JVM Object Allocation and References

Bloom Filters and Effective Caching

Effective Software - Lecture 11

David Šišlák
david.sislak@fel.cvut.cz

[1] Tarkoma, S., Esteve, R., Lagerspetz, E.: Theory and Practice of Bloom Filters for Distributed Systems. IEEE Communications Surveys and Tutorials, vol. 14, no. 3, 2012.

[2] Oaks, S.: Java Performance, 2nd Edition. O'Reilly, 2020.

[3] OpenJDK source code: http://openjdk.java.net

Outline

- Object allocation
  - Thread-local allocation buffers (TLABs)
  - NUMA alignment
  - Escape analysis
- Bloom filters
  - Extensions
    - Counting Bloom filter
    - Bitwise Bloom filter
- Reference objects
  - Weak, soft, final and phantom references
  - Object reachability

Fast Object Allocation

- Based on the **bump-the-pointer** technique
  - tracks the previously allocated object
  - fits the new object into the remaining free space
- **Thread-local allocation buffers** (TLABs)
  - each thread has a small exclusive area (a few percent of Eden in total), **NUMA- aligned**
  - removes the concurrency bottleneck
    - no synchronization among threads (eliminates slower atomic operations)
    - eliminates false sharing (each cache line is used by only one CPU core)
  - exclusive allocation takes only a ***few native instructions***
  - infrequent TLAB refills imply synchronization (based on ***lock inc***)
  - thread-based adaptive resizing of TLABs
    - does not work well for thread pools with varying allocation pressure
- tuning options
  - -XX:+UseTLAB ; -XX:AllocatePrefetchStyle=1; -XX:+PrintTLAB
  - -XX:AllocateInstancePrefetchLines=1 ; -XX:AllocatePrefetchLines=3
  - -XX:+ResizeTLAB ; -XX:TLABSize=10k ; -XX:MinTLABSize=2k

Fast Object Allocation - Example

C2 compiler, **standard OOP**, object size: **96 bytes**

```
mov     0x60(%r15),%r11
mov     %r11,%r10
add     $0x60,%r10
cmp     0x70(%r15),%r10
jae     0x0000000107895244
mov     %r10,0x60(%r15)
prefetchnta 0xc0(%r10)
mov     %r11,%rdi
add     $0x10,%rdi
mov     $0xa,%ecx
movabs $0x220558080,%r10  ;   {metadata('Structure')}
mov     0xa8(%r10),%r8
mov     %r8,(%r11)
mov     %r10,0x8(%r11)
xor     %rax,%rax
shl     $0x3,%rcx
rep rex.W stos %al,%es:(%rdi)  ;*new
                               ; - StructureTest::allocate@4 (line 5)
```

read the **TLAB** allocation pointer

bump the pointer

check whether it fits into the **TLAB**

store the **TLAB** allocation pointer

prefetch three cache lines ahead

prepare for **zero-initialization**
RDI = object data; ECX = 10 qwords

fill the **object header**

zero-initialize the instance

```
class Structure {
    private boolean boolean1;
    private byte byte1;
    private char char1;
    private short short1;
    private int int1;
    private long long1;
    private float float1;
    private double double1;
    private Object object1;
    private boolean boolean2;
    private byte byte2;
    private char char2;
    private short short2;
    private int int2;
    private long long2;
    private float float2;
    private double double2;
    private Object object2;

    Structure(int value, Object r

    @Override
    public String toString() {...
}
```

8B - mark word
4B / 8B – Klass ref.
… object data

Note: all examples are from a 64-bit JVM 8 on an Intel Haswell CPU and use AT&T syntax.

Using Flight Recorder to Analyze TLABs

Example with one million allocations of Structure, using compressed OOPs.

General | Allocation in New TLAB | Allocation Outside TLABs

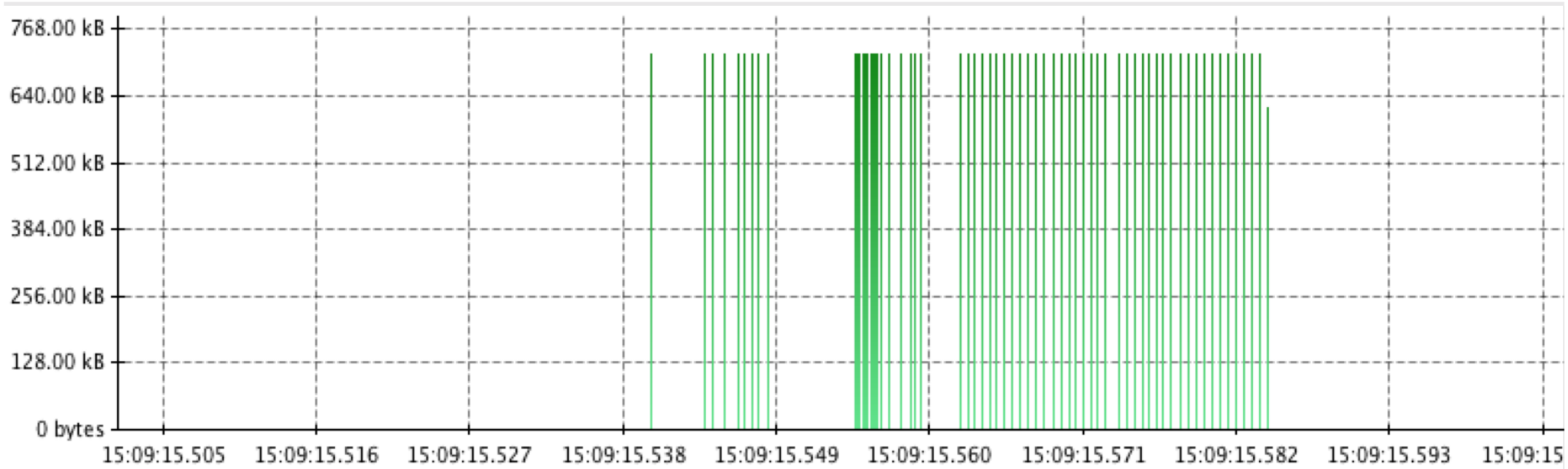
Thread Local Allocation Buffer (TLAB) Statistics

