Building Memory-efficient Java Applications: Practices and Challenges

Nick Mitchell, Gary Sevitsky (presenting)
IBM TJ Watson Research Center   Hawthorne, NY USA

Copyright is held by the author/owner(s).
ACM SIGPLAN PLDI 2009, Dublin, Ireland

Quiz

Small boxes?

Q: What is the size ratio of Integer to int?

a. 1 : 1

b. 1.33 : 1

c. 2 : 1

d. ?

Assume 32-bit platform

Small things?

Q: How many bytes in an 8-character String?

a. 8

b. 16

c. 28

d. ?

Assume 32-bit platform

Bigger? Better?

Q: Which of the following is true about HashSet relative to HashMap

a. does less, smaller

b. does more, smaller

c. similar amount of functionality, same size

d. ?

The big pile-up

Heaps are getting bigger

- Grown from 500M to 2-3G or more in the past few years
- But not necessarily supporting more users or functions

Surprisingly common:

- requiring 1G memory to support a few hundred users
- saving 500K session state per user
- requiring 2M for a text index per simple document
- creating 100K temporary objects per web hit

Consequences for scalability, power usage, and performance

Common thread

- It is easy to build systems with large memory requirements for the work accomplished
- Overhead of representation of data can be 50-90%
  - Not counting duplicate data and unused data

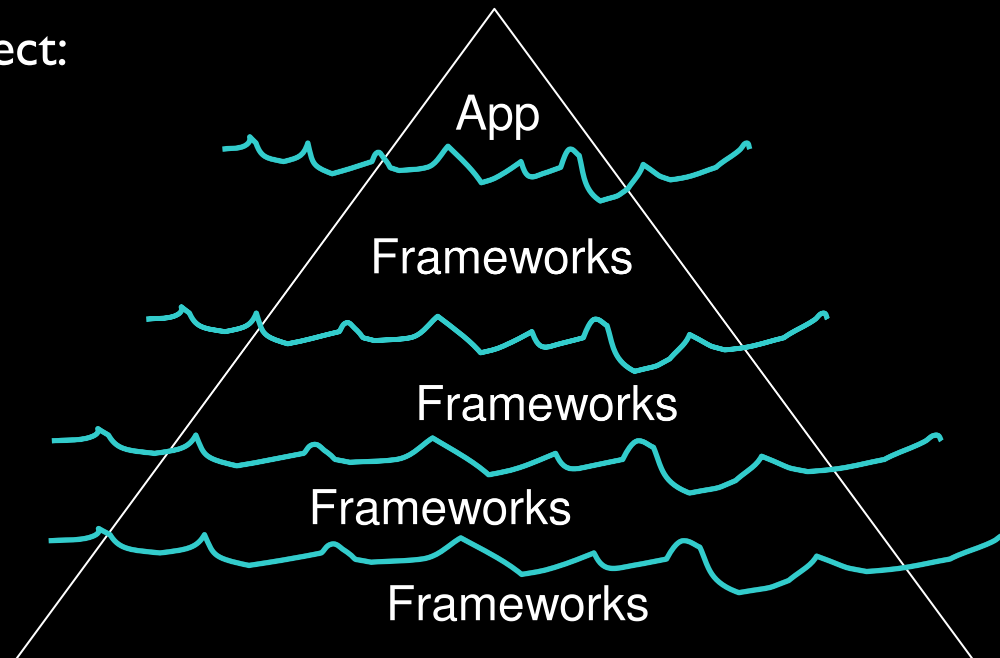
The big pile-up

Not a reflection on the quality of programmers – many are expert

More abstractions = less awareness of costs

- It is easy for costs to pile up, just piecing together building blocks

The iceberg effect:



Myths

Things are fine

Objects (or Strings, HashMaps, …) are cheap

Frameworks are written by experts, so they've been optimized (for my use case!)

The JIT and GC will fix everything

Things are not fine

I knew foo was expensive; I didn't know it was this expensive!

It's no use: O-O plus Java is always expensive

Efficiency is incompatible with good design

Data type health

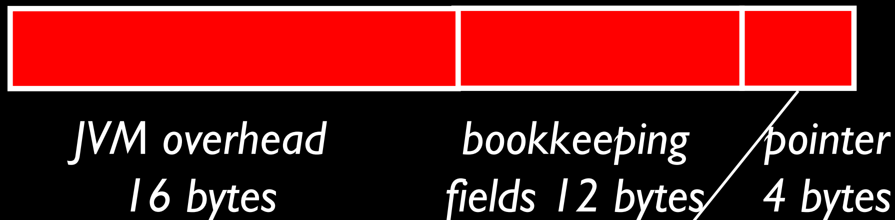
One Double

- 33% is actual data
- 67% is the representation overhead
- From one 32-bit JVM. Varies with JVM, architecture.

Data type health

Example: An 8-character String

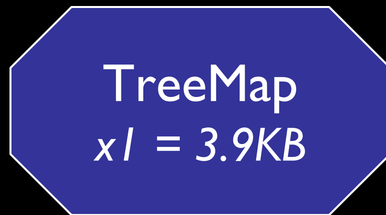
8-char String
64 bytes

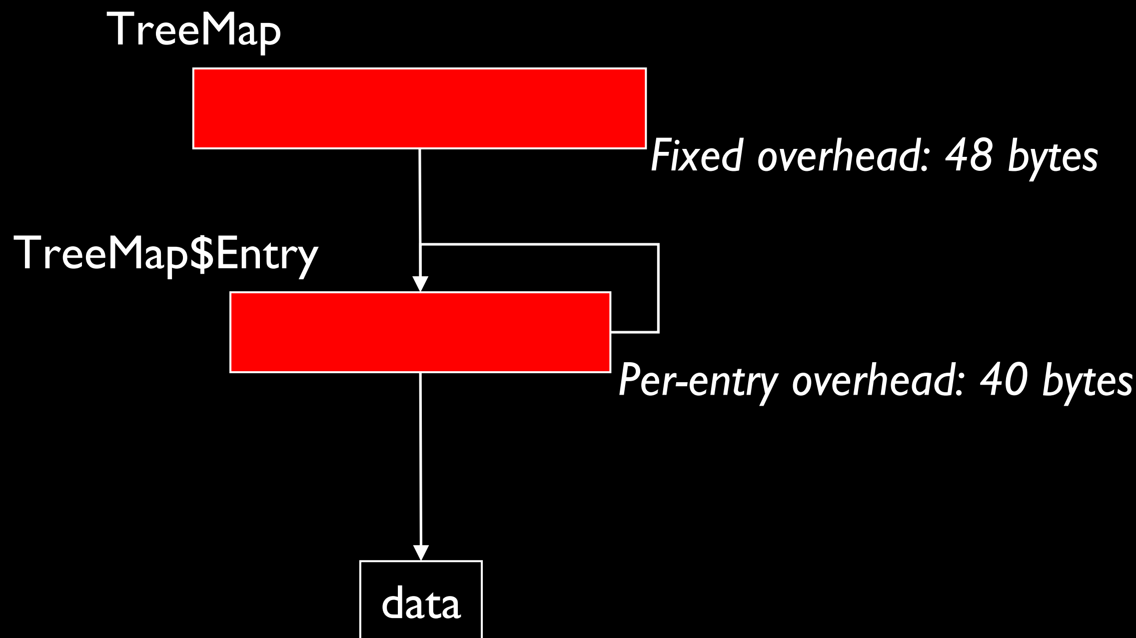


- only 25% is the actual data
- 75% is overhead of representation
- would need 96 characters for overhead to be 20% or less

Collection health

A 100-entry TreeMap

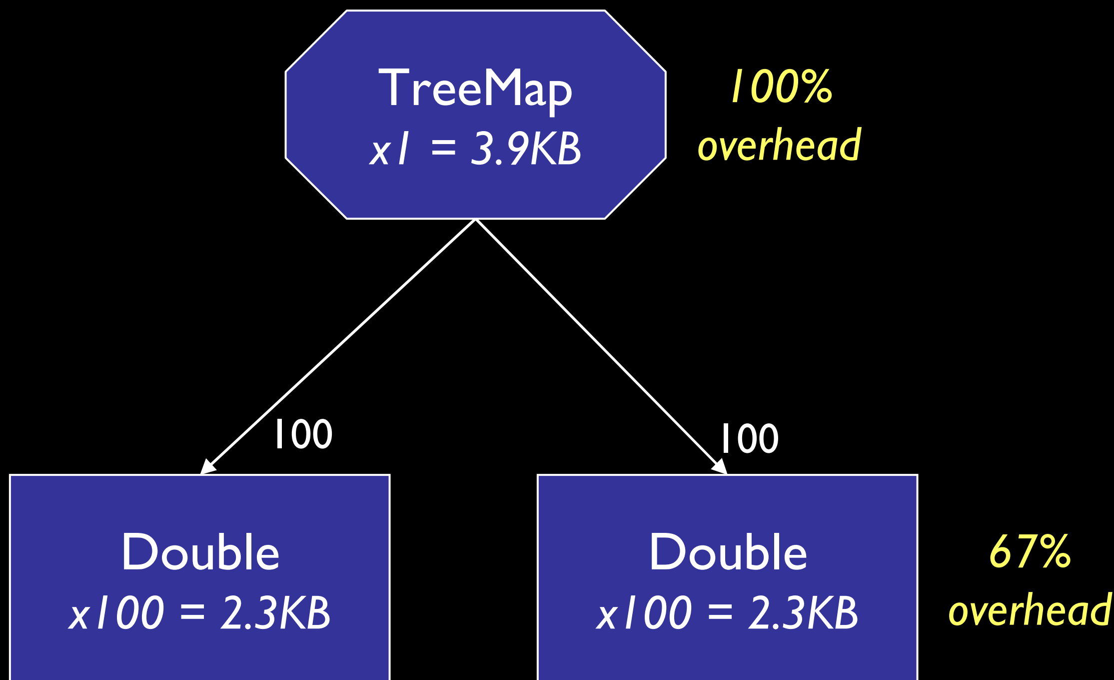




- How does a TreeMap spend its bytes?
- Collections have fixed and variable costs

Data structure health

TreeMap<Double, Double> (100 entries)



- 82% overhead overall
- Design enables updates while maintaining order
- Is it worth the price?

