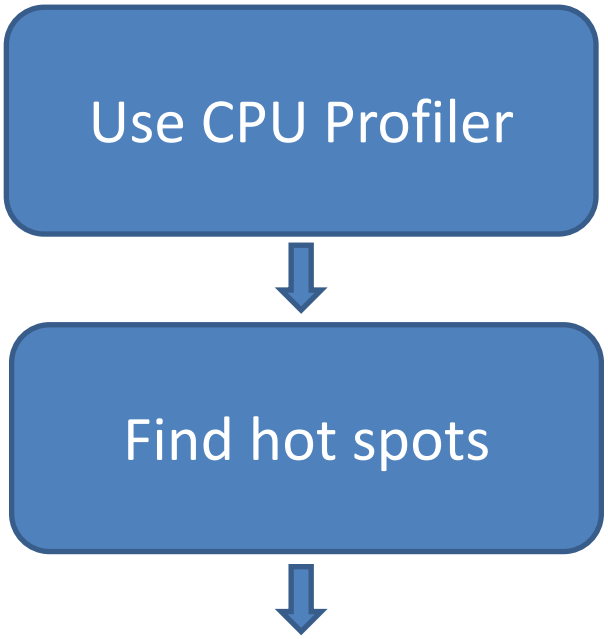
THE SECRET TO SUCCESS IS KNOWING WHO TO BLAME FOR YOUR FAILURES.

Troubleshooting

C/C++

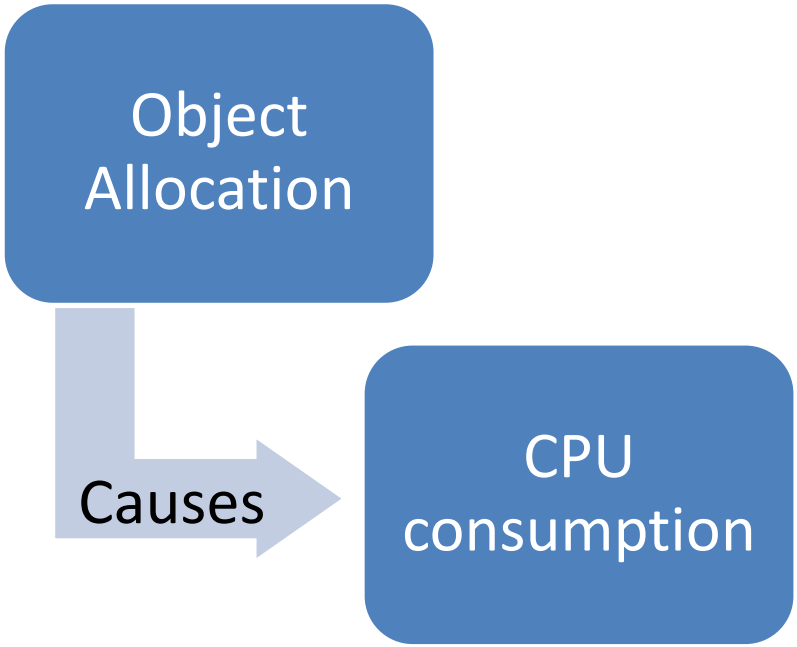


Java



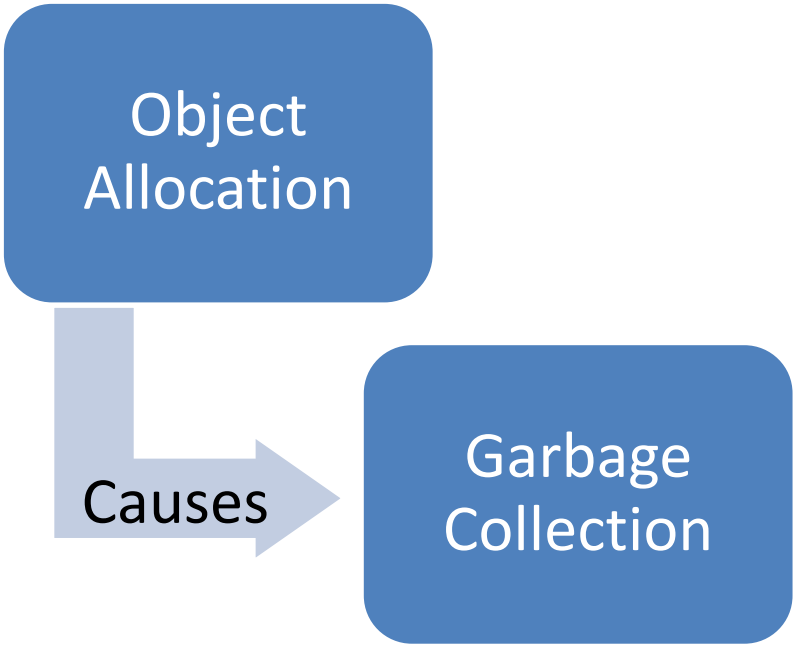
Wait! Allocation is FAST!
(does not consume CPU)

