HotSpot Under the Hood

Alex Blewitt
@alblue

Docklands.LJC presentation – June 2016

Something to talk about

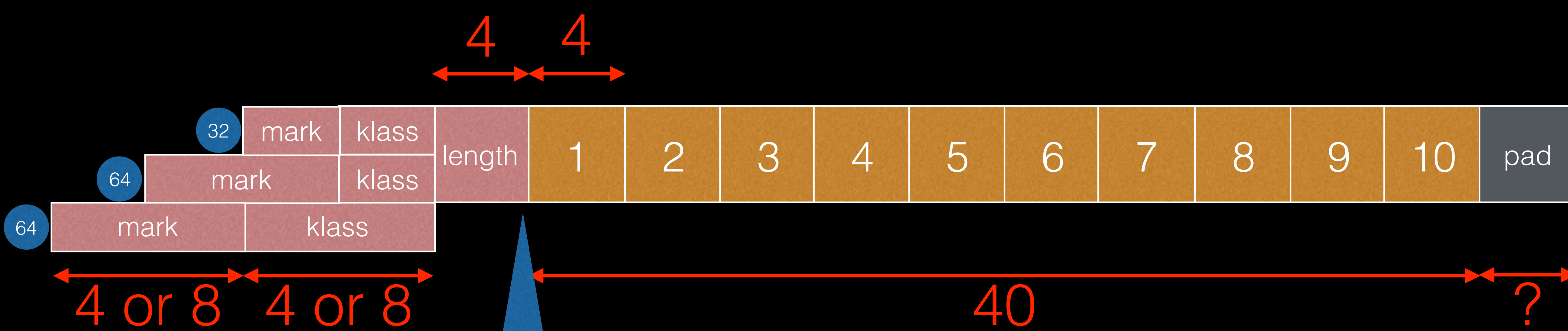
- Need a code sample to talk about

```
int thing[] = new int[10];

int sum(int[] thing) {
  int total = 0;
  for(int t : thing) {
    total += t;
  }
  return total;
}
```

int[] thing

- Arrays are variable sized objects on the heap



Types may also have padding for data alignment

Objects are multiples of 8*

16, 24, 32, 40, 48, 56, 64 …

* when object alignment is 8

Klass field

- The klass field is a pointer to the object's type
  - Think `getClass()` in Java …
- Present for every object/array instance
- Can be 4 or 8 bytes wide
  - 32 bit JVM - 4 bytes
  - 64 bin JVM - 4 bytes or 8 bytes

Klass field can be compressed

64 Compressed OOPS

- Compressed Ordinary Object Pointers
  - Store an object reference in 32 bits

F 0 0 2	zero extend	0 0 0 0 F 0 0 2	< 4G
F 0 0 2	shift extend	0 0 0 7 8 0 1 0	< ~30G
F 0 0 2	shift + base	0 0 1 7 8 0 1 0	< 32G

```
-XX:+/-UseCompressedOops
-XX:+/-UseCompressedClassPointers
-XX:ObjectAlignmentInBytes=8
```

=16	0 0 1 F 0 0 2 0	< 64G

64

Compressed OOPS

- Handled efficiently by generated code
  - In many cases, don't need to expand
  - Uses addressing modes to pack/unpack

```
mov 0xc(%r12,%r10,8),%r11d
r11 = *(r10 * 8 + r12 + 12)
```