Data structure health

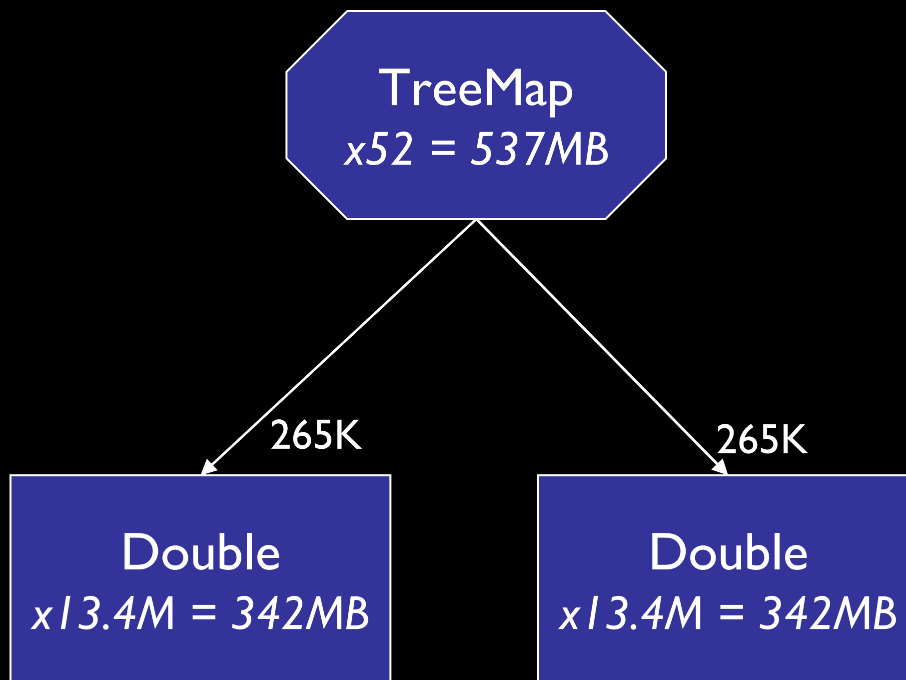
Alternative implementation (100 entries)

double[] *1x = 816 bytes*	double[] *1x = 816 bytes*	*2% overhead*

- Binary search against sorted array
- Less functionality – suitable for load-then-use scenario
- 2% overhead

Collections involving scalars

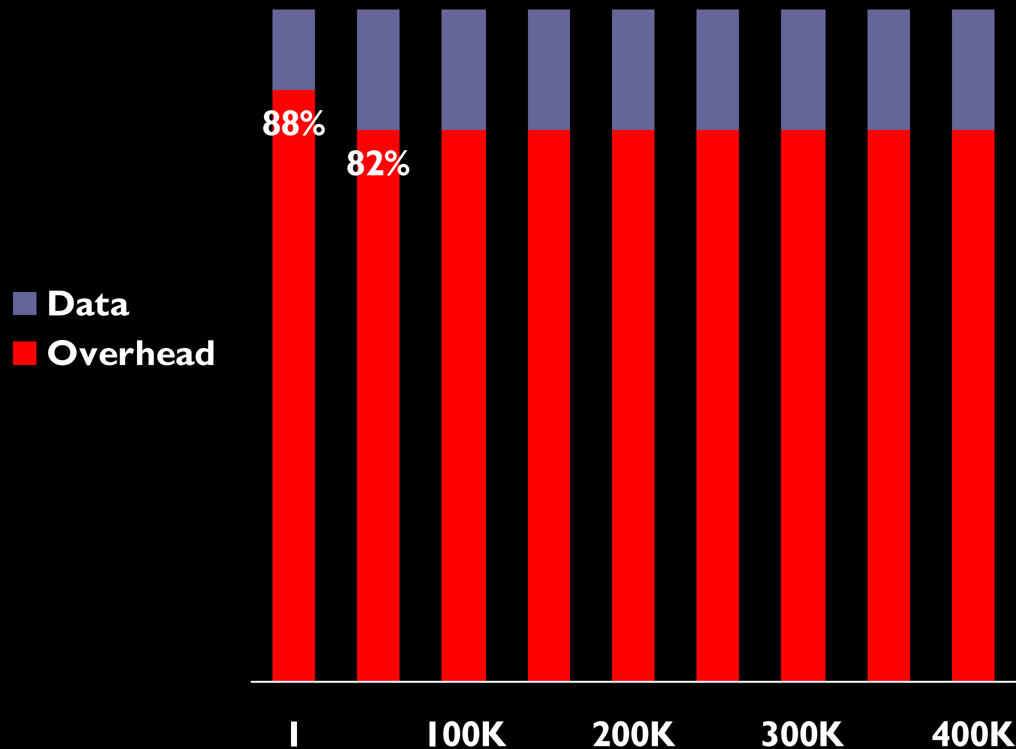
Case study: monitoring infrastructure



- Data structure took 1.2GB
- Overhead is still 82% at this giant scale
- Some alternative scalar maps/collections available, with much lower overhead

Health as a gauge of scalability

TreeMap<Double, Double>

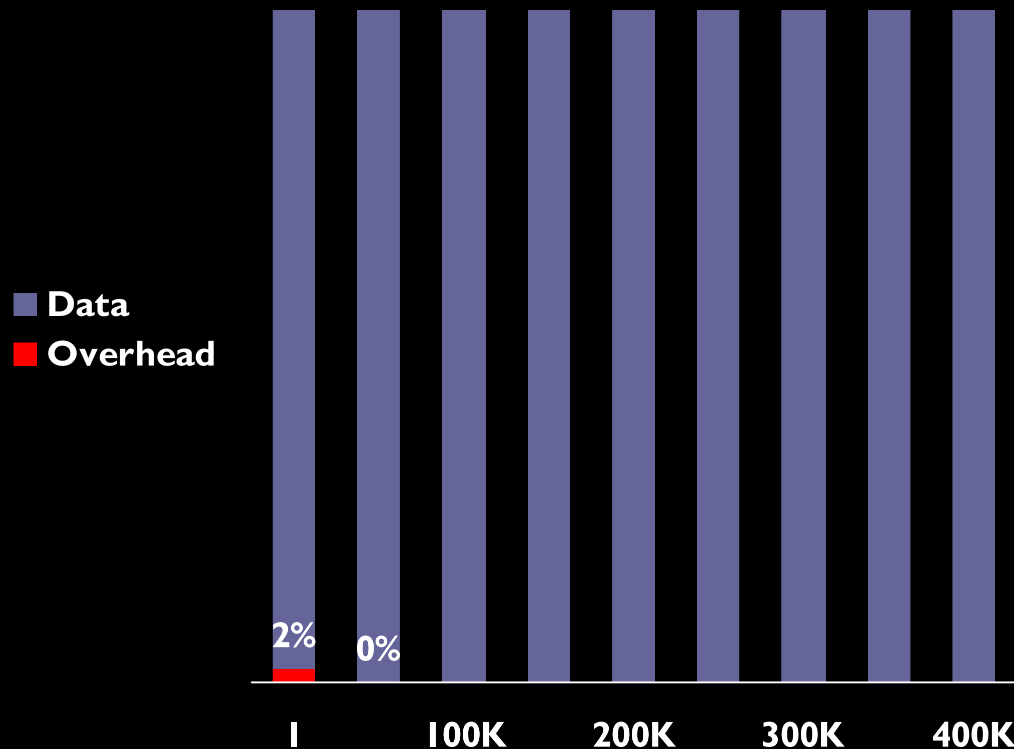


- Overhead is still 82% of cost
- Overhead is not amortized in this design
- High constant cost per element: 88 bytes

Health as a gauge of scalability

Alternative implementation

~ 0% overhead



- Overhead starts out low, quickly goes to 0
- Cost per element is 16 bytes, pure data

Background: the cost of objects

Boolean
16 bytes

header *12 bytes*　*boolean* *1 byte*　*alignment* *3 bytes*

Double
24 bytes

header *12 bytes*　*double* *8 bytes*　*alignment* *4 bytes*

char[2]
24 bytes

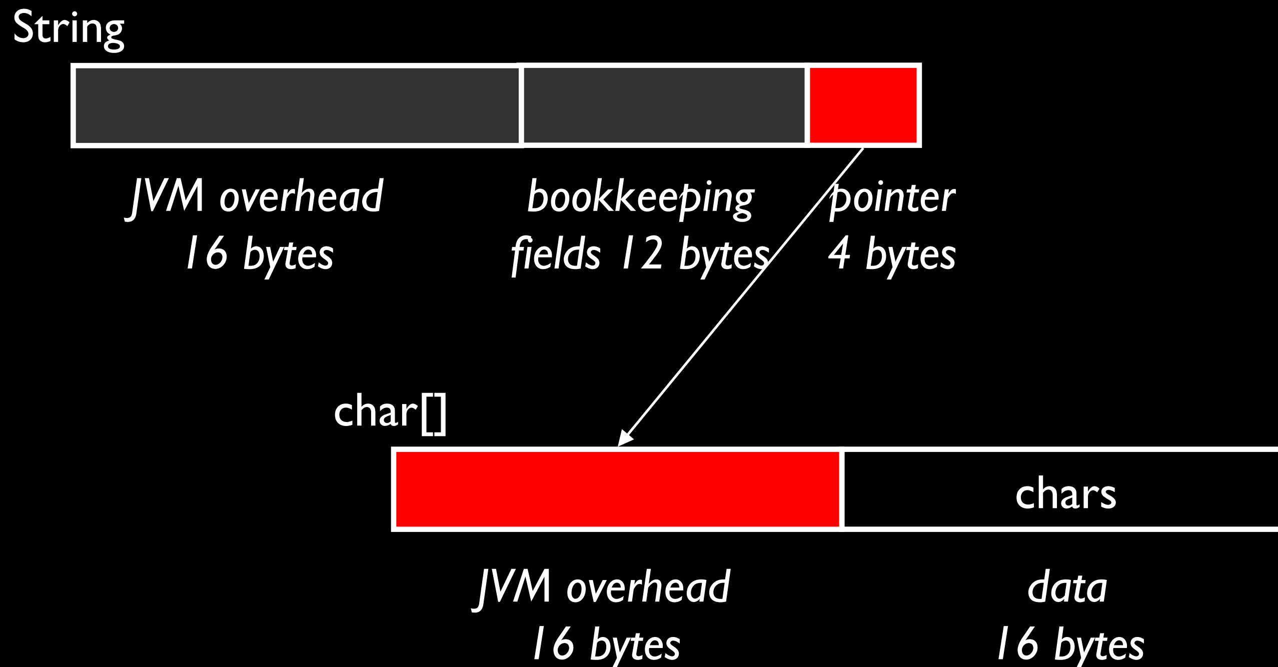
header *16 bytes*　*2 chars* *4 bytes*　*alignment* *4 bytes*

From experiment on one 32-bit JVM

- JVM & hardware impose costs on objects. Can be substantial for small objects
- Headers enable functionality and performance optimizations
- 8-byte alignment in this JVM
- Costs vary with JVM, architecture

