The Shape of an Object

Optimizing for space and time in IBM's J9 Java VM

Peter Burka

peter.burka@twosigma.com

December 11, 2014

J9

- IBM's Java virtual machine
- Cleanroom implementation
- Originally an embedded JVM
- Used by thousands of IBM customers

** Disclaimer: I'm not speaking on behalf of IBM or Two Sigma*

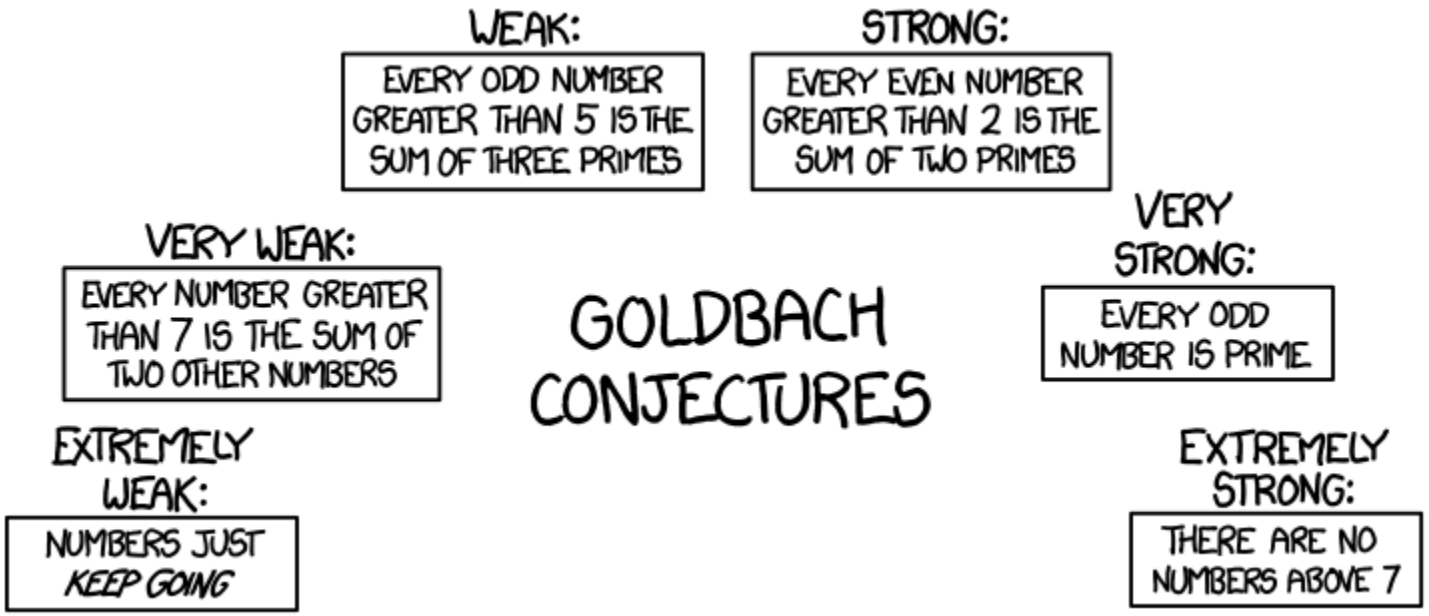
Objects

- Object = a value stored in memory
- Design questions:
  - Interoperability (ABI)
  - Identity and mutability (==, =)
  - Dynamic vs. static shape
  - Size / Speed
  - Subtyping / multiple inheritance
  - Reflection
  - Architectural considerations (e.g. alignment)

Some options

- Dictionaries *(Python, Javascript)*
  - Completely dynamic
- 2-Tuples *(Lisp)*
  - Uniformly size
- Contiguous structs *(C, C++, Fortran, Java)*
  - Space and time efficient
  - Shape known early

A conjecture...



http://xkcd.com/1310/

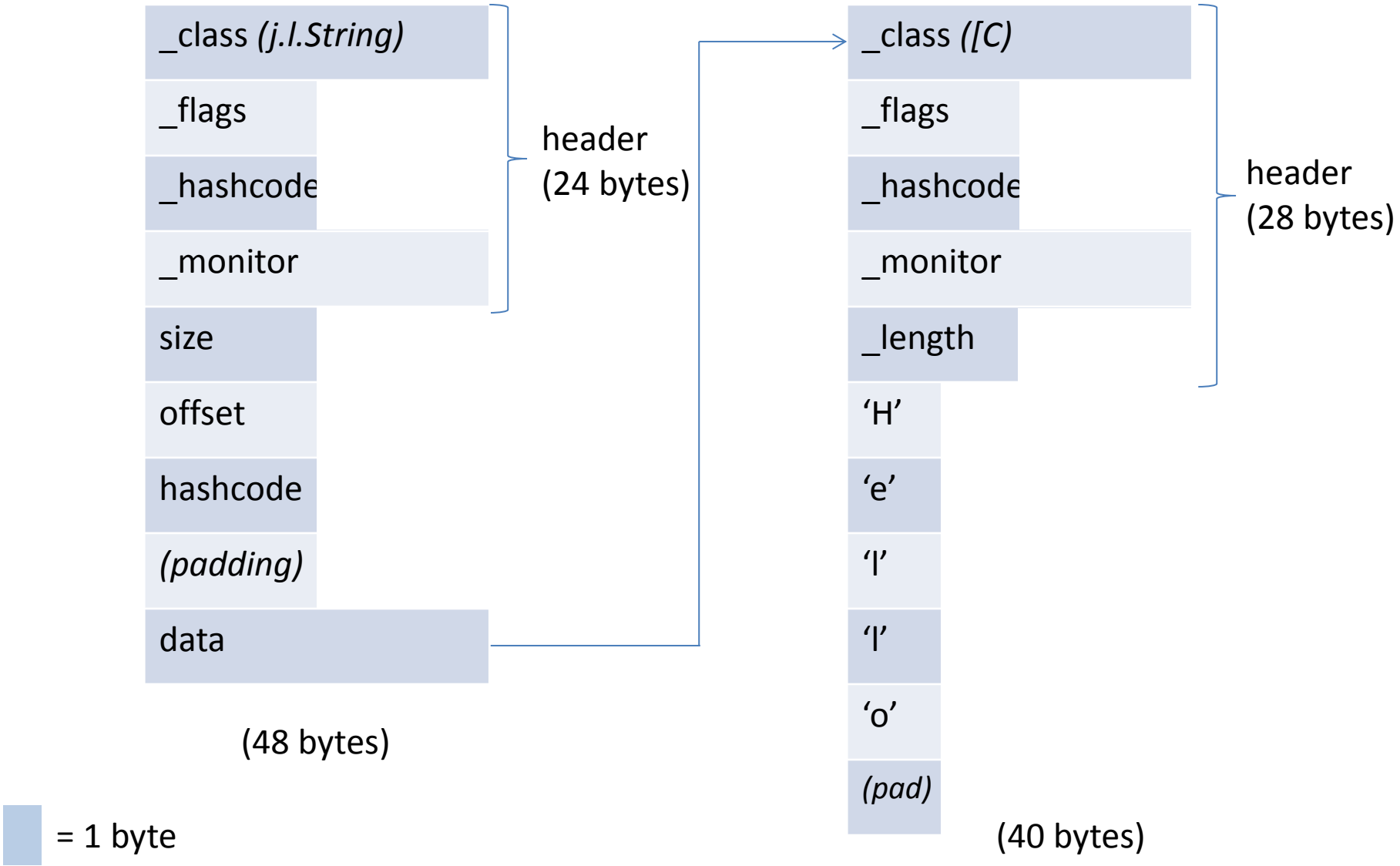
A conjecture...

- All objects are strings

A conjecture...

- All objects are strings
- And the remainder are arrays.

