Lilliput

Shrinking object headers in the Hotspot JVM

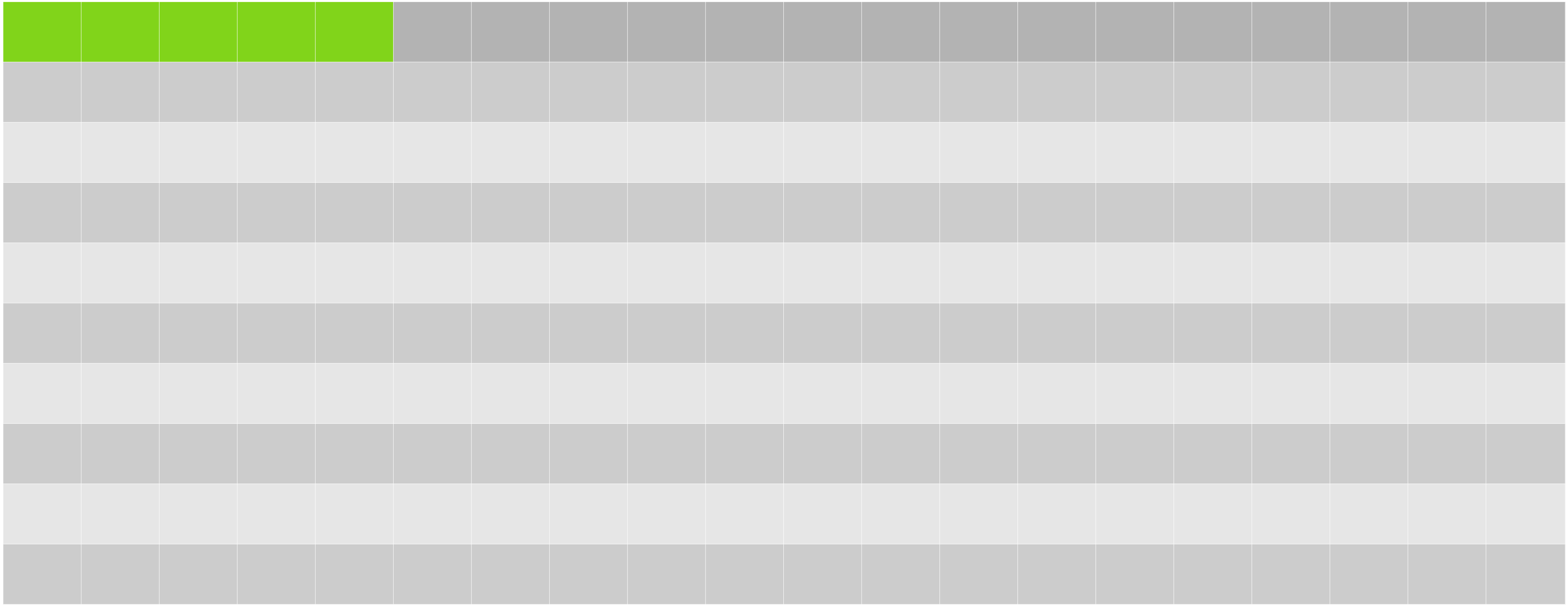
Roman Kennke, Red Hat
@rkennke

Agenda

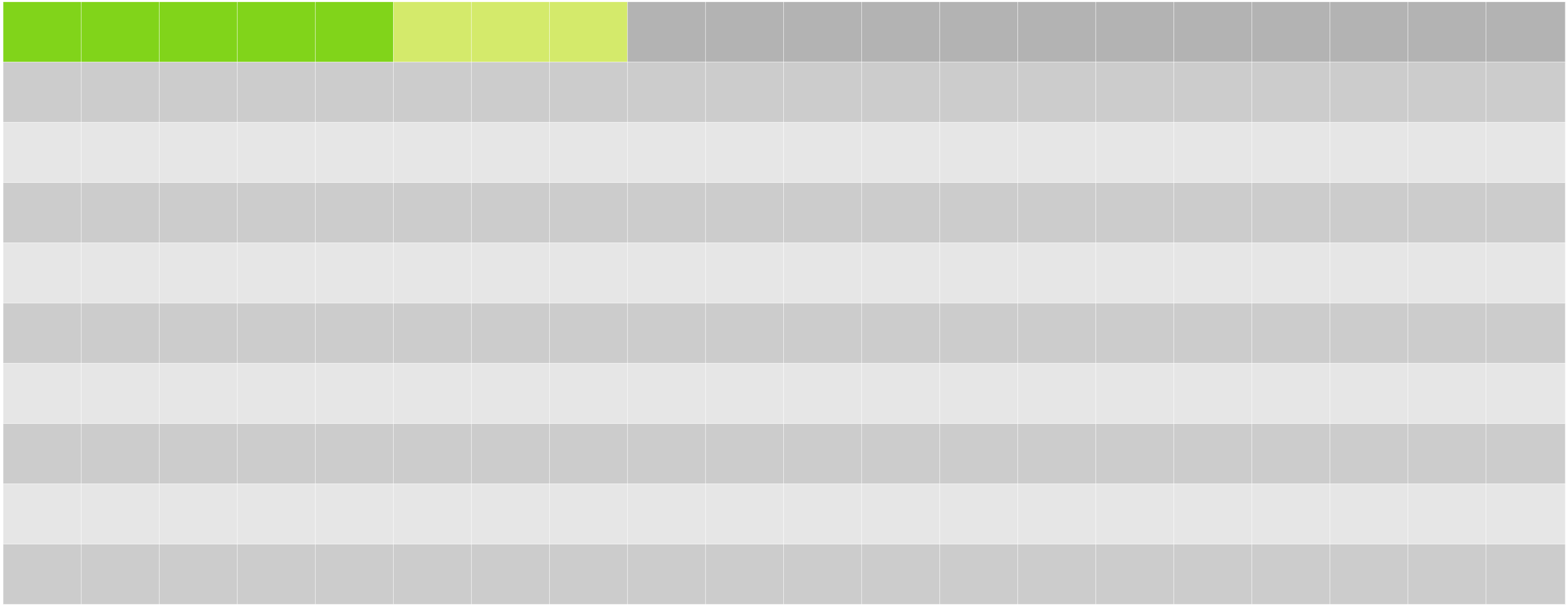
- Introduction: Heap and object layout
- Goals of Project Lilliput
- GC Forwarding pointers
- Identity Hash-Code
- Locking
- Look into the future