The cost of delegation

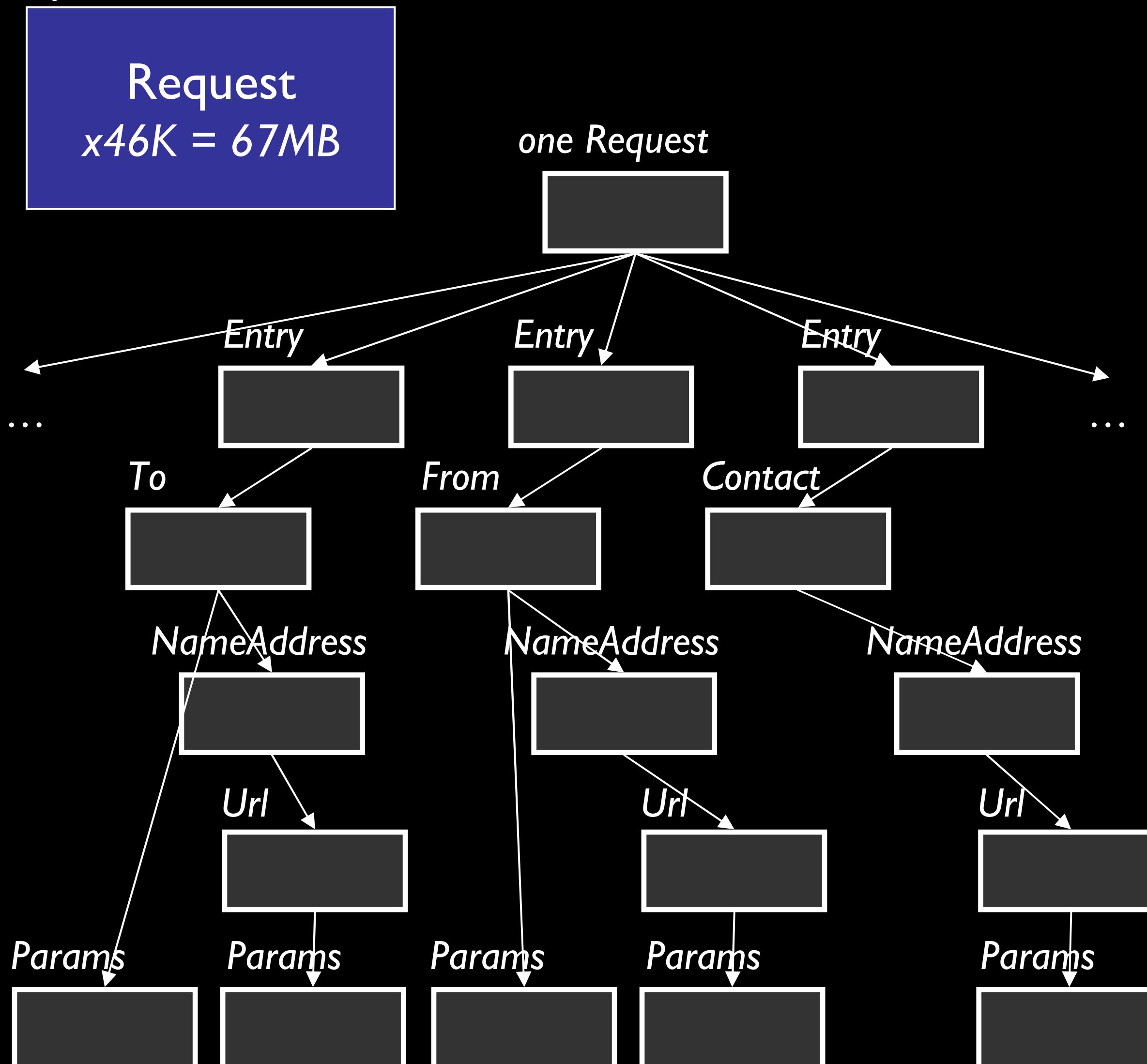
Example: An 8-character String



- 31% is overhead due to modeling as two objects
- Effect varies with size of String

Fine-grained modeling

Case study: server framework, part of connection



- 34 instances to represent a request. Cost: 1.5K per request. Will not scale.
- 36% of cost is delegation overhead
- Constant overhead per Request
- Can magnify the costs of other choices

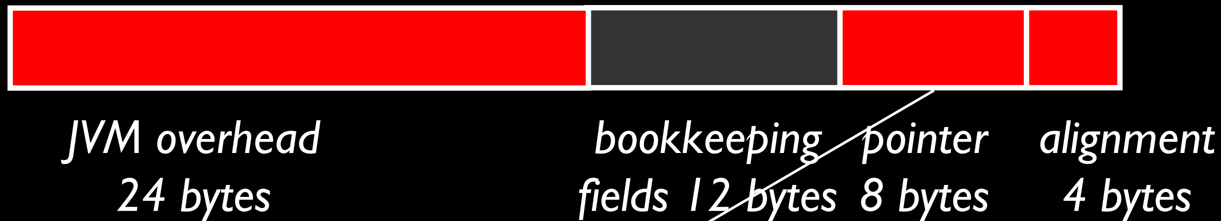
32- vs. 64-bit

Example: An 8-character String

8-char String
96 bytes

- 50% larger
- Delegated design is responsible for extra object header and pointer costs
- Fine-grained designs incur especially high costs

String



char[]

Data type modeling: challenges for developers

- Java's limited data modeling means tradeoffs require care
  - Moving rarely-used fields to side objects incurs delegation costs
  - Moving sparse fields to a map incurs high map entry costs
  - Verifying actual costs and benefits is essential
- Fixing problems of high-overhead data usually means refactoring data models
  - Not easy late in the cycle
  - Using interfaces and factories up front can help

Data type modeling: community challenges

- Many more objects and pointers than other languages
  - x high per-object cost = 35% delegation overhead avg in heaps

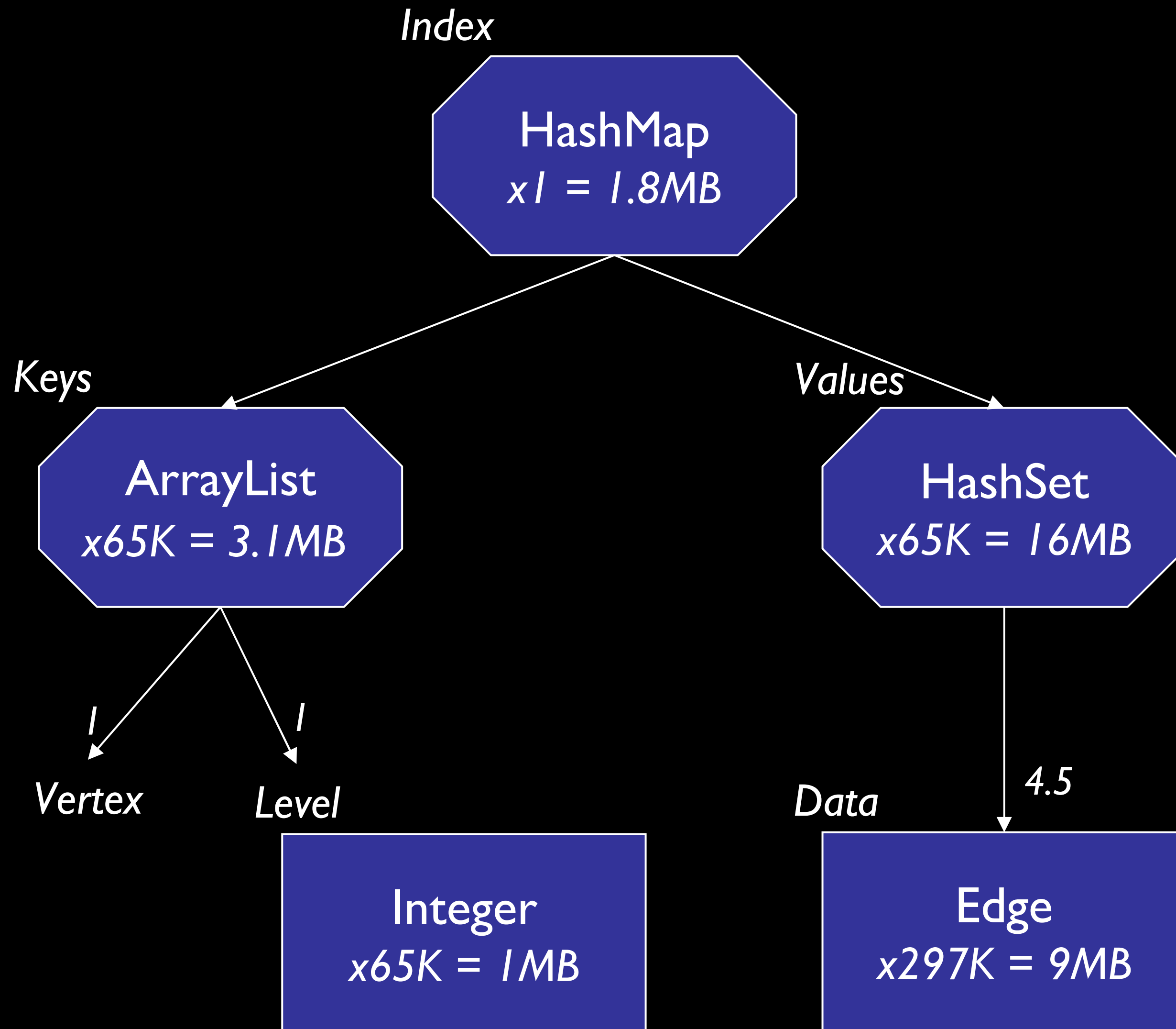
- Only two options for achieving variation – both are expensive
  - delegation vs. unused fields (large base classes)
  - both limit higher-level choices and magnify carelessness

- Consequences all the way up the stack
  - Primitives as object(s): String, Date, BigDecimal, boxed scalars
  - Collections suffer these limitations
  - Many layers of frameworks implementing systems functionality

- Solutions in the language / runtime?

Small collections in context

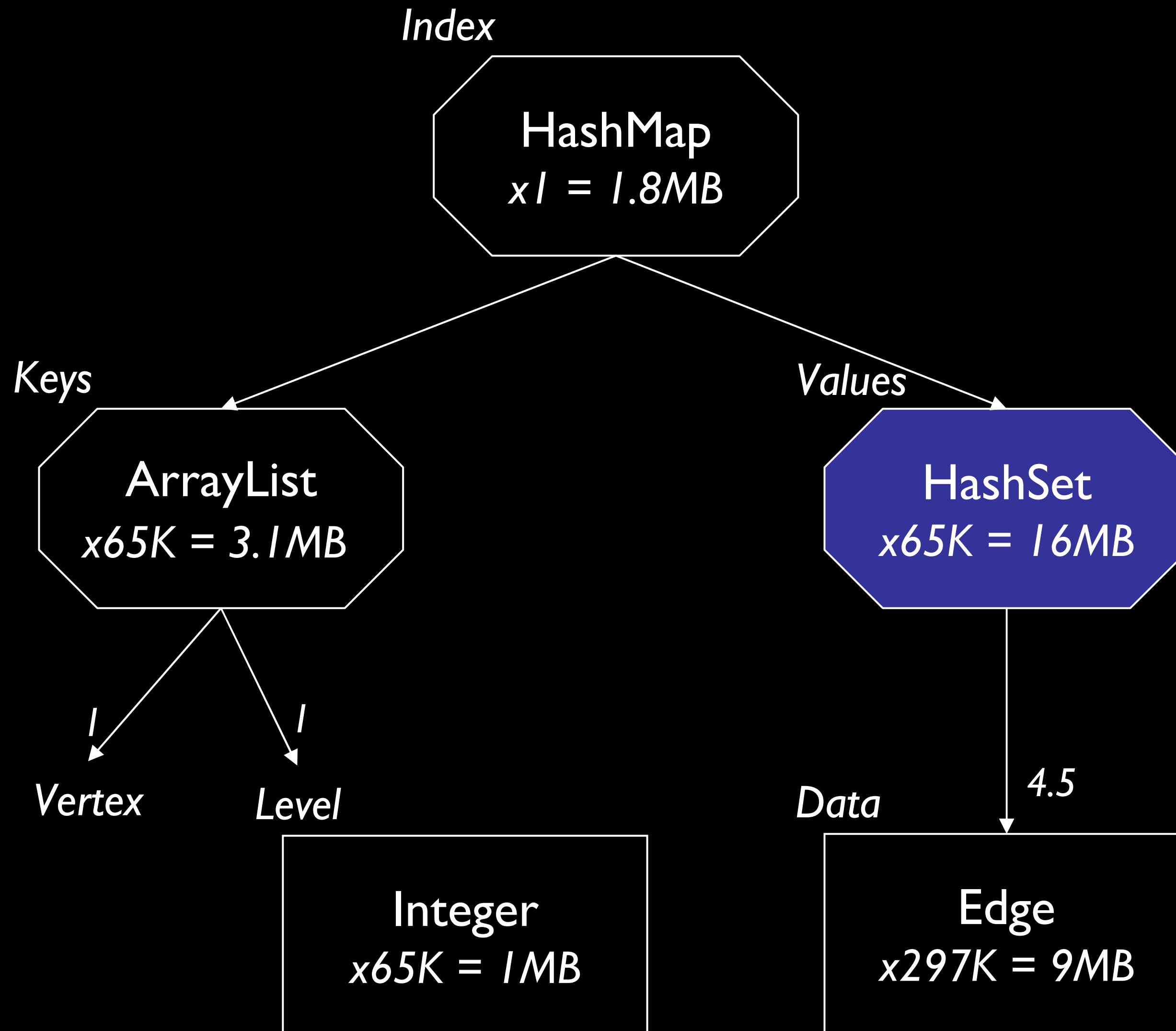
Case study: Planning system, level graph edges