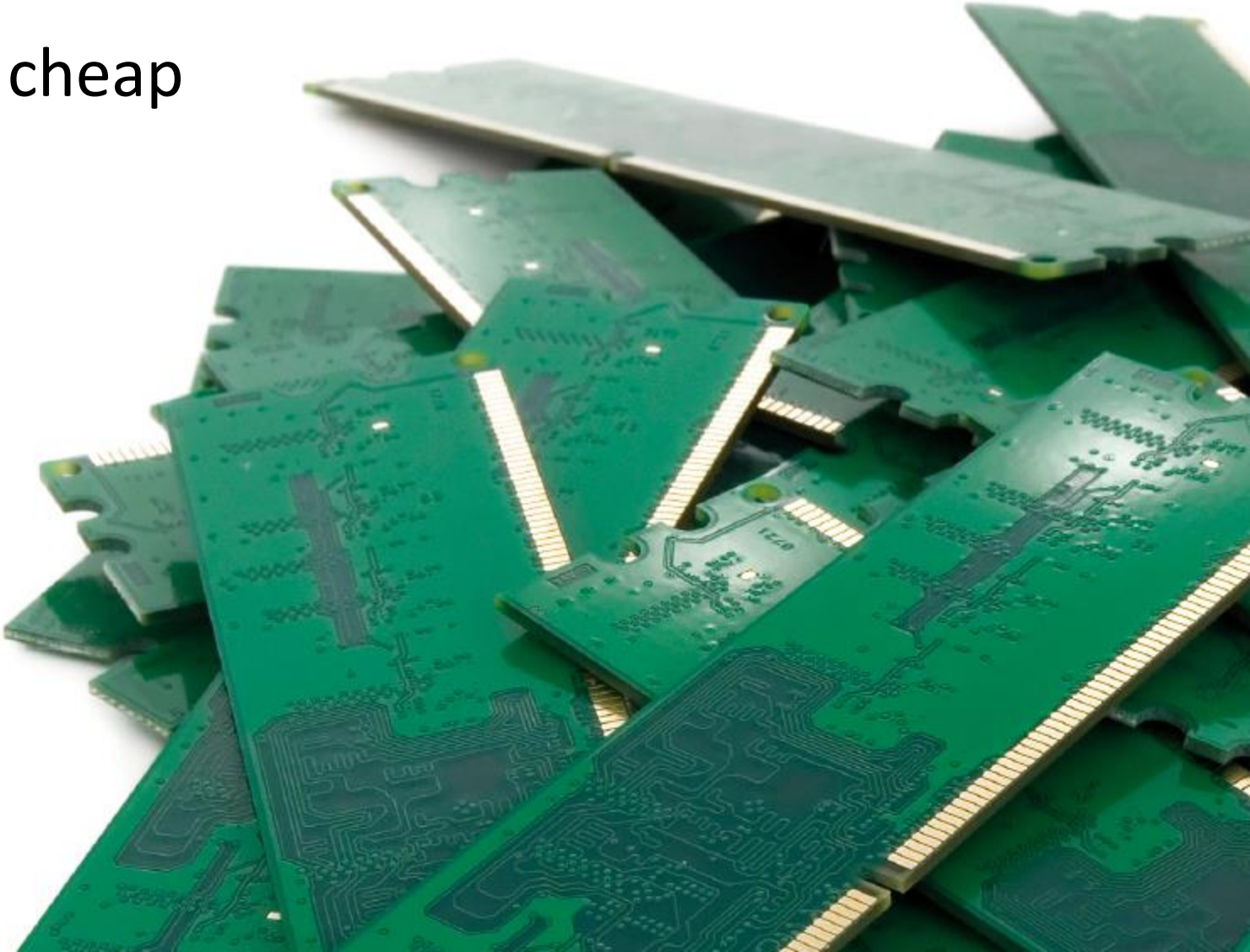
A Java string



Can we make these smaller?

Should ~~Can~~ we make these smaller?

- RAM is cheap



Should ~~Can~~ we make this smaller?

- Address space is cheap (64-bits)

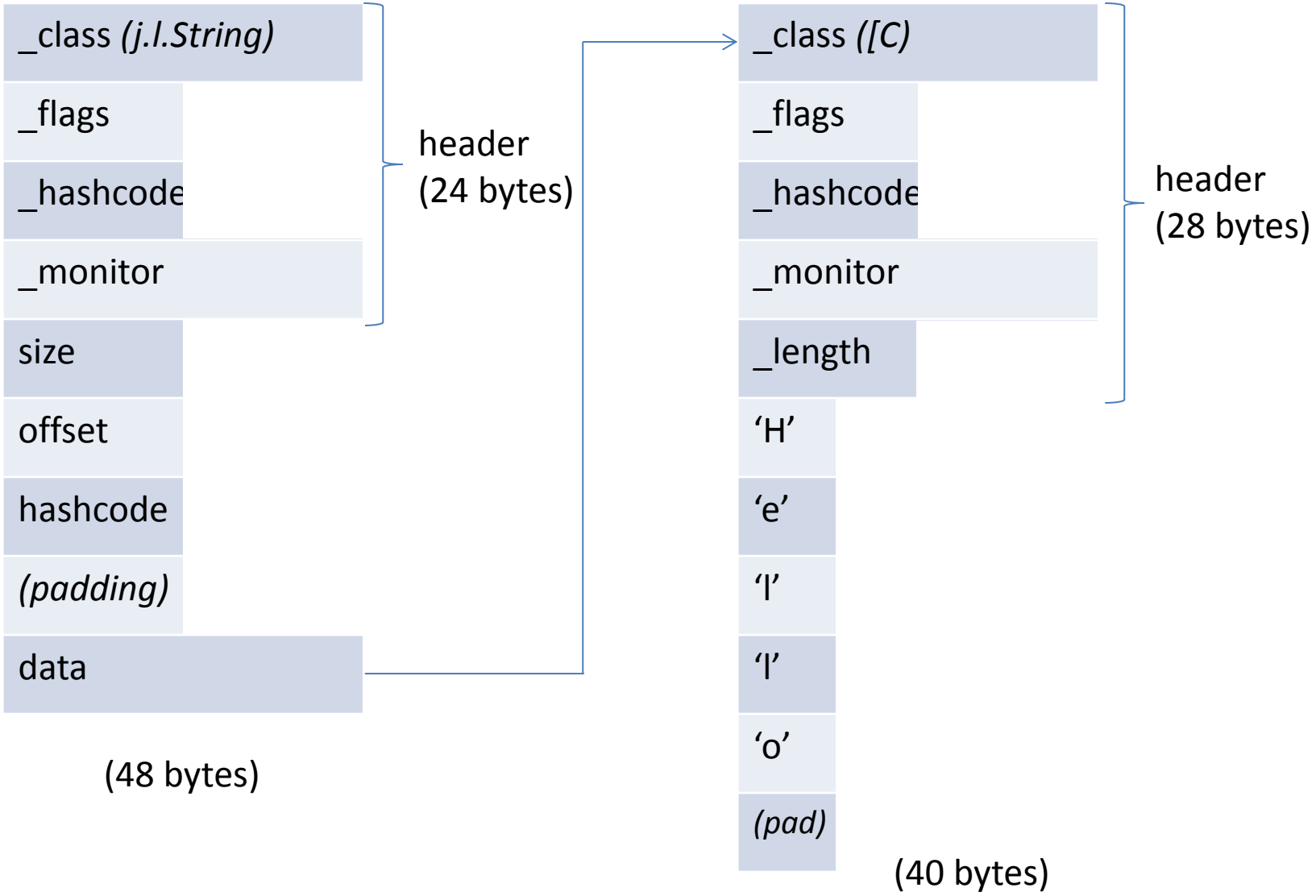
Should ~~Can~~ we make these smaller?

