Software Qualities: Memory Efficiency

Marco Faella

Why space efficiency?

Because you might have **little space** or **a lot of stuff**

Little space:

- Embedded systems
- Modern cars have 50-100 microcontrollers on board
- Most microcontrollers come with less than 1MB RAM

A lot of stuff:

- Recent AAA videogames take over 50GB
- Big data applications
- Billions of water containers!

Object Layout

Memory overheads

Java objects include an *object header* with auxiliary information

The header supports the following functionalities:

1. Type awareness and reflection
2. Multi-threading
3. Garbage collection

The size of a reference

In theory:

References (aka pointers) take 32 or 64 bits depending on the architecture

In practice:

Enters *Compressed ordinary object pointers (COOPs):*

- Most programs don't address more than 32GB
- Encode references using 32 bits
- When using an address, add 3 zeros at the end
- You get 35 bit addresses: 32GB of addressable memory

ON by default in *HotSpot*

It can be turned off with cmdline arguments to the JVM

Type awareness overhead

Each object must know its *dynamic type*

This is used by `instanceof`, `getClass` and other reflective operations

The header contains **a pointer to a Class object**

Multi-threading overhead

In theory:

Java assigns a **monitor** to each object

- Similar to a mutex
- Accessed via the `synchronized` keyword

In practice:

Current versions of HotSpot instantiate the monitor *on demand*

This reduces (but does not avoid) the memory overhead

Garbage collection overhead

In theory:

A reference count for each object

In practice:

- HotSpot employs *tracing* (mark-and-sweep) and *generational* GCs
- Tracing:
  - No reference count
  - Instead, follow all references starting from GC *roots*

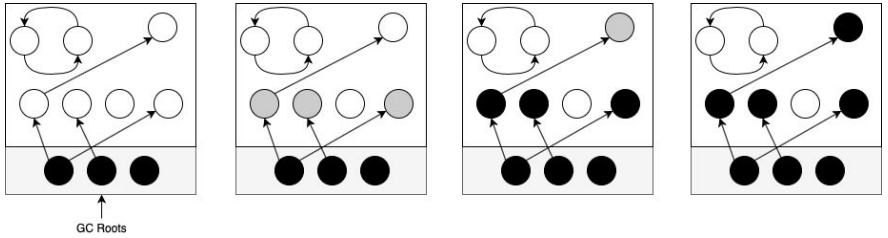


Image by
Ali Dehghani

- Generational:
  - Objects are arranged in groups called generations, based on the time since creation
  - Objects carry an *age* field

Other overheads: alignment and padding

In most architectures, memory accesses are more efficient if they are ***word-aligned***

In a 64-bit arch, an address is word-aligned if it is a **multiple of 8**

- lower 3 bits equal to zero
- just like COOPs!

Effective object sizes are multiples of 8 bytes

If necessary, empty space is inserted into objects (**padding**)

Summary of overheads

64bit arch with COOPs:

• Pointer to Class object	4 bytes
• Monitor + GC	8 bytes (complex dynamic encoding)
Total:	**12 bytes**

For simplicity, ignore alignment and padding

For details, see HotSpot source

- File src/share/vm/oops/markOop.hpp
- at https://hg.openjdk.java.net/jdk10/jdk10/hotspot

Array overheads

Arrays store:

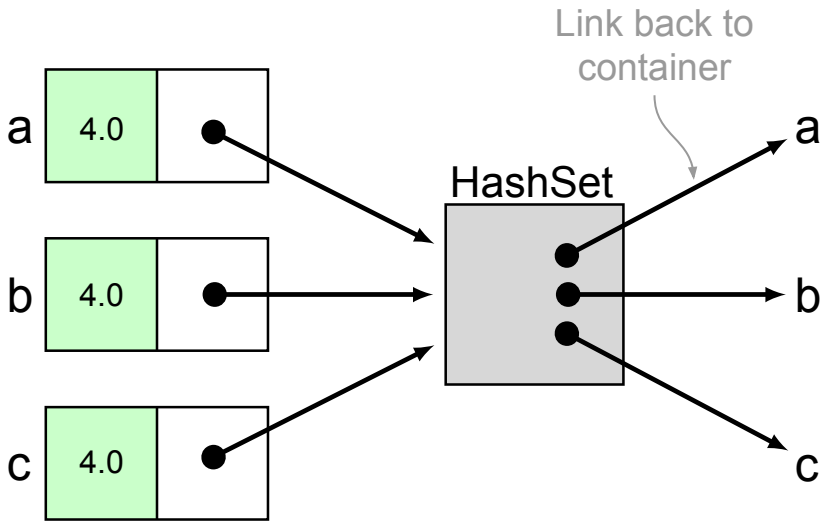
- Their size
  - 4 additional bytes
- The static type of their cells
  - No additional overhead
  - This information is part of their type
  - A String[] contains a reference to the Class object for String[]