Heap layout

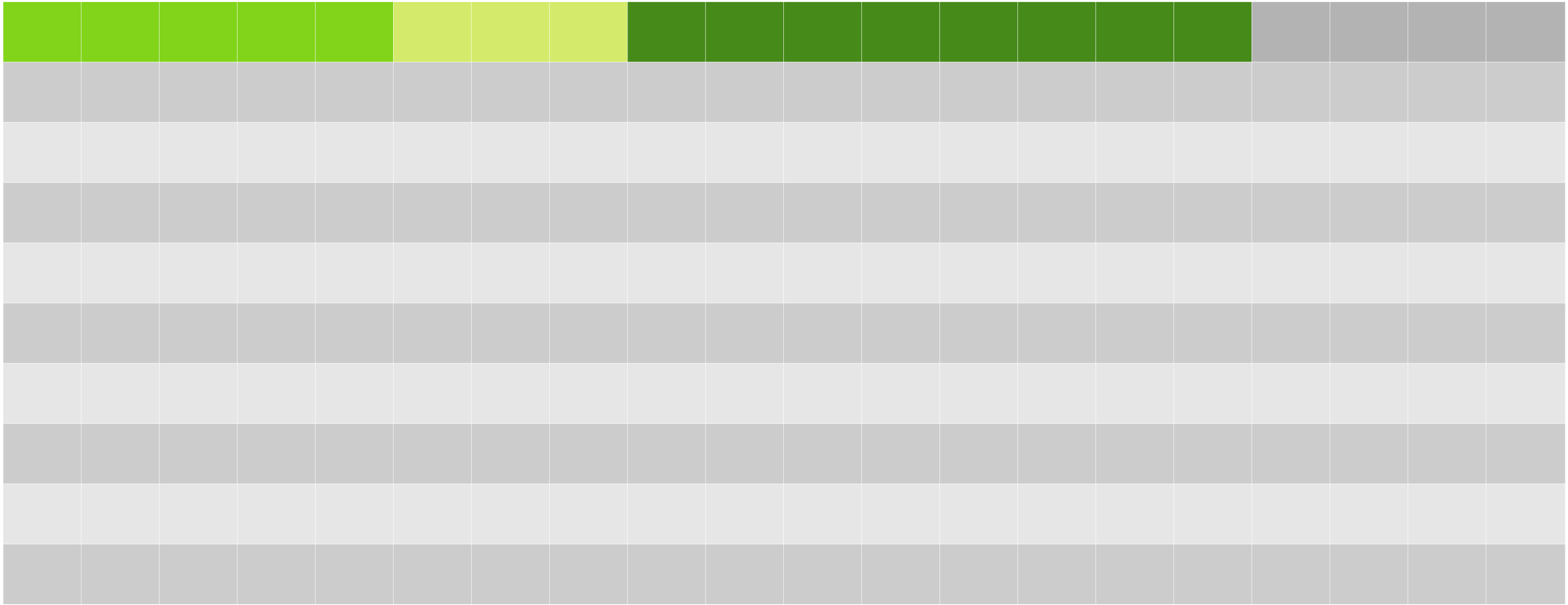
Heap layout



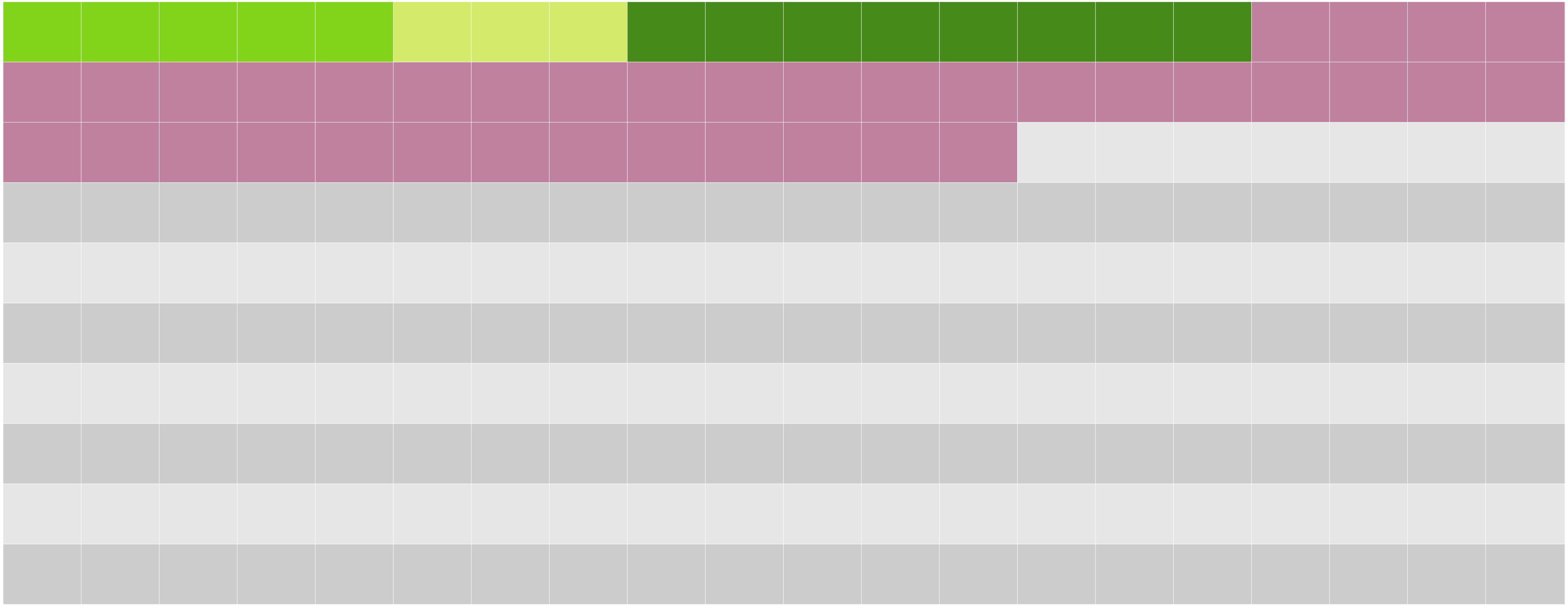
Heap layout



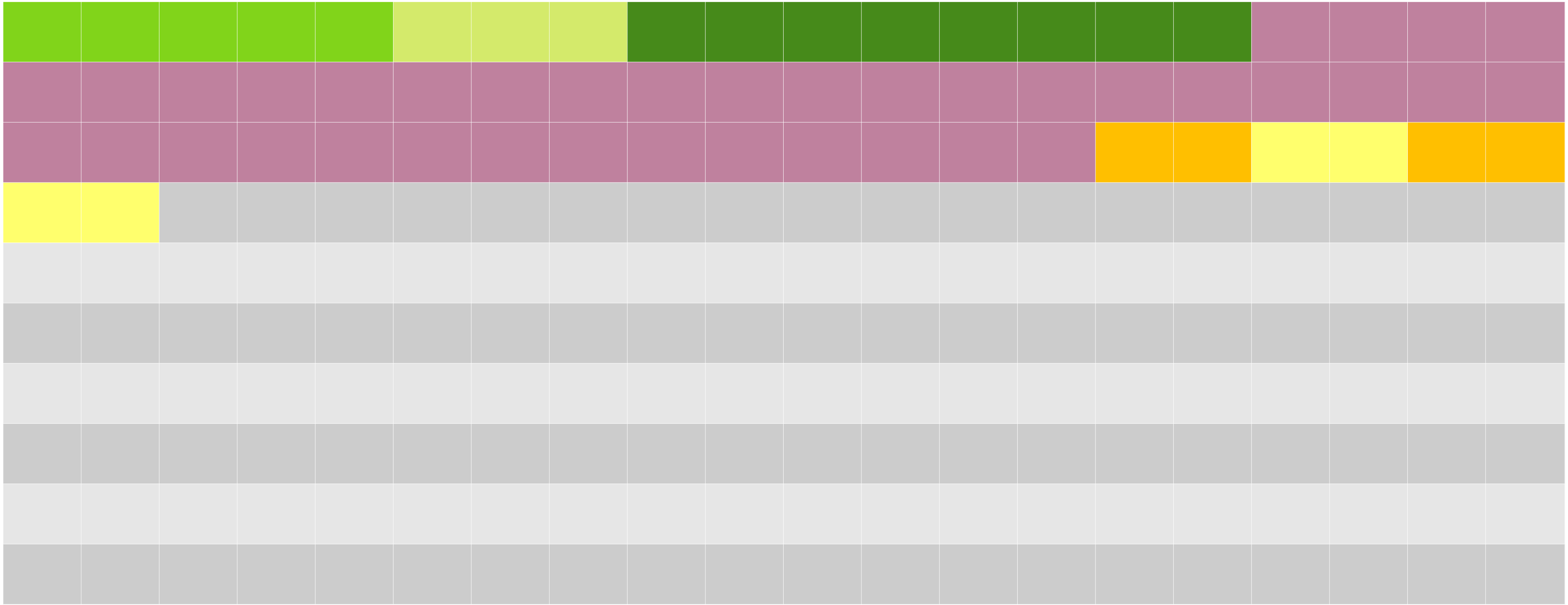
Heap layout



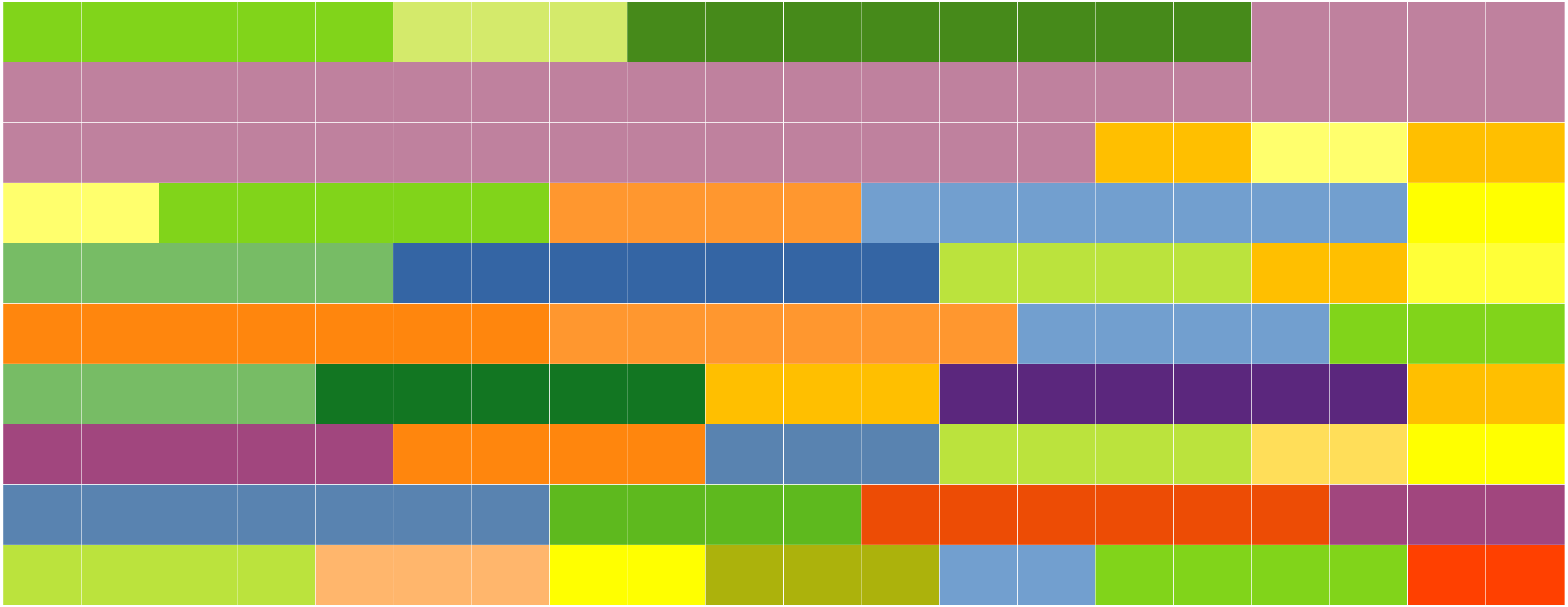
Heap layout



Heap layout



Heap layout



Anatomy of a Java object

```
public class Point {
  int x;
  int y;
  int z;
}
```

Anatomy of a Java object

```
public class Point {
  int x;
  int y;
  int z;
}
```

```
int x = 42;
```

Anatomy of a Java object

```
public class Point {
  int x; // 32 bit
  int y; // 32 bit
  int z; // 32 bit
}
```

```
int x = 42;  int y = 68;
```

Anatomy of a Java object

```
public class Point {
  int x; // 32 bit
  int y; // 32 bit
  int z; // 32 bit
}
```

int x = 42; int y = 68;
int z = 17;

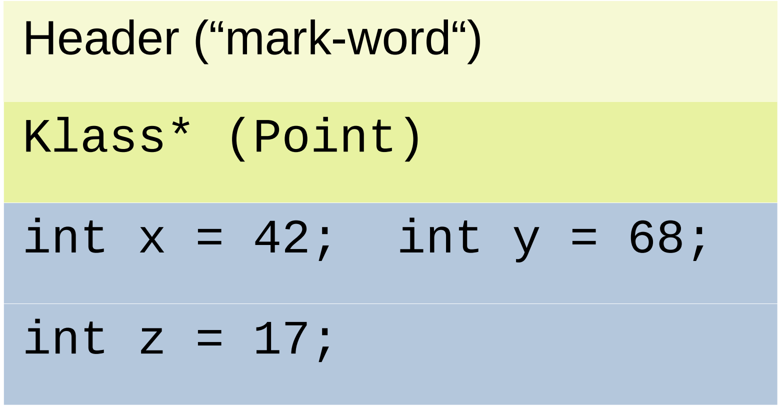
Anatomy of a Java object

```
public class Point {
  int x; // 32 bit
  int y; // 32 bit
  int z; // 32 bit
}
```

Klass* (Point)
int x = 42; int y = 68;
int z = 17;

Anatomy of a Java object

```
public class Point {
  int x; // 32 bit
  int y; // 32 bit
  int z; // 32 bit
}
```



Anatomy of a Java object

```
public class Point {
  int x; // 32 bit
  int y; // 32 bit
  int z; // 32 bit
}
```

Header (“mark-word“)
`narrowKlass; int x = 42;`
`int y = 68;  int z = 17;`

`-XX:+UseCompressedClassPointers`

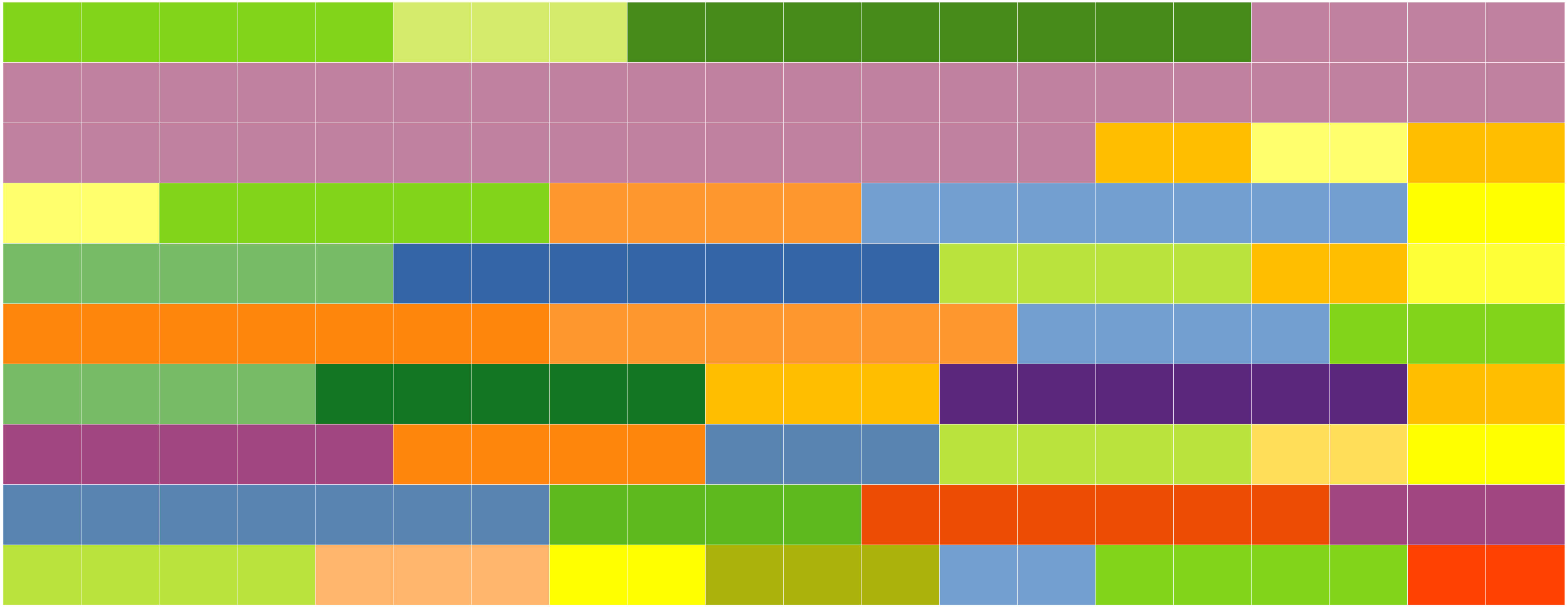
Anatomy of a Java object

```java
public class Point {
  int x; // 32 bit
  int y; // 32 bit
  int z; // 32 bit
}
```