TLAB Count	65
Maximum TLAB Size	720.58 kB
Minimum TLAB Size	615.77 kB
Average TLAB Size	718.96 kB
Total Memory Allocated for TLABs	45.64 MB
Allocation Rate for TLABs	439.59 MB/s

Statistics for Object Allocations (Outside TLABs)

Object Count	0
Maximum Object Size	N/A
Minimum Object Size	N/A
Average Object Size	N/A
Total Memory Allocated for Objects	N/A
Allocation Rate for Objects	N/A

Allocation



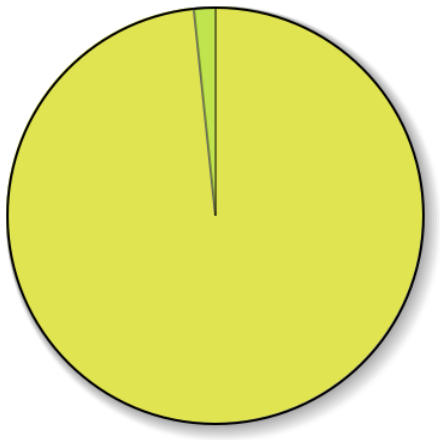
Using Flight Recorder to Analyze TLABs

Example with one million allocations of Structure, using compressed OOPs.

General | Allocation in New TLAB | Allocation Outside TLABs

Allocation by Class | Allocation by Thread | Allocation Profile

Allocation Pressure



Class	Average Object Size	TLABs	Total TLAB Size	Pressure
Structure	80 bytes	64	44.93 MB	98.46%
java.lang.Object[]	88 bytes	1	720.58 kB	1.54%

Stack Trace

Stack Trace	TLABs	Total TLAB Size	Pressure
▶ StructureTest.allocate(int, Object)	64	44.93 MB	100.00%

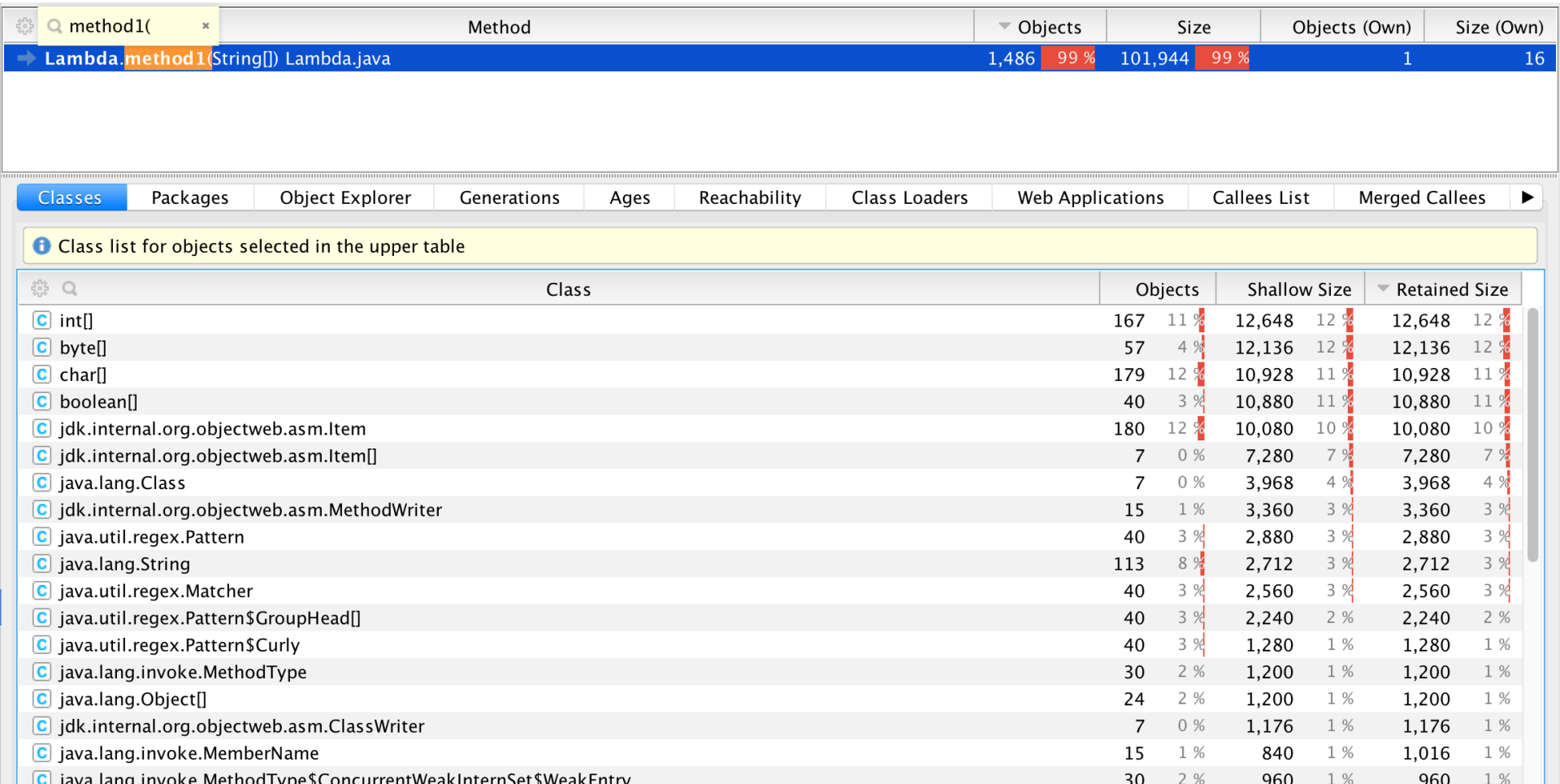
Example – Dynamic Memory Analysis

```
public static String[] method1(String[] args) {
    return Arrays.stream(args).
            filter(t -> t.matches( regex: "[^0-9]+")).
            sorted(Comparator.<String,String>comparing(String::toLowerCase).reversed()).
            collect(Collectors.toList()).toArray(new String[0]);
}
```

Example – Dynamic Memory Analysis

```
public static String[] method1(String[] args) {
    return Arrays.stream(args).
            filter(t -> t.matches( regex: "[^0-9]+")).
            sorted(Comparator.<String,String>comparing(String::toLowerCase).reversed()).
            collect(Collectors.toList()).toArray(new String[0]);
}
```