Address in memory

Field offset

Compressed OOP

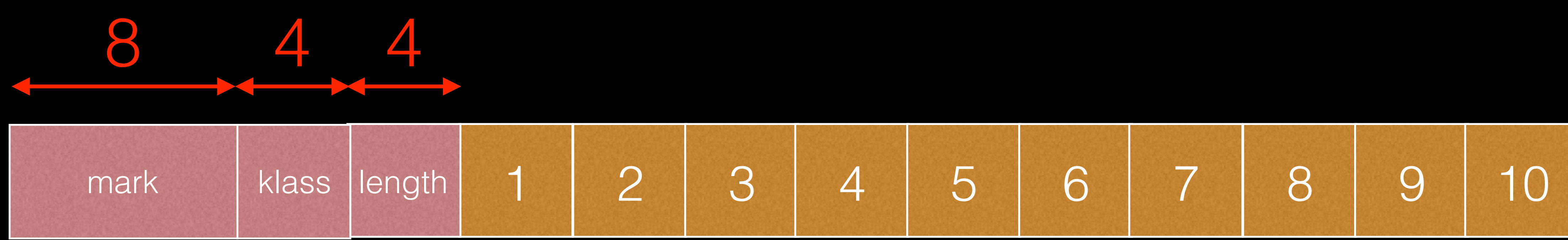
r12 is Heap base

* when object alignment is 8

64

Array length

- Getting the length of an array




```
length = *( r10 * 8 + r12 + 12)
```

address

+12 0xC

<< shift + base