- Two examples of small high-overhead collections
- 297K edges cost 31MB
- Overhead of representation: 83%
- Overhead will not improve with more vertices

Small collections in context

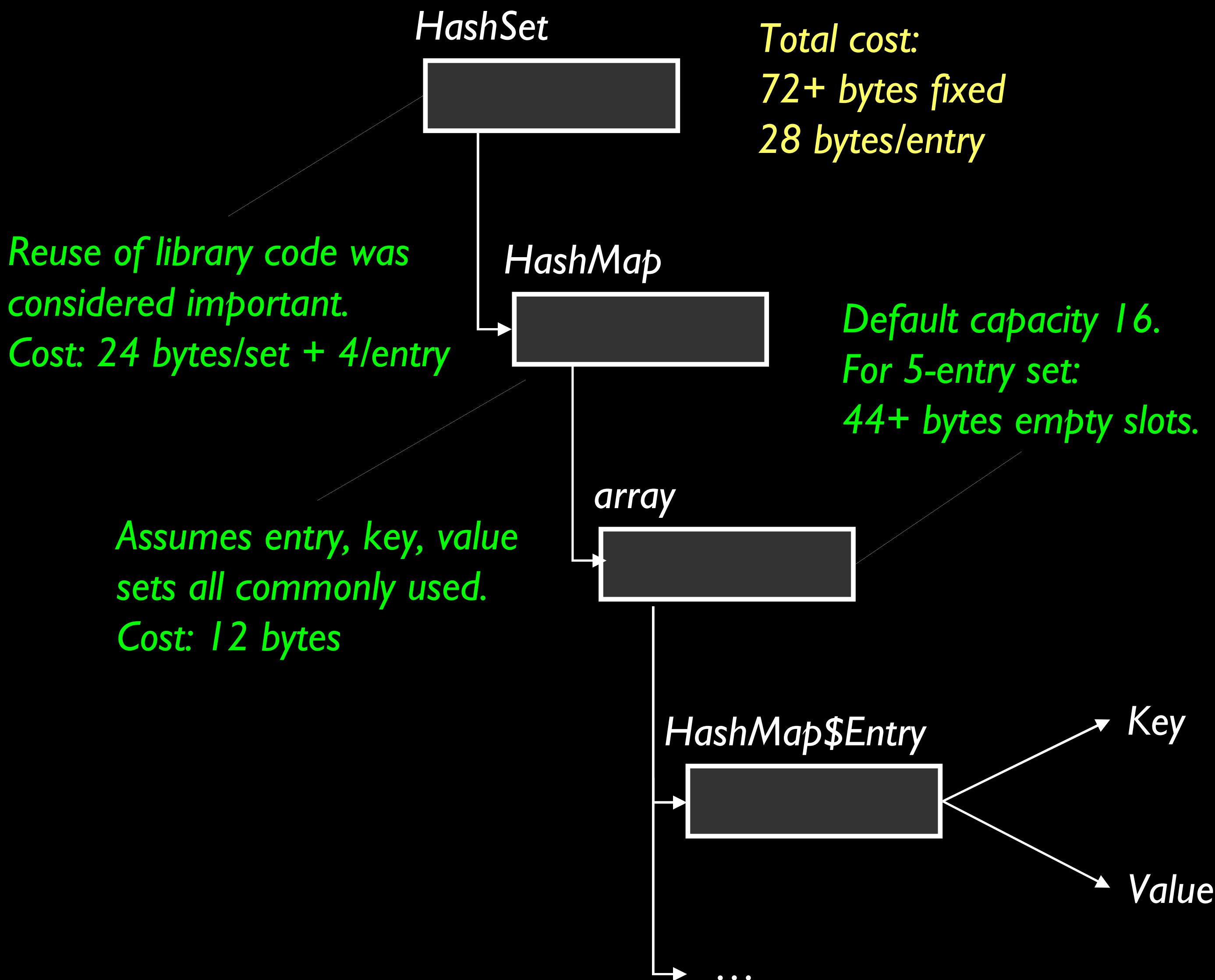
Map with multiple values per entry



- Only 5% of sets had more than a few elements each

Inside the Java collections

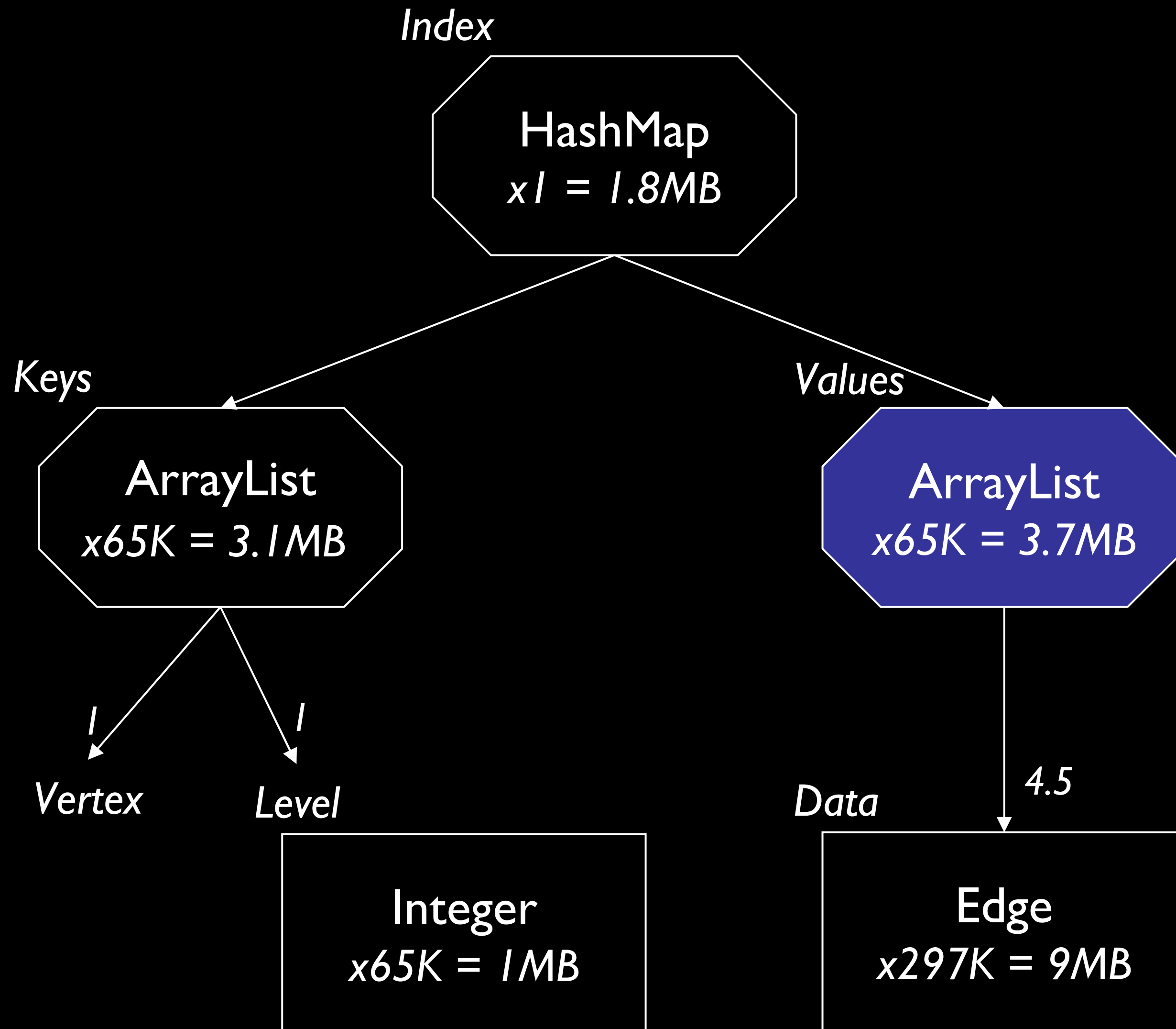
HashSet: many embedded usage assumptions



- Not a good choice for small collections
- Users, look before you leap – always measure
- Framework designers, beware making usage assumptions