C/C++



malloc/free cost

Java



Memory allocation profiling

- "-Xaprof" option in **old** JVM (<= 1.6)
  - Prints something like this *on process termination*:

```
Allocation profile (sizes in bytes, cutoff = 0 bytes):

____________Size__Instances__Average__Class_______________
      555807584   34737974        16  java.lang.Integer
         321112       5844        55  [I
         106104        644       165  [C
          37144         63       590  [B
          13744        325        42  [Ljava.lang.Object;
… <the rest>
```

Top alloc'd

That is where **aprof** project got its name from

Where memory is allocated?

Where memory is allocated? (1)

Fundamental ways to allocate memory

Since Java 1.0

- **Java**
  - **new** CName(…)
  - **new** <prim>[…]
  - **new** CName[…]
  - **new** CName[…][…]…[…]

Allocate memory

- **Bytecode**
  - **new**
  - ***newarray***
  - ***anewarray***
  - ***multianewarray***

Quiz for audience: What else?

Where memory is allocated? (2)

Boxing – syntactic sugar to allocate memory

Since Java 5

- Integer i = 1234;

- Integer i = Integer.valueOf(1234);

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

"Enhanced for loop" also desugars to method calls

Quiz for audience: What else?

Where memory is allocated? (3)

- java.lang.Object.clone
  - Yet another true way of object allocation
- Reflection & deserialization
  - Gets compiled into bytecode after a few invokes

    but Array.newInstance needs separate care
- sun.misc.Unsafe.allocateInstance
  - Could be tracked, but is not in current aprof
- JNI
  - Can be tracked via native JVMTI

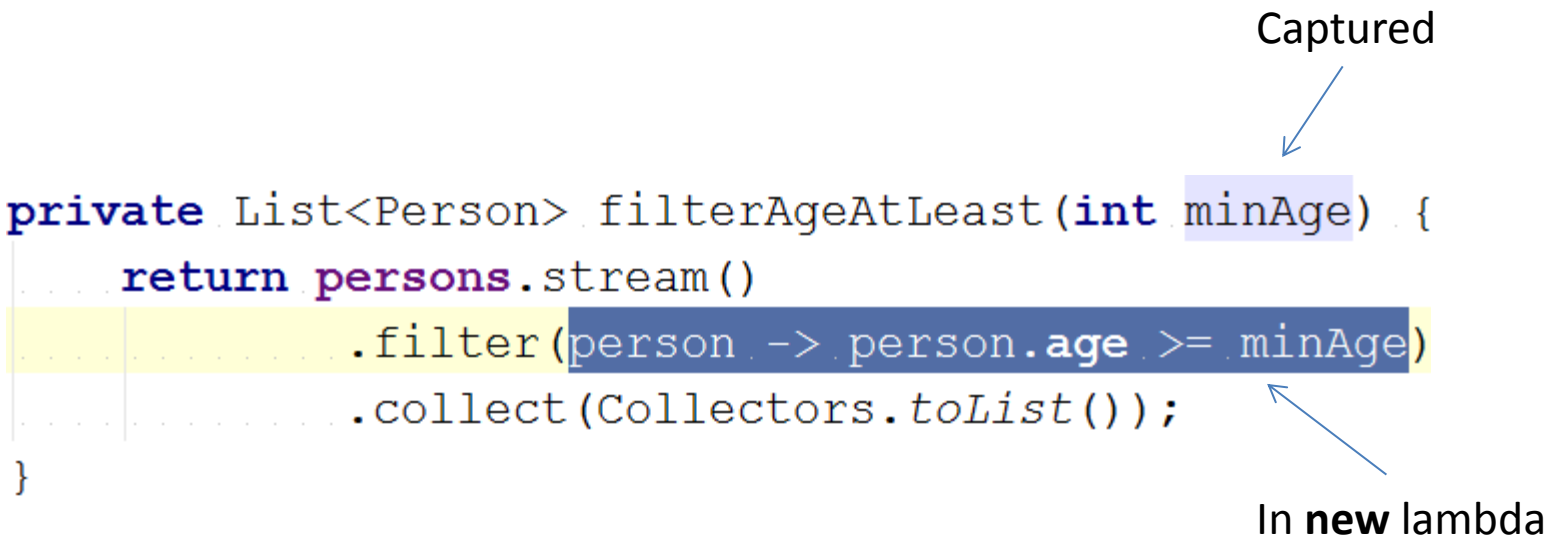
    aprof is a pure Java tool now, does not track it