Allocations for a call with 40 elements (27 without digits):



Dynamic Memory Analysis – Optimized Example

```
private static Comparator<String> reverseIgnoreCaseComparator =  String.CASE_INSENSITIVE_ORDER.reversed();

public static String[] reversedAlphabeticalOnlyOrderOptimized(String[] args) {
    String[] arr = new String[args.length];
    int i = 0;
    for (String arg : args) {
        boolean filterOut = false;
        for (int k = 0; k < arg.length(); k++) {
            char c = arg.charAt(k);
            if ((c >= '0') && (c <= '9')) {
                filterOut = true;
                break;
            }
        }
        if (!filterOut) arr[i++] = arg;
    }
    Arrays.sort(arr, fromIndex: 0, i, reverseIgnoreCaseComparator);

    return Arrays.copyOf(arr, i);
}
```

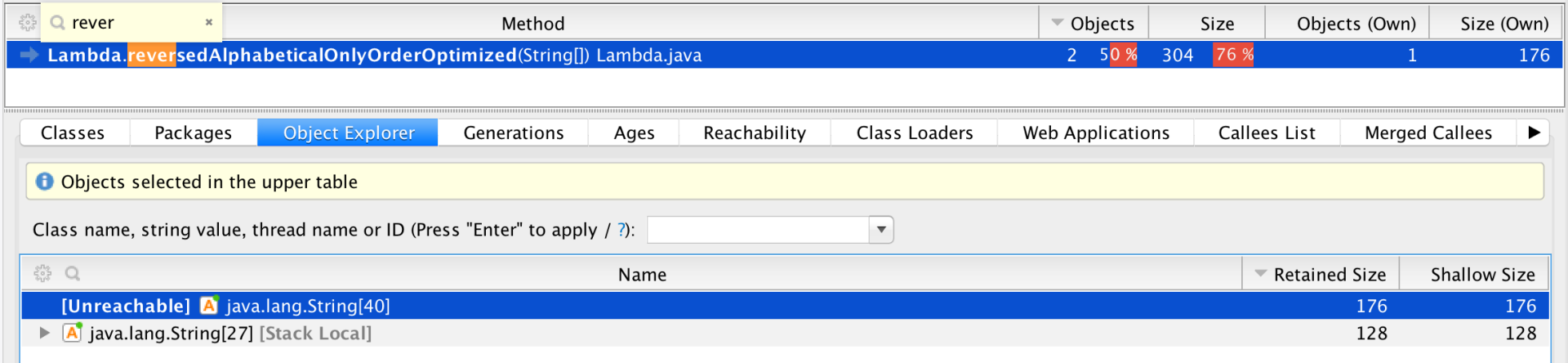
Dynamic Memory Analysis – Optimized Example

```
private static Comparator<String> reverseIgnoreCaseComparator =  String.CASE_INSENSITIVE_ORDER.reversed();

public static String[] reversedAlphabeticalOnlyOrderOptimized(String[] args) {
    String[] arr = new String[args.length];
    int i = 0;
    for (String arg : args) {
        boolean filterOut = false;
        for (int k = 0; k < arg.length(); k++) {
            char c = arg.charAt(k);
            if ((c >= '0') && (c <= '9')) {
                filterOut = true;
                break;
            }
        }
        if (!filterOut) arr[i++] = arg;
    }
    Arrays.sort(arr, fromIndex: 0, i, reverseIgnoreCaseComparator);

    return Arrays.copyOf(arr, i);
}
```

Allocations for a call with 40 elements (27 without digits):



Understand Your Application's Behavior

- Simple code can be very inefficient – **understand what you are using.**
- Large numbers of **small, short-lived objects** still slow down your application.
  - Allocations in TLABs are quite fast, but still slower than no allocation.
    - Check whether ***escape analysis*** applies, or change your code.
  - Objects allocated in TLABs benefit from cache **locality** and are **NUMA-aligned.**
  - **No false sharing** occurs between cores (data in a cache line are used by only one CPU core).
  - This increases pressure on the young generation and therefore on minor GC activity.
    - Other objects are promoted earlier to the old generation.
    - This increases the number of major GCs.
- Large numbers of **long-lived objects** slow your application even more.
  - During each collection, all live objects must be traversed.
  - Compacting GCs must copy objects.
    - This breaks the original data locality.
    - This can introduce false sharing between cores.

Escape Analysis – Not All Objects Require Heap Allocation

» The **C2 compiler** performs **escape analysis** on new objects ***after inlining hot methods.***

» Each new object allocation is classified into one of the following categories:

- ***NoEscape*** – the object does not escape the method in which it is created.
  - All of its uses are inlined.
  - It is never assigned to a static or object field, only to local variables.
  - At every point, its use must be determinable at JIT compile time and must not depend on unpredictable control flow.
  - If the object is an **array**, indexing into it must be a JIT-time constant.
- ***ArgEscape*** – the object is passed as an argument to a method, or referenced from one, but does not escape the current thread.
- ***GlobalEscape*** – the object is accessed from a different method or thread.

Escape Analysis – Not All Objects Require Heap Allocation

» *NoEscape* objects are **not allocated on the heap;** instead, the JIT performs **scalar replacement.**

- The object is decomposed into its constituent fields (stored on the stack or in registers).
- These values disappear automatically when the stack frame is popped (i.e., when the method returns).
- There is no GC impact at all, and references do not need to be tracked via write barriers.

» *ArgEscape* objects are allocated on the heap, but monitor operations can be eliminated.