Small collections in context

Map with multiple values per entry



Remedy

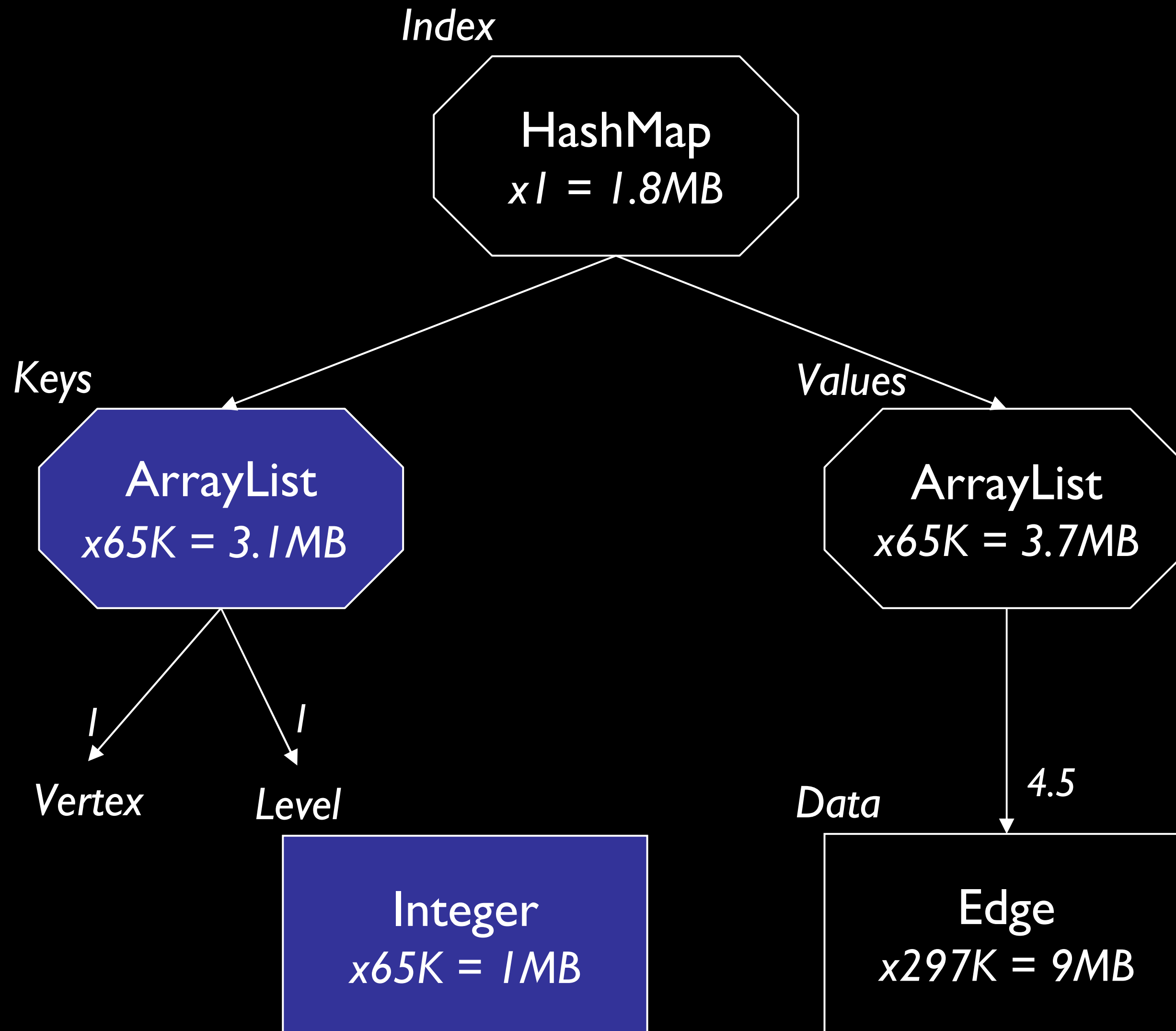
- Switched to ArrayList. Saved 77% of that region.
- HashSet functionality was not worth the cost. Uniqueness already guaranteed elsewhere

Wish list

- Gracefully-growing collections

Small collections in context

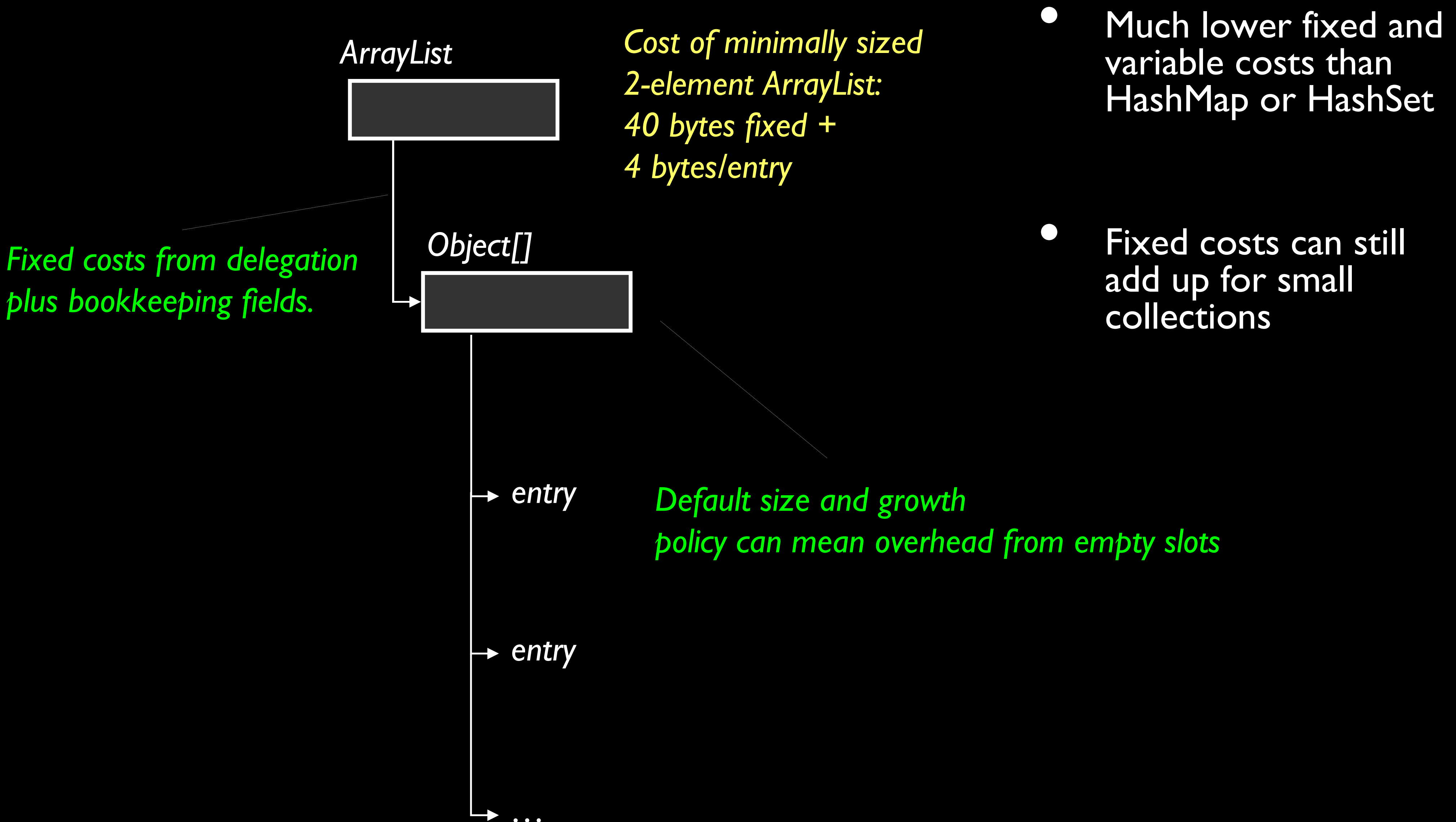
Multipart key as 2-element ArrayList



- ArrayList has a high fixed cost. Also required boxing of integers.

Inside the Java collections

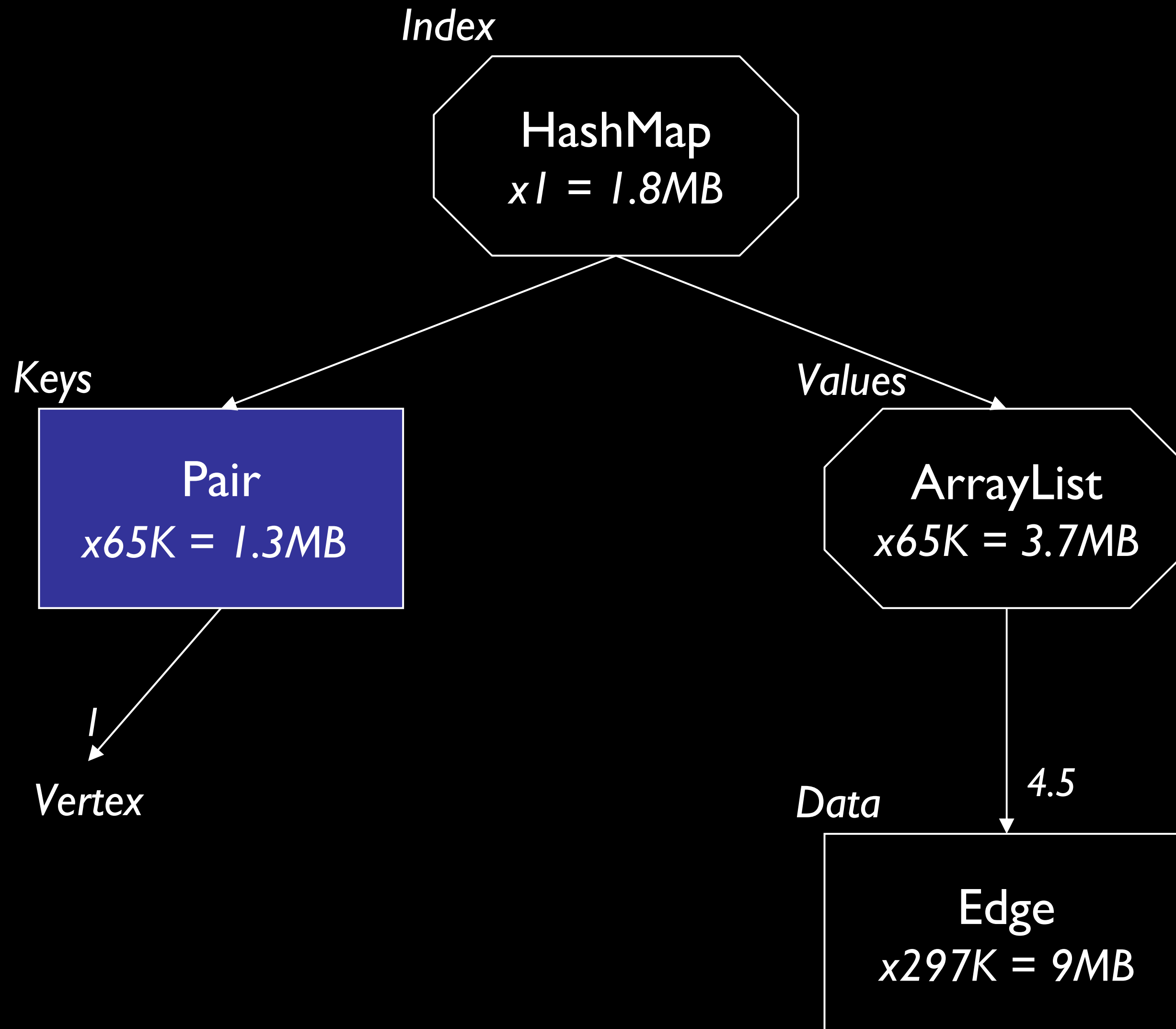
ArrayList



- Much lower fixed and variable costs than HashMap or HashSet
- Fixed costs can still add up for small collections