Quiz for audience: What else?

Where memory is allocated? (4)

Capturing Java 8 Lambda Closure

Since Java 8



Let's instrument bytecodes

```
java –javaagent:aprof.jar …
```

JVM API to write bytecode instrumenting agents in Java

Define Premain-Class

aprof.jar!META-INF/MANIFEST.MF

```
Premain-Class: com.devexperts.aprof.AProfAgent
Boot-Class-Path: aprof.jar
Can-Redefine-Classes: true
```

Install transformer

Ah… The secret sauce!

```
public static void premain(String agentArgs, Instrumentation inst)
```

```
inst.addTransformer(transformer);
```

```
public class AProfTransformer implements ClassFileTransformer {
   public byte[] transform(ClassLoader loader,
       String internalClassName, Class<?> classBeingRedefined,
       ProtectionDomain protectionDomain, byte[] classFileBuffer)
       throws IllegalClassFormatException
```

That is what we need!

Manipulate bytecode with ASM

- ObjectWeb ASM is an open source lib to help
  - Easy to use for bytecode manipulation
  - Extremely fast (suited to on-the-fly manipulation)

Instrument around bytecodes

```
public void visitTypeInsn(int opcode, String desc) {
    switch (opcode) {
        case Opcodes.NEW:
            visitAllocateBefore(desc);
            mv.visitTypeInsn(opcode, desc);
            visitAllocateAfter(desc);
            break;
        case Opcodes.ANEWARRAY:
            String arrayDesc = desc.startsWith("[") ?
                    "[" + desc : "[L" + desc + ";";
            visitAllocateArrayBefore(arrayDesc);
            mv.visitTypeInsn(opcode, desc);
            visitAllocateArrayAfter(arrayDesc);
            break;
        default:
            mv.visitTypeInsn(opcode, desc);
    }
}
```

Generate calls to profiling methods

New array bytecode

Needs length to compute object size

Index of location

```
protected void visitAllocateArrayBefore(String desc) {
    mv.dup(); // keep array size to be allocated
    pushAllocationPoint(desc);
    Type type = Type.getType(desc);
    Type elementType = type.getElementType();
    String name = elementType.getSort() == Type.OBJECT ||
        elementType.getSort() == Type.ARRAY ? "object" :
        elementType.getClassName();
    mv.visitMethodInsn(Opcodes.INVOKESTATIC,
        context.getAprofOpsImplementation(),
        name + "AllocateArraySize",
        TransformerUtil.INT_INT_VOID, false);
}
```

invoke static profiling method

Generate calls to profiling methods

Regular new object bytecode

```
protected void visitAllocateBefore(String desc) {
    pushAllocationPoint(desc);
    pushClass(desc);
    mv.visitMethodInsn(Opcodes.INVOKESTATIC,
        context.getAprofOpsImplementation(),
        "allocateSize",
        TransformerUtil.INT_CLASS_VOID, false);
}
```

Needs class to compute object size

Example transformation

```
public int[] newArray(int n) {
    return new int[n];
}
```

```
public int[] newArray(int);
  Code:
     0: iload_1
     1: newarray       int
     3: areturn
```

```
java -javaagent:aprof.jar=dump.classes.dir=dump …
```

```
public int[] newArray(int);
   Code:
      0: iload_1
      1: dup
      2: sipush        4137
      5: invokestatic  #31
      8: newarray       int
     10: areturn
```

Assigned index

Method com/devexperts/aprof/AProfOps.intAllocateArraySize

return new int[n];

We cannot measure the system without affecting it

Need to minimize measurement effect

Measure garbage without producing garbage (do not allocate memory)

Garbage-free code?