Escape Analysis Example

```
public static class Vector {
    private final int a1, a2;

    public Vector(int a1, int a2) {
        this.a1 = a1;
        this.a2 = a2;
    }

    public Vector add(Vector v) {
        return new Vector(a1+v.getA1(),a2+v.getA2());
    }

    public int mul(Vector v) {
        return v.getA1()*a1 +  v.getA2()*a2;
    }

    public int getA1() {
        return a1;
    }

    public int getA2() {
        return a2;
    }
}


public int compute(int val) {
    Vector v = new Vector(val+1, val*2);
    synchronized (v) {
        return v.add(v).mul(v);
    }
}
```

Escape Analysis Example

C2 compilation with inlining:

```
        74   22   !   4       EscapeExample::compute (37 bytes)
                                @ 10   EscapeExample$Vector::<init> (15 bytes)   inline (hot)
                                  @ 1   java.lang.Object::<init> (1 bytes)   inline (hot)
                                @ 20   EscapeExample$Vector::add (26 bytes)   inline (hot)
                                  @ 9   EscapeExample$Vector::getA1 (5 bytes)   accessor
                                  @ 18   EscapeExample$Vector::getA2 (5 bytes)   accessor
                                  @ 22   EscapeExample$Vector::<init> (15 bytes)   inline (hot)
                                    @ 1   java.lang.Object::<init> (1 bytes)   inline (hot)
                                @ 24   EscapeExample$Vector::mul (20 bytes)   inline (hot)
                                  @ 1   EscapeExample$Vector::getA1 (5 bytes)   accessor
                                  @ 10   EscapeExample$Vector::getA2 (5 bytes)   accessor
sub     $0x18,%rsp
mov     %rbp,0x10(%rsp)
mov     %edx,%r11d
add     %edx,%r11d
mov     %edx,%r10d
shl     %r10d
mov     %r10d,%r8d
add     %r10d,%r8d
imul    %r10d,%r8d
add     $0x2,%r11d
inc     %edx
imul    %r11d,%edx
mov     %edx,%eax
add     %r8d,%eax
add     $0x10,%rsp
pop     %rbp
test    %eax,-0x21742ec(%rip)
retq
```

```
public int compute(int val) {
    Vector v = new Vector(val+1, val*2);
    synchronized (v) {
        return v.add(v).mul(v);
    }
}
```

```
# this:     rsi:rsi   = 'EscapeExample'
# parm0:    rdx       = int
#           [sp+0x20]  (sp of caller)
```

no allocation at all; no synchronization

all computation is performed in registers only

Bloom Filter

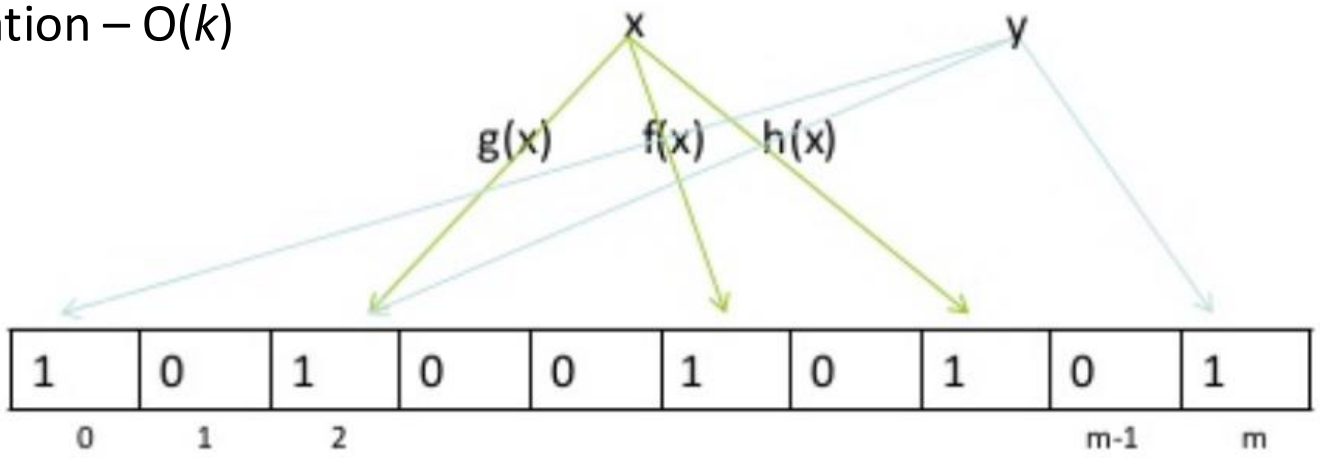
- **Bloom filter** operations
  - **add** a new object to the set
  - **test** whether a given object is a member of the set
  - **deletion** is not supported
- **significant memory savings** (a few bits per element) compared with other collections
  - at the cost of a **small false-positive** rate (typically about 1%)
  - guaranteed absence of **false negatives**
  - elements are not stored explicitly (unlike in *standard collections*)
- add and query operations have **constant complexity**, even in the presence of collisions
- very useful in **big-data** processing and many other applications
  - used to test whether the **object is certainly not present**
  - e.g., they can eliminate many I/O operations by ruling out reads of full collections from a file when the Bloom filters are kept in RAM or can be read quickly from disk

Bloom Filter

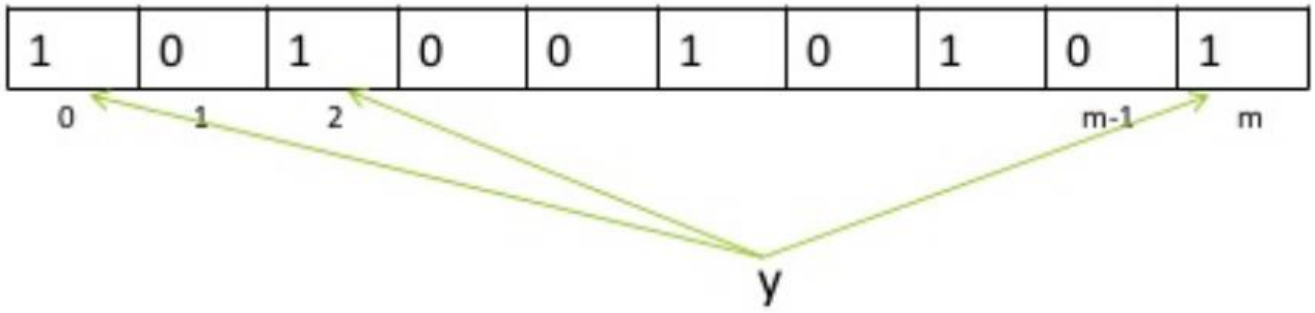
» use a **bit array** with ***m*** bits

» use ***k* independent hash functions**

» **add** operation – O(*k*)