Small collections in context

Multipart key class

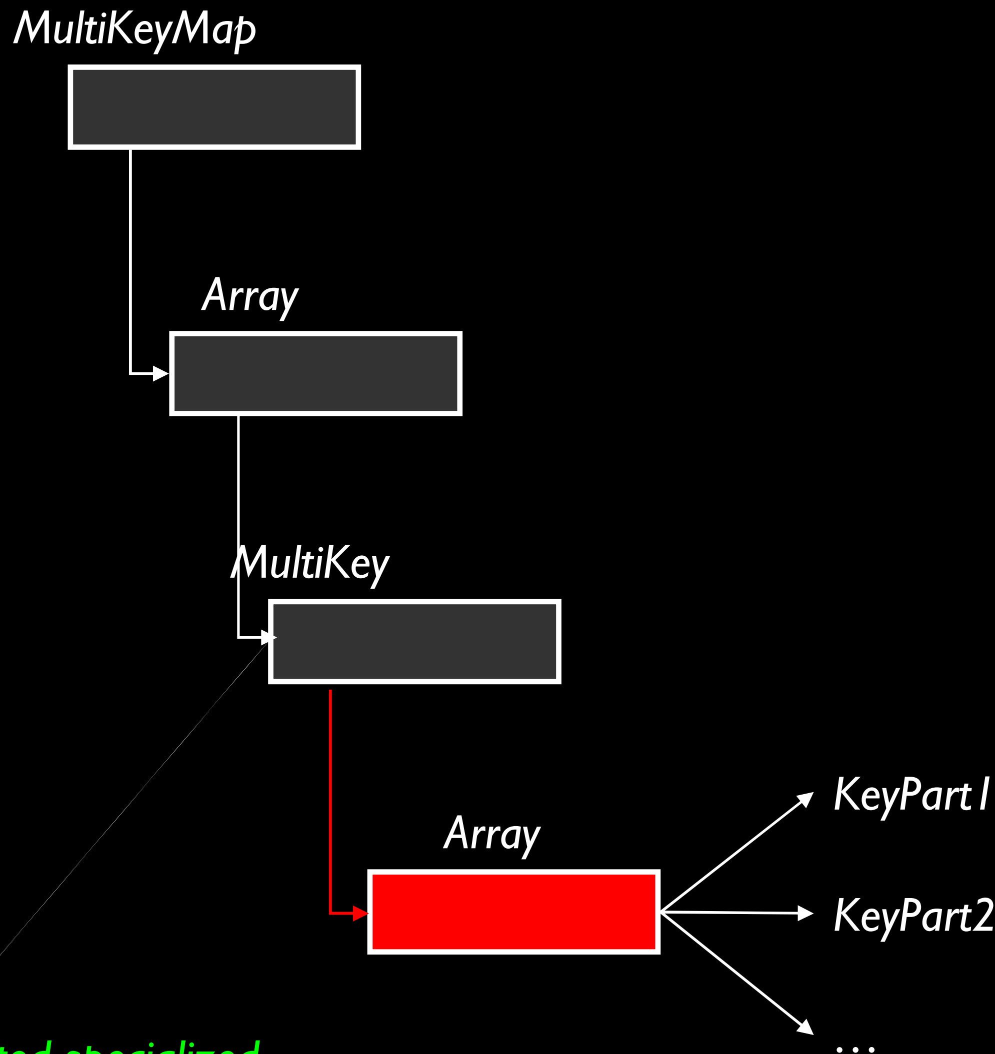


Remedy:

- Introduced Pair class (Vertex, int level)
- Again, functionality of original design was not worth the cost
- Reduced key overhead by 68%

Multipart key

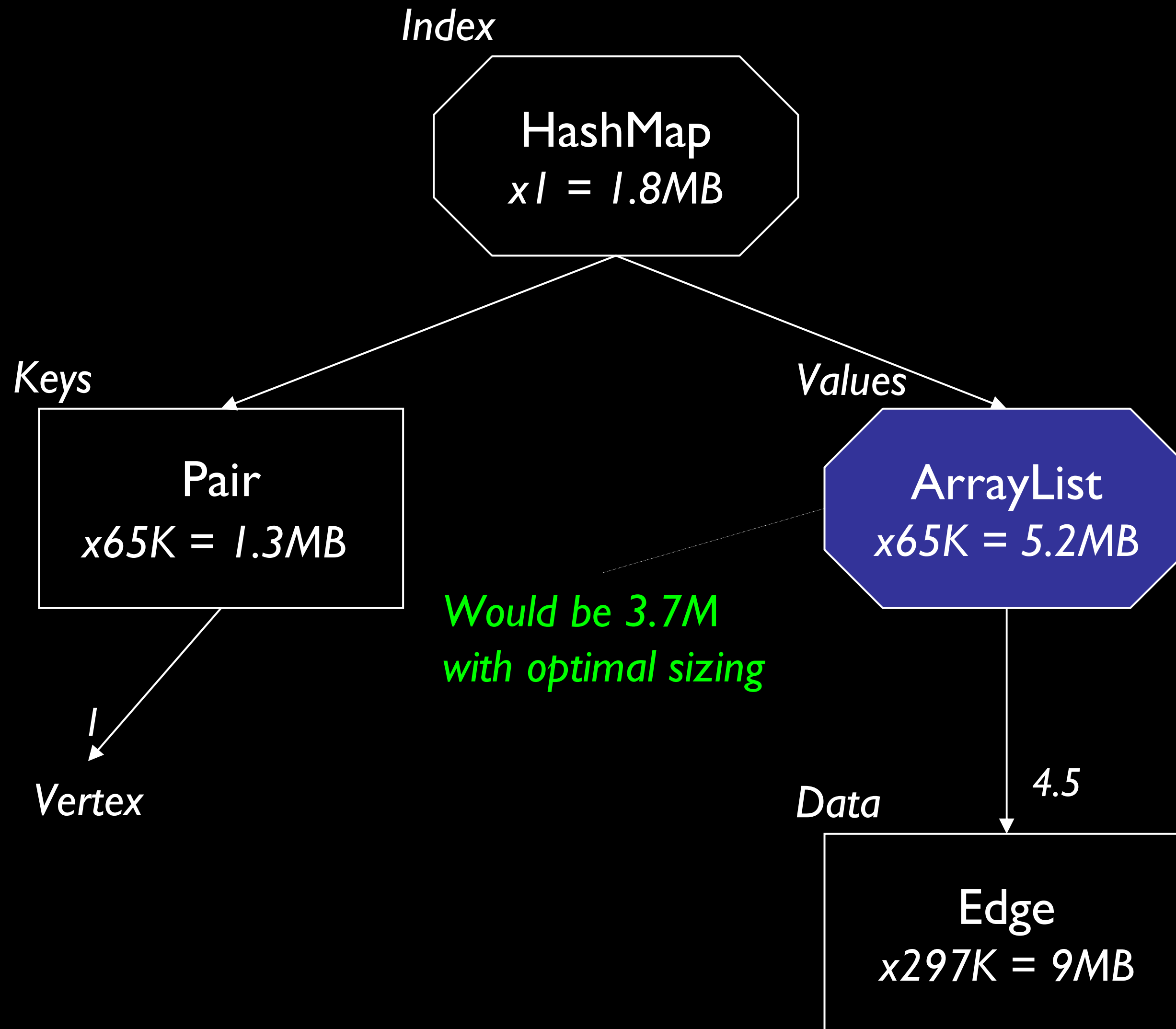
Case study: Apache Commons MultiKeyMap



- Apache Commons collections frameworks has the same pattern
- Paying for flexibility that's not needed
- Cost: additional 20 bytes per entry

Growth policies

Example: creating default-size ArrayLists



- 28% overhead in ArrayLists just from empty slots
- collections optimized for growth
- large defaults and jumps – doubling
- 10% tax on some copies

Remedies:

- Set initial capacity
- trimToSize() after load

Inside the Java Collections

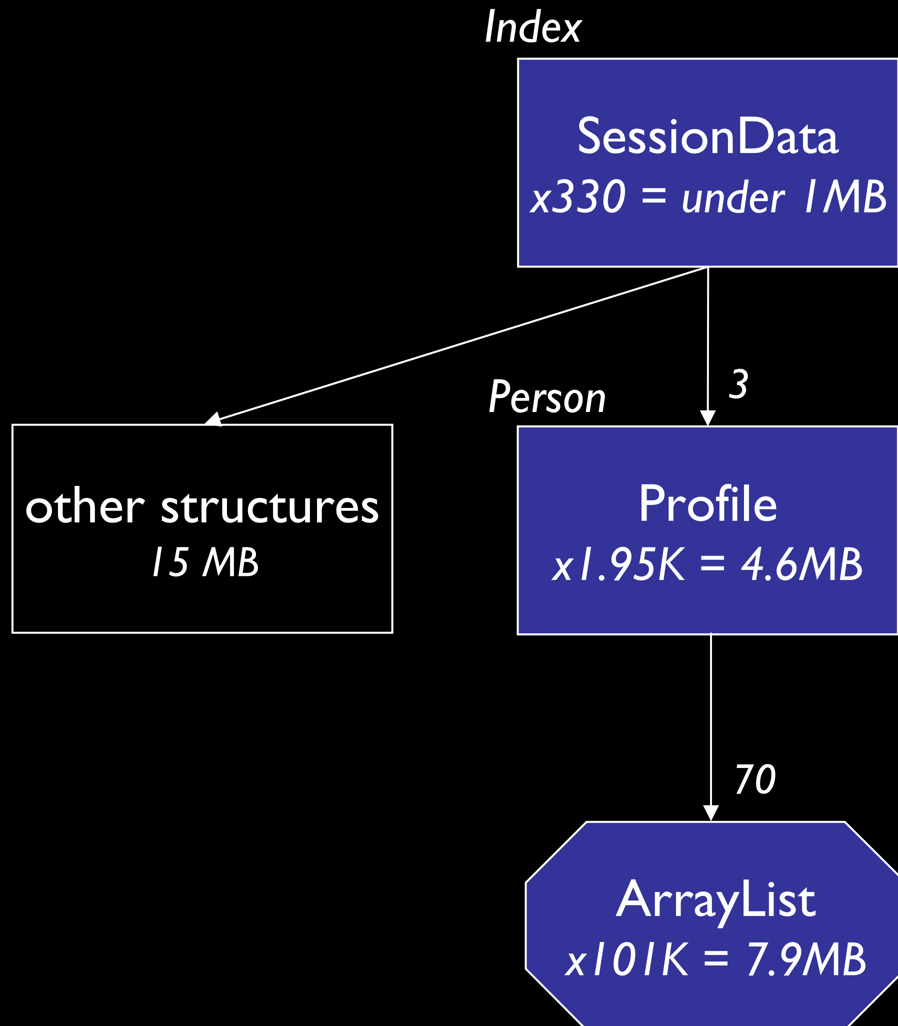
Cost of a 2-element collection

	Minimal size (bytes)	Default size (bytes)	# of slots for 2 elements using default size
LinkedList	96	96	3
ArrayList	48 or 56	80 or 88	10
HashMap	116 or 168	168	16
HashSet	132 or 184	184	16

From experiments with a few different JVMs, all 32-bit.