- [ ] Class-transformation
  - Only during load (don't care)
- [ ] Object-class size computation
  - Only once per class (don't care)
- [x] Count individual allocations
  - It **has to be** garbage-free
    - and it is (once all allocation locations were visited)

Count all allocations

```
public static void objectAllocateArraySize(int length, int index) {
    if (length < 0) return; // will not allocate anyway
    IndexMap map = getRootIndex(index);
    map.incrementArraySizeAndCount(length,
            objectArraySize(length));
}
```

Uses **fast hash** to keep one object per index; (http://elizarov.livejournal.com/tag/hash)
Index enumerates (location, type) pairs

```
public static void allocateSize(int index, Class objectClass) {
    RootIndexMap rootIndex = getRootIndex(index);
    rootIndex.incrementCount();
    DatatypeInfo datatypeInfo = rootIndex.getDatatypeInfo();
    if (datatypeInfo.getSize() == 0)
        datatypeInfo.setSize(getObjectSizeByClass(objectClass));
}
```

Compute object size

Array size

```
private static long OBJECT_BASE_OFFSET =
        UnsafeHolder.UNSAFE.arrayBaseOffset(Object[].class);
private static long OBJECT_INDEX_SCALE =
        UnsafeHolder.UNSAFE.arrayIndexScale(Object[].class);
```

```
public static long objectArraySize(int length) {
    long realSize = OBJECT_BASE_OFFSET + OBJECT_INDEX_SCALE * length;
    return (realSize + 7) & ~7L; // align
}
```

… very fast computation

Compute object size

Regular object size

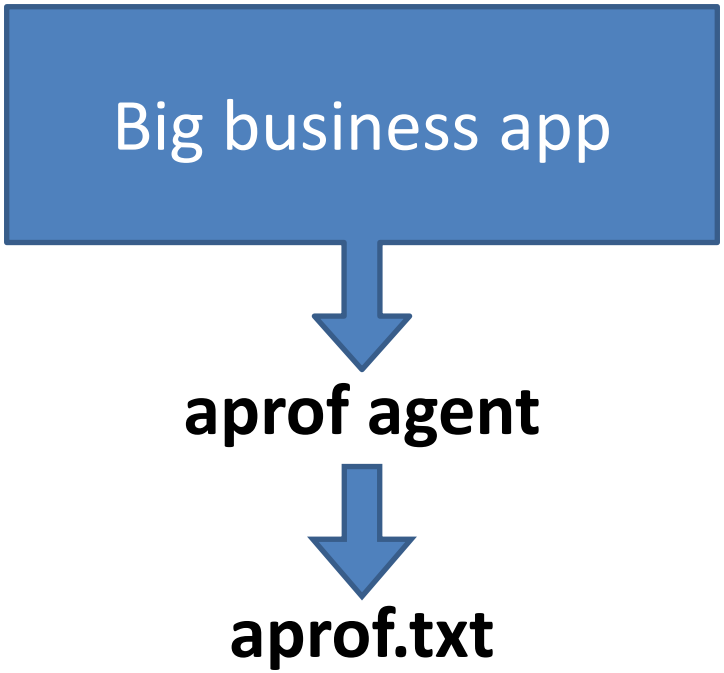
```
public static long getObjectSizeByClass(Class objectClass) {
    long size;
    try {
        size = inst.getObjectSize(
            UnsafeHolder.UNSAFE.allocateInstance(objectClass));
    } catch (InstantiationException e) {
        size = -1;
    }
    return size;
}
```

Precise and actual size (!)

… and **cache** the size for class

Let’s try it

http://3.bp.blogspot.com/-vUgjzK5P-kQ/ToYOTLjqBTI/AAAAAAAAFu0/2ZGrGV9y0c4/s640/tumblr_komtamvGna1qzvjtno1_500.jpg




```
Top allocated data types with locations
-----------------------------------------------------------------
char[]: 330,657,880 bytes in 5,072,613 objects
       java.util.Arrays.copyOfRange: 131,299,568 bytes in …
```

Oops! That is not informative

We knew that it will produce a lot of char[] garbage in strings!