- Cache is not cheap
  - No significant size increase in 10 years
  - (when measured per thread)

	L1	L2	L3	Main
Gallatin ('04)	8K	512K	4.0M	2G
Haswell ('14)	32K	128K	1.5M	128G

(cache sizes are per-thread)

A Java string



Step 1: compress pointers

- Use 32-bit pointers for
  - Class pointer
  - Monitor pointer
  - Object pointers

Compress pointers

- Limited to 4GB space?
  - Classes, monitors & objects can each have own space
  - Exploit alignment

Minimum alignment	Maximum heap
4 bytes	16 GB
8 bytes	32 GB
16 bytes	64 GB
32 bytes?	128 GB

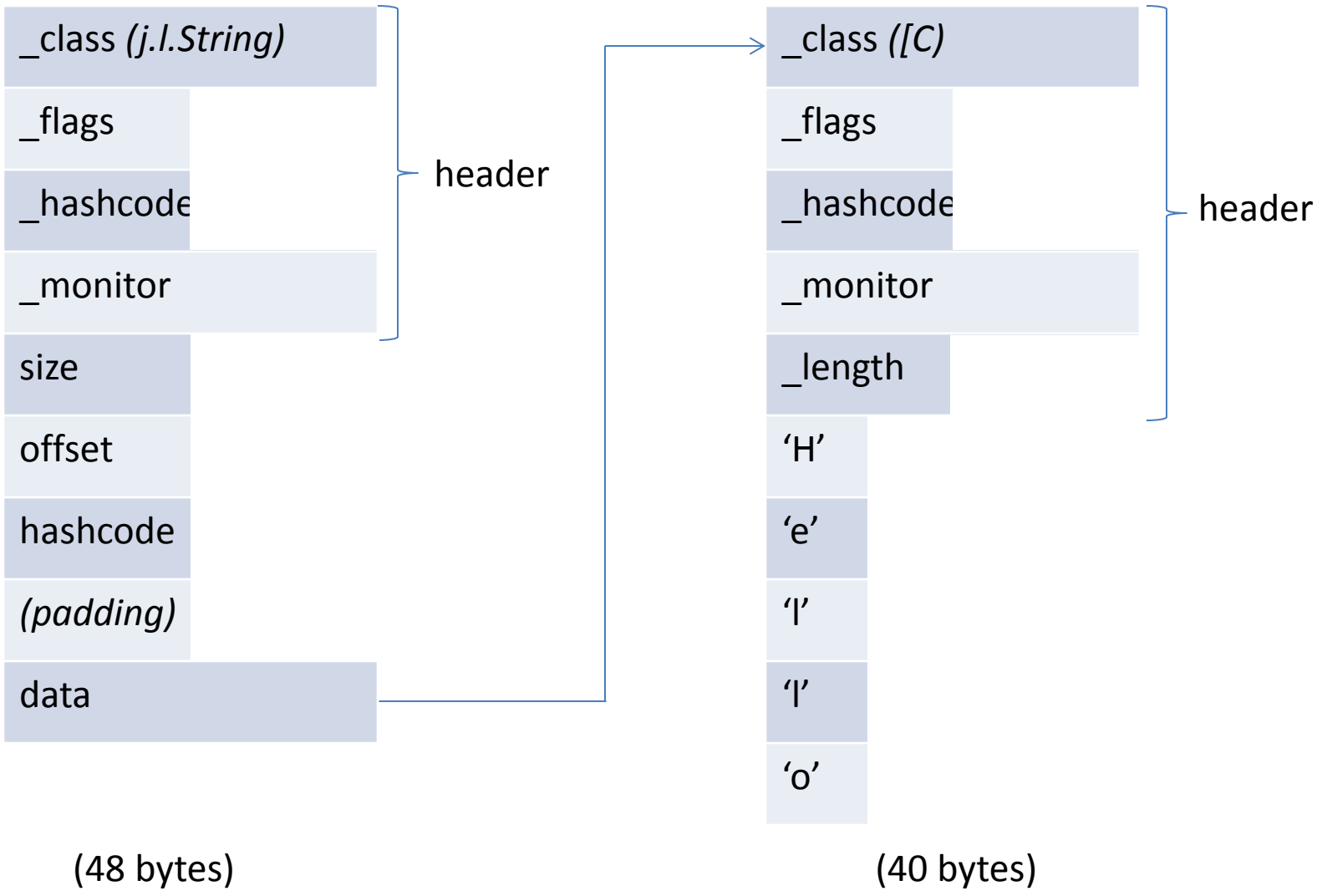
Compress pointers

```
def decompress(comp_ptr) {
    if (comp_ptr == 0)
        return NULL;
    else
        return base + (comp_ptr << scale);
}
```

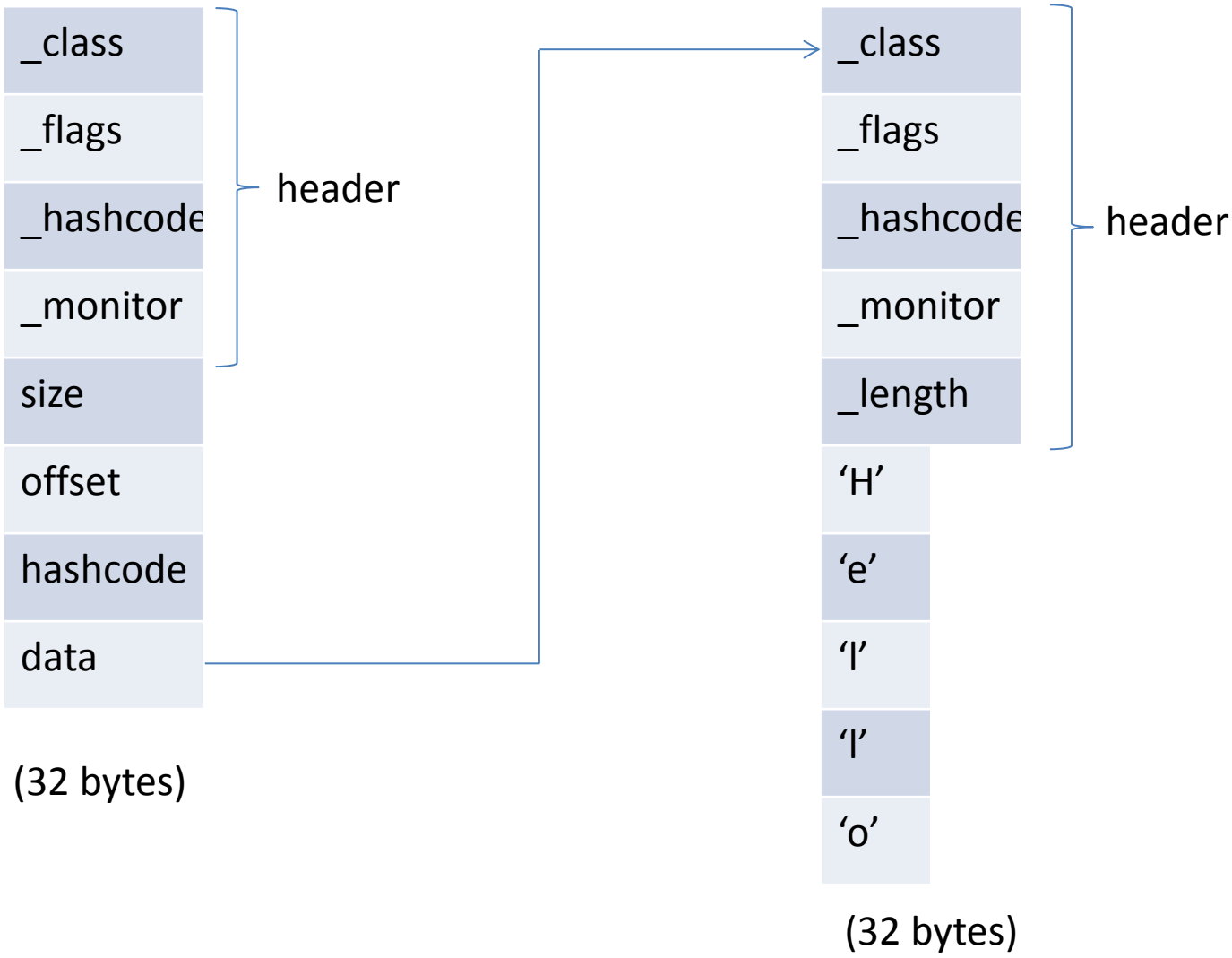
Compress pointers

```
def decompress(comp_ptr) {
#if (base == 0)
    return comp_ptr << bits;
#else
    if (comp_ptr == 0)
        return NULL;
    else
        return base + (comp_ptr << scale);
#endif
}
```

Before


_class (j.l.String)
_flags
_hashcode
_monitor
size
offset
hashcode
(padding)
data
header
(48 bytes)
_class ([C)
_flags
_hashcode
_monitor
_length
'H'
'e'
'l'
'l'
'o'
header
(40 bytes)