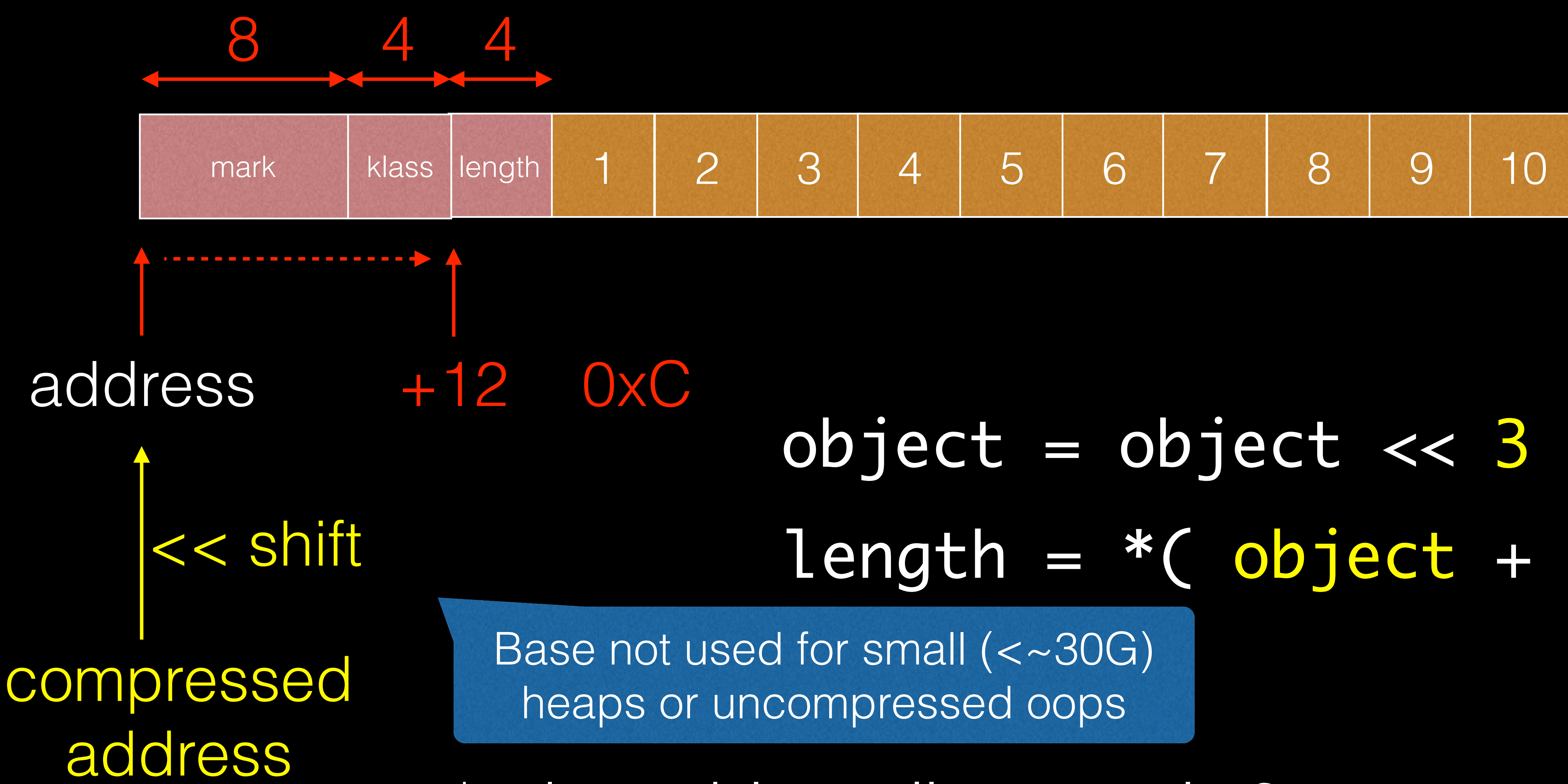
compressed address

Base used for large (>~30G) heaps with compressed oops

* when object alignment is 8

Array length

- Getting the length of an array



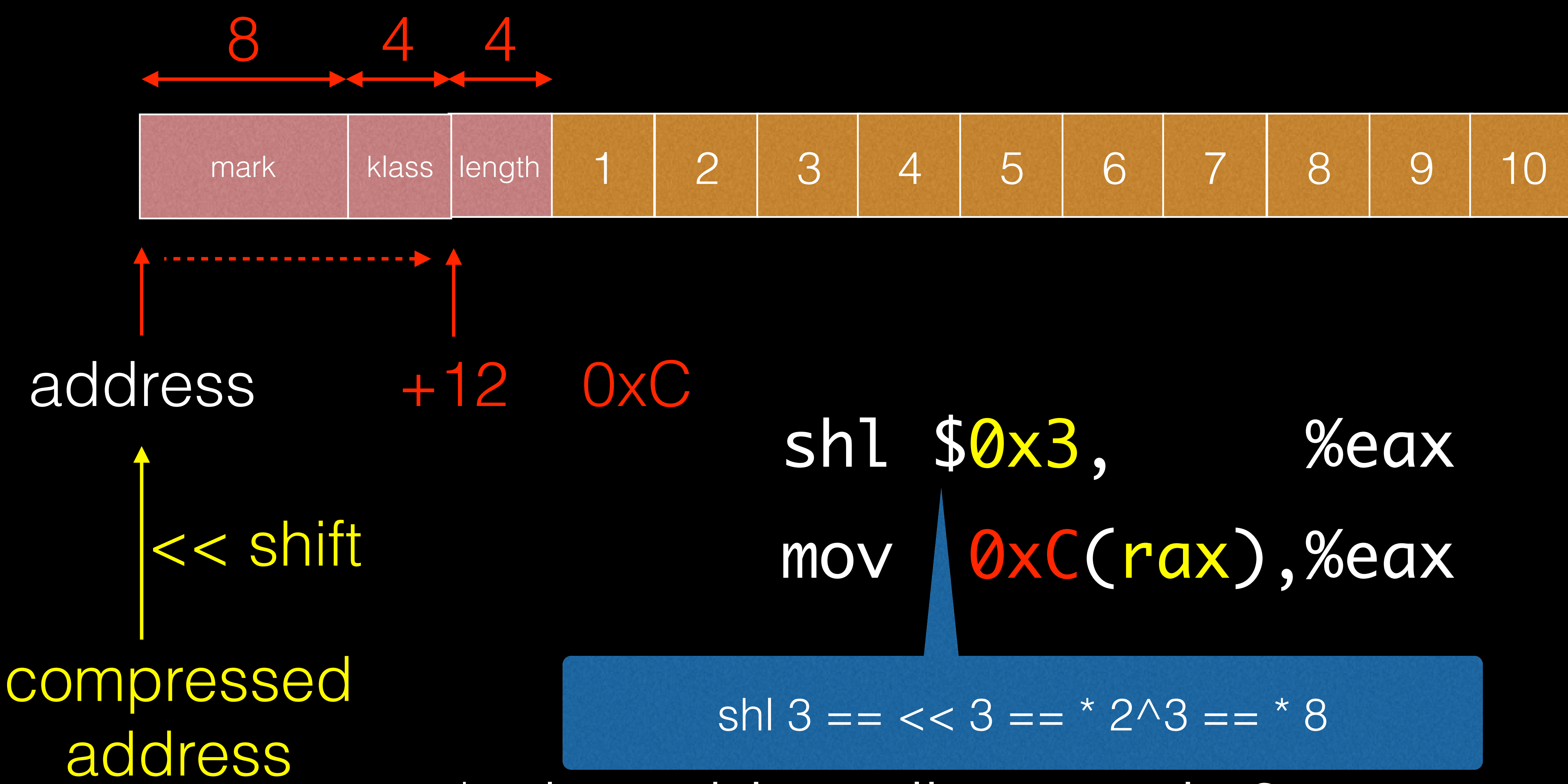

```
object = object << 3
length = *( object + 12)
```