» **query** operation



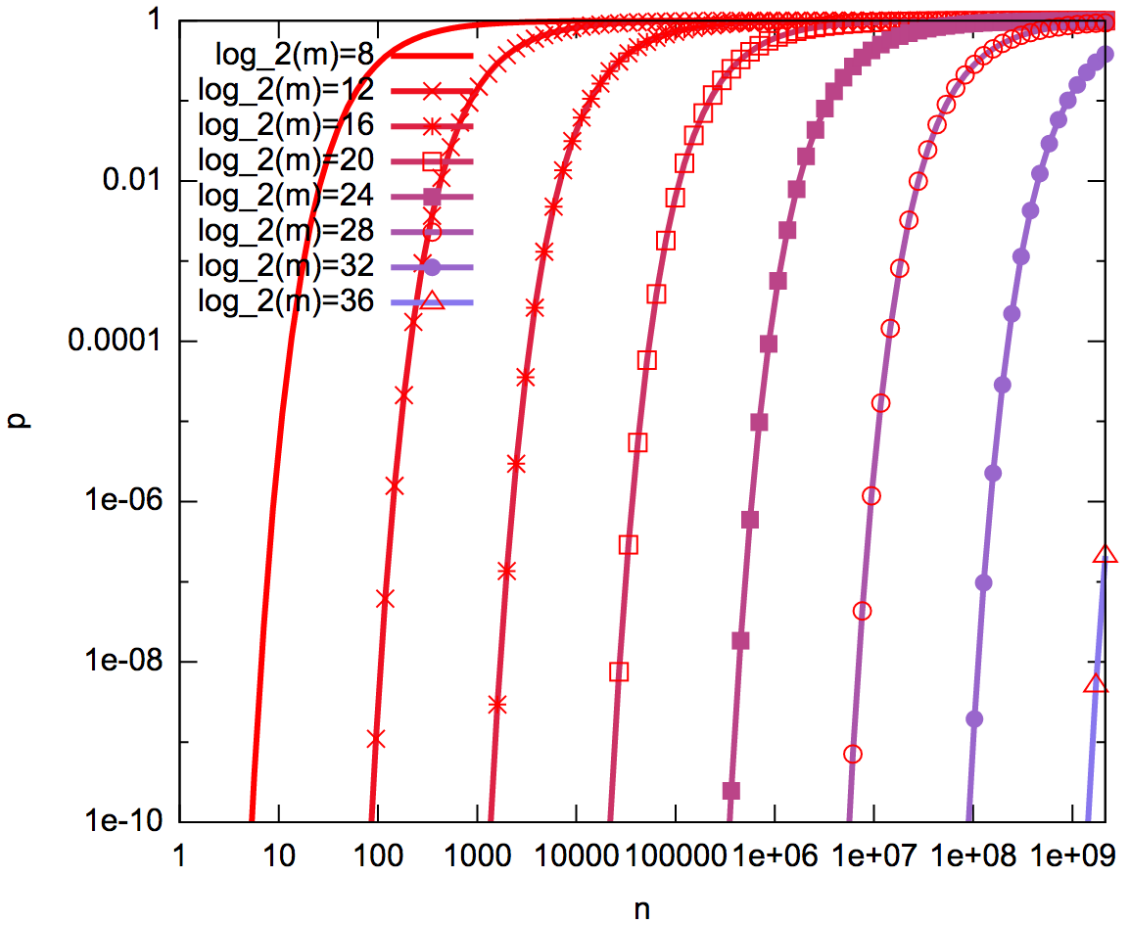
Bloom Filter

» Number of bits in the filter

$$ceil\left(\frac{n \cdot \ln(p)}{\ln\left(\frac{1}{2^{\ln(2)}}\right)}\right)$$

» Number of hash functions

$$round\left(\frac{\ln(2) \cdot m}{n}\right)$$



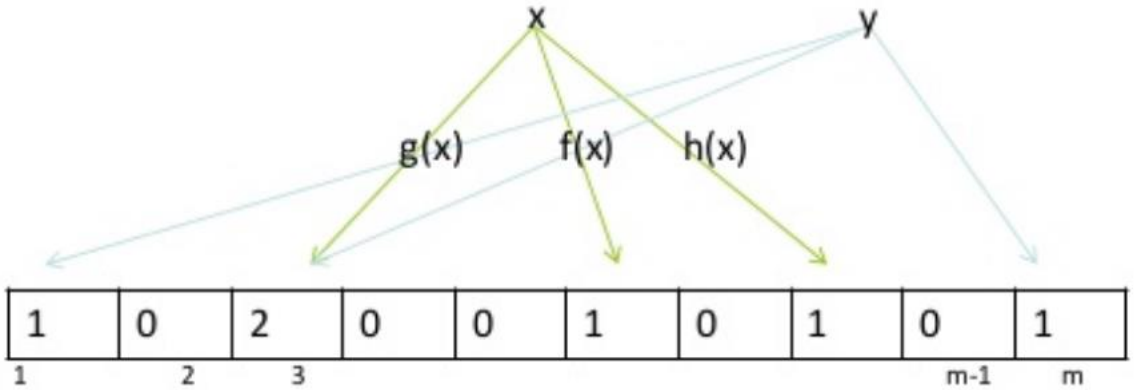
» Example – store 1 million strings with a total size of 25 MB

- Set<String> requires >50 MB of retained size
- A Bloom Filter with a 1% FP rate requires 1.13 MB and 7 hash functions
  - This is more than 44x smaller, and 99% of negative queries are true negatives.

Extensions of Bloom Filters

» **Counting Bloom filter**

- supports **delete** and **count-estimate** operations



- each position in the filter is a bucket (e.g., 3 bits) that acts as a counter
  - add – increment
  - delete – decrement the counter; the estimated count is the minimum counter value
  - query – test for nonzero counters
- **bucket-overflow problem**
  - no further increments are possible once the counter reaches its max value
  - deletions can increase the false-negative rate

» **Bitwise Bloom filter**

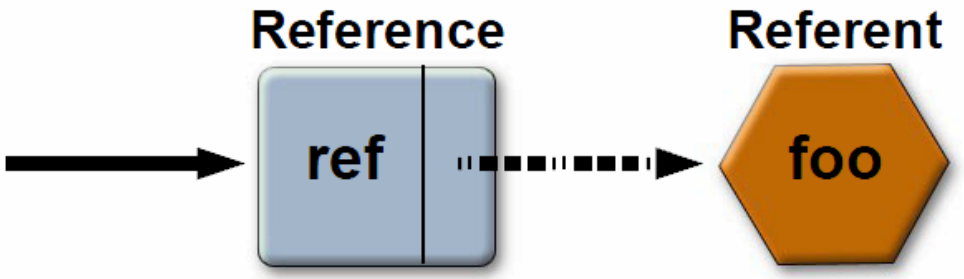
- uses multiple counting filters, added dynamically, to address the issues above

Reference Objects

» Pre- and post-mortem hooks are more **flexible** than finalization.