The cost of empty collections

Case study: CRM system, part of session data

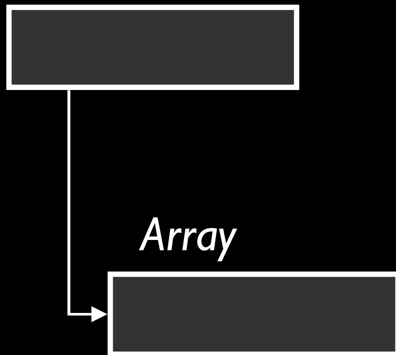


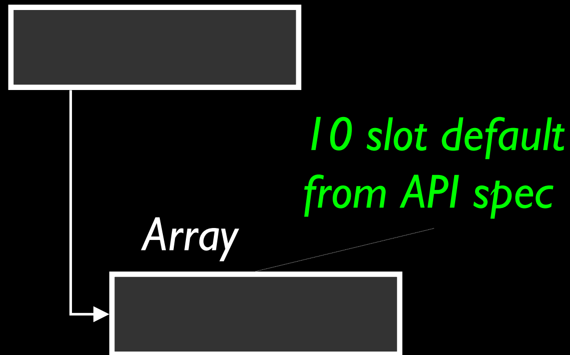
- Small run had 26M of session data. Will not scale.
- 210 empty collections per session = 28% of session cost

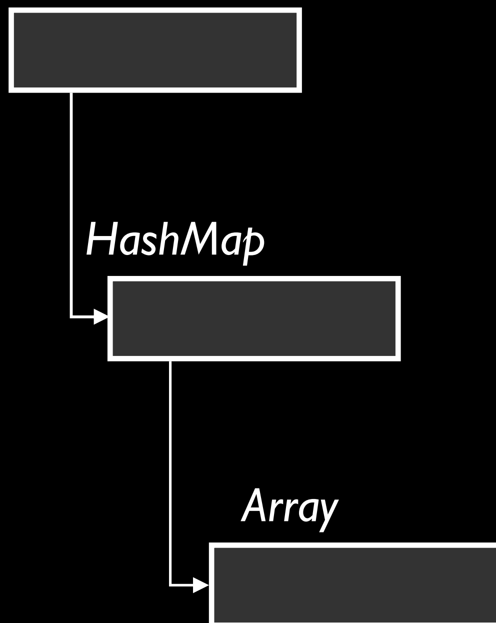
Remedies:

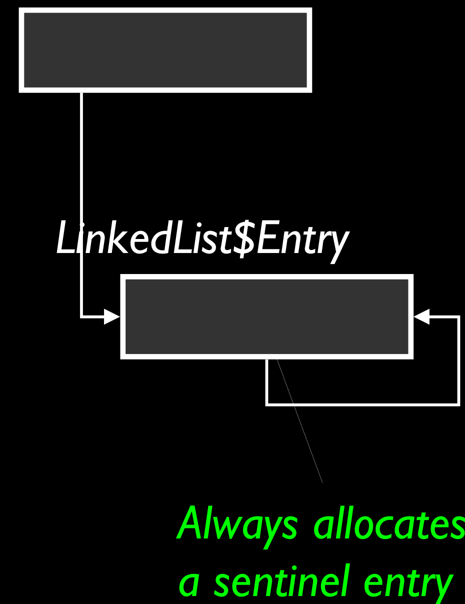
- Lazily allocate
- Collections.emptySet()
- Avoid giving out references

The Not-so-empty Collections









- Minimum of 2 objects each – component parts are always allocated

- Default sizing increases cost (e.g. 16 elements for HashMap/HashSet)

Inside the Java Collections

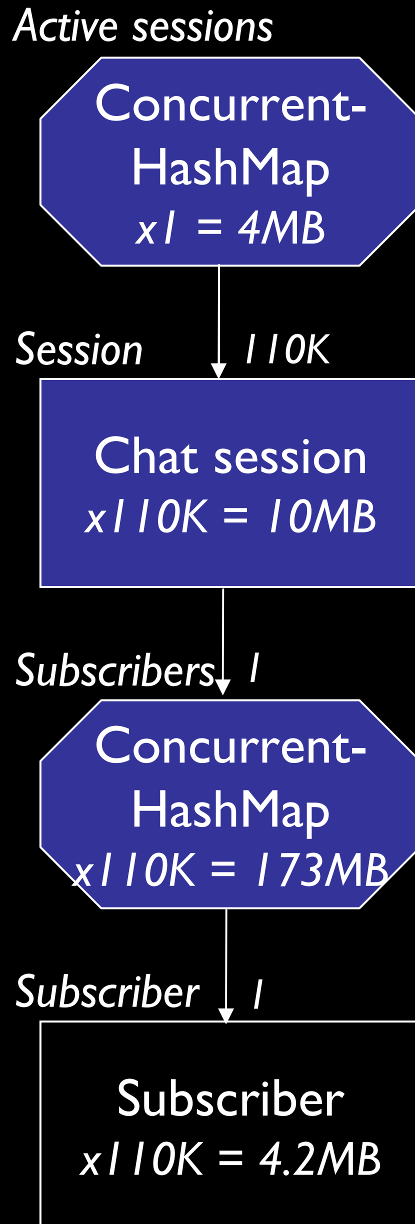
Cost of an empty collection

	Minimal size (bytes)	Default size (bytes)	Default # of slots
LinkedList	48	48	1 sentinel entry
ArrayList	40 or 48	80 or 88	10
HashMap	56 or 120	120	16
HashSet	72 or 136	136	16

From experiments with a few different JVMs, all 32-bit.

Small concurrent maps

Case study: Chat server framework



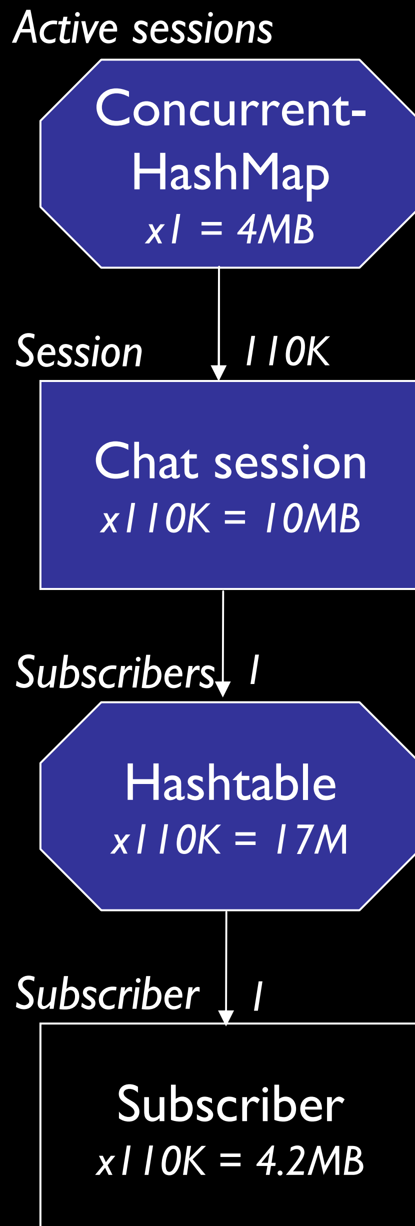
- Nested CHMs: > 1600 bytes each!
- Cost was 90% of this structure; 10-20% of total heap

What went wrong:

- Library not intended for use at this scale
- Concurrency requirements were different at fine vs. coarse grain

Small concurrent maps

Case study: Chat server framework



Remedies:

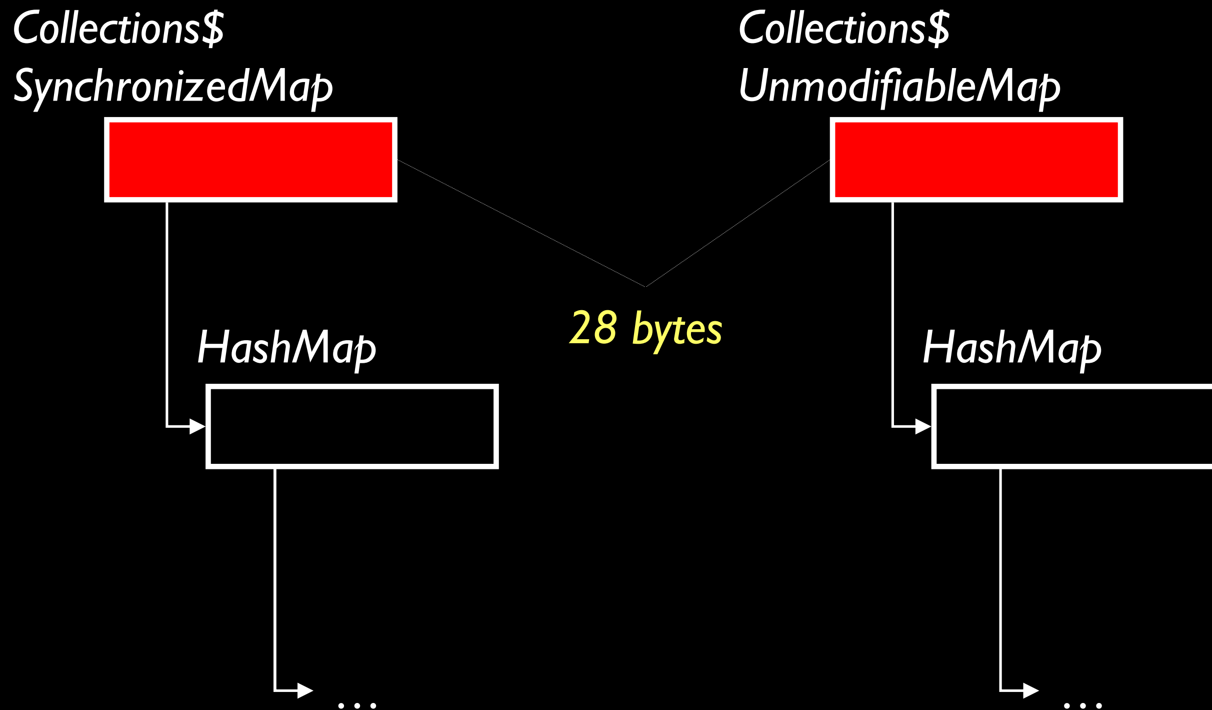
- First considered reducing width of inner ConcurrentHashMap from 16 to 3. Savings: 67%
- Used Hashtable, since high level of concurrency not needed. Savings: 90^{+}%

Note:

- Hashtable less expensive than similar Collections$ SynchronizedMap

Inside the Java Collections

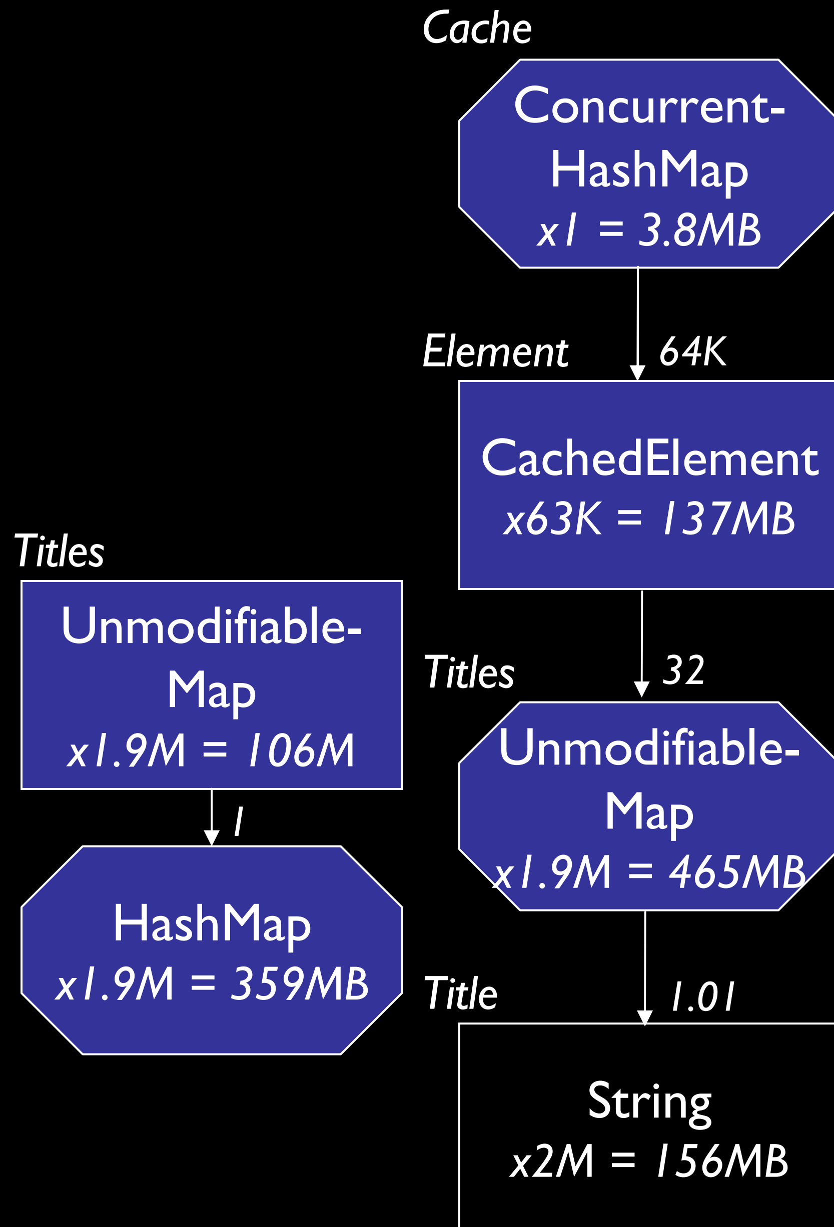
Wrapped collections



- Design is based on delegation
- Costs are significant when collections are small
- Fine for larger collections

Small wrapped collections

Case study: media e-commerce site (64-bit)



- 108MB for UnmodifiableMap wrappers. 56 bytes each
- Twice the cost as on a 32-bit JVM

What went wrong:

- Functionality not worth the cost at this scale. *Unmodifiable* serves a development-time purpose

Inside the Java Collections

Standard collections: per-entry costs.

	Per-entry cost (bytes)
LinkedList	24
ArrayList	4
HashMap	28 or 36
HashSet	28 or 36

- Plus any overhead of introducing a key or value object

From experiments with a few different JVMs, all 32-bit.

Excludes amortized per-collection costs such as empty array slots. Includes pointer to entry.

The standard collections

JDK Standard Collections

- Speed has been the focus, not footprint

IBM (Harmony) and Sun implementations not that different in footprint

Hard-wired assumptions, few policy knobs (e.g. growth policies)

Specialized collections are worth learning about:

- IdentityHashMap, WeakHashMap, ConcurrentHashMap, etc.

Collections alternatives

Apache Commons

- Many useful collections:
  - Flat3Map, MultiMap, MultiKeyMap
- Focus is mostly on functionality. Maps allow some extension.
- Footprint similar to standard, with a few exceptions

GNU Trove

- Many space-efficient implementations
- e.g. scalar collections
- e.g. list entries without delegation cost

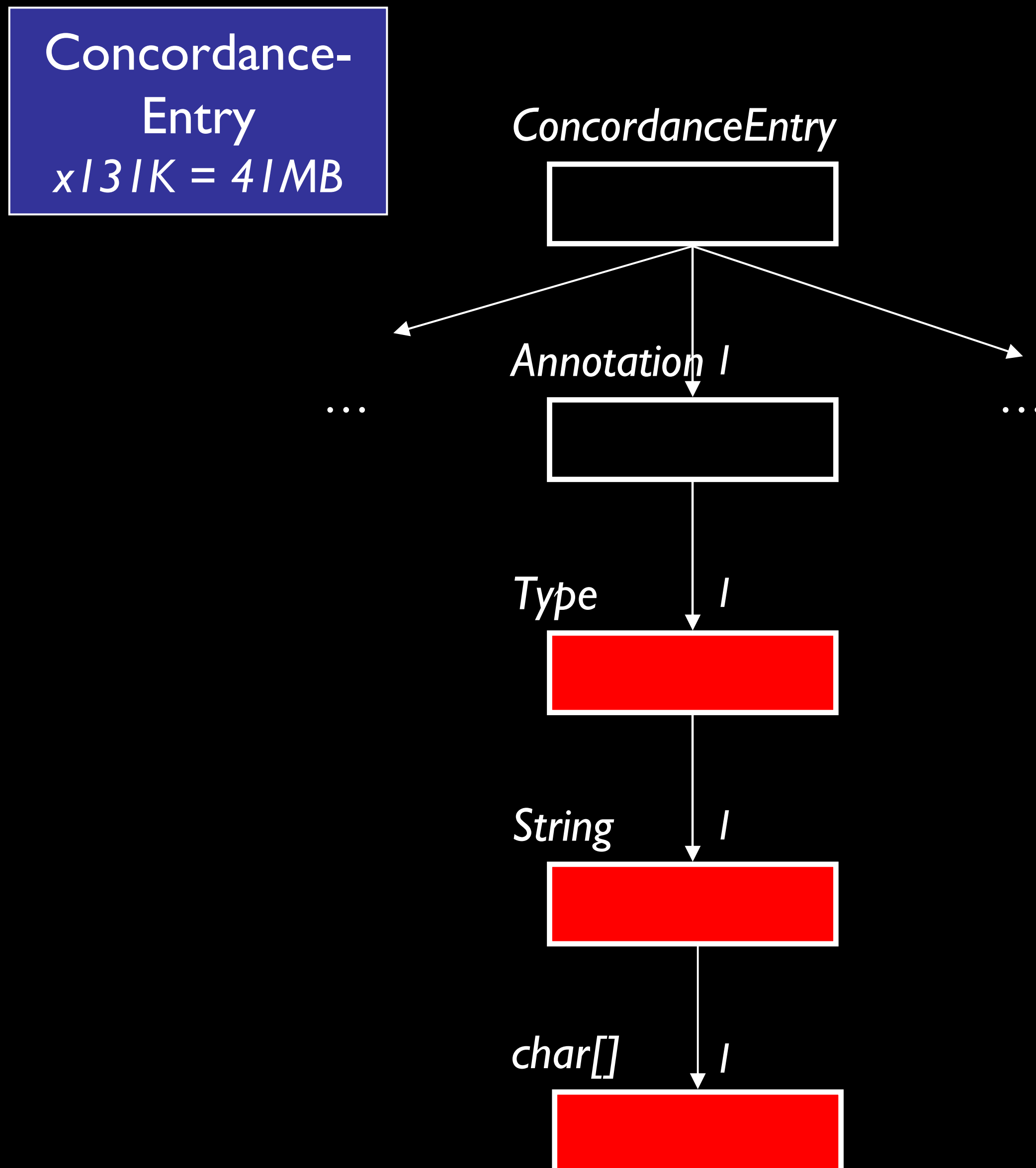
Cliff Click nonblocking; Javolution; Amino

Specialized collections within frameworks you use

Important: check your corporate policy re: specific open source frameworks

Duplicate, immutable data

Case study: Text analysis system, concordance



- 17% of cost due to duplication of Type and its String data
- Only a small number of immutable Types

What went wrong?

- Interface design did not provide for sharing
- Full cost of duplication was hidden

Remedy

- Use shared immutable factory pattern

Background: sharing low-level data

String.intern()

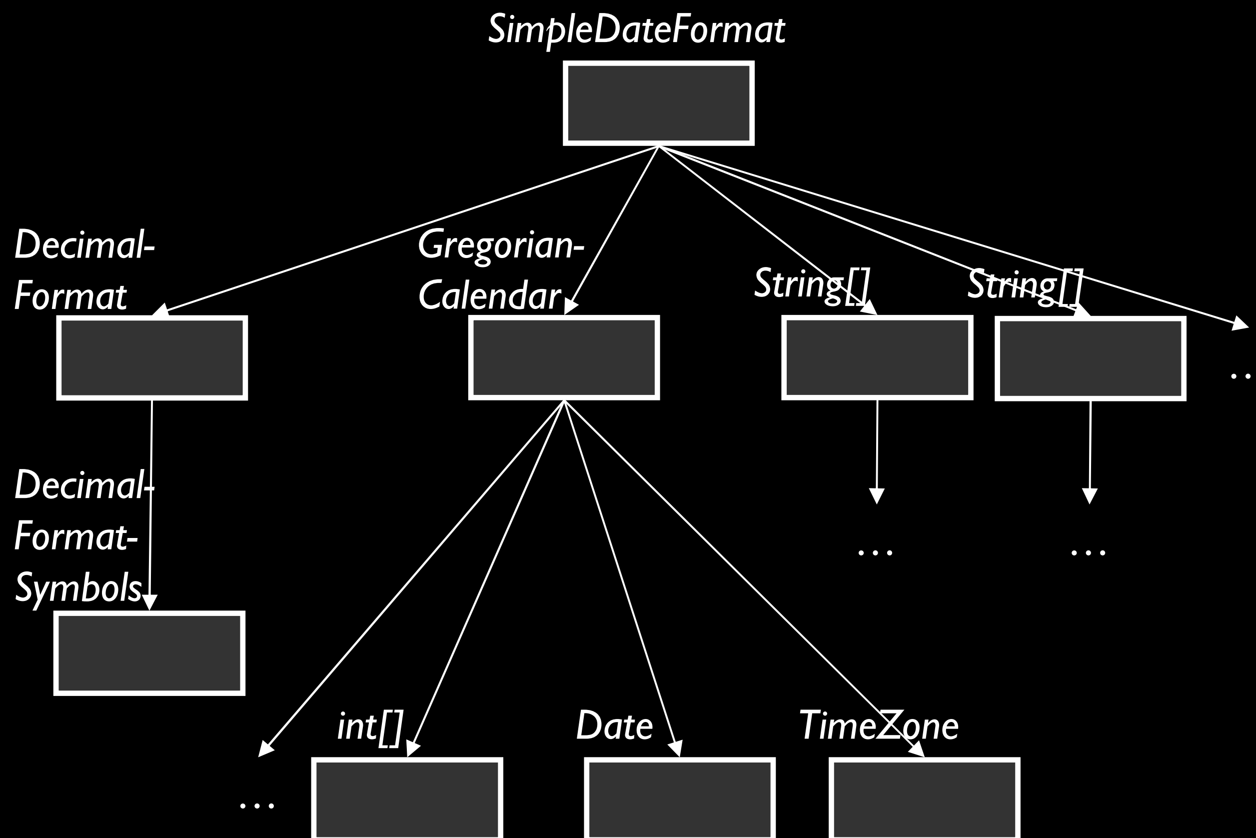
- You specify which Strings to share
- Shares the String object and the character array
- Make sure it’s worth it, since there is a space cost
- Myth that is causes memory leaks
  - Though can hit permspace limits

Boxed scalars

- Integer.valueOf(), Boolean.valueOf(), etc.
- Shares some common values (not all)
- Make sure you don’t rely on ==

Expensive temporaries

Example: SimpleDateFormat



- Costly construction process. Each call to the default constructor results in:
  - 123 calls to 55 distinct methods
  - 44 new instances
- Designed for costs to be amortized over many uses
- Remedy: reuse via a local variable or thread-local storage