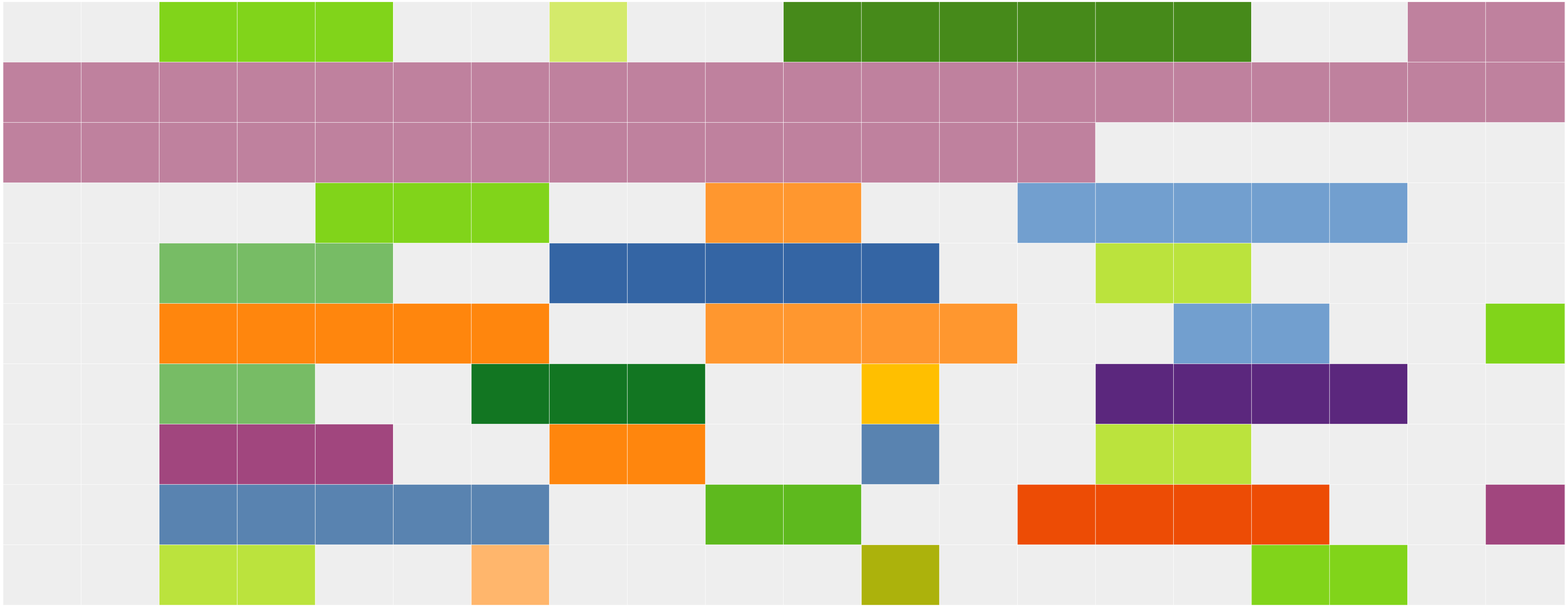
Header (“mark-word“)
narrowKlass; int x = 42;
int y = 68; int z = 17;

The full picture:
https://shipilev.net/jvm/objects-inside-out/

Heap layout – Headers vs payload



Heap layout – Headers vs payload



In practice

- 2 words / 16 bytes minimum object size
- ~4-10 words / 32-80 bytes avg object size in typical workloads

→ up to ~20-50% overhead for object header

How about other languages?

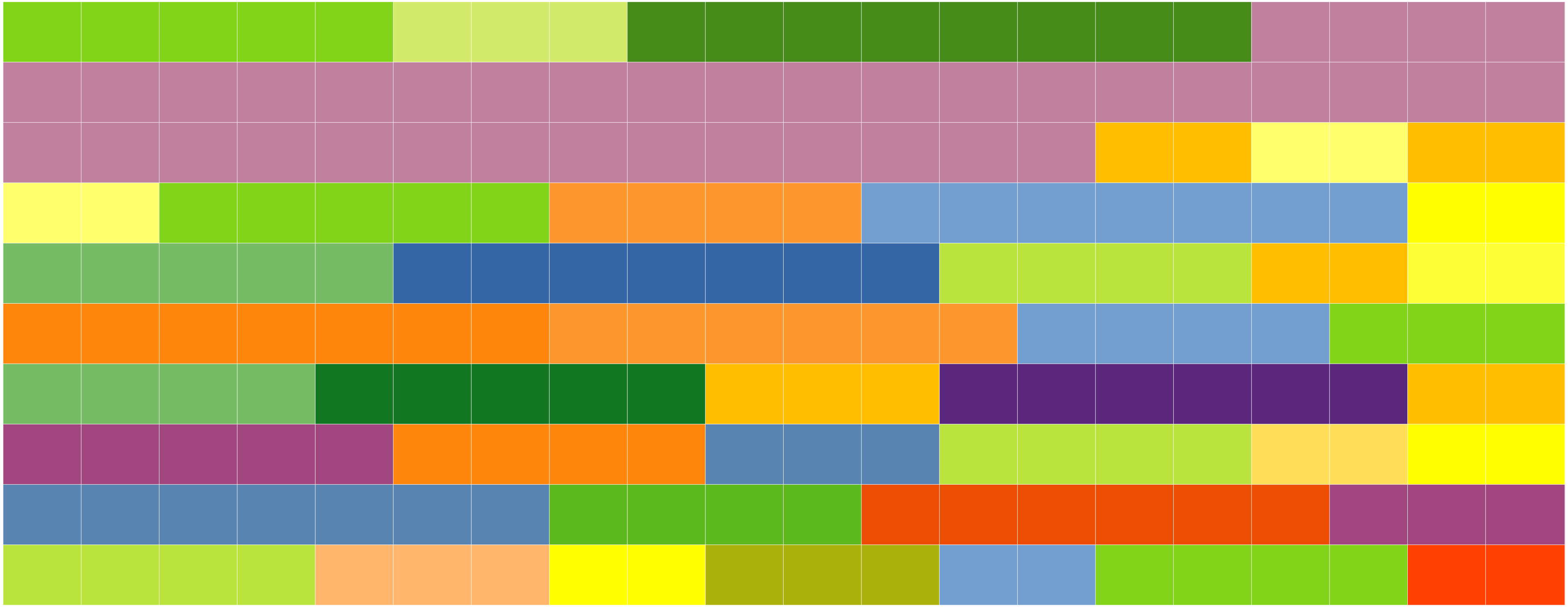
- C: 0 words
- C++: 0 or 1 words
- Rust: 0..? words

Why are Java headers so big?

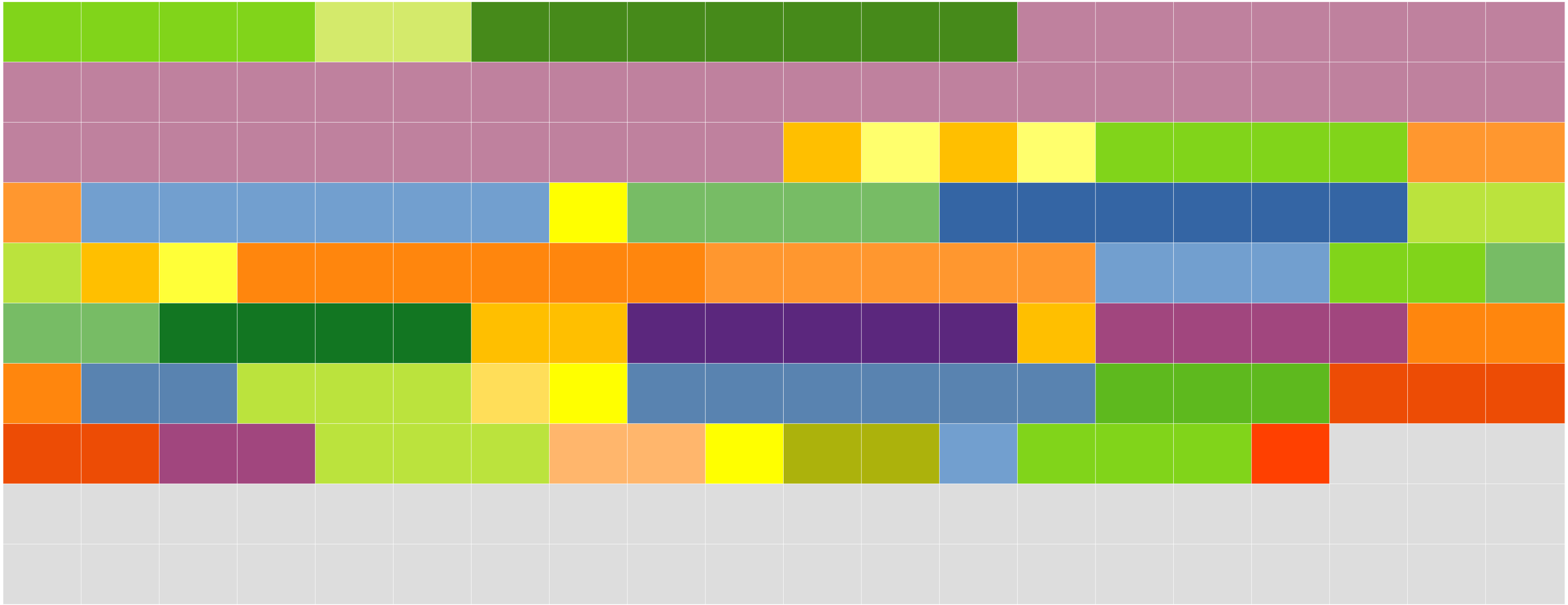
- Built-in support for:
  - Type info (instanceof, virtual calls, …)
  - Locking (synchronized { … })
  - Garbage Collection
  - Identity Hash-Code

Can we do better?

Heap layout – Big headers



Heap layout – Lilliput headers



Advantages of smaller headers

- Better memory footprint
- Higher memory density (lower cache pressure)
- Higher payload allocation rate
- Less GC pressure

Advantages of smaller headers

- Better memory footprint
- Higher memory density (lower cache pressure)
- Higher payload allocation rate
- Less GC pressure
- (Energy savings)

Advantages of smaller headers

- Better memory footprint
- Higher memory density (lower cache pressure)
- Higher payload allocation rate
- Less GC pressure
- (Energy savings)
- ($$$ savings)

Estimates

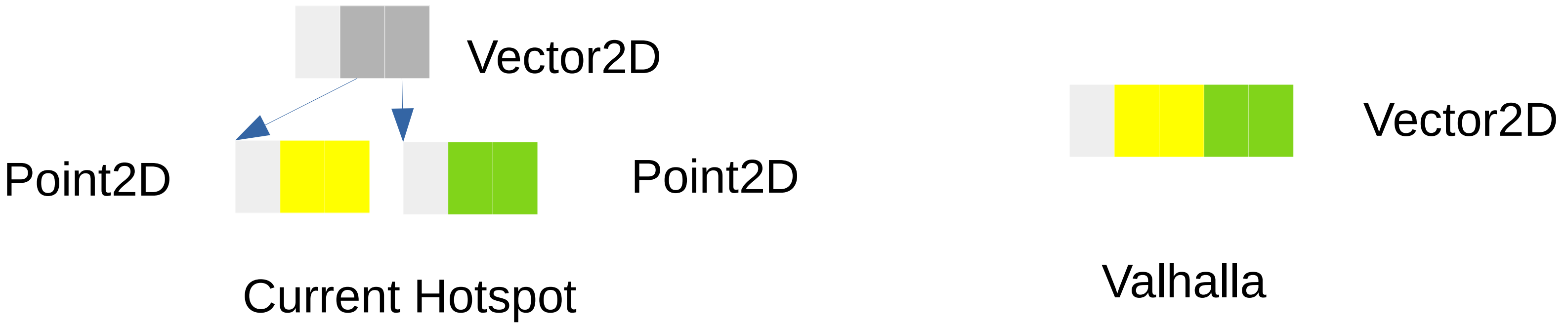
- Up to 33% footprint savings
- Average savings over SPECjvm2008, dacapo and Renaissance benchmarks ~15%