Need allocation context

```
public String getData(int i) {
    return "my data is " + i;
}
```

```
public String getData(int i) {
    return new StringBuilder()
        .append("my data is ")
        .append(i)
        .toString();
}
```

Call stack:

- MyDataClass.getData(int)
- java.lang.StringBuilder.toString()
- Java.lang.String.String(char[], int, int)
- Java.util.Arrays.copyOfRange(char[], int, int[])
- new char[]

We want this location

But it is allocated here

Aprof Tracked Methods

```
java –jar aprof.jar export details.config
```

details.config

```
…
java.lang.StringBuilder
     <init>
     append
     appendCodePoint
     ensureCapacity
     insert
     setLength
     subSequence
     substring
     trimToSize
     toString
…
```

Java RT methods that might allocate memory

How it is done?

- Tracked via thread-local instance of class **com.devexperts.aprof.LocationStack**
- Local variable is injected into
  - methods that allocate memory
  - methods that invoke tracked methods
  - tracked methods themselves
- Local variable is initialized on first use

Example transformation (actual)

```
public int[] newArray(int n) {
    return new int[n];
}
```

```
public int[] newArray(int);
   Code:
      0: aconst_null
      1: astore_2
      2: iload_1
      3: dup
      4: aload_2
      5: dup
      6: ifnonnull     15
      9: pop
     10: invokestatic  #25
     13: dup
     14: astore_2
     15: sipush        4137
     18: invokestatic  #31
     21: newarray       int
     23: areturn
```

LocationStack stack = null; // local var

If (stack == null)
    stack = LocationStack.get();

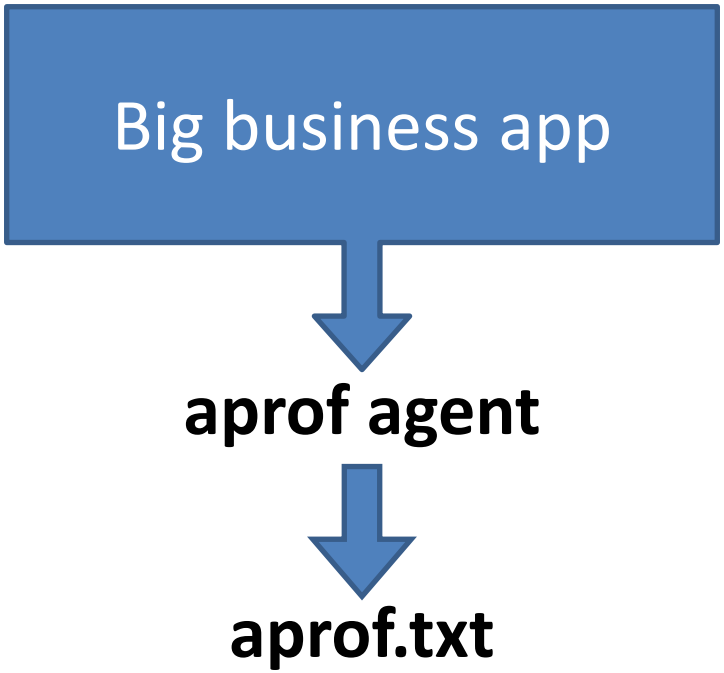
return new int[n];

Looks scary

… But fast in practice

- Long-running methods retrieve LocationStack from ThreadLocal only once
- HotSpot optimizes this code quite well
- It only affects code that does memory allocations or uses tracked (memory allocating!) methods
- Does not allocate memory during stable operation

It has no performance impact on dense garbage-free computation code, various getters, etc




```
Top allocated data types with reverse location traces
-----------------------------------------------------
char[]: 330,657,880 bytes in 5,072,613 objects
       java.util.Arrays.copyOfRange: 131,299,568 bytes in …
              java.lang.StringBuilder.toString: …
                     MyBusinessMethod: …
                     … (more!)
```

Got you!

Aprof dump vs actual call stack

Actual call stack:

…

- **MyBusinessMethod**
- **java.lang.StringBuilder.toString()**
- Java.lang.String.String(char[], int, int)
- **Java.util.Arrays.copyOfRange(char[], int, int[])**
- **new char[]**

(4) Caller of the outermost tracked method

(3) Outermost tracked method

(2) Where it was allocated

(1) Type that was allocated

```
Top allocated data types with reverse location traces
-----------------------------------------------------
char[]: 330,657,880 bytes in 5,072,613 objects
        java.util.Arrays.copyOfRange: 131,299,568 bytes in …
                java.lang.StringBuilder.toString: …
                        MyBusinessMethod: …
                        … (more!)
```

(1) (2) (3) (4)

At most 4 items are displayed in aprof dump

But what if

… caught **MyFrameworkMethod** allocating memory instead?