» **reference types** (ordered from strongest to weakest):

- **{strong reference}**
- **soft reference** – optional reference queue
- **weak reference** – optional reference queue
- **{final reference}** – mandatory reference queue
- **phantom reference** – mandatory reference queue

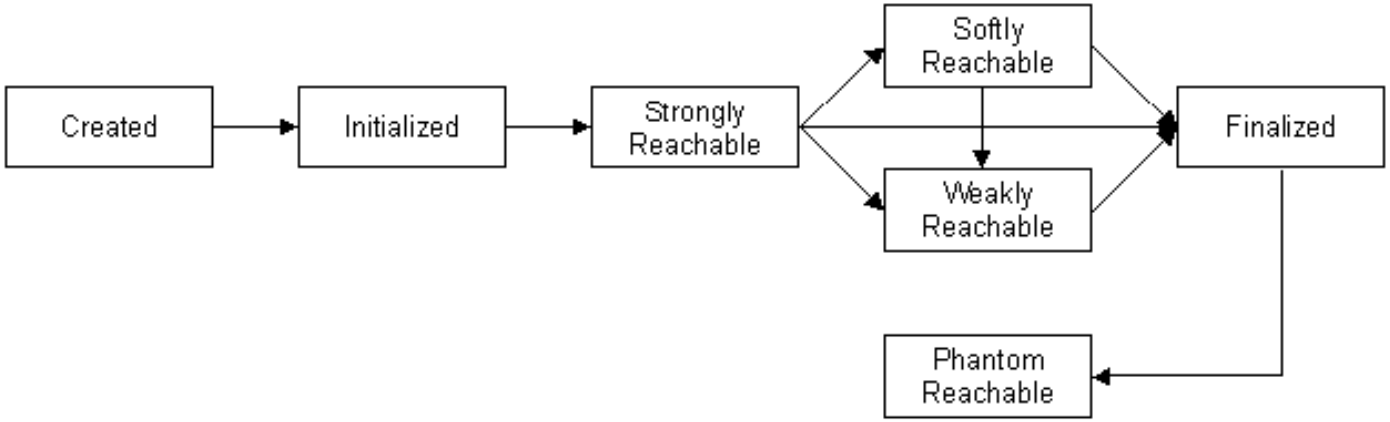



```
ref  = new WeakReference(foo, rq);
```

Reference Objects

- A reference object can be enqueued on a designated **ReferenceQueue** when the GC determines that its referent has become less reachable; the referent is then released.
- References are enqueued **only if you keep a strong reference to the reference object itself.**
- The GC must run first; then the **Reference Handler** thread enqueues them into the **reference queue.**
- A Reference is just **another heap object** – 48 bytes in a 64-bit JVM with standard OOPs.

Object Reachability



- **strongly reachable** – reachable from GC roots
- **softly reachable** – not strongly reachable, but reachable via a soft reference
- **weakly reachable** – not strongly or softly reachable, but reachable via a weak reference; the referent link is cleared and the object becomes eligible for finalization
- **eligible for finalization** – not strongly, softly or weakly reachable, and has a non-trivial finalize method
- **phantom reachable** – not strongly, softly, or weakly reachable; already finalized or without a finalize method; but still reachable via a phantom reference
- **unreachable** – none of the above; eligible for reclamation by the GC

Weak Reference

- pre-finalization processing
- Typical uses:
  - **do not keep the object alive solely because of this reference**
  - the target is not owned, e.g., listeners
  - canonicalizing maps – e.g., ObjectOutputStream
  - can be used to implement **a more flexible form of finalization**:
    - prioritize cleanup
    - decide when to run cleanup
- get() returns:
  - the referent, if it has not been cleared
  - null otherwise
- **the referent is cleared** by the GC (when passed to the Reference Handler) and **can then be reclaimed**
- **copy the referent to a strong reference and check that it is not null before using it**
- WeakHashMap<K,V> uses weak keys; cleanup occurs during standard operations

Weak Reference – External Resource Cleanup

» **cleanup** approach using ReferenceQueue<T>

- a **dedicated thread**

```
ReferenceQueue<Image3> refQueue =
    NativeImage3.referenceQueue();
while (true) {
    NativeImage3 nativeImg =
        (NativeImage3) refQueue.remove();
    nativeImg.dispose();
}
```

- perform cleanup **before creating new** objects
  - limit cleanup work to avoid long pauses
  - use poll() – a non-blocking retrieval of the next element

Custom Finalizer Example

```
public abstract class CustomFinalizer extends WeakReference<Object> {
    private static final ReferenceQueue<Object> referenceQueue = new ReferenceQueue<>();
    private static final CustomFinalizer circularEnd = new CustomFinalizer() {...};

    private CustomFinalizer next, prev;

    public CustomFinalizer(Object referent) {...}

    private CustomFinalizer() {...}

    private void executeCustomFinalize() {...}

    public abstract void customFinalize();

    static {
        Thread cleanupThread = new Thread(() -> {
            for (;;) {
                try {
                    CustomFinalizer toCleanup = (CustomFinalizer) referenceQueue.remove();
                    toCleanup.executeCustomFinalize();
                } catch (InterruptedException e) {
                }
            }
        }, name: "Custom finalizer");
        cleanupThread.setDaemon(true);
        cleanupThread.start();
    }
}
```

Custom Finalizer Example

```
public CustomFinalizer(Object referent) {
    super(referent, referenceQueue);
    synchronized (circularEnd) {
        next = circularEnd.next;
        circularEnd.next.prev = this;
        prev = circularEnd;
        circularEnd.next = this;
    }
}

private void executeCustomFinalize() {
    if (next == null) return;
    synchronized (circularEnd) {
        prev.next = next;
        next.prev = prev;
    }
    next = prev = null;
    customFinalize();
}
```