Total: **16 byte** overhead

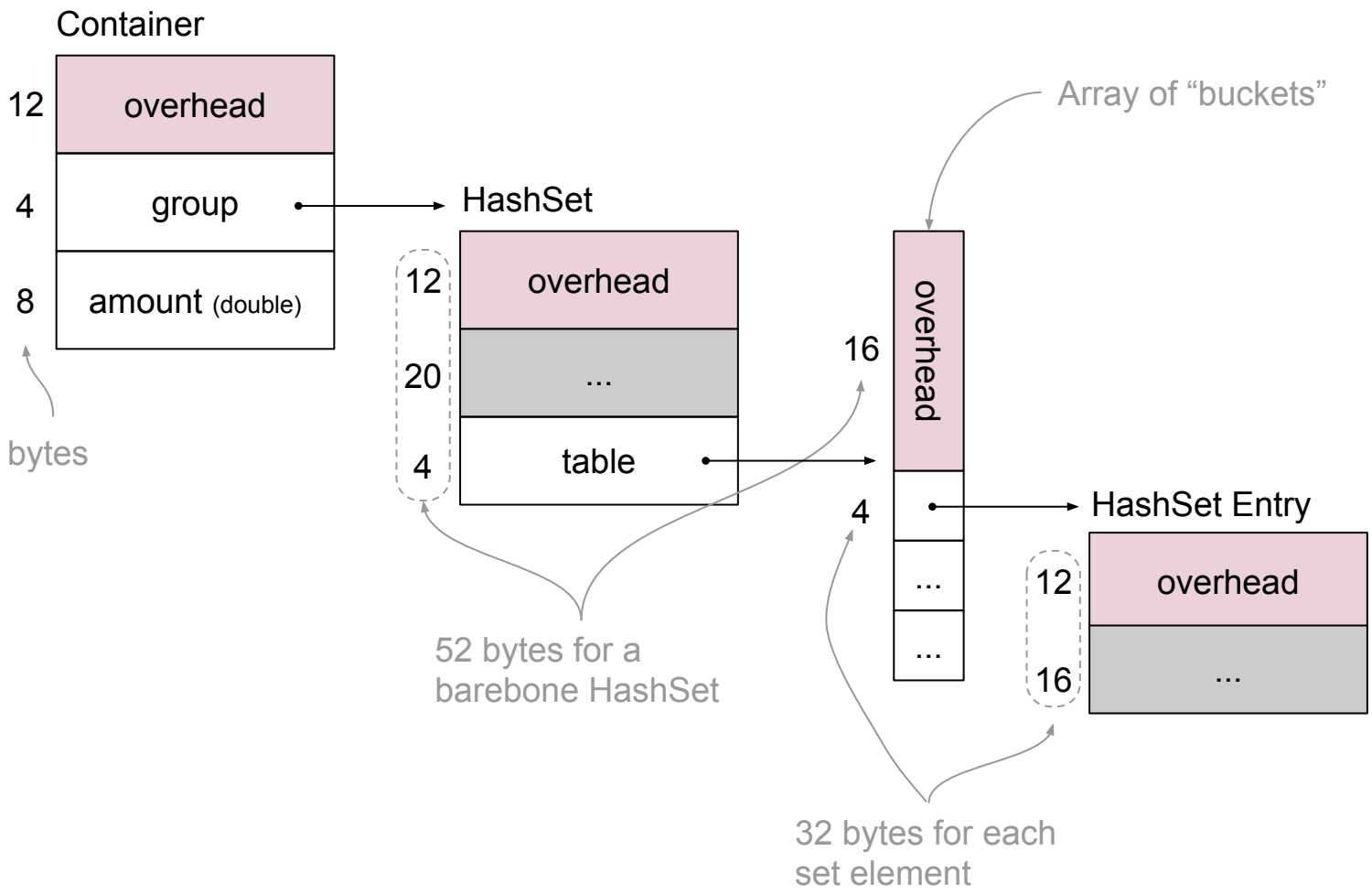
Reference implementation

The **fields**:

`double amount`	Amount of water in this container
`Set<Container> group`	Containers connected *directly or indirectly* to this one, including this one