Estimates

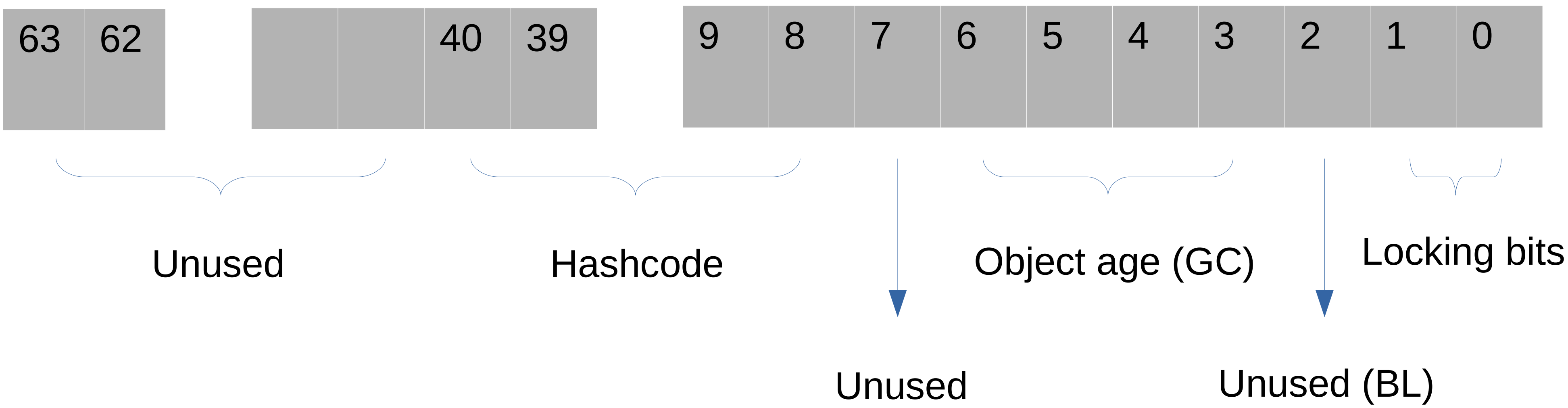
- Up to 33% footprint savings
- Average savings over SPECjvm2008, dacapo and Renaissance benchmarks ~15%
- Difficult to estimate CPU impact
- Experience from Shenandoah brooks pointer optimization: ~10%-~20% gains

Excursion: Valhalla

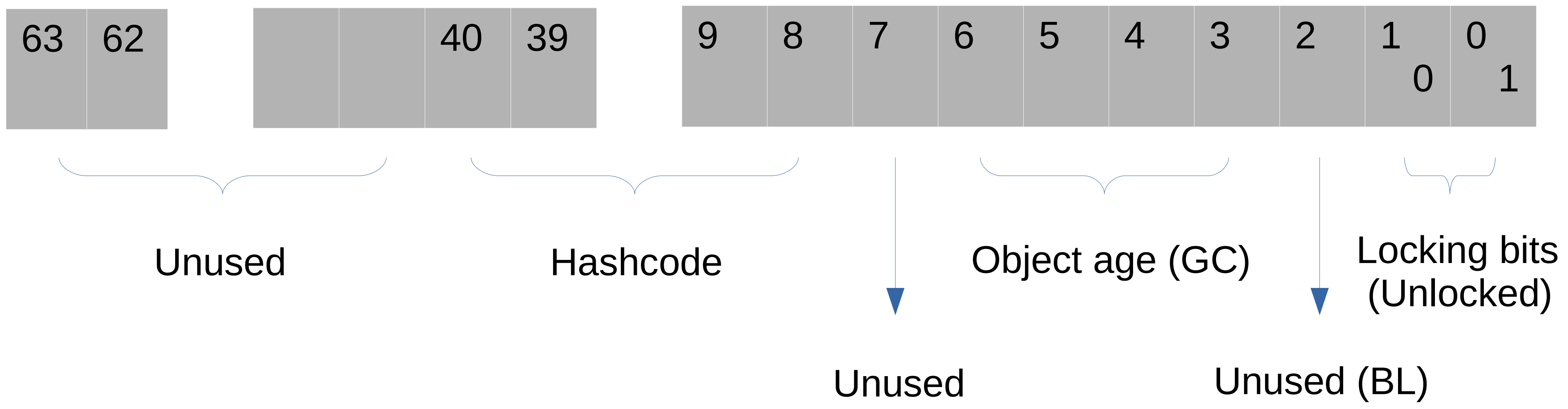
- Valhalla aims to improve payload layout
- Flatten object structure, value-types
- Complements Lilliput nicely



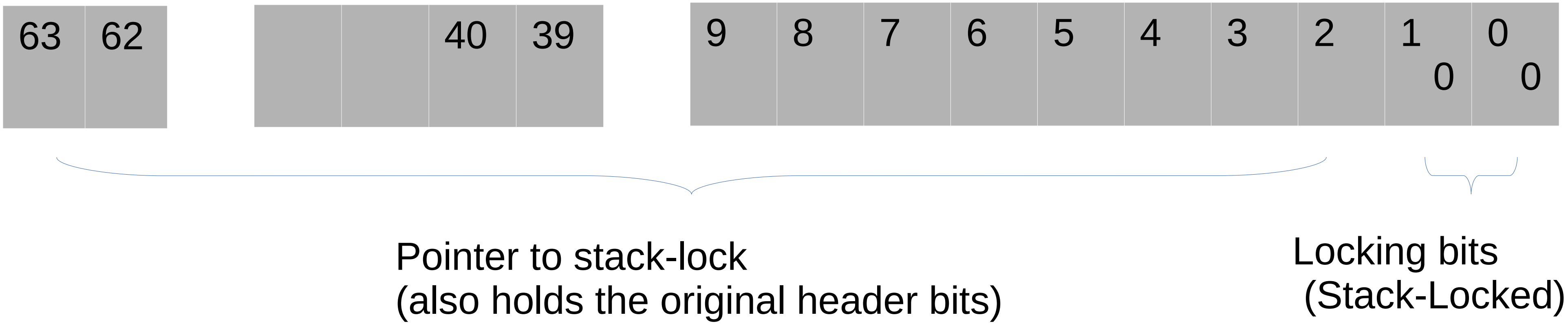
Anatomy of object headers



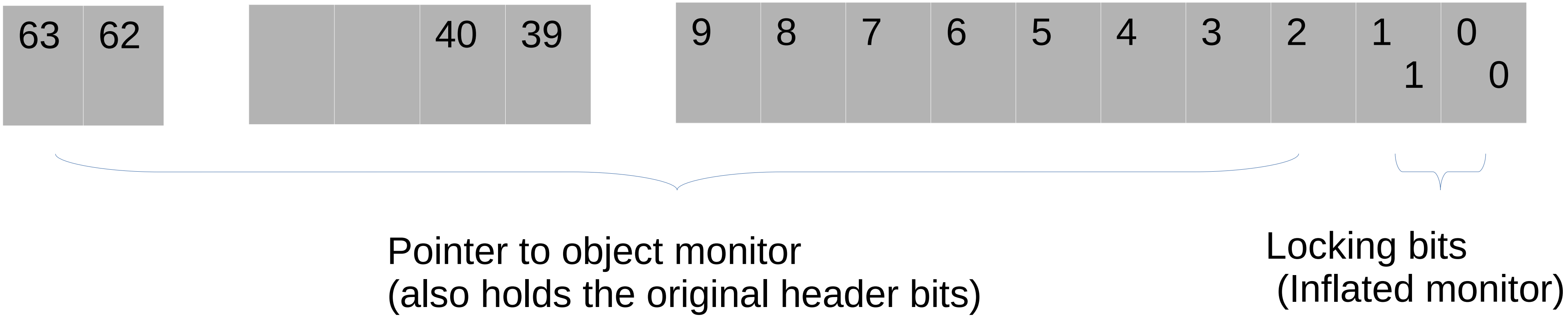
Anatomy of object headers



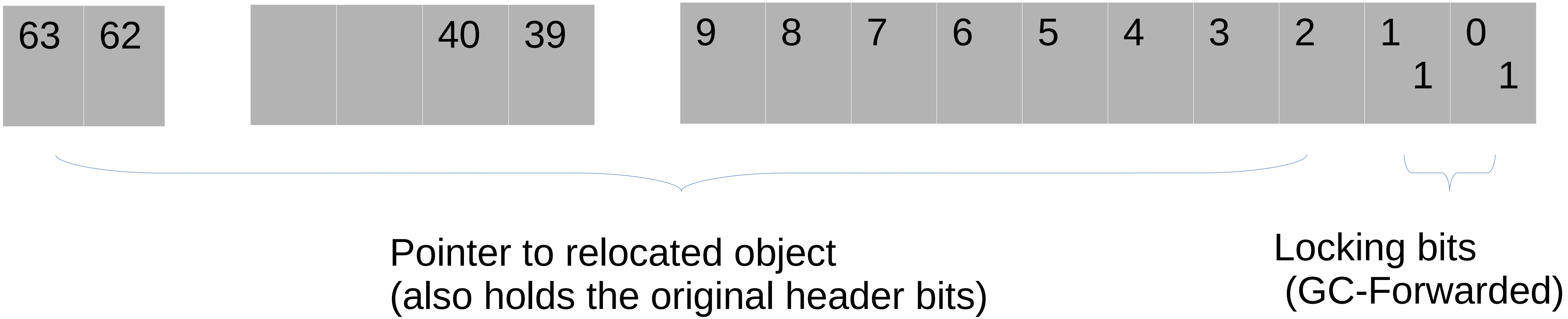
Anatomy of object headers



Anatomy of object headers



Anatomy of object headers



Klass* field

- Pointer to Klass structure:
  - Type information (name, superclass, instanceof...)
  - Vtable/Itable
  - Etc

Klass* field

- Pointer to Klass structure:
  - Type information (name, superclass, instanceof...)
  - Vtable/Itable
  - Etc
- Can be compressed to 32 bit:
  - -XX:+UseCompressedClassPointers

Pointer Compression in Hotspot

- 32-bit mode
  - Pointers to lowest 4GB have highest 32 bits zero
  - Lowest 4GB can be addressed directly by a 32-bit address
  - Simplest mode

Pointer Compression in Hotspot

- Zero-based
  - Pointers to aligned object have lowest bits 0
  - Example: 8-byte alignment means lowest 3 bits are 0 ($2^3 = 8$)
  - We can shift the address to extend addressable range of aligned objects.
  - E.g. 2^3 alignment → 2^(32+3) → 32GB

Pointer Compression in Hotspot

- Non-zero-based
  - Same as zero-based, but add a base-address
  - Allows 2^(32+shift) address range anywhere

→ Compressed class pointers always use this mode

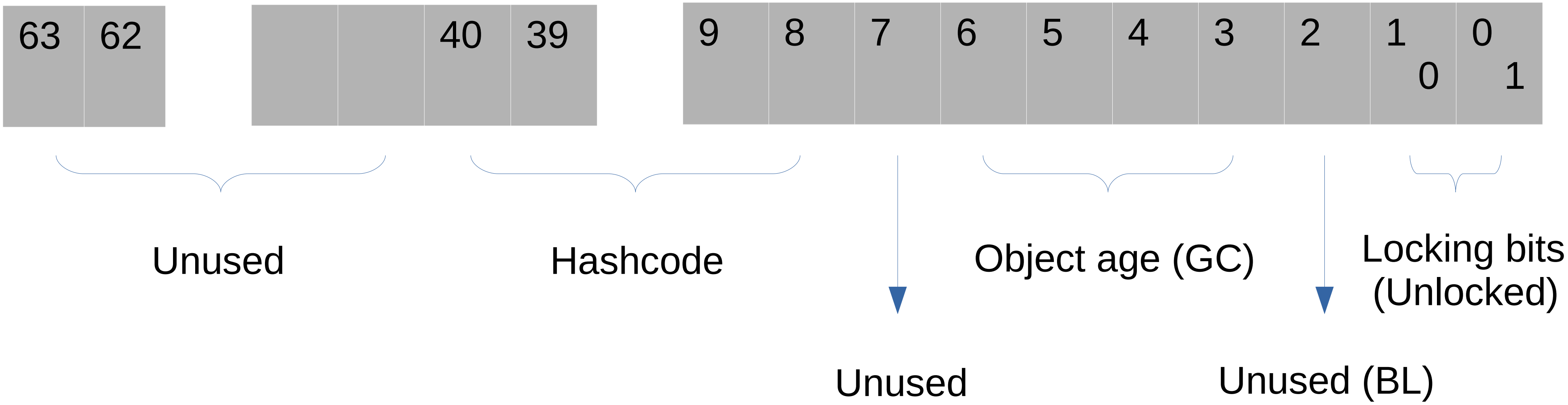
Pointer Compression in Hotspot

- Generalized version:
  - Allows addressing of 2^(num_bits + shift) bytes with
  - num_bits: number of compressed bits
  - shift: byte-alignment of addressed objects, 2^shift
- Example:
  - Using alignment on 1024 bytes, e.g. shift = 10
  - And 22 bits
  - We can address 2^(22+10) = 2^32 bytes = 4GB