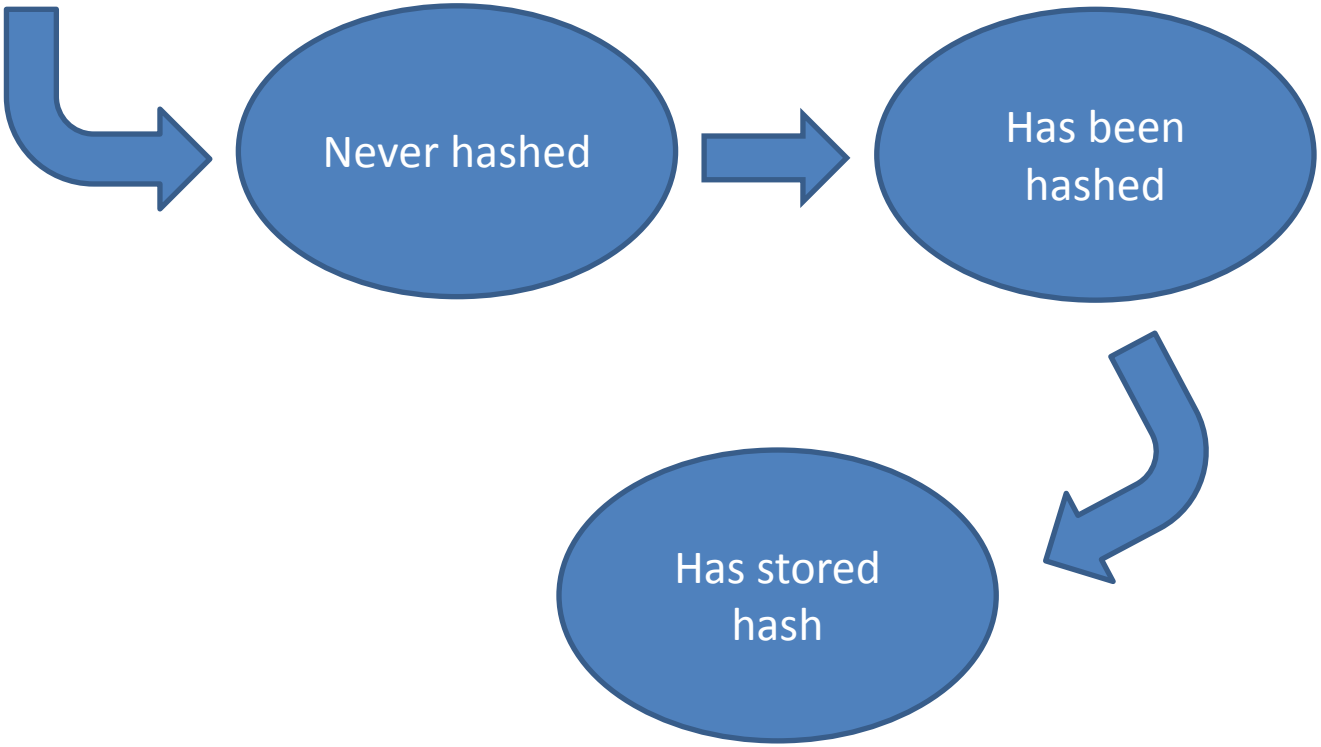
After


_class
_flags
_hashcode
_monitor
size
offset
hashcode
data
header
(32 bytes)
_class
_flags
_hashcode
_monitor
_length
'H'
'e'
'l'
'l'
'o'
header
(32 bytes)

Step 2: Get rid of hashcode

- All objects have ‘identity’ and identity hash:
  - `System.identityHashcode()`
- Stored in header, because objects move
- In practice, < 2% hashed
- Can we store it lazily?