Base not used for small (<~30G) heaps or uncompressed oops

* when object alignment is 8

Array length

- Getting the length of an array




```
shl $0x3,       %eax
mov  0xC(rax),%eax
```

shl 3 == << 3 == * 2^3 == * 8

* when object alignment is 8

Bytecode

- JavaC translates Java to bytecode
  - Stack-based byte oriented code
  - Local vars `istore_1`
  - Object loads `aload_2`
  - Array length `arraylength`

```
 0: iconst_0
 1: istore_1
 2: aload_0
 3: astore_2
 4: aload_2
 5: arraylength
 6: istore_3
 7: iconst_0
 8: istore        4
10: iload         4
12: iload_3
13: if_icmpge     33
16: aload_2
17: iload         4
19: iaload
20: istore        5
22: iload_1
23: iload         5
25: iadd
26: istore_1
27: iinc          4, 1
30: goto          10
33: iload_1
34: ireturn
```

Bytecode execution

- HotSpot uses `-XX:+TieredCompilation`
  - Starts off with interpreter
  - Hot spots get compiled as they get executed
- JIT compilers
  - C1 (aka `-client`)
  - C2 (aka `-server`)

Interpreter

- An interpreter sounds simple …

```
switch(bytecode) {
  case nop: break;
  case aconst_null: push(null); break;
  case iconst_m1:   push(-1); break;
  case iconst_0:    push(0); break;
  case iconst_1:    push(1); break;
  …
}
```

Template Interpreter

- HotSpot uses a *template interpreter*

```
Runnable[] ops = new Runnable[] {
  () -> {},
  () -> push(null),
  () -> push(-1),
  () -> push(0),
  () -> push(1),
  …
}
```

```
ops[index++].run()
```

* this is a Java approximation only

Template Interpreter

- Assembly, dumped with -XX:+PrintInterpreter

```
arraylength  190
  0x000000001068fe9a0: pop    %rax
  0x000000001068fe9a1: mov    0xc(%rax),%eax
  0x000000001068fe9a4: movzbl 0x1(%r13),%ebx
  0x000000001068fe9a9: inc    %r13
  0x000000001068fe9ac: movabs $0x106293760,%r10
  0x000000001068fe9b6: jmpq   *(%r10,%rbx,8)
  0x000000001068fe9ba: nopw   0x0(%rax,%rax,1)
```

Template Interpreter

- Get address of array into 64-bit **`rax`** register

```
arraylength  190
  0x000000001068fe9a0: pop    %rax
  0x000000001068fe9a1: mov    0xc(%rax),%eax
  0x000000001068fe9a4: movzbl 0x1(%r13),%ebx
  0x000000001068fe9a9: inc    %r13
  0x000000001068fe9ac: movabs $0x106293760,%r10
  0x000000001068fe9b6: jmpq   *(%r10,%rbx,8)
  0x000000001068fe9ba: nopw   0x0(%rax,%rax,1)
```

Template Interpreter

- Load `*(address + 12)` into 32-bit `eax`

```
arraylength  190
  0x00000001068fe9a0: pop    %rax
  0x00000001068fe9a1: mov    0xc(%rax),%eax
  0x00000001068fe9a4: movzbl 0x1(%r13),%ebx
  0x00000001068fe9a9: inc    %r13
  0x00000001068fe9ac: movabs $0x106293760,%r10
  0x00000001068fe9b6: jmpq   *(%r10,%rbx,8)
  0x00000001068fe9ba: nopw   0x0(%rax,%rax,1)
```

Template Interpreter

- Load byte **`*(r13 + 1)`** into 32-bit **`ebx`**; **`r13++`**

```
arraylength  190
  0x00000001068fe9a0: pop    %rax
  0x00000001068fe9a1: mov    0xc(%rax),%eax
  0x00000001068fe9a4: movzbl 0x1(%r13),%ebx
  0x00000001068fe9a9: inc    %r13
  0x00000001068fe9ac: movabs $0x106293760,%r10
  0x00000001068fe9b6: jmpq   *(%r10,%rbx,8)
  0x00000001068fe9ba: nopw   0x0(%rax,%rax,1)
```

* r13 is the bytecode index pointer

Template Interpreter

- Load byte `*(r13 + 1)` into 32-bit `ebx`; `r13++`

```
arraylength  190
  0x000000001068fe9a0: pop    %rax
  0x000000001068fe9a1: mov    0xc(%rax),%eax
  0x000000001068fe9a4: movzbl 0x1(%r13),%ebx
  0x000000001068fe9a9: inc    %r13
```