Add it to tracked methods list

a) `java -javaagent:aprof.jar=track=MyFrameworkMethod …`

b) `java -javaagent:aprof.jar=track.file=details.config …`

Repeat

Poor man’s catch-all

```
java –javaagent:aprof.jar=+unknown …
```

Injects the profiling code right into java.lang.Object constructor

- Does some mental accounting to exclude allocation that were already counted, reports the difference
- The difference can appear from JNI object allocations. But location is unknown anyway.
- Does not help with tracking JNI array allocations at all (no constructor invocation)


HotSpot strikes back

```java
private void sumALot(List<Integer> list) {
    for (int i = 0; i < 1000000000; i++)
        sum += sumList(list);
}

private int sumList(List<Integer> list) {
    int sum = 0;
    for (Integer j : list)
        sum += j;
    return sum;
}
```

Iterator is allocated here

```
Top allocated data types with reverse location traces
-----------------------------------------------------
java.util.ArrayList$Itr: 32,000,006,272 bytes …
      java.util.ArrayList.iterator: 32,000,006,272 bytes …
            IterateALot.sumList: 32,000,000,000 bytes …
```

Shouldn’t GC work like hell?

```
java -XX:+PrintGCDetails -XX:+PrintGCTimeStamps …
```

```
[PSYoungGen: 512K->400K(1024K)] 512K->408K(126464K)
[PSYoungGen: 912K->288K(1024K)] 920K->296K(126464K)
[PSYoungGen: 800K->352K(1024K)] 808K->360K(126464K)
[PSYoungGen: 864K->336K(1536K)] 872K->344K(126976K)
[PSYoungGen: 1360K->352K(1536K)] 1368K->360K(126976K)
[PSYoungGen: 1376K->352K(2560K)] 1384K->360K(128000K)
[PSYoungGen: 2400K->0K(2560K)] 2408K->284K(128000K)
[PSYoungGen: 2048K->0K(4608K)] 2332K->284K(130048K)
```

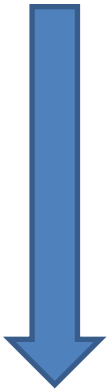
And that is it! No more GC!

What is going on here?

HotSpot allocation elimination

Aprof allocation elimination checking

```
java –javaagent:aprof.jar:+check.eliminate.allocation
-XX:+UnlockDiagnosticVMOptions -XX:+LogCompilation …
```



1. HotSpot writes hs_xxx.log files
2. Aprof parses them to learn eliminated allocation locations

```
Top allocated data types with reverse location traces
-----------------------------------------------------
java.util.ArrayList$Itr: 32,000,006,272 bytes …
      java.util.ArrayList.iterator: 32,000,006,272 bytes …
            IterateALot.sumList: … ; possibly eliminated
```

Additional options/features

- **histogram** – track array allocation separately for different size brackets
- **file** – configure dump file, use #### in file name to auto-number files
- **file.append** – append dump to file every, instead of overwriting
- **time** – time period to write dumps, defaults to a minute

Advanced topics

- Profiling the profiler
  - **aprof** records and reports its own allocations
- Retransforming classes that were loaded before **aprof** Pre-Main had even got control
- Caching of class meta-information for each class-loader
  - for fast class transformation
- Aggregate classes with dynamic names like "sun.reflect.GeneratedConstructorAccessorX"
  - to avoid memory leaks (out of memory)

The source

https://github.com/devexperts/aprof

GPL 3.0

Your contributions are welcome

Known issues

Where you can help

- Lambda capture memory allocations are not tracked nor reported in any way
  - Need to track metafactory calls and unnamed classes it generates
- Java 8 library methods (collections, streams, etc) are not included into default list of tracked methods
  - Need to work through them and include them
- JNI allocations are not properly tracked
  - Need to write native JVMTI agent to track them

Questions?
Feedback?

aprof@devexperts.com
elizarov@devexperts.com