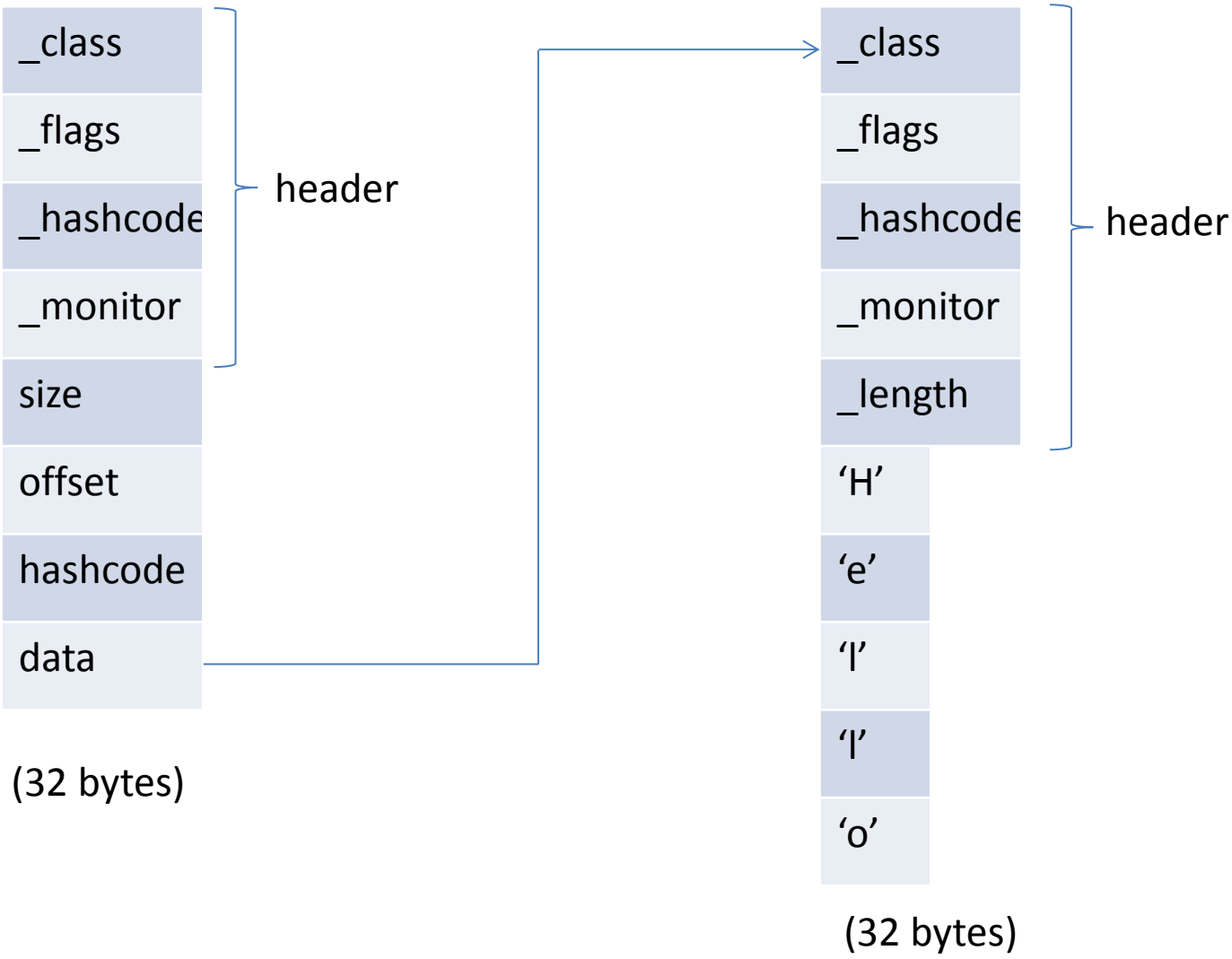
A hash state machine



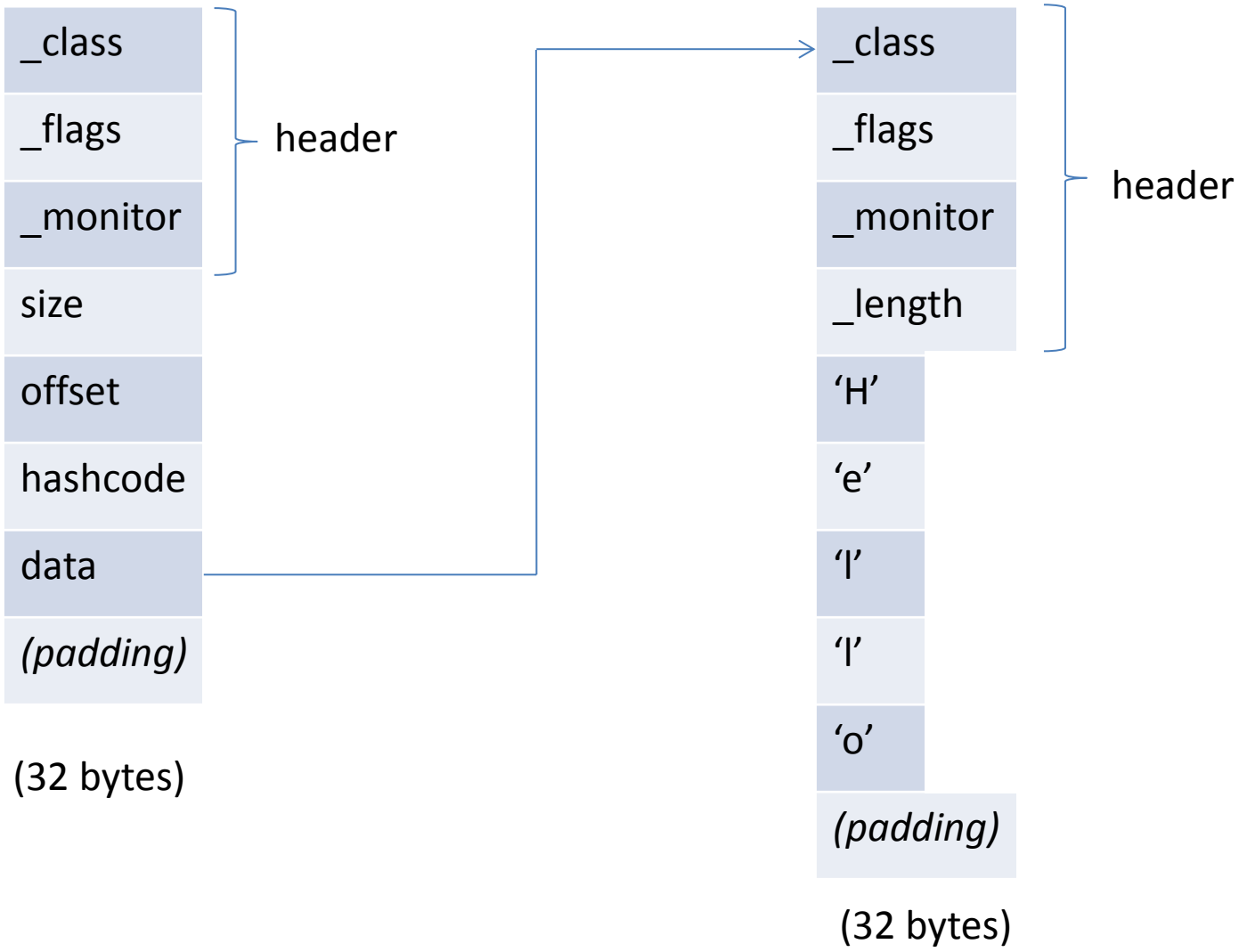
Hashcode algorithm

- First time object is hashed:
  - Generate hash based on address
  - Record ‘has been hashed’
- If object moves:
  - Store hash at end of object
  - Record ‘has been moved’
- Subsequent hashes:
  - Determine if hash is stored or not
  - Read hash, or generate hash based on address

Before


_class
_flags
_hashcode
_monitor
size
offset
hashcode
data
header
(32 bytes)
_class
_flags
_hashcode
_monitor
_length
'H'
'e'
'l'
'l'
'o'
header
(32 bytes)

After


_class
_flags
_monitor
size
offset
hashcode
data
(padding)
header
(32 bytes)
_class
_flags
_monitor
_length
'H'
'e'
'l'
'l'
'o'
(padding)
header
(32 bytes)

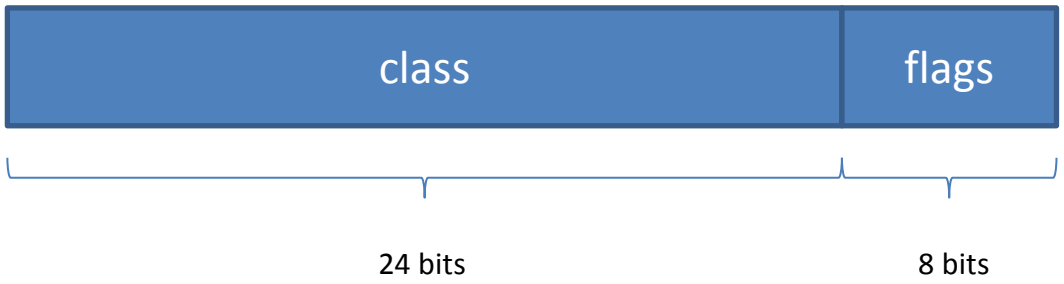
Step 3: Get rid of flags slot

- What's in the flags?

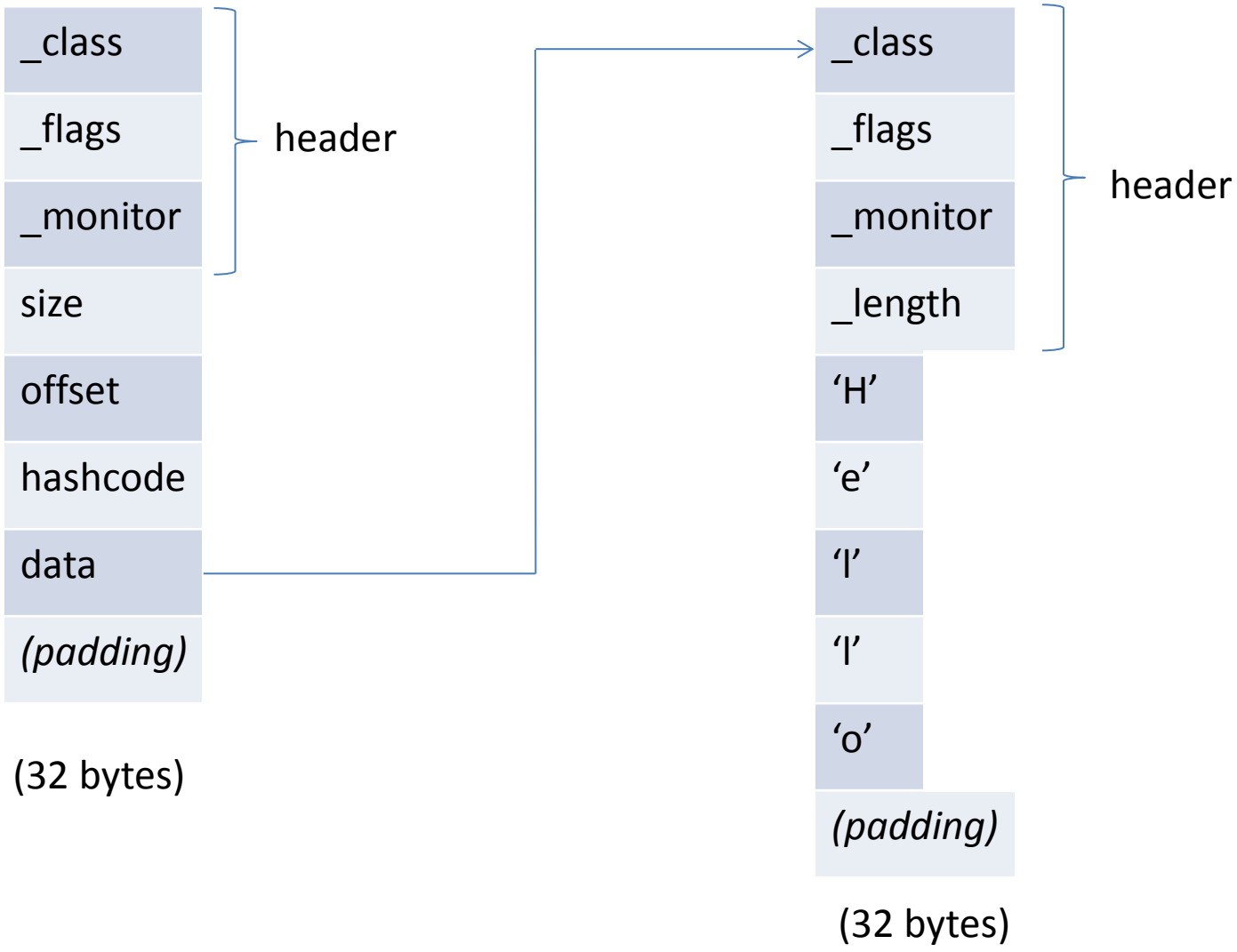
Hash state	2 bits
GC info (e.g. age, remembered)	4-12 bits
Object type (e.g. array)	3 bits
Misc. other stuff	Expands to fill available space