Logically equivalent to:

```
inc %r13    ; %r13++
movzbl (%r13), %ebx
```

but HotSpot's approach is faster since the naïve implementation would cause a data dependency on `%r13` between the prior instruction and the subsequent one

* r13 is the bytecode index pointer

Template Interpreter

- Load table address **`0x10…60`** into 64-bit **`r10`**

```
arraylength  190
  0x000000001068fe9a0: pop    %rax
  0x000000001068fe9a1: mov    0xc(%rax),%eax
  0x000000001068fe9a4: movzbl 0x1(%r13),%ebx
  0x000000001068fe9a9: inc    %r13
  0x000000001068fe9ac: movabs $0x106293760,%r10
  0x000000001068fe9b6: jmpq   *(%r10,%rbx,8)
  0x000000001068fe9ba: nopw   0x0(%rax,%rax,1)
```

* 0x106293760 is the start of the template table

Template Interpreter

- Jump to `r10 + rbx * 8`

```
arraylength  190
  0x00000001068fe9a0: pop    %rax
  0x00000001068fe9a1: mov    0xc(%rax),%eax
  0x00000001068fe9a4: movzbl 0x1(%r13),%ebx
  0x00000001068fe9a9: inc    %r13
  0x00000001068fe9ac: movabs $0x106293760,%r10
  0x00000001068fe9b6: jmpq   *(%r10,%rbx,8)
  0x00000001068fe9ba: nopw   0x0(%rax,%rax,1)
```

* rbx is the next bytecode loaded earlier

Template Interpreter

- Nop instruction (slightly bigger nop)*

```
arraylength  190
  0x000000001068fe9a0: pop    %rax
  0x000000001068fe9a1: mov    0xc(%rax),%eax
  0x000000001068fe9a4: movzbl 0x1(%r13),%ebx
  0x000000001068fe9a9: inc    %r13
  0x000000001068fe9ac: movabs $0x106293760,%r10
  0x000000001068fe9b6: jmpq   *(%r10,%rbx,8)
  0x000000001068fe9ba: nopw   0x0(%rax,%rax,1)
```

* fills gap until next alignment

Template Interpreter

- Arraylength = *(address of object + `0xc`)

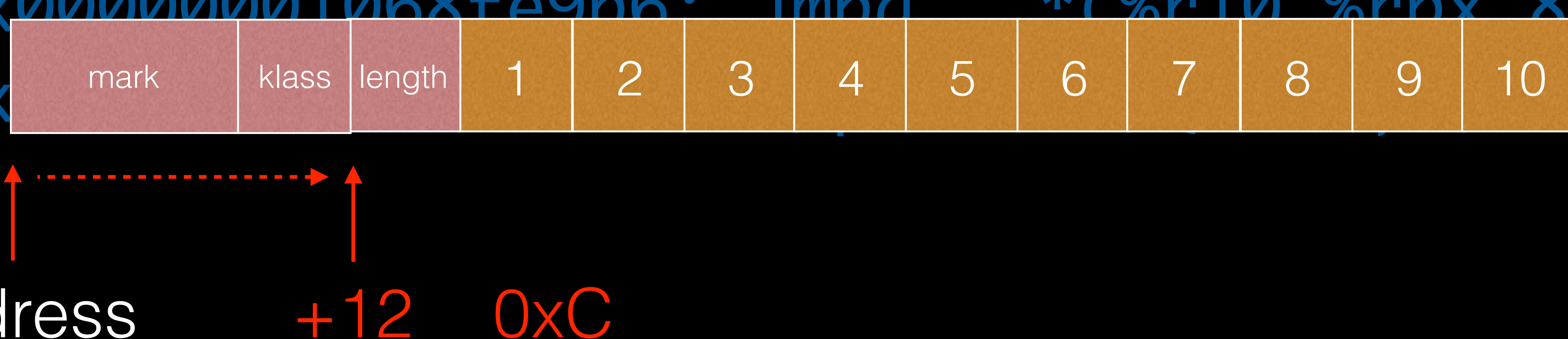
```
arraylength  190
  0x00000001068fe9a0: pop    %rax
  0x00000001068fe9a1: mov    0xc(%rax),%eax
  0x00000001068fe9a4: movzbl 0x1(%r13),%ebx
  0x00000001068fe9a9:
  0x00000001068fe9ac:                93760,%r10
  0x00000001068fe9b6:                %rbx,8)
  0x00000001068fe9ba: nopw   0x0(%rax,%rax,1)
```