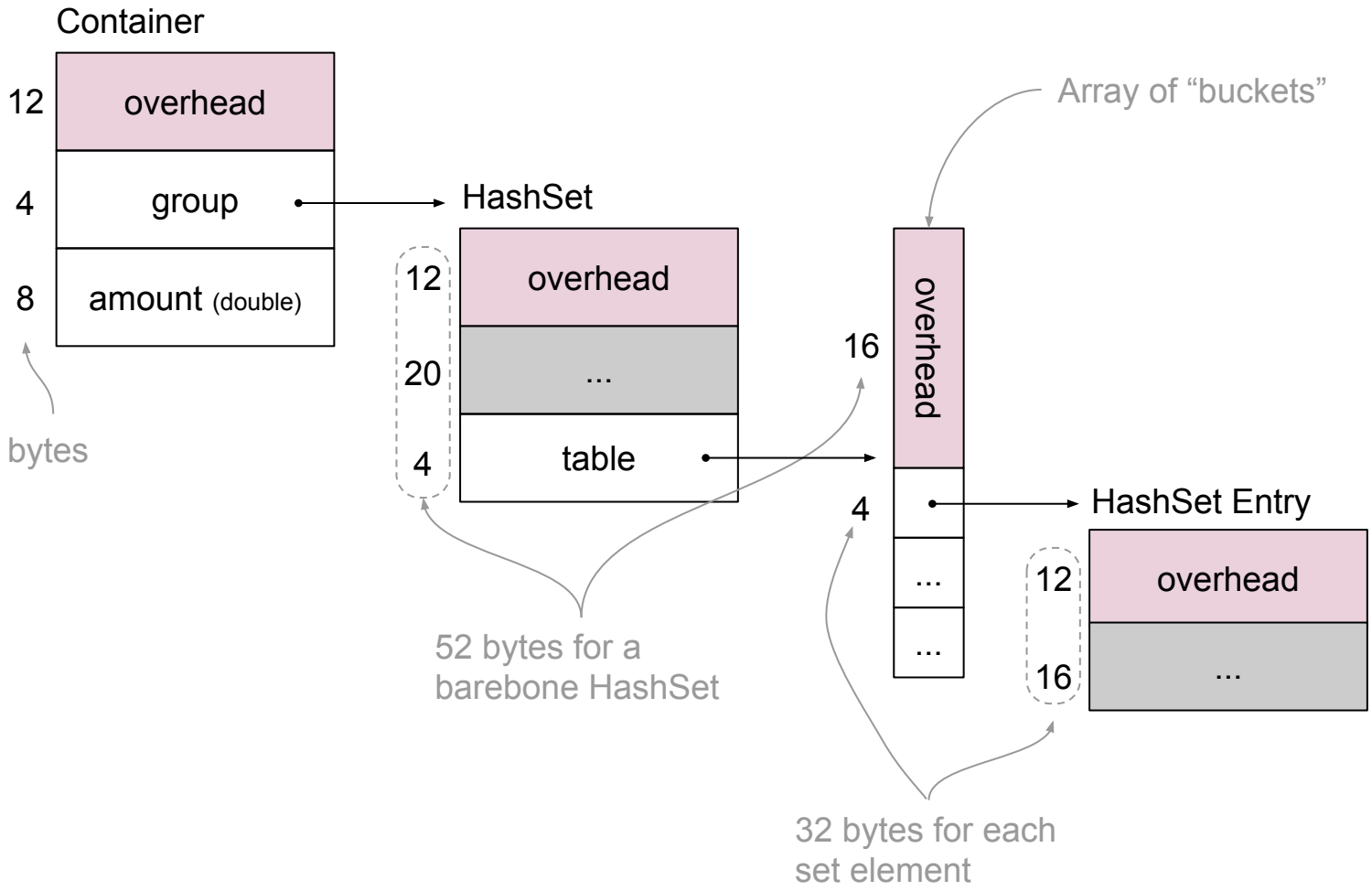
Detailed memory layout of Reference



Con

Table 2.1 Memory requirements of *Reference* in two conventional scenarios

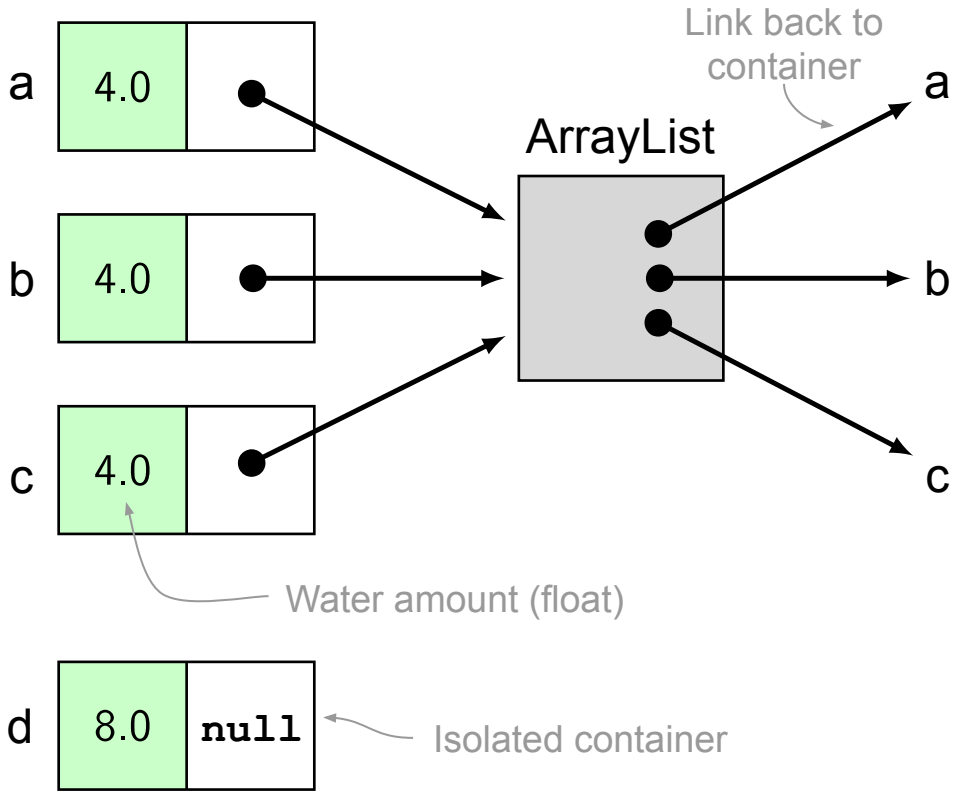
Scenario	Size (calculations)	Size (bytes)
1000 isolated	$1000 * (12 + 8 + 4 + 52 + 32)$	108000
100 groups of 10	$1000 * (12 + 8 + 4) + 100 * (52 + 10 * 32)$	61200