Hiding flags

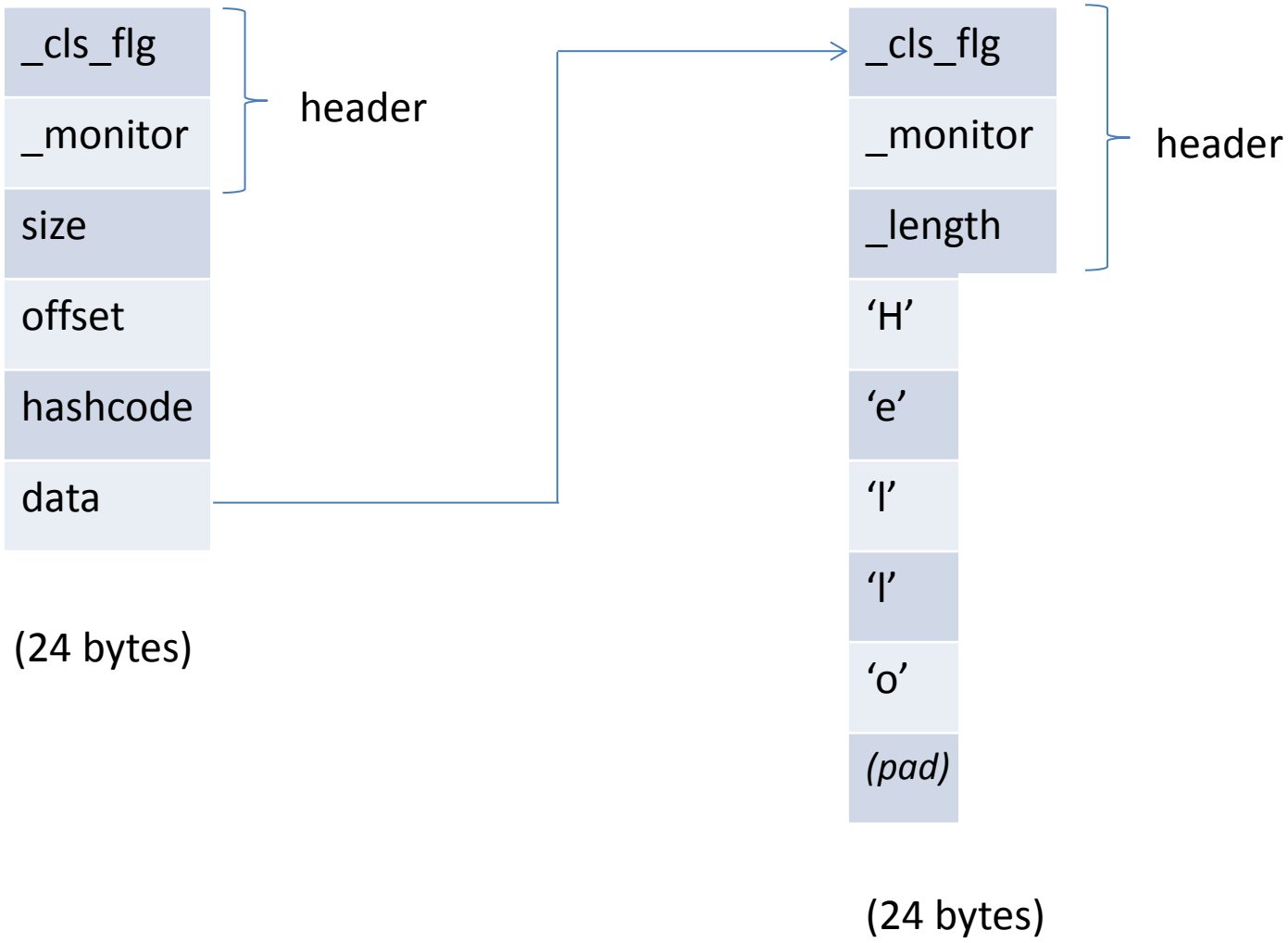
- Infer some flags from class
  - One extra indirect
- Hide the rest in the class pointer
  - Classes must be 256-byte aligned
  - One extra mask instruction



Before


_class
_flags
_monitor
size
offset
hashcode
data
(padding)
header
(32 bytes)
_class
_flags
_monitor
_length
'H'
'e'
'l'
'l'
'o'
(padding)
header
(32 bytes)

After


_cls_flg
_monitor
size
offset
hashcode
data
header
(24 bytes)
_cls_flg
_monitor
_length
'H'
'e'
'l'
'l'
'o'
(pad)
header
(24 bytes)

Step 4: Get rid of monitor slot

- All objects have a monitor
  - synchronized, wait, notify
- Very few objects use the monitor
  - Strings are immutable
  - Arrays are usually wrapped in other objects
- Use monitor slot for some objects
  - “lock nursery” (i.e. hash table) for others

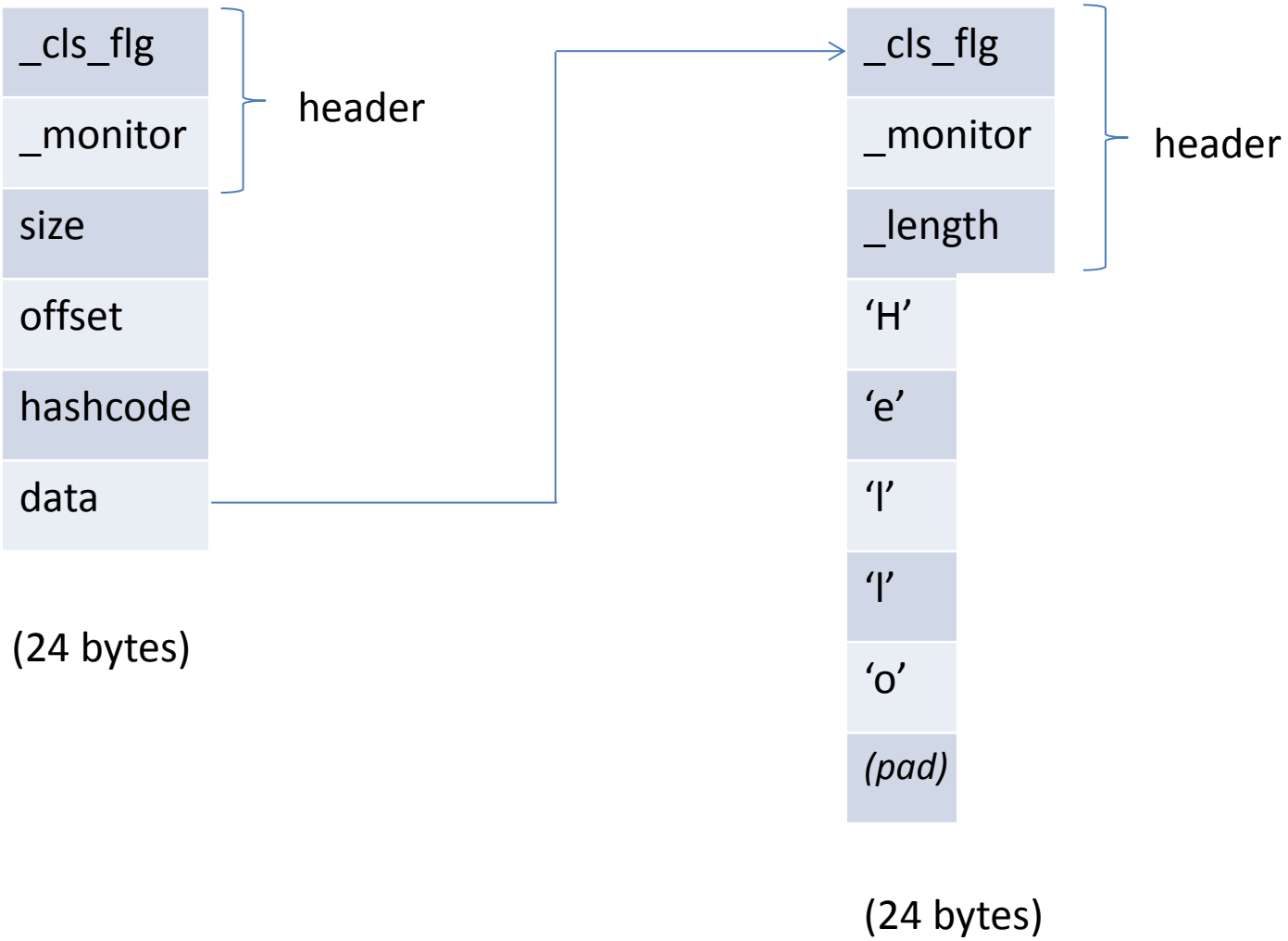
Can we guess where it's needed?