This is the key part of the **`arraylength`** bytecode

Template Interpreter

- Arraylength = *(address of object + `0xc`)

```
arraylength  190
  0x0000000001068fe9a0: pop    %rax
  0x0000000001068fe9a1: mov    0xc(%rax),%eax
  0x0000000001068fe9a4: movzbl 0x1(%r13),%ebx
  0x0000000001068fe9a9: inc    %r13
  0x0000000001068fe9ac: movabs $0x106293760,%r10
  0x0000000001068fe9b6: jmpq   *(%r10,%rbx,8)
  0x ... ,1)
```



Null Checks

- Null checks are automatically handled
  - The assembly code is generated from:

```
void TemplateTable::arraylength() {
  transition(atos, itos);
  __ null_check(rax, arrayOopDesc::length_offset_in_bytes());
  __ movl(rax, Address(rax, arrayOopDesc::length_offset_in_bytes()));
}
```

```
0x0000000001068fe9a1: mov     0xc(%rax),%eax
```

If **`rax`** is null, **`*(0+0xc)`** is a deref of a zero page memory location - causes **`SIGSEGV`**

JVM **`SIGSEGV`** handler translates this to **`NullPointerException`**

Top of Stack

- It's a little more complicated than that …
- HotSpot caches top-of-stack in a register
  - Faster access
  - Different register based on type
    - `rax` – long/int/short/char/byte/boolean
    - `xmm0` – double/float
  - Different implementations needed for `pop`

Popping off

Entry points for different types

object → 0x00000001068f5440
float → 0x00000001068f5446
double → 0x00000001068f5454
long → 0x00000001068f5462
int → 0x00000001068f546f

```
pop   87 pop
0x00000001068f5440: push   %rax
0x00000001068f5441: jmpq   0x00000001068f5470
0x00000001068f5446: sub    $0x8,%rsp
0x00000001068f544a: vmovss %xmm0,(%rsp)
0x00000001068f544f: jmpq   0x00000001068f5470
0x00000001068f5454: sub    $0x10,%rsp
0x00000001068f5458: vmovsd %xmm0,(%rsp)
0x00000001068f545d: jmpq   0x00000001068f5470
0x00000001068f5462: sub    $0x10,%rsp
0x00000001068f5466: mov    %rax,(%rsp)
0x00000001068f546a: jmpq   0x00000001068f5470
0x00000001068f546f: push   %rax
0x00000001068f5470: add    $0x8,%rsp
```

Top of Stack state

- The type of value on the top affects entry point

TemplateTable

Byte code	Byte	Bool	Char	Short	Int	Long	Float	Double	Object	Void	
array length	X	X	X	X	X	X	X	X	fe9a0	fe9a0	Entry
pop	f546f	f546f	f546f	f546f	f546f	f546f	f5446	f5454	f5440	f5440	Entry
iadd	f5920	X	f5920	f5920	f5920	X	X	X	X	f5920	Entry
ladd	X	X	X	X	X	f5980	X	X	X	f5908	Entry

Wide and safepoint

- Wide extends certain instructions
  - load i -> load ii, fstore i -> fstore ii
  - iinc i -> iinc ii
- Different table when interpreting 'wide mode'
  - `_template_table`, `_template_table_wide`
- Can be used to implement safepoint
  - Update entry points to use safepoint handler

Fast bytecodes

- Some bytecodes are re-written on the fly
    - getfield -> fast_agetfield, fast_igetfield etc.
    - putfield -> fast_aputfield, fast_iputfield etc.
    - iload -> fast_iload
    - aload_0 -> fast_aload_0