Memory Efficiency

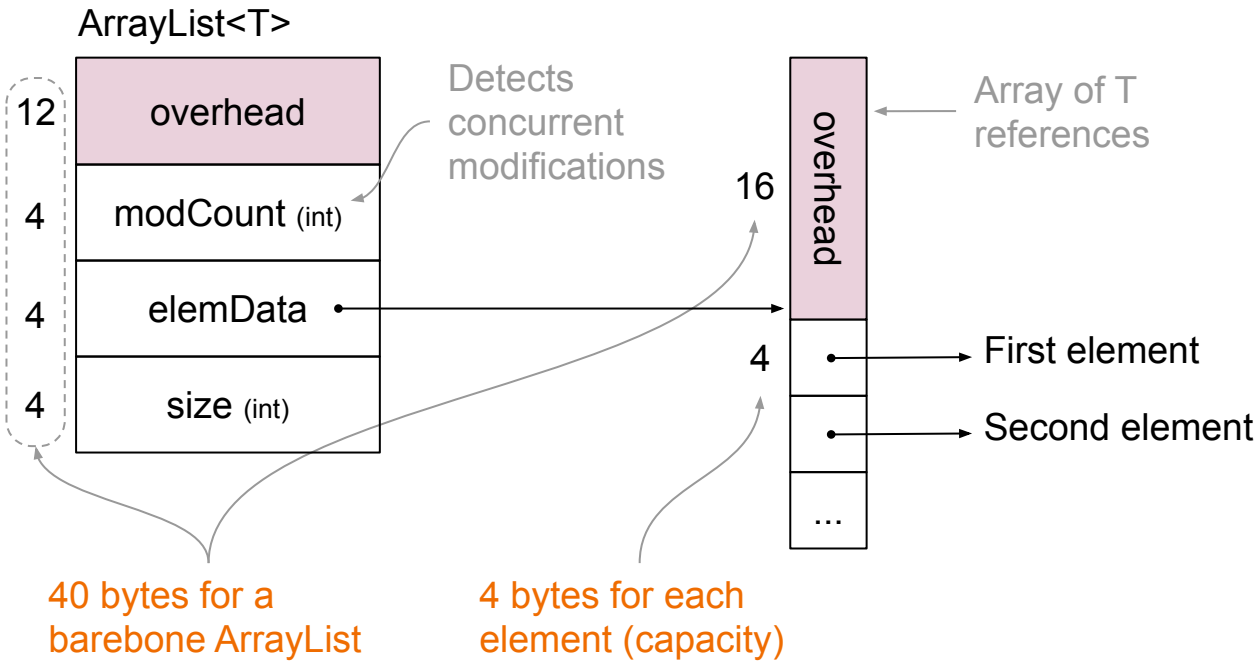
Gently squeezing [Memory1]