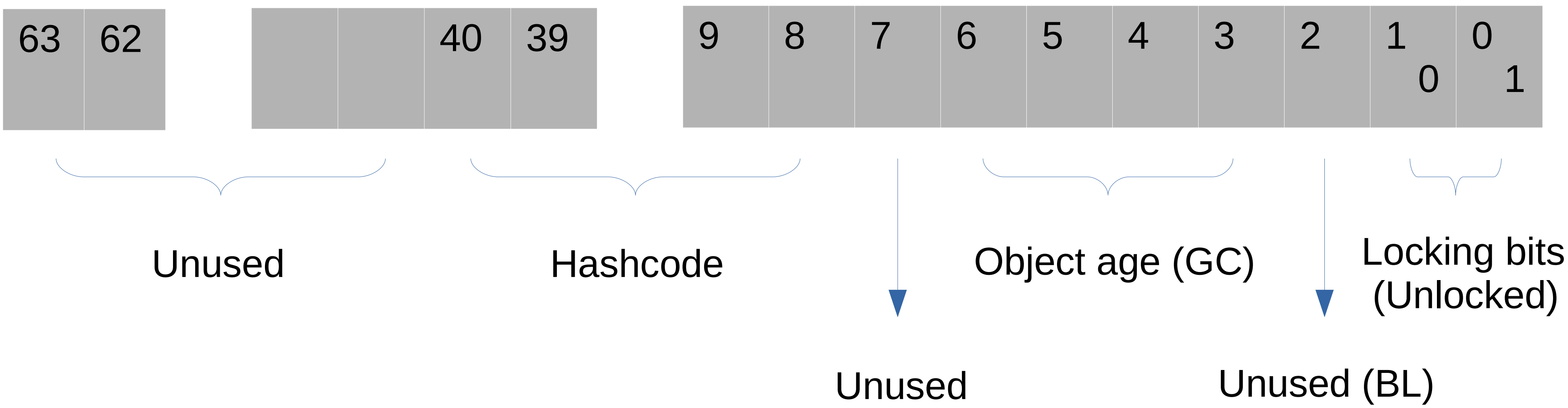
Lilliput – The Big Picture

- What do we want to achieve?

Existing object header - reminder

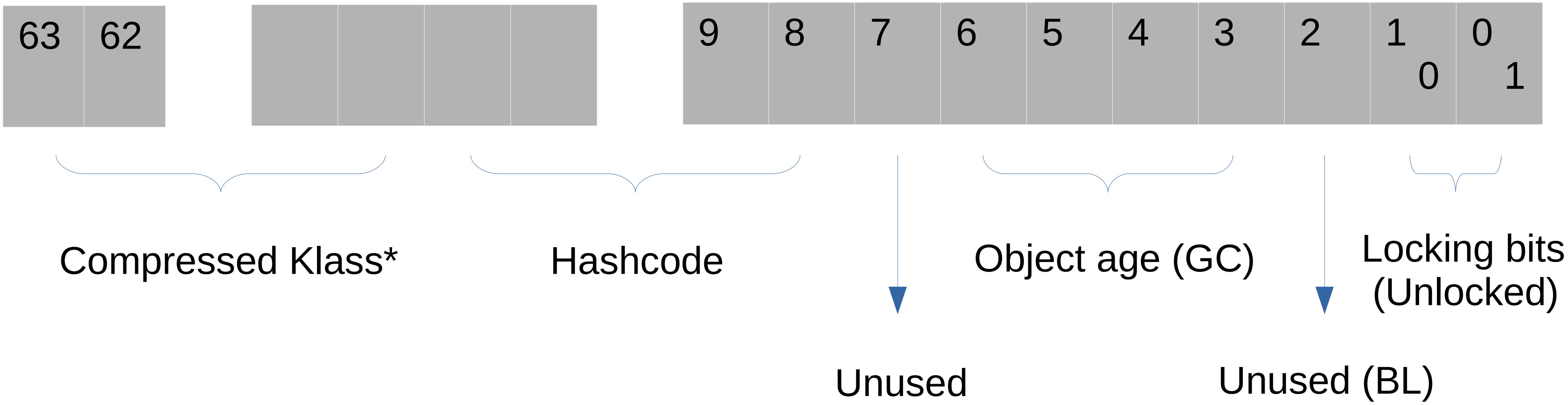


Existing object header - reminder



PLUS: Another 32 or 64 bits for the Klass*

Lilliput header

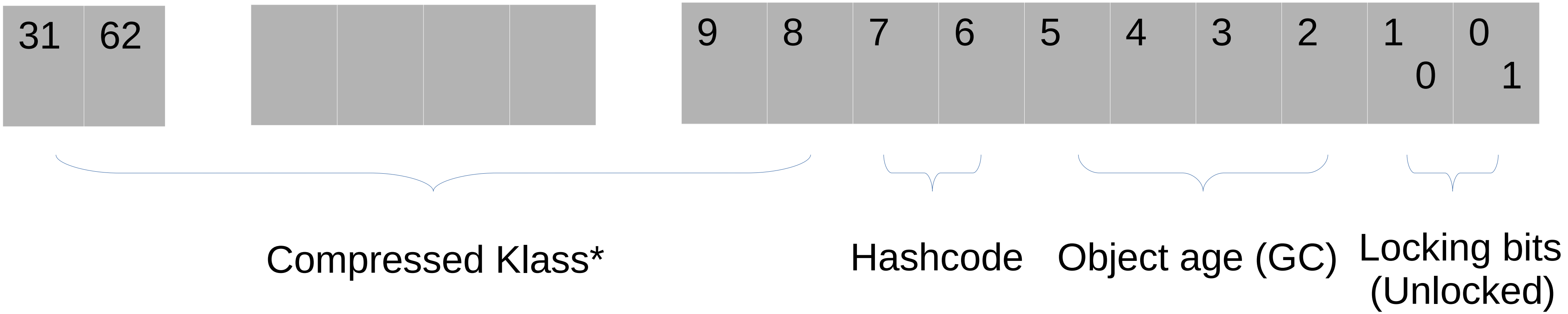


Possibly trade hash-code bits with compressed-class-ptr bits

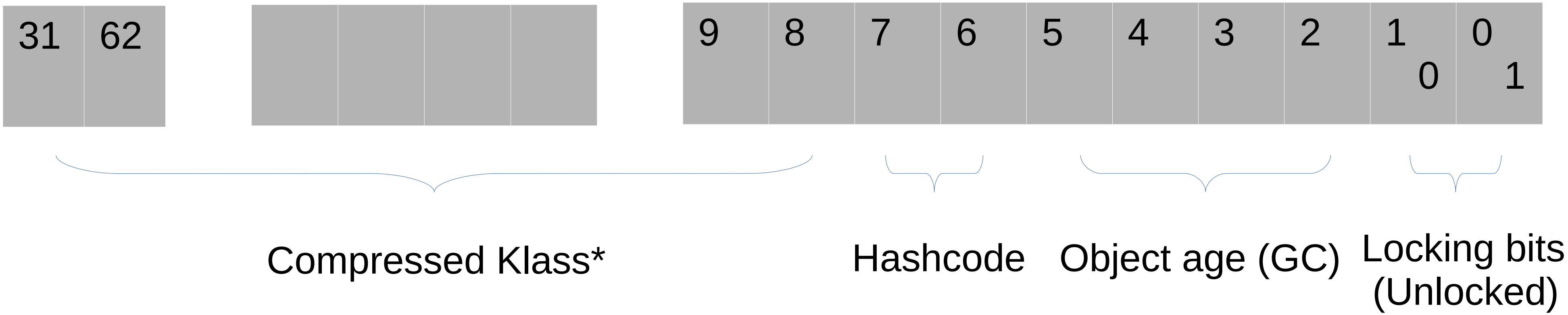
Lilliput – The Big Picture

- Can we do better?

Lilliput header – 32 bits version

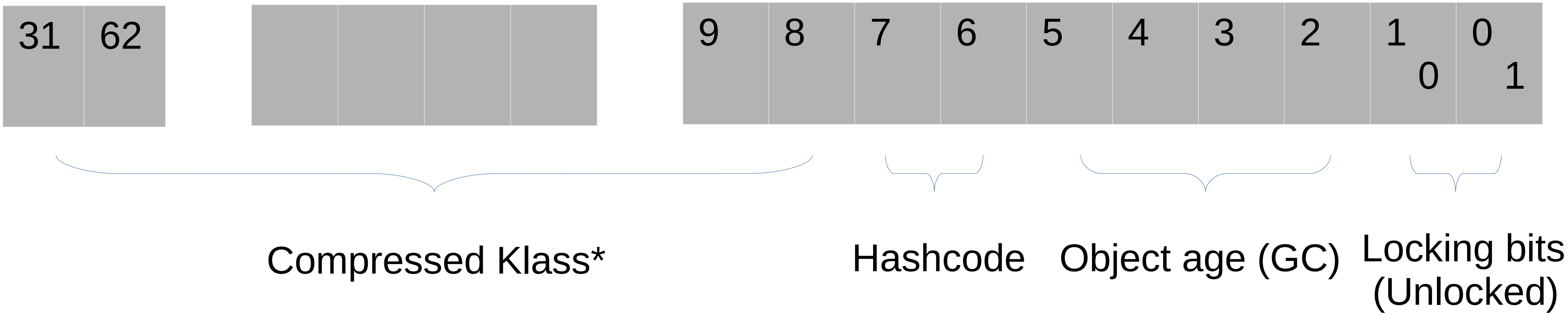


Lilliput header – 32 bits version



What to do with the upper 32 bits?

Lilliput header – 32 bits version



What to do with the upper 32 bits?
→ Store arraylength
→ Store first couple of fields

Identity Hash-Code

- You already know:
  - Object.hashCode() and Object.equals()
  - a.equals(b) → a.hashCode() == b.hashCode()
  - hash-code **should** be well-distributed

Identity Hash-Code

- Identity hash-code
  - System.identityHashCode() and ==
  - a == b → System.identityHashCode(a) == System.identityHashCode(b)
  - Default implementation of Object.hashCode()
  - Matches default implementation of Object.equals() (==)

I-hash approaches

- Use constant, e.g. return 42 for all objects
  - Valid, but worst distribution
  - Useful for debugging (-XX:hashCode=2)

I-hash approaches

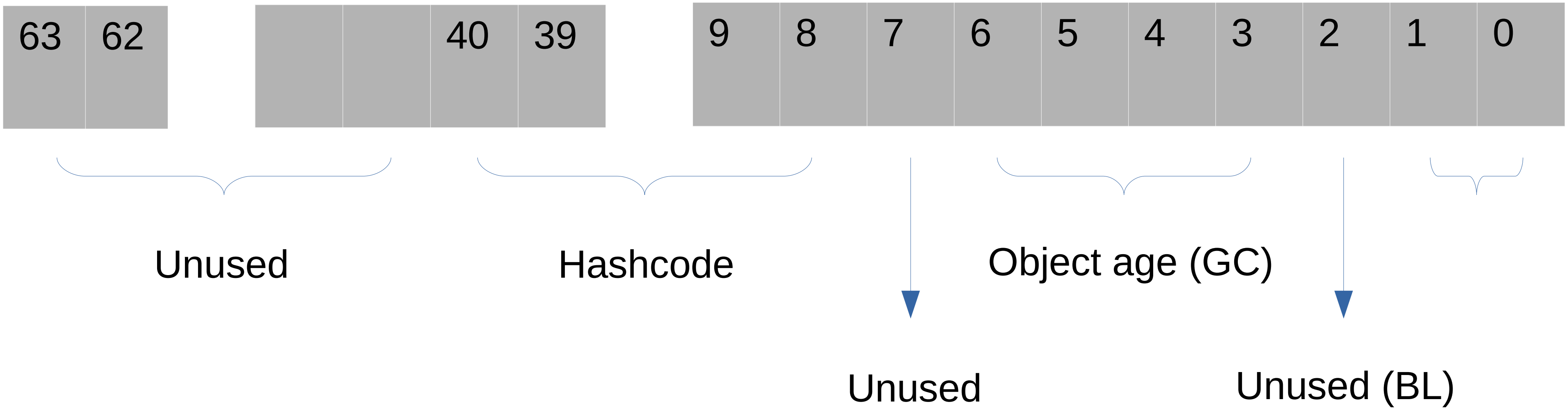
- Use object address
  - Bad hash distribution
  - Doesn‘t work with relocating GCs (IOW, all Hotspot GCs)
- Variant: Use some function of object address
  - e.g. murmur3(address)
  - Can solve distribution problem, but not relocation