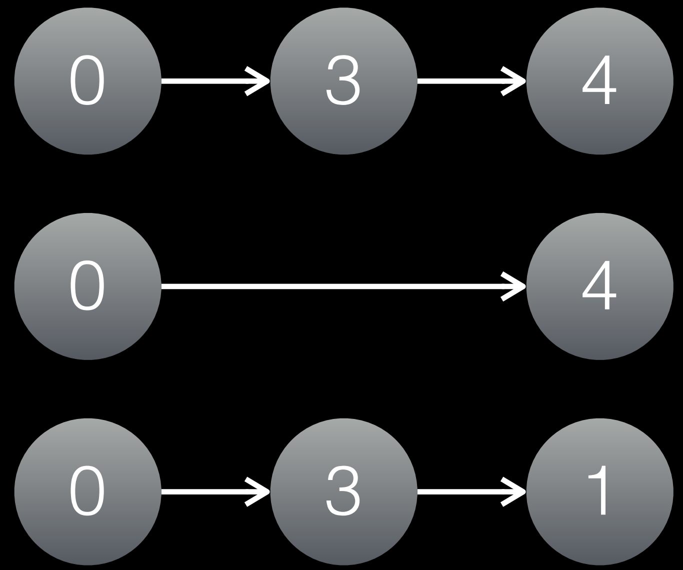
`aload_0` stores `this` for instances

Getting Faster

- Interpreter is fast, but still slower than native
- Methods get compiled at hot spots
- Pipeline for compiled methods at different levels
  - C1
  - C2
- Can be (re)compiled multiple times

Compilation levels

- HotSpot as a number of compilation levels
  - 0 – interpreter
  - 1 – pure C1
  - 2 – C1 with invocation and backedge counting
  - 3 – C1 with full profiling
  - 4 – C2 (full optimisation)



Optimisations

- Optimisations generally occur due to:
  - Method inlining
  - Dead code/path elimination
  - Heuristics for optimising call sites
  - Constant folding
- C2 performs more optimisations

Intrinsics

- Implemented in native code directly
  - Native code included instead of caller

```
InterpreterGenerate::generate_math_entry(kind) {
  switch (kind) {
    case Interpreter::java_lang_math_sin:
      __ trigfunc('s'); break
    case Interpreter::java_lang_math_abs:
      __ fabs(); break;
    …
  }
}
```

Intrinsics

- Implemented in native code directly
  - Native code included instead of caller

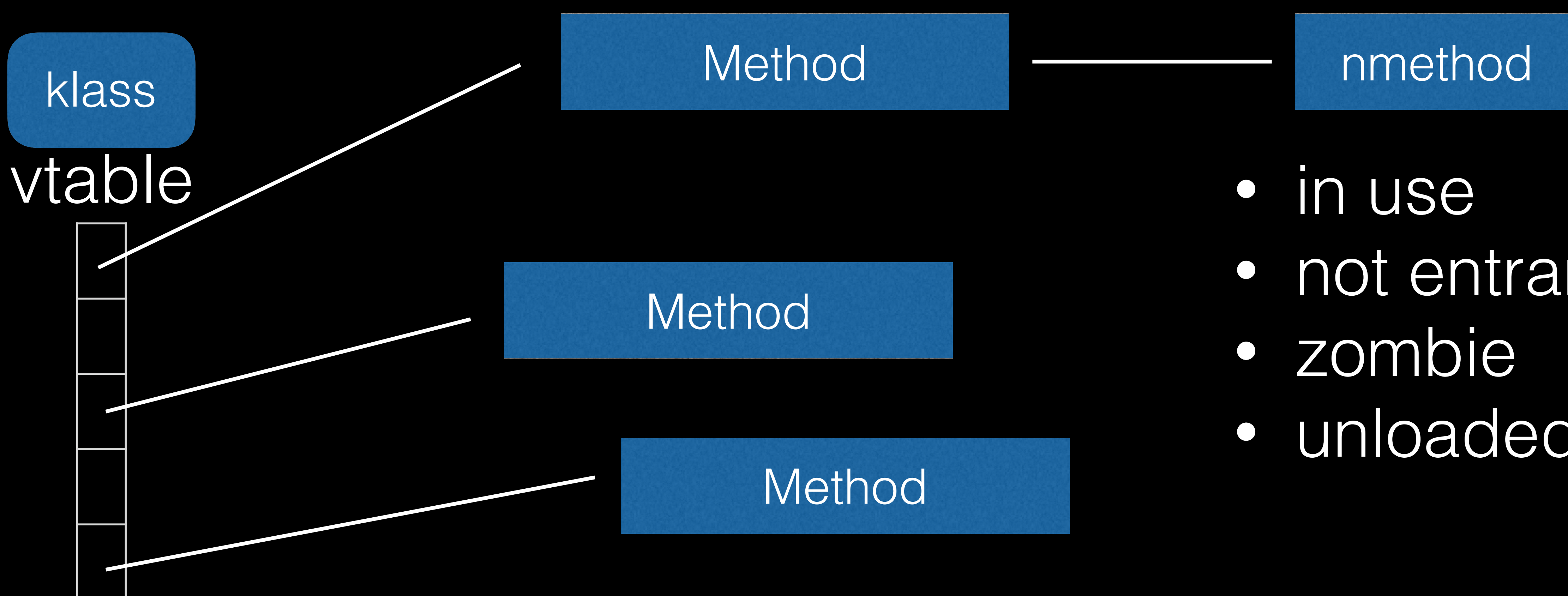
```
LibraryCallKit::inline_math_native(id) {
  switch (id) {
    case vmIntrinsics::_dsin:
      inline_trig(id); break
    case vmIntrinsics::_dabs:
      inline_math(id); break;
    …
  }
}
```