1. Amount field: Switch from double to float
2. Group field: Switch from HashSet to a *lazily instantiated* ArrayList



Note: methods keep the same complexity as Reference

Memory requirements of ArrayList



Memory requirements of Memory1

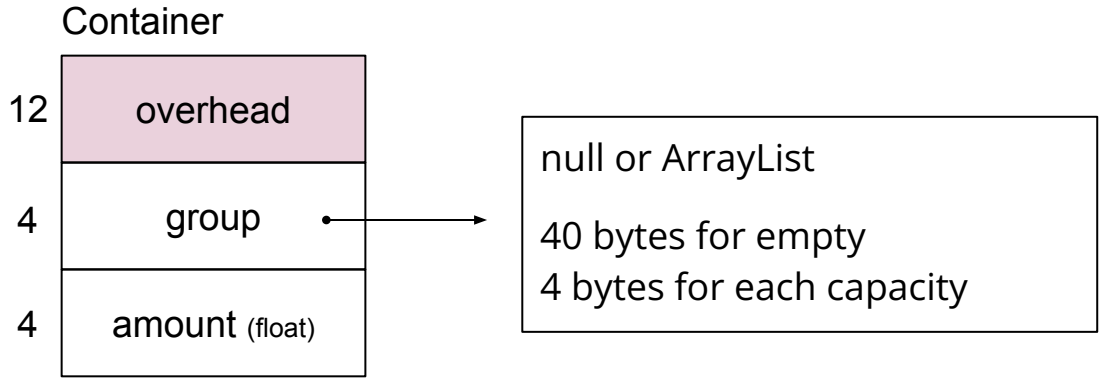


Table 4.1 Memory requirements of *Memory1*

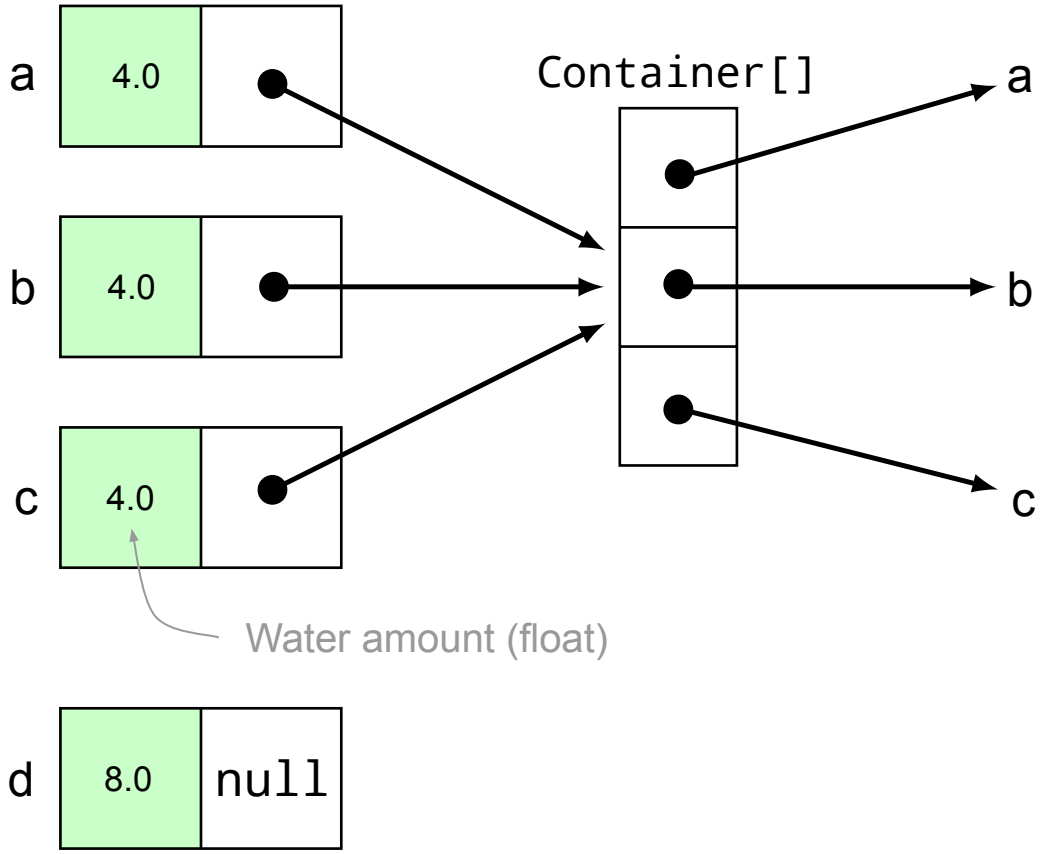
Scenario	Size (calculations)	Size (bytes)	% of reference
1000 isolated	$1000 * (12 + 4 + 4)$	20000	19%
100 groups of 10	$1000 * (12 + 4 + 4) + 100 * (40 + 10 * 1.25 * 4)$	29000	47%

Expected excess capacity

Plain arrays

[Memory2]

Represent group as an **array of Containers**



Memory requirements of Memory2

Table 4.4 Memory requirements of common collections, assuming that the capacity of the `ArrayList` and the `HashSet` is equal to their size. The second column is tagged "barebone" instead of "empty" because it doesn't take into account the default initial capacity of that collection.

Type	Size (barebone)	Size (each extra element)
array	16	4
`ArrayList`	40	4
`LinkedList`	24	24
`HashSet`	52	32

Table 4.3 Memory requirements of *Memory2*

Scenario	Size (calculations)	Size (bytes)	% of reference
1000 isolated	$1000 * (12 + 4 + 4)$	20000	19%
100 groups of 10	$1000 * (12 + 4 + 4) + 100 * (16 + 10 * 4)$	25600	42%

A modest improvement over Memory1!

An object-less API

Containers identified by **integers** instead of objects

```
int a = Container.newContainer(),
    b = Container.newContainer(),
    c = Container.newContainer(),
    d = Container.newContainer();

Container.addWater(a, 12);
Container.addWater(d, 8);
Container.connect(a, b);
System.out.println(Container.getAmount(a));
```

Replaces the constructor

Constructor and methods become **4 static methods**

Not object-oriented!