I-hash approaches

- Random number
  - Is not idempotent, thus not valid i-hash

I-hash approaches

- Random number
  - Is not idempotent, thus not valid i-hash
- Solution: compute once, store i-hash in header



I-hash approaches

- Compute once, store i-hash in header
  - Can use any computation approach, e.g. RNG
  - Requires ~32 bits in object header
  - Most objects (>99%) are never hashed
  - This is what is currently implemented in Hotspot

I-hash approaches

- Compute hash, store when object moves
  - Can use any computation approach, e.g. RNG
  - Requires 2 bits in object header
  - Allocates storage only when needed
  - Many objects can fit 32bit hashcode in alignment gap
  - Requires support by GC

I-hash approaches

- Compute hash, store when object moves
  - Uses murmur3(address) as long as object doesn‘t move
  - When GC moves hashed object, it allocates extra storage, if needed (at the end of object)
  - Hash bits in header: 00 – not hashed, 01 – hashed, 10 hash installed, (11 – hashed & installed)

Storing the Klass*

- Plain pointer
  - Requires 64 bits
  - Currently implemented in Hotspot

Storing the Klass*

- Compressed pointer
  - Requires 32 bits
  - Currently implemented in Hotspot
  - Use remaining 32 bits for arraylength or first fields

Storing the Klass*

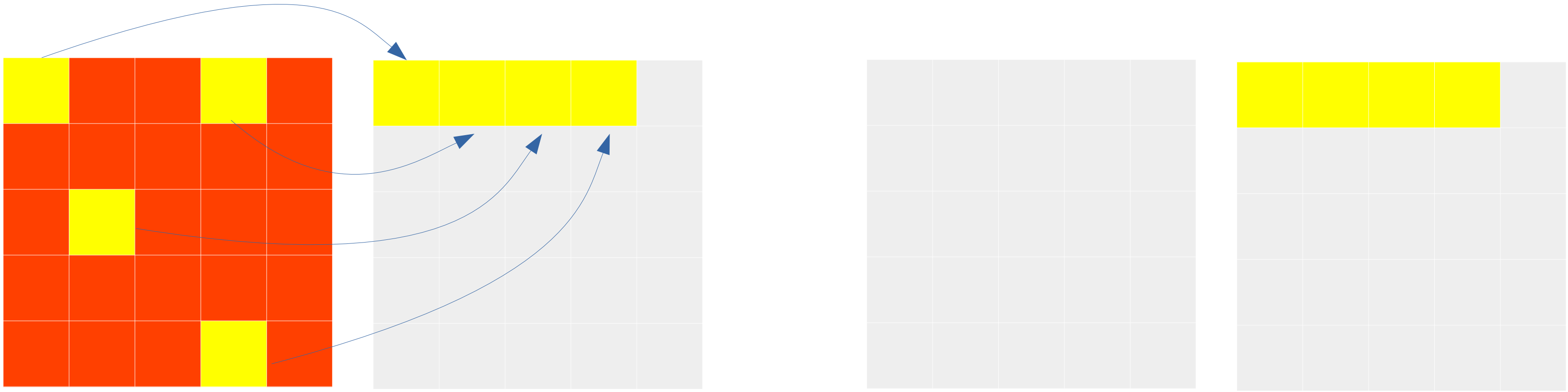
- Compressed pointer in mark-word
  - Requires 32 bits (or less)
  - Needs careful coordination with locking and GC to avoid overriding the Klass*
  - Can address 2^(nbits + shift) bytes of class space
  - High alignments (e.g. 1K) sensible because Klass objects are typically ‘large‘

Storing the Klass*

- Index into Klass* lookup table
  - Requires 32 bits (or less)
  - Needs careful coordination with locking and GC to avoid overriding the Klass*
  - Can address 2^nbits number of classes
  - Useful as last resort, if compressed classes is not enough (i.e. huge amount of classes)

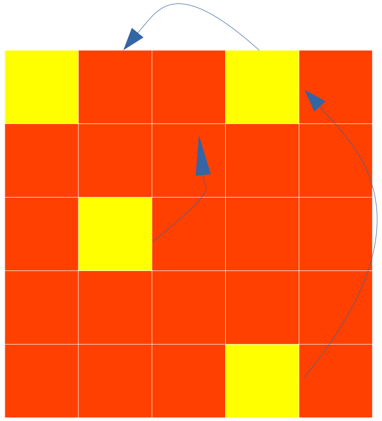
GC Forwarding

- Copying GCs (all Hotspot GCs) copy objects:
  - Copy live objects to empty to-space
  - Reclaim the whole from-space (all garbage now)



GC Forwarding

- Sliding GCs (most Hotspot GCs) copy objects:
  - Copy live objects ‘bottom sediment‘
  - Useful when no more room for to-space



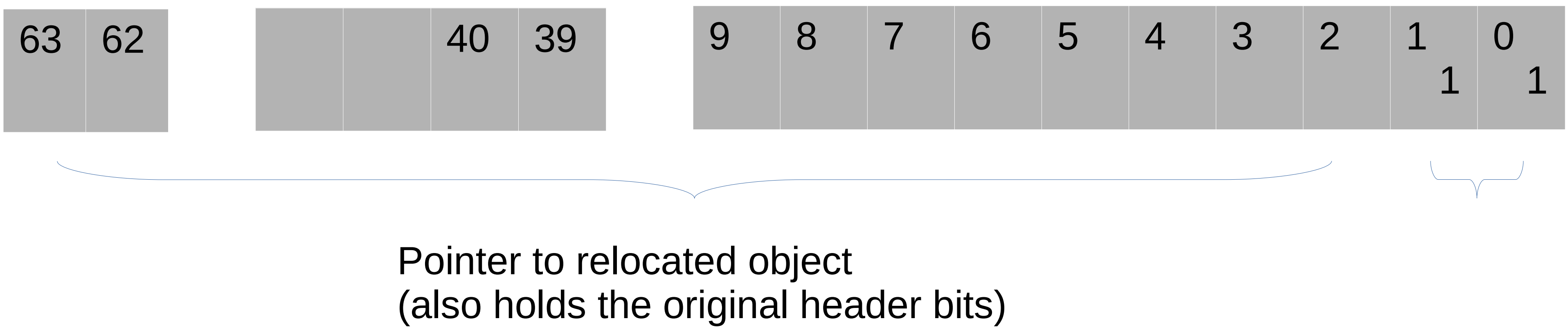
GC Forwarding

- All references to moved live objects must be updated

→ We need to store new location somewhere

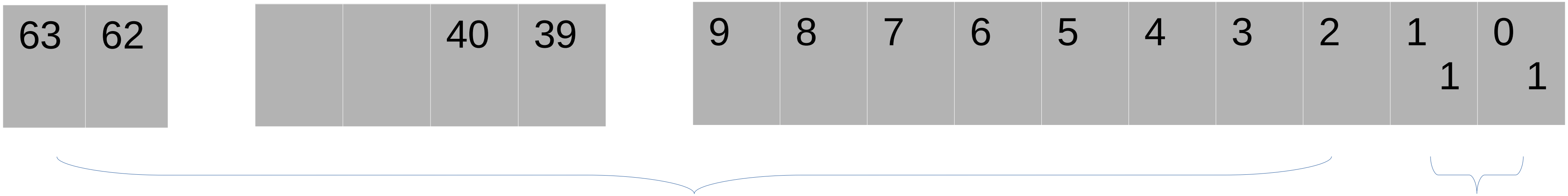
GC Forwarding

- Current solution: store in object header
- Interesting lower bits preserved in side-table



GC Forwarding

- Current solution: store in object header
- Interesting lower bits preserved in side-table
- Lilliput troubles: We override Klass* info



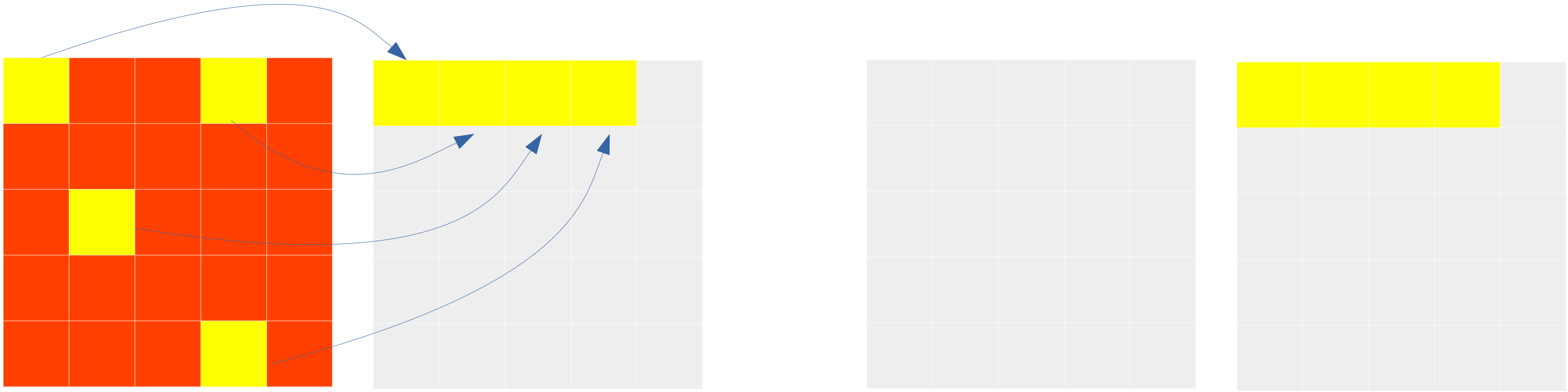
Pointer to relocated object
(also holds the original header bits)

GC Forwarding

- Forwarding Table
  - Requires off-heap storage
  - Access more complicated and potentially less performant than simple pointer-read-decode
  - Used by ZGC

GC Forwarding

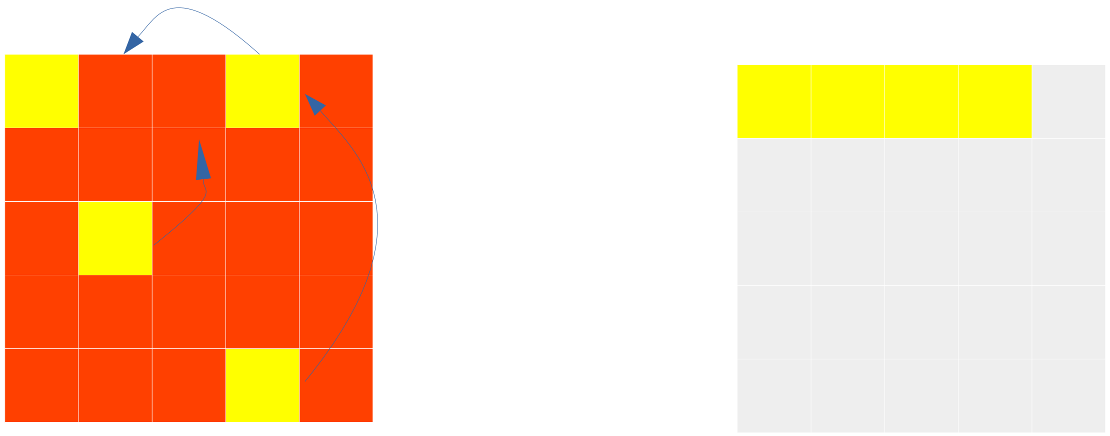
- Copying GCs (all Hotspot GCs) copy objects:
  - Copies are made **before** overriding old header
  - Careful iteration can avoid accessing old header