Common intrinsics

- Thread.currentThread()
- System.arrayCopy()
- System.clone()
- System.nanoTime(), currentTimeMillis()
- String.indexOf()
- Math.*

Calling Methods

- Code looks up method through vtable
  - klazz.vtable[id].method -> native code



- in use
- not entrant
- zombie
- unloaded

Morphism

- Most methods are invoked on a single type
  - Not final, but only one class seen
  - Method records the first type, assumes mono
  - Can be specialised into bimorphic
  - Falls back to slow path

Verified Entry Point

- Code has an entry point and verified entry point
  - Entry point is where code starts

```
Code:
[Entry Point]
  # {method} {0x000000011a54b000} 'hashCode' in 'java/lang/String'
  #           [sp+0x40]  (sp of caller)
  0x0000000001067dac80: mov    0x8(%rsi),%r10d
  0x0000000001067dac84: shl    $0x3,%r10
  0x0000000001067dac88: cmp    %rax,%r10
  0x0000000001067dac8b: jne    0x000000010671fb60  ;   {runtime_call}
  0x0000000001067dac91: data32 data32 nopw 0x0(%rax,%rax,1)
  0x0000000001067dac9c: data32 data32 xchg %ax,%ax
[Verif
```

rsi is the String instance
rsi+8 is klass

shl3 == *8
Expanding compressed oop

Fall back if not correct

rax is the expected type (String)

Verified Entry Point

- Code has an entry point and verified entry point
    - Verified Entry point is where type holds

```
Code:
[Entry Point]
…
[Verified Entry Point]
  0x000000001067daca0: mov    %eax,-0x14000(%rsp)
  0x000000001067daca7: push   %rbp
  0x000000001067daca8: sub    $0x30,%rsp
  0x000000001067dacac: movabs $0x11a70ccb0,%rax
   ; {metadata(method data for {method}
   ;  {0x000000011a54b000} 'hashCode' '()I' in 'java/lang/String')}
  0x000000001067dacb6: mov    0xdc(%rax),%edi
  0x000000001067dacbc: add    $0x8,%edi
  0x000000001067dacbf: mov    %edi,0xdc(%rax)
```

Stack banging/
StackOverflowError

Method data for
String's hashCode
implementation

Recompilation

- Methods get recompiled frequently
- Use -XX:+PrintCompilation to see when
  - % osr = on stack replacement

```
70    1     n 0       java.lang.System::arraycopy (native)   (static)
71    2       3       java.lang.Object::<init> (1 bytes)
73    3       3       java.lang.String::hashCode (55 bytes)
75    5       3       java.lang.String::charAt (29 bytes)
76    6       3       java.lang.String::length (6 bytes)
76    7       3       java.lang.String::indexOf (70 bytes)
76    4       3       java.lang.Math::min (11 bytes)
76    9       1       java.lang.Object::<init> (1 bytes)
76    2       3       java.lang.Object::<init> (1 bytes)   made not entrant
```

Summary

- HotSpot has lots of optimisations
- Lots of routines are generated with assembly
- Native code is modified at runtime
  - Assumptions about target types
  - In-lining for performance
- Interpreter, C1 and C2 generate different code

HotSpot Under the Hood

Alex Blewitt
@alblue

Docklands.LJC presentation – June 2016