Two IDs and two arrays [Memory3]

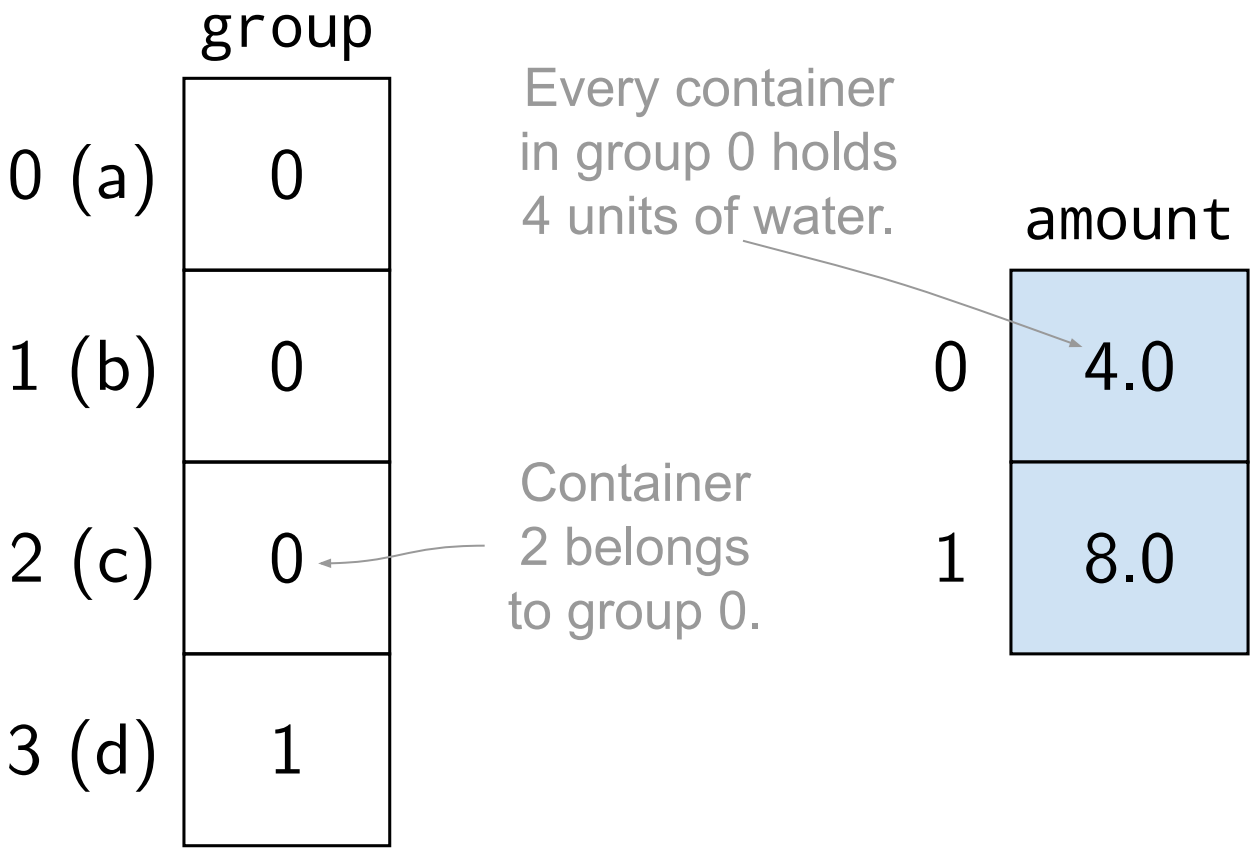
- Each container is identified by an ID (**containerID**)
- Each group of containers is identified by an ID (**groupID**)

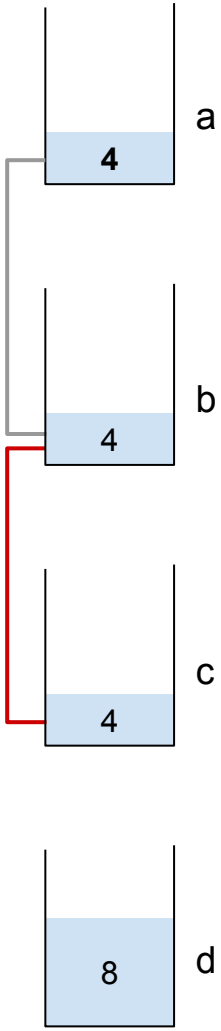
Class sketch:

```
public class Container {
   // From containerID to groupID
   private static int group[] = new int[0];
   // From groupID to the amount in each container in the group
   private static float amount[] = new float[0];

   public static float getAmount(int containerID) {
       int groupID = group[containerID];
       return amount[groupID];
   }
...
```

Memory layout of Memory3





Method implementation

Method **newContainer**:

- Adds a new containerID and a new groupID
- Enlarge both arrays
- **Linear** time (*)

Method **addWater**:

- Counts the size of the group (scans the group array)
- Updates the amount array
- **Linear** time (*)

Method **connect**:

- Merges two groups
- Removes one groupID
- To discover all containers in a group, you have to iterate over **all containers**
- **Linear** time (*)

All methods except getAmount are linear (*)