» Usage example: beware of the **implicit strong reference to this** in an instance context.

```
new CustomFinalizer(monitoredObjectForFinalization) {
    @Override
    public void customFinalize() {
        // custom finalization
    }
};
```

Soft Reference

» pre-finalization processing

» Typical uses:

- **would like to keep the referent, but can lose it**
- suitable for **caches** – use strong references to keep data alive while needed
  - objects with expensive initialization
  - frequently used information
- reclaim only under "**memory pressure**", based on heap usage

now – timestamp > (SoftRefLRUPolicyMSPerMB * amountOfFreeMemoryInMB)

-XX:SoftRefLRUPolicyMSPerMB=N (default 1000)

- all soft references are cleared before OutOfMemoryError is thrown

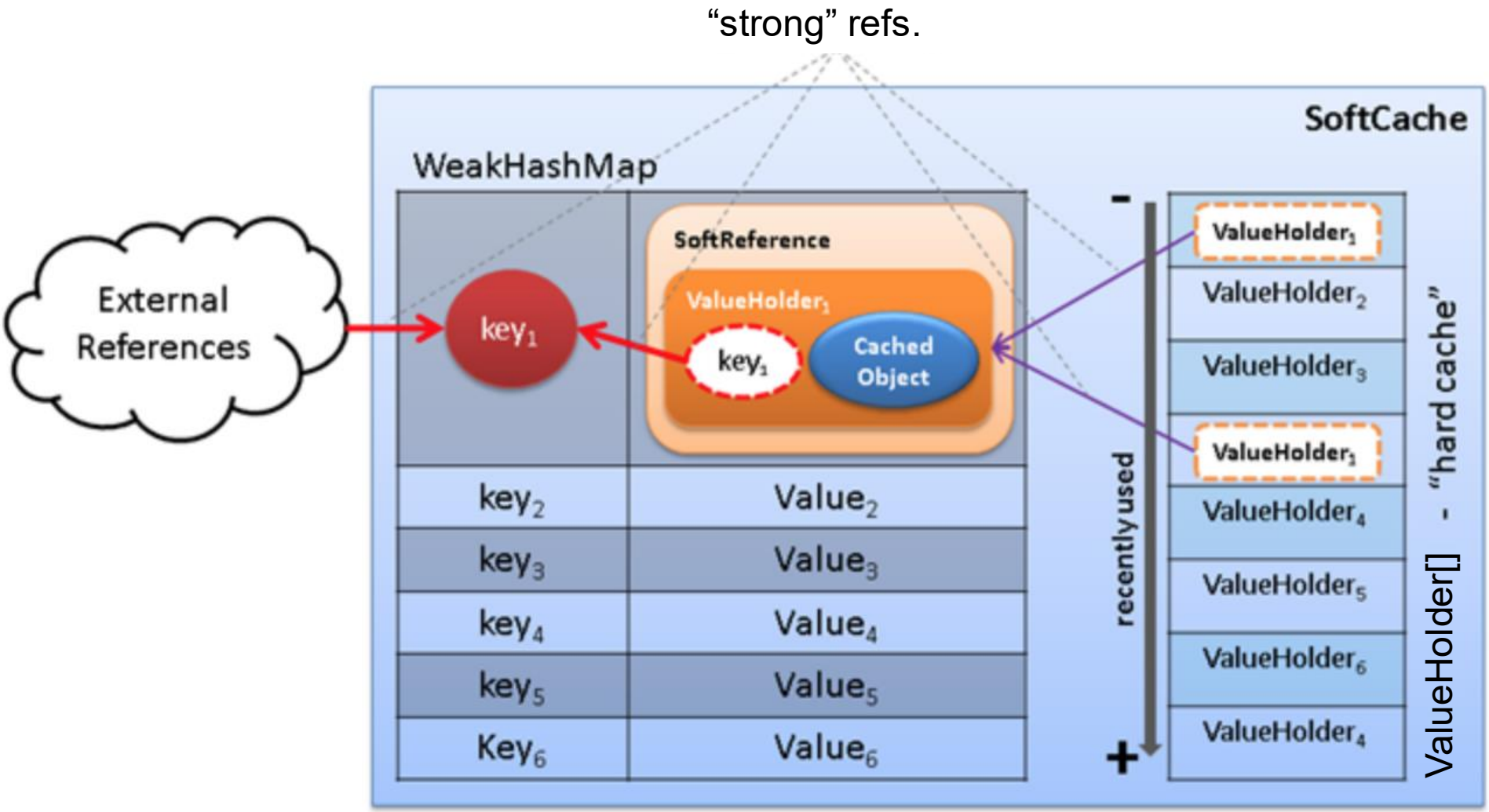
» get() returns:

- the referent, if it has not been cleared; null otherwise
- **updates the last-used timestamp** (so recently used objects may stay alive longer)

» **the referent is cleared** by the GC (when passed to the Reference Handler) and **can then be reclaimed**

Efficient Cache Example

efficient **LRU** tracking combined with memory-pressure handling for older



Final Reference – Objects with Non-Trivial finalize() Methods

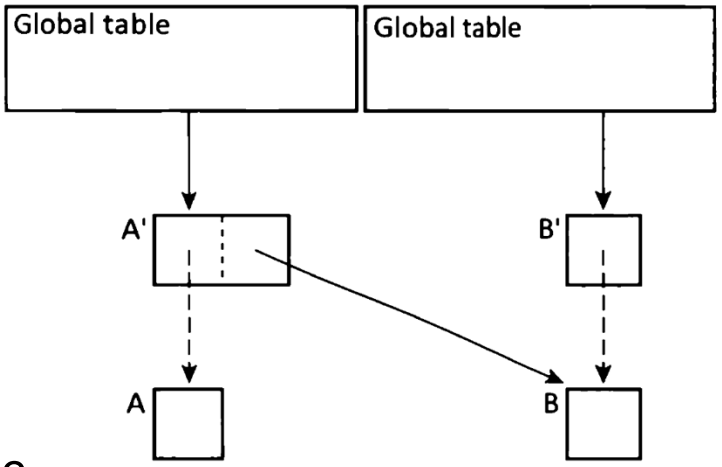
- the final-reference hook cannot be used directly (the underlying class is package-private)
- instance allocation for an object with a **non-trivial finalize() method**
  - allocation is slower than for standard objects
  - runtime call to **Finalizer.register,** with a possible global safepoint
    - not inlined; all references must be saved on the stack in an OopMap
  - allocates a **FinalReference** instance and performs synchronized tracking
- **the referent is neither cleared nor reclaimed** before finalization
  - **all objects referenced** from it also remain unreclaimed
- there is only one **Finalizer** thread for all FinalReferences of all types
  - calls the **finalize()** method and then **clears** the referent
    - **problematic** if finalize() recreates a **strong reference**
    - there is no priority control among multiple finalize() methods
    - a long-running finalize() delays all other finalization
  - because it is a daemon thread, the JVM can terminate before all finalization has completed
- finalized objects can be reclaimed during a **subsequent GC cycle**