GC Forwarding

- Sliding GCs (most Hotspot GCs) copy objects:
  - Objects are copied **after** we override old header

    → we need to preserve original Klass*

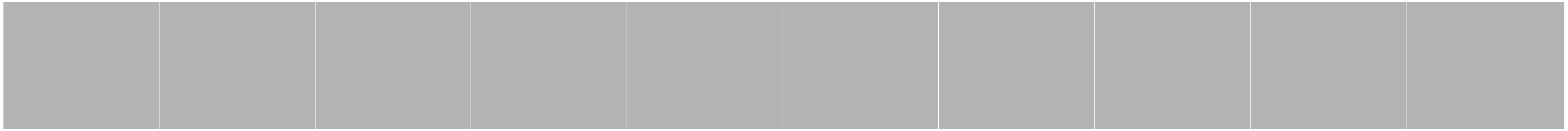


GC Forwarding – preserving Klass*

- Klass* resides in upper ~half of object header
- We can use lower ~half for storing compressed pointers
- Regular pointer compression (+UseCompressedOops) not generally available (e.g. >32GB heaps)
- For sliding compaction, we can do better

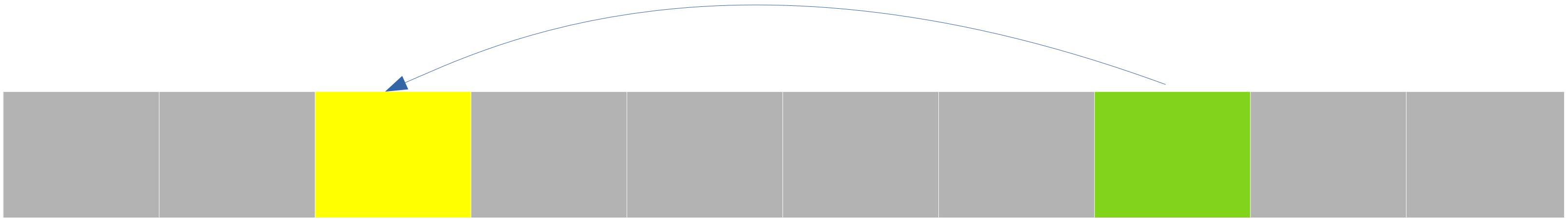
Sliding forwarding ptr compression

- Divide heap into sliding windows



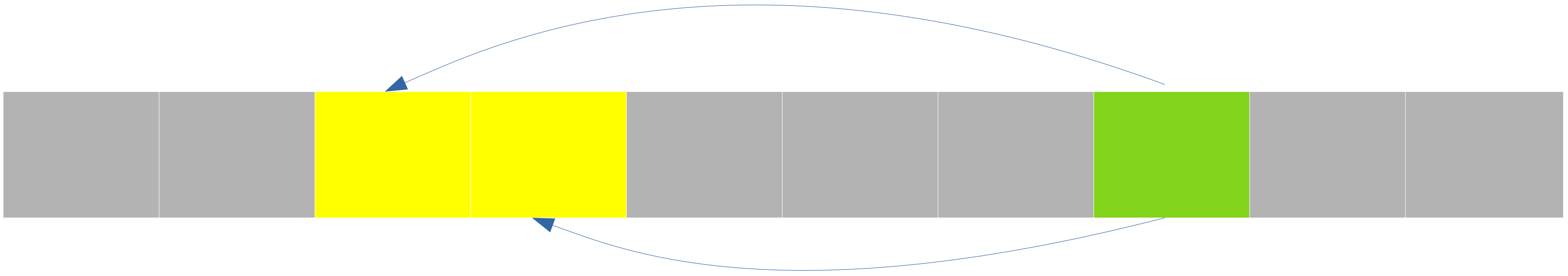
Sliding forwarding ptr compression

- Divide heap into sliding windows
- Objects from each one window only ever ‚slide‘ to one of two possible target windows

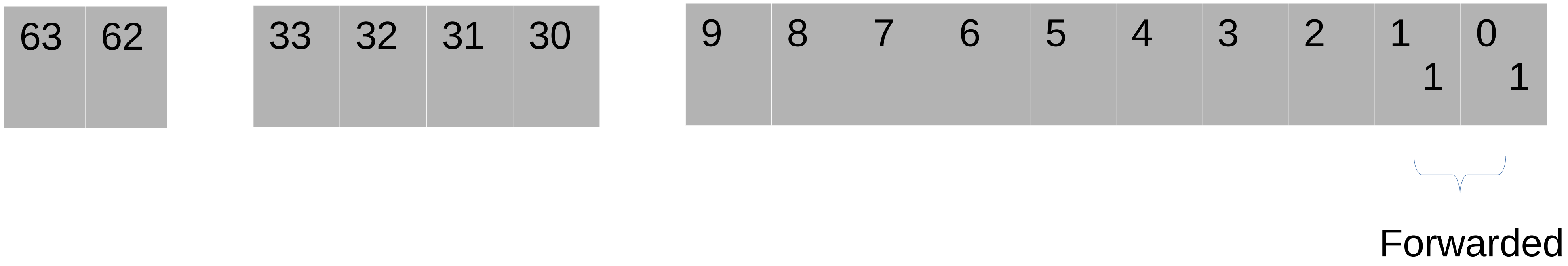


Sliding forwarding ptr compression

- Divide heap into equal-sized sliding windows
- Objects from each window only ever ‚slide‘ to one of two possible target windows