(*) in the total number of containers

Memory requirements of Memory3

Table 4.6 Memory requirements of *Memory3*

Scenario	Size (calculations)	Size (bytes)	% of reference
1000 isolated	$4 + 16 + 1000 * 4 + 4 + 16 + 1000 * 4$	8040	7%
100 groups of 10	$4 + 16 + 1000 * 4 + 4 + 16 + 100 * 4$	4440	7%

The black hole [Memory4]

A **single array** for both *groups* and *amounts*

```
public class Container {
   private static float nextOrAmount[] = new float[0];
```

When positive:

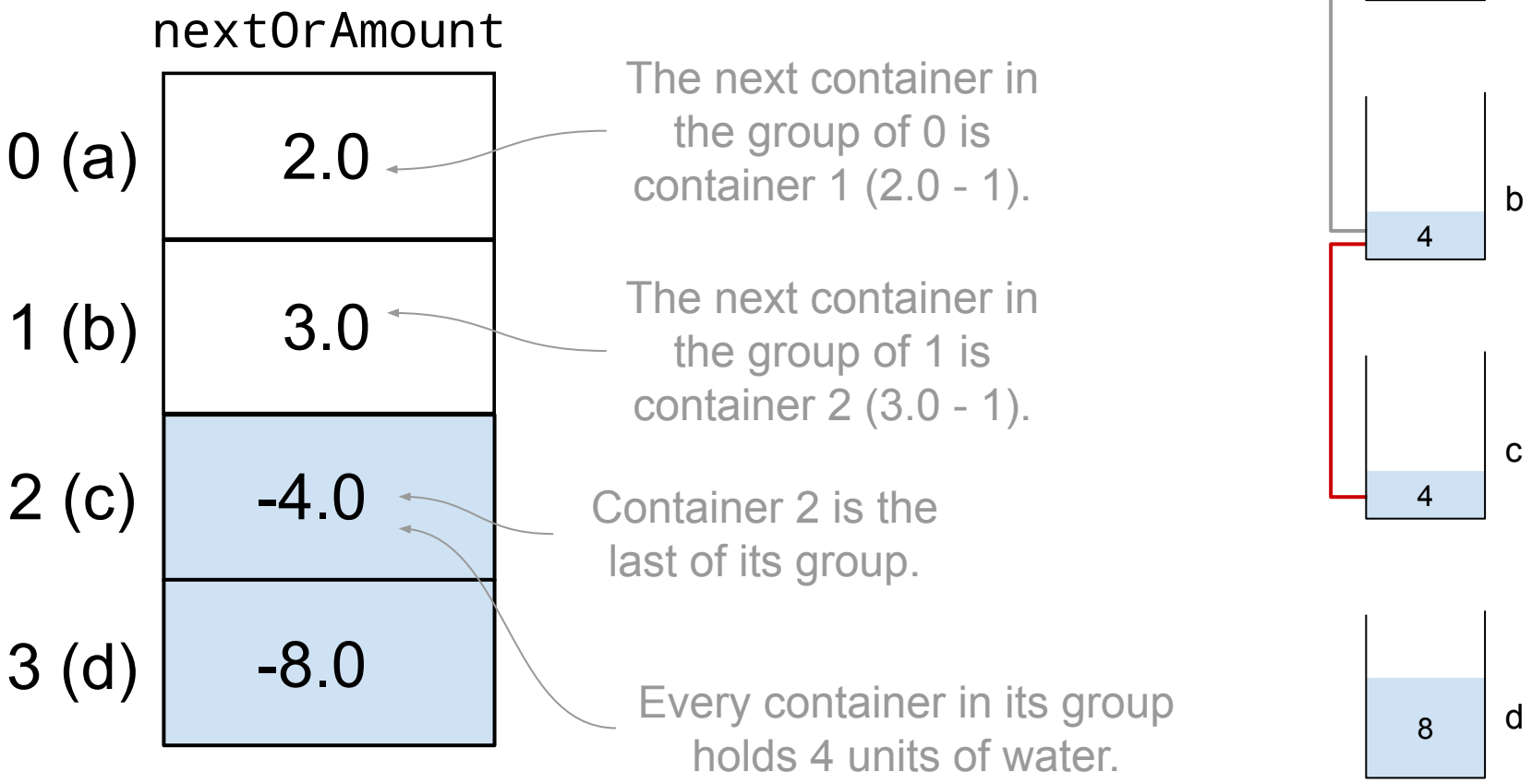
the index of the **next** container in this group, with a +1 bias

When negative or zero:

the *opposite* of the water **amount** in each container of this group

```
   public static float getAmount(int containerID) {
       while (nextOrAmount[containerID]>0)
           containerID = (int)nextOrAmount[containerID] -1;
       return -nextOrAmount[containerID];
   }
```

Memory layout of Memory3



Weakness of Memory3

- It works as long as the index of the next container can be represented by a float
- How many containers are supported?
- Recall that a float is 32 bits (24 bits significand + 8 bit exponent)
- The *uninterrupted integer range* of the type float is 0 to 2^24

```
> jshell
> float f = 1 << 24;
1.6777216E7
> f+1 == f;
true
> f-1 == f;
false
```