Phantom Reference

» post-finalization processing; a pre-mortem hook

» Typical uses:

- **indicates that the object is no longer in use,** before it is reclaimed
- used to guarantee a specific **order of finalization for objects** (not possible with weak references)
  - A and B – finalizable objects
  - A' and B' - PhantomReferences



» get() returns:

- null always
- the referent can still be read via reflection
  - this avoids recreating a strong reference

» a reference queue **must** be specified in the constructor

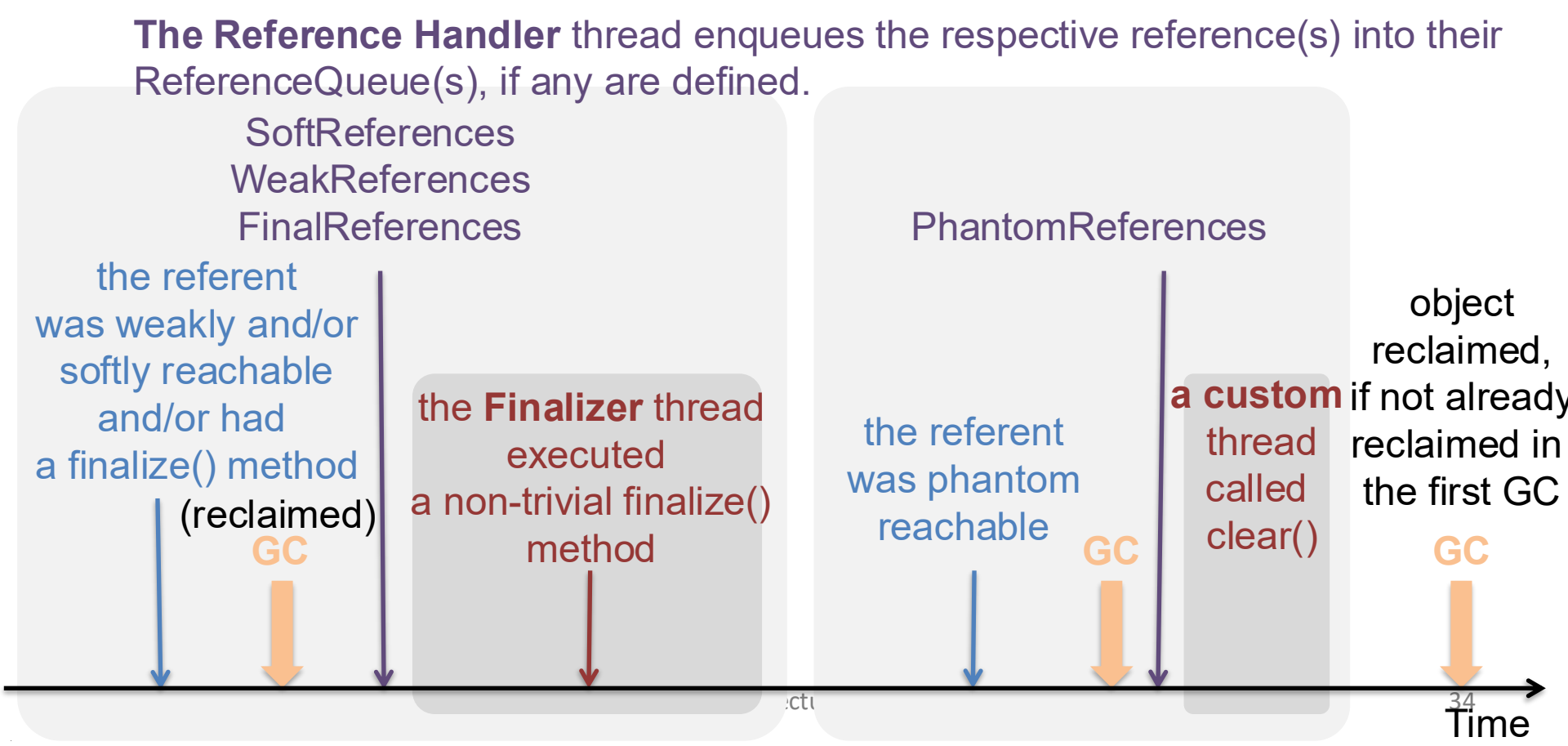
» **the referent is not cleared or reclaimed** until all phantom references become unreachable or are manually cleared using clear()

» **all objects referenced** from it also cannot be reclaimed

Reference Object Lifecycle

» Only **one GC cycle** is needed to reclaim the *referent* if only soft or weak references to the same object exist.

» **Multiple GC cycles** are needed to reclaim a *referent* that has at least one final or phantom reference.



Performance Costs of References

- **creation** cost
  - instance allocation
  - synchronization for tracking Reference objects (via strong references)
- **garbage-collection** cost (-XX:+PrintReferenceGC –XX:+PrintGCDetails)
  - tracking live reference objects without following their referents
  - construct a list of live Reference objects during each GC cycle
    - via the discovered field in Reference
  - per-reference traversal overhead, regardless of whether the referent is collected
    - for soft, weak/finalizable, and phantom references
  - reference objects themselves are also subject to garbage collection
- **enqueue** cost
  - Reference Handler enqueue operations require synchronization
- **reference-queue processing** cost
  - synchronized queue consumption

Object Reachability