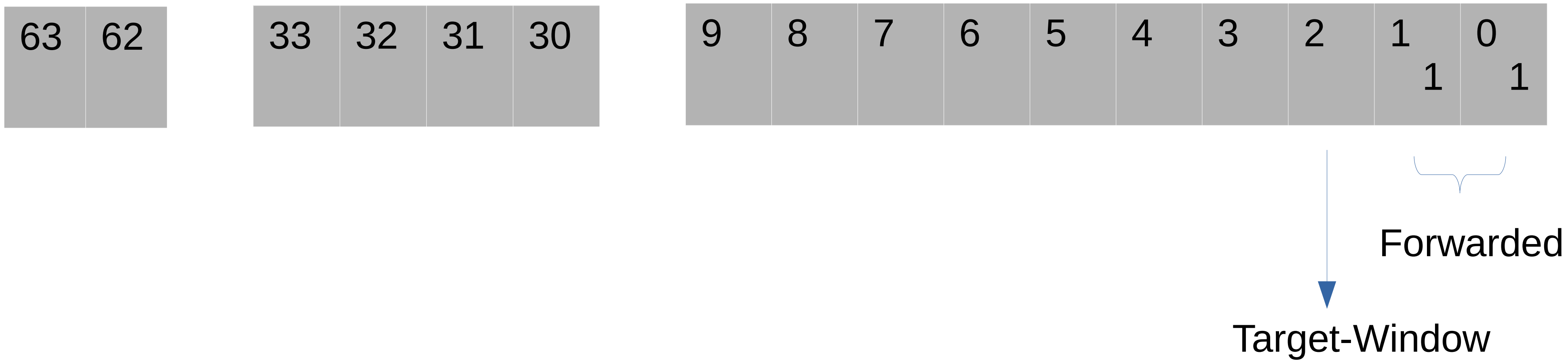


Sliding forwarding ptr compression



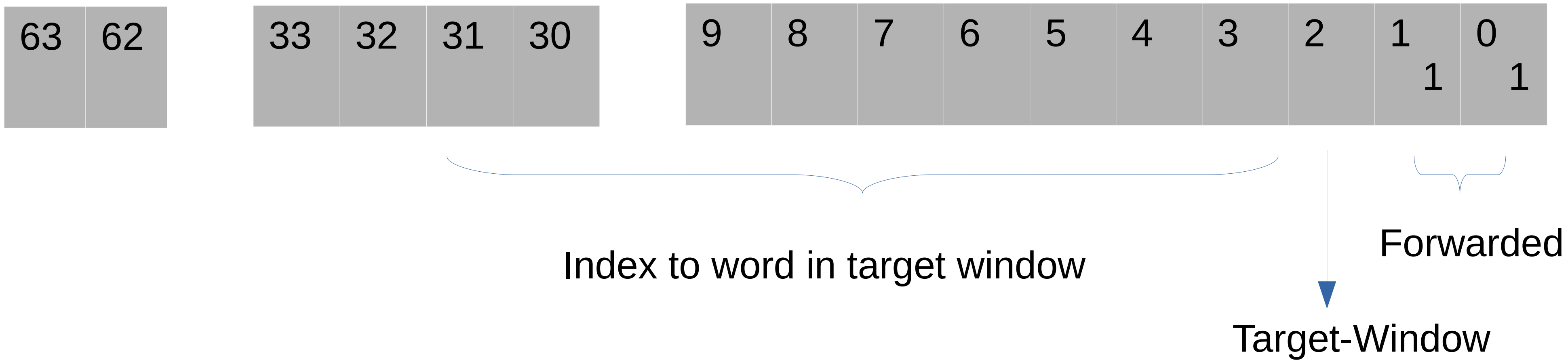
Sliding forwarding ptr compression

- Side-table: maps window → target window



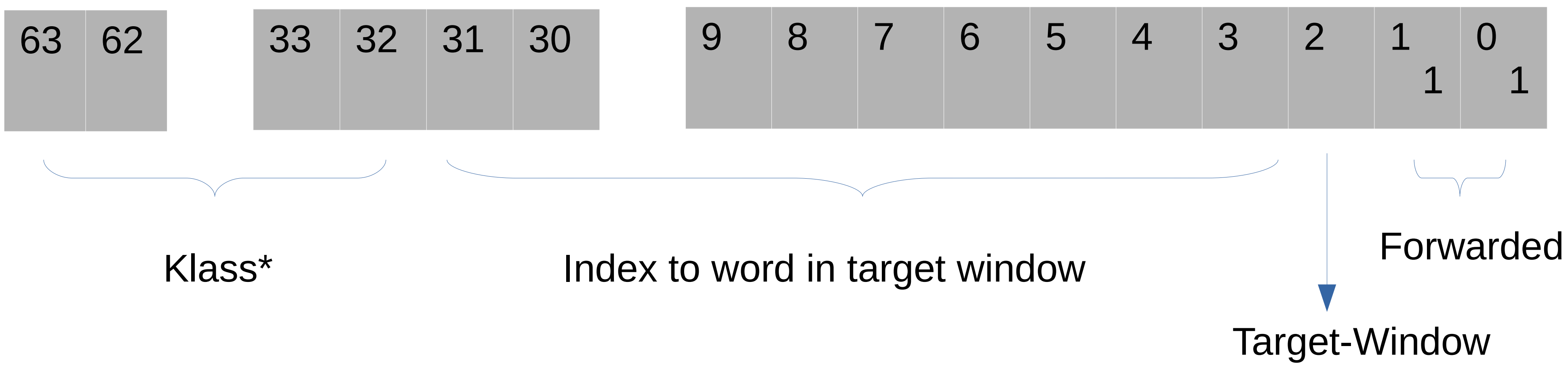
Sliding forwarding ptr compression

- Side-table: maps window → target window
- 2^28 = 268M words = 2GB per window



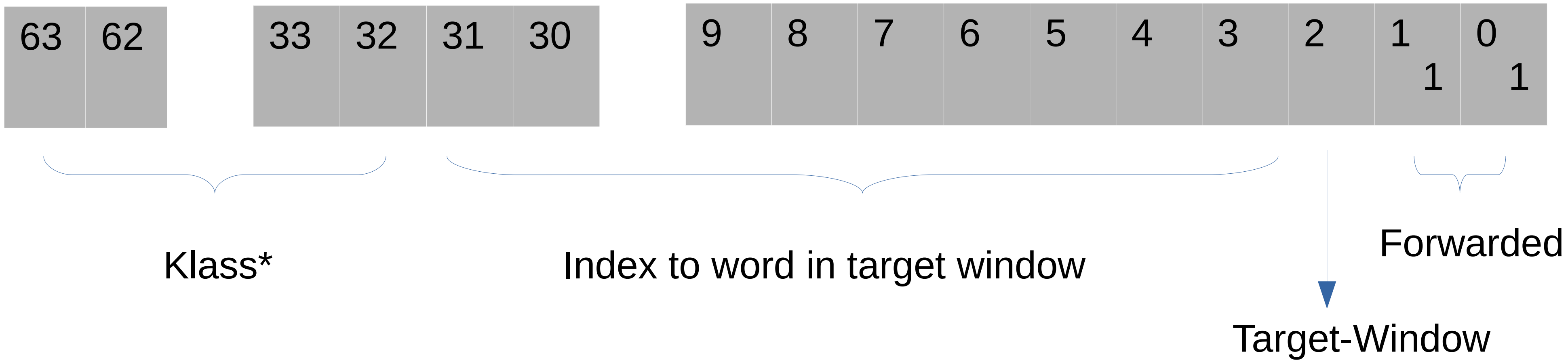
Sliding forwarding ptr compression

- Side-table: maps window → target window
- 2^28 = 268M words = 2GB per window



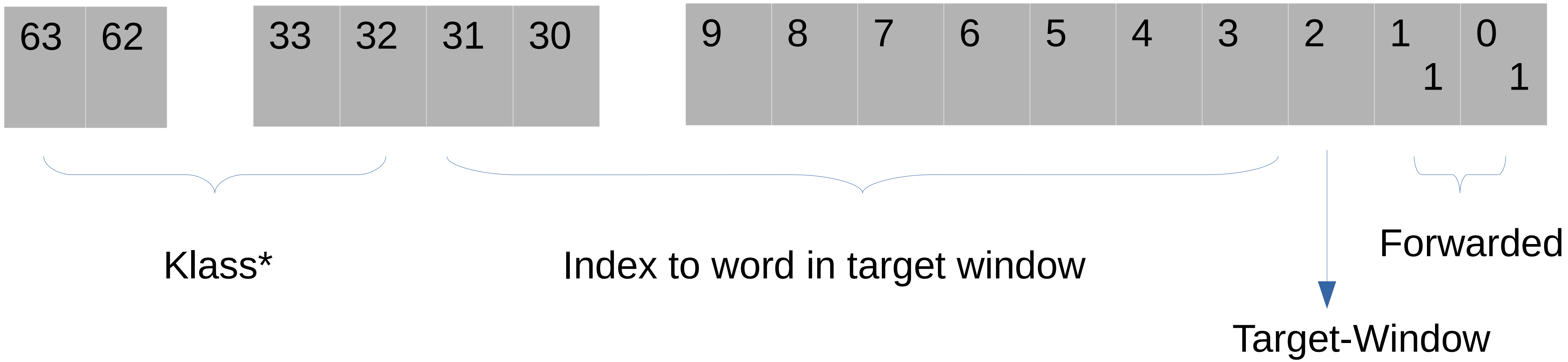
Sliding forwarding ptr compression

- Side-table: maps window → target window
- 2^29 = 512M words = 4GB per window
- Interesting lower bits preserved in side-table



Sliding forwarding ptr compression

- Max 4GB per window
- Windows can be chosen arbitrarily
- Some GCs (G1, Shenandoah) can use regions



GC Age

- What to do with the 4 GC age bits?

GC Age

- What to do with the 4 GC age bits?

Nothing: leave them alone

GC Age

- What to do with the 4 GC age bits?

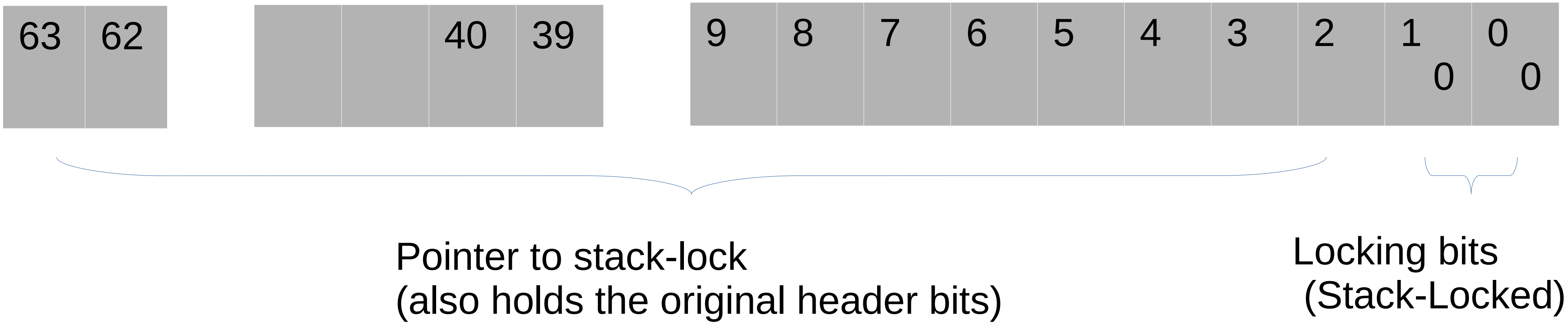
Nothing: leave them alone

(maybe cut 1 bit if really needed)

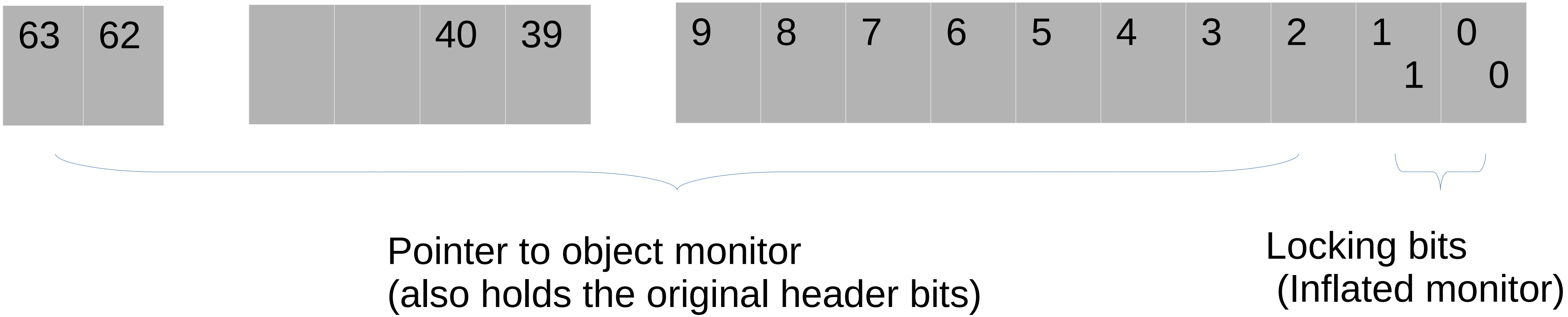
Locking

- Most research-y topic

Anatomy of object headers



Anatomy of object headers



Locking

- Thin (stack-) locks:
  - For simplest locking/unlocking ops
  - Header points to location into locking thread
  - Becomes inflated to full monitor upon contention
  - Fast and racy (wrt header access)

Locking

- Fat locks / monitors
  - Inflated from stack-locks, upon contention
  - Wait/Notify support
  - JNI
  - Deflated concurrently since JDK 15

Locking

- Lilliput troubles:
  - Displacement of header
  - How to safely access Klass* when locking messes with the header?

Locking

- Thin locks
  - Option 1: Lightweight inflation protocol
  - Temporarily install INFLATING token to prevent concurrent threads from making mess
  - Then access mark-word/Klass* without race
  - Don‘t actually inflate to monitor
  - Could degrade performance, especially GC threads

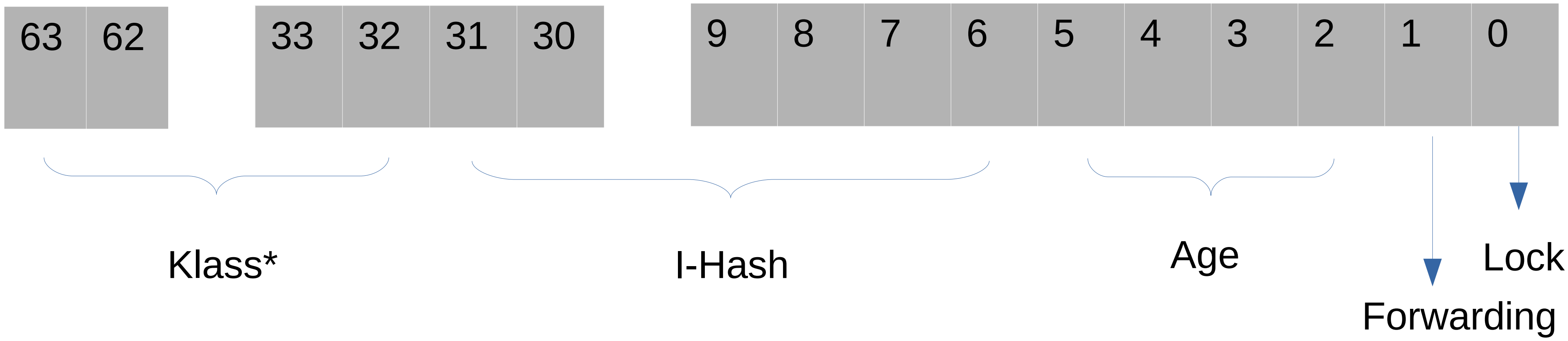
Locking

- Thin locks
  - Option 2: Remove thin-locks altogether
  - Greatly simplifies locking and header-access code
  - Not as important as it was 20 years ago
  - Might be useful (required?) for Project Loom
  - Performance gains elsewhere probably outweight performance loss in locking

- Fat locks
  - Less trouble: once installed, can be accessed safely from Java threads
  - Need to rendezvous GC threads to avoid race with deflation
  - Pointers to monitors should not be difficult to compress to <32bits

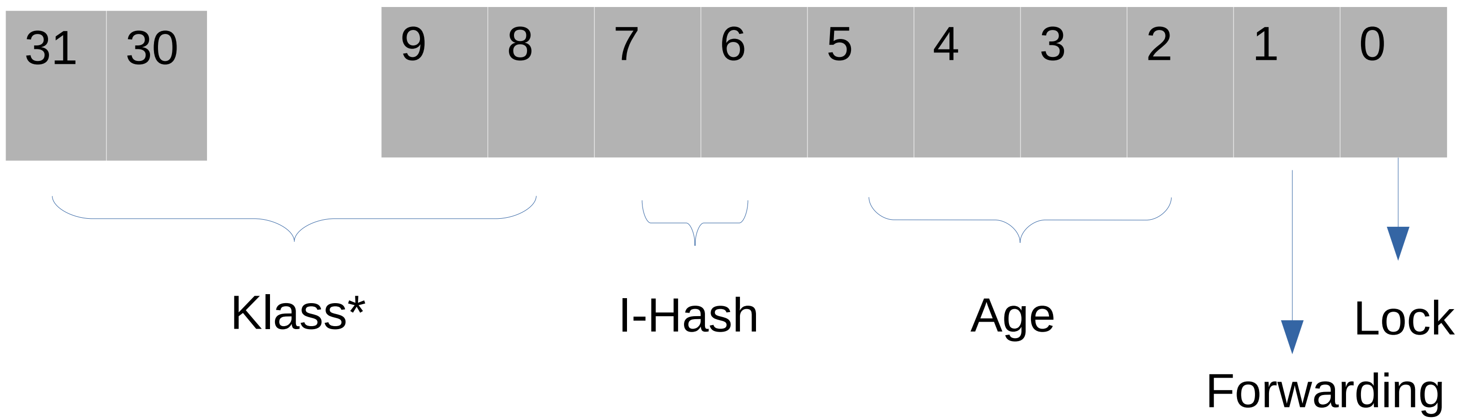
Putting it all together

- First stage: 64 bit header



Putting it all together

- Second stage: 32 bit header



Interferences

- Valhalla:
  - May want 1 or more header bits
  - We can probably trade Klass-bits

Interferences

- Valhalla:
  - May want 1 or more header bits
  - We can probably trade Klass-bits
- Loom
  - Rewrite locking
  - Benefits Lilliput (avoids displaced headers)

Lilliput Release

- Lilliput will show up in your friendly JDK release...

Lilliput Release

- Lilliput will show up in your friendly JDK release…   when it‘s ready.

Lilliput

- https://openjdk.java.net/projects/lilliput/
- https://wiki.openjdk.java.net/display/lilliput