Method implementation

Method **addWater**:

- Find the last container in the group
- Count the size of the group
- **Quadratic** time (in the total number of containers)

Method **connect**:

- Find all containers in both groups
- **Quadratic** time (in the total number of containers)

Table 4.9 Time complexities of *Memory4*

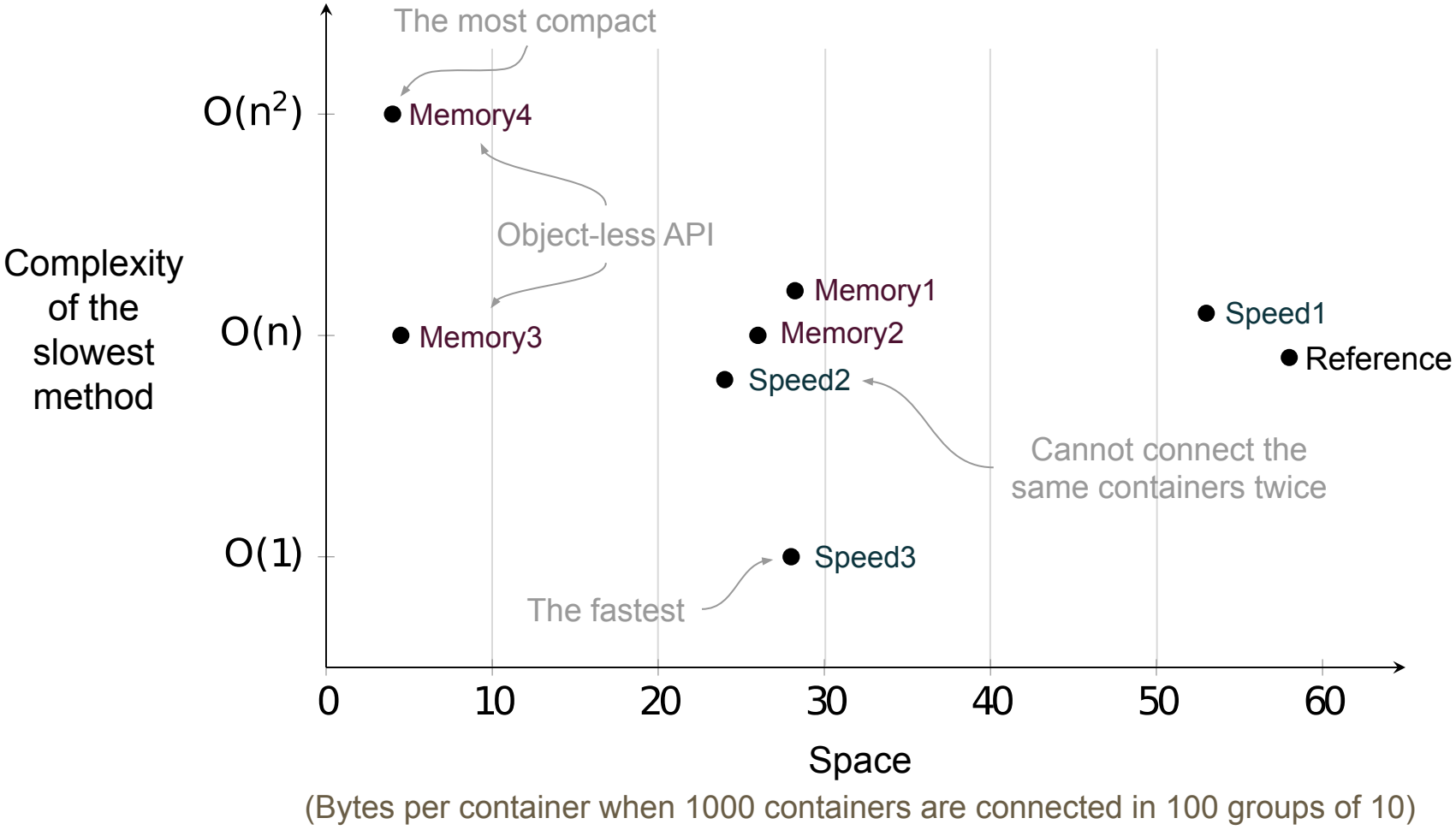
Method	Time complexity
`getAmount`	$O(n)$
`connectTo`	$O(n^2)$
`addWater`	$O(n^2)$

Memory requirements

Table 4.10 Memory requirements of all implementations from this chapter, plus *Reference*. Recall that *Memory3* and *Memory4* expose a different, object-less API.

Scenario	Version	Bytes	% of *Reference*
1000 isolated	*Reference*	108000	100%
	Memory1	20000	19%
	Memory2	20000	19%
	Memory3	8040	7%
	Memory4	4020	4%
100 groups of 10	*Reference*	61200	100%
	Memory1	29000	47%
	Memory2	25600	42%
	Memory3	4440	7%
	Memory4	4020	7%

Comparing implementations



(Bytes per container when 1000 containers are connected in 100 groups of 10)