- Maybe
- Static analysis can help
- But fails in some common cases
  - `Object _lock = new Object();`
- J9 was conservative
  - Removed monitor from a small set of classes
    - String, Number, Boolean, ...

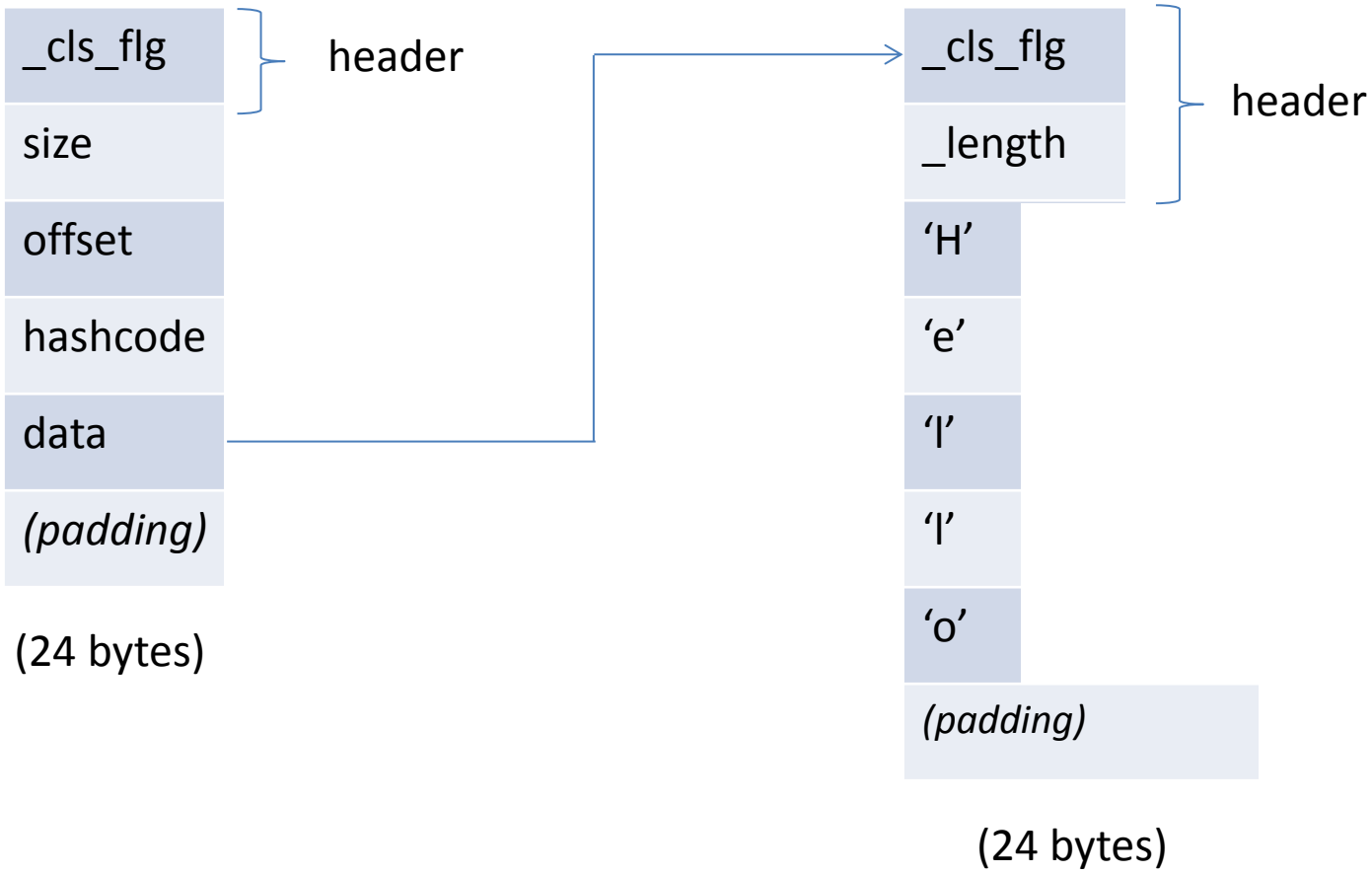
Experimental solution

- Grow monitors on demand
  - Moving objects is expensive
    - Must update all incoming pointers (usually)
  - Combine with a small nursery to amortize cost

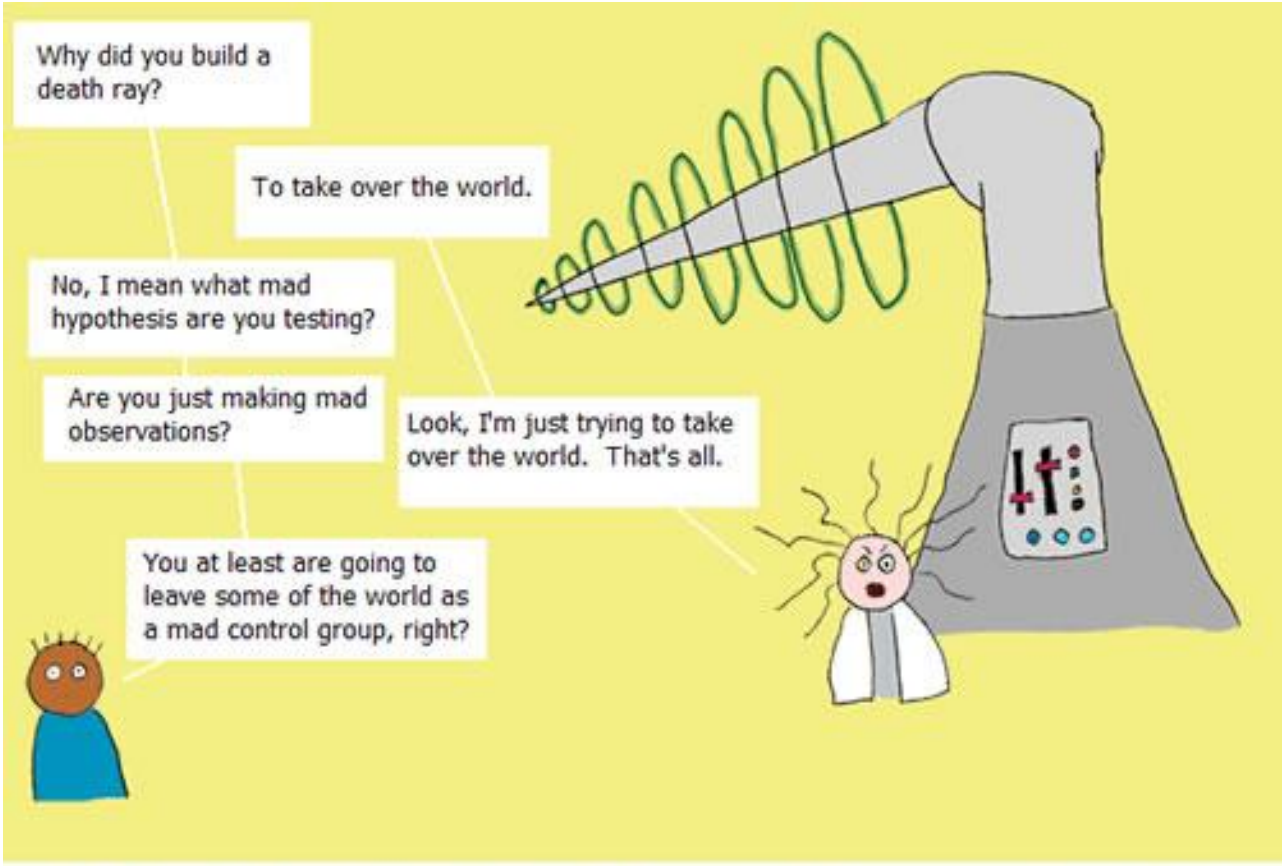
Before


_cls_flg
_monitor
size
offset
hashcode
data
header
(24 bytes)
_cls_flg
_monitor
_length
'H'
'e'
'l'
'l'
'o'
(pad)
header
(24 bytes)

After


_cls_flg
size
offset
hashcode
data
(padding)
header
(24 bytes)
_cls_flg
_length
'H'
'e'
'l'
'l'
'o'
(padding)
header
(24 bytes)

Some more ideas



Sad truth: Most "mad scientists" are actually just mad engineers

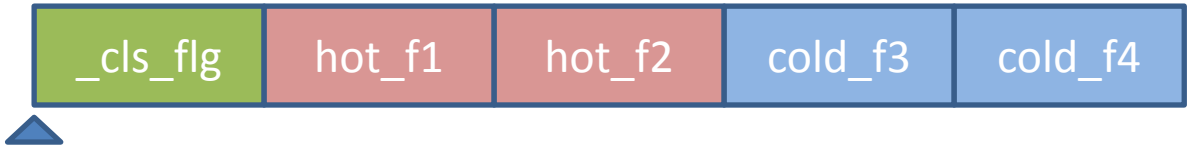
http://www.neatorama.com/2009/01/01/mad-scientists-are-actually-just-mad-engineers/

Idea 1: Hot and cold fields

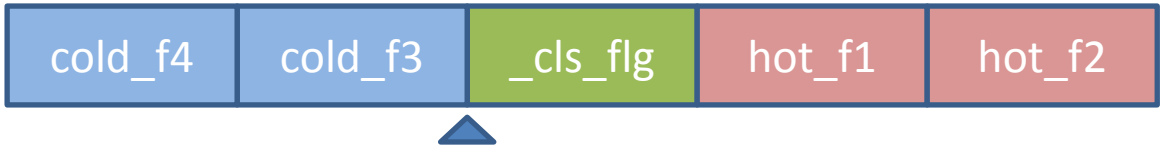
- Large objects may have rarely used fields
- Use runtime profiling to identify hot fields
- Split hot from cold for cache efficiency
- Caveat:
  - changing object layout at runtime is costly

Hot & cold layouts

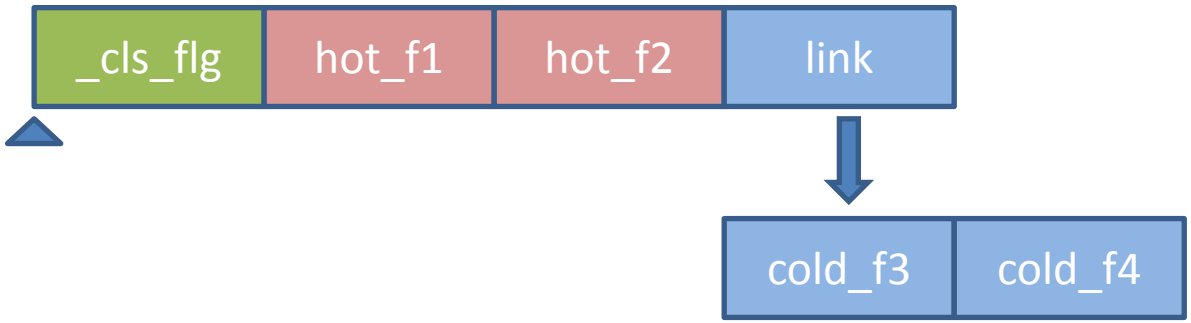
- Sort hot fields to front



- Bidirectional objects

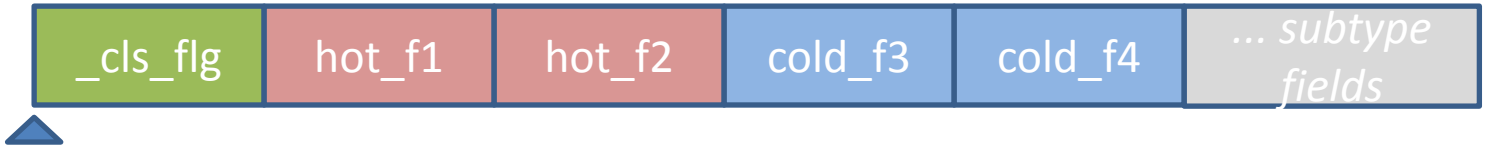


- Linked objects

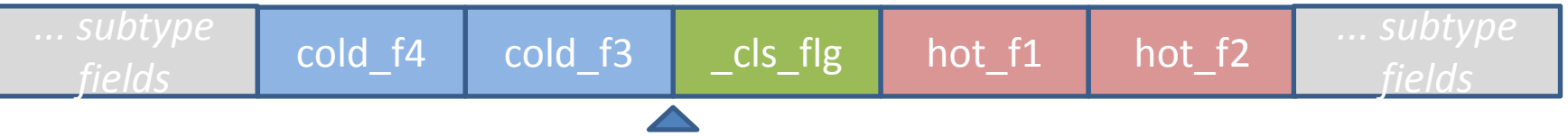


Hot & cold layouts

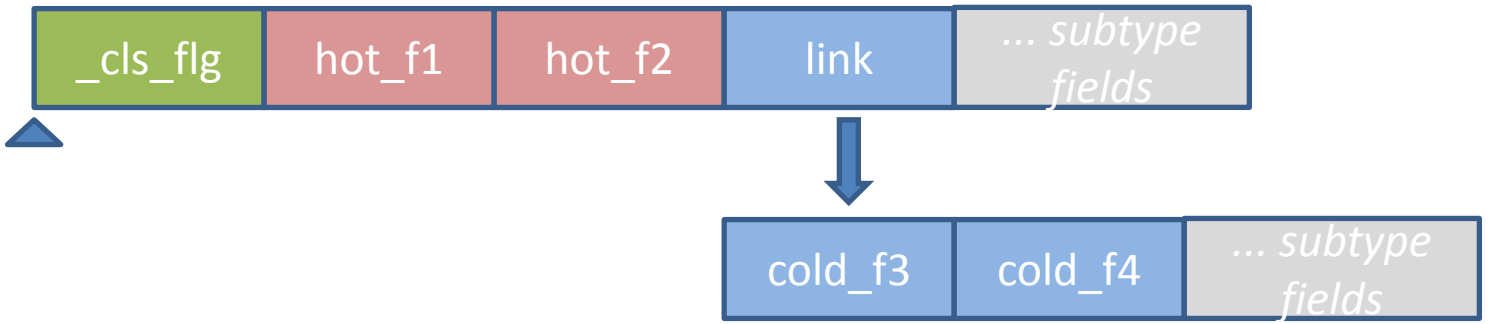
- Sort hot fields to front



- Bidirectional objects



- Linked objects



Idea 2: Headerless objects?

- Can we delete the class pointer?
- Infer class from pointer
  - `class = object & 0xFFFFFFFFF000000`

Class A	Obj 1	Obj 2
Obj 3	Obj 4	Obj 5
Obj 6	Obj 7	Obj 8
Obj 9	Obj 10	Free
Free	Free	Free

Class B	Obj 11	Obj 12
Obj 13	Free	Free
Free	Free	Free
Free	Free	Free
Free	Free	Free

Headerless objects (cont.)

- Wastes memory
  - But RAM and address space are cheap
- Organizes objects by class
  - Splits up related objects
  - Could be bad for cache

Lessons

- Object shape affects performance
- Language affects object shape
  - Dynamic vs. static
  - Hash codes
  - Synchronization
  - Compatibility

Further reading

- Bacon, Fink and Grove. “Space- and Time-Efficient Implementation of the Java Object Model”, 2002
- Adl-Tabatabai, *et al.* “Improving 64-Bit Java IPF Performance by Compressing Heap References”, 2004
- Domborwski, *et al.* “Dynamic monitor allocation in the Java virtual machine”, 2013

Idea 3: Objlets

- Break large objects up into trees of smaller objects
- Simplifies allocation
- Avoids defragmentation
- Enables realtime allocation guarantees