Effective Software

Lecture 11: JVM - Object Allocation, Bloom Filters, References, Effective Caching

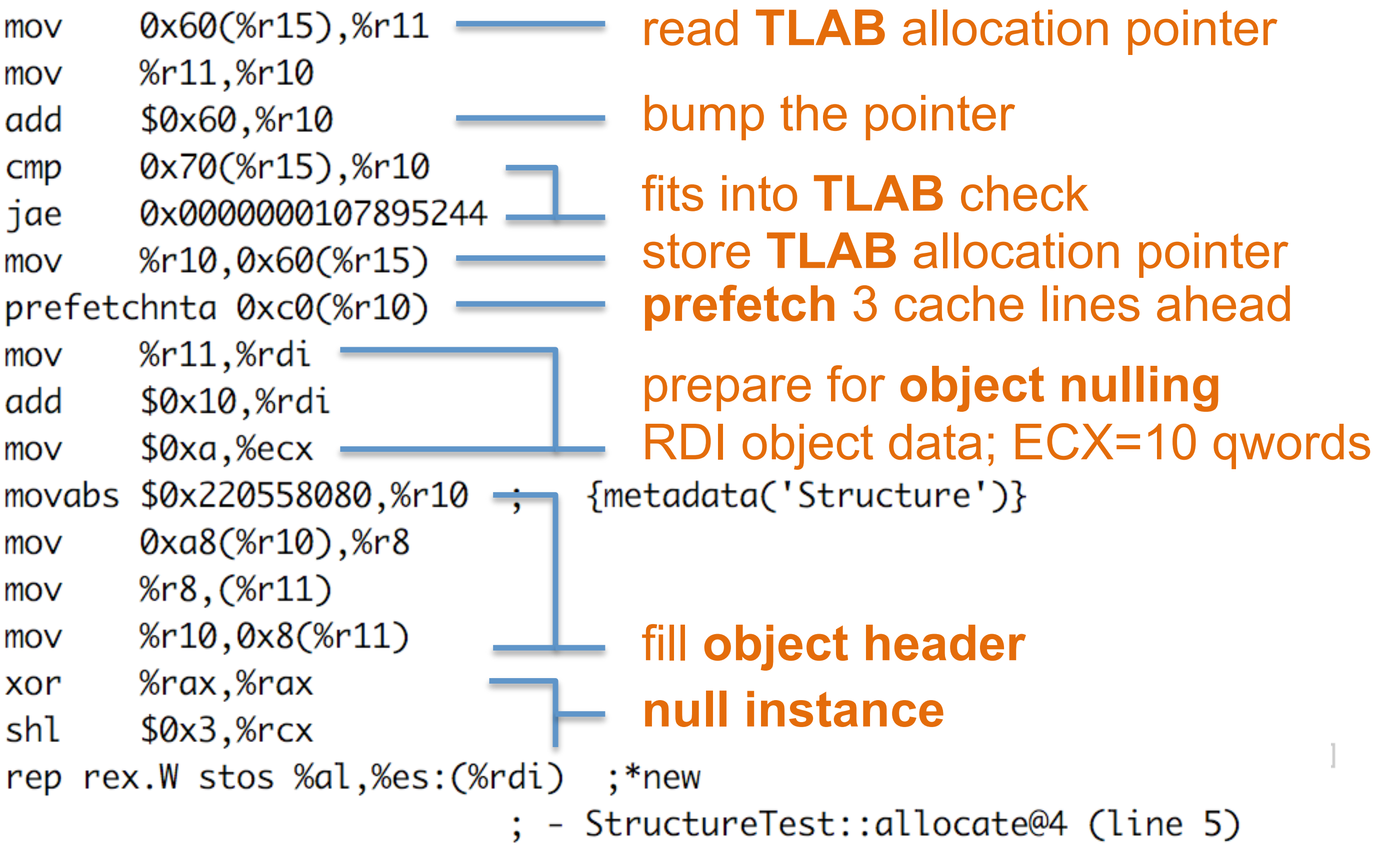
David Šišlák
david.sislak@fel.cvut.cz

Fast Object Allocation

- based on **bump-the-pointer** technique
  - track previously allocated object
  - fit new object into remainder of generation end
- **thread-local allocation buffers** (TLABs)
  - each thread has small exclusive area (few % of Eden in total) **aligned NUMA**
  - remove concurrency bottleneck
    - no synchronization among threads (remove slower atomics)
    - remove false sharing (cache line used just by one CPU core)
  - exclusive allocation takes about *few native instructions*
  - infrequent full TLABs implies synchronization (based on *lock inc*)
  - thread-based adaptive resizing of TLAB
    - not working 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, 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 **TLAB** allocation pointer

bump the pointer

fits into **TLAB** check

store **TLAB** allocation pointer

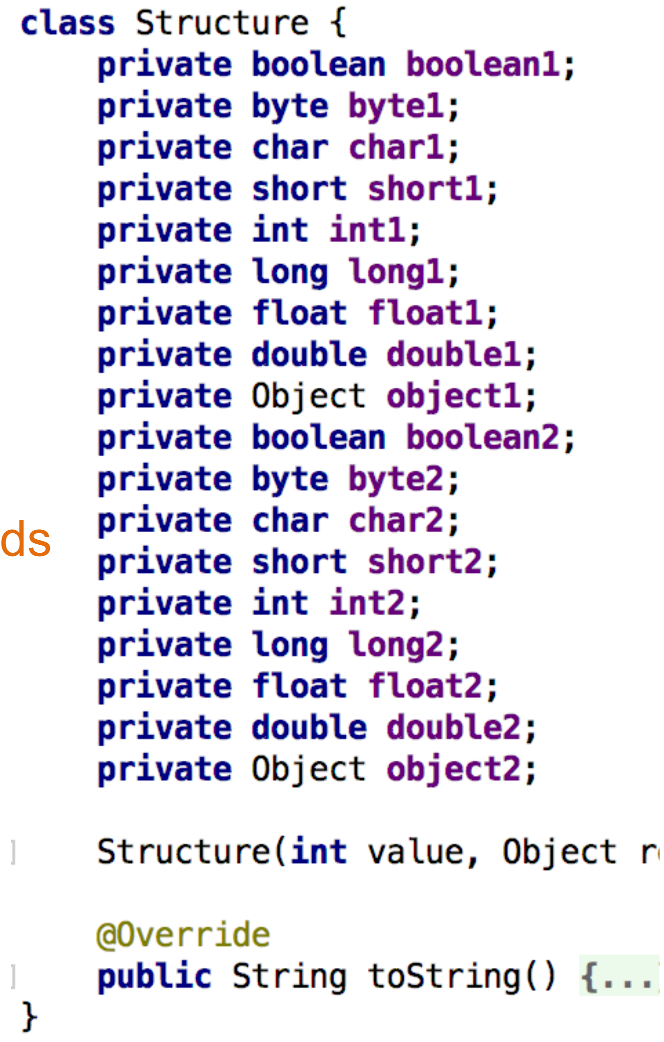
prefetch 3 cache lines ahead

prepare for **object nulling**

RDI object data; ECX=10 qwords

fill **object header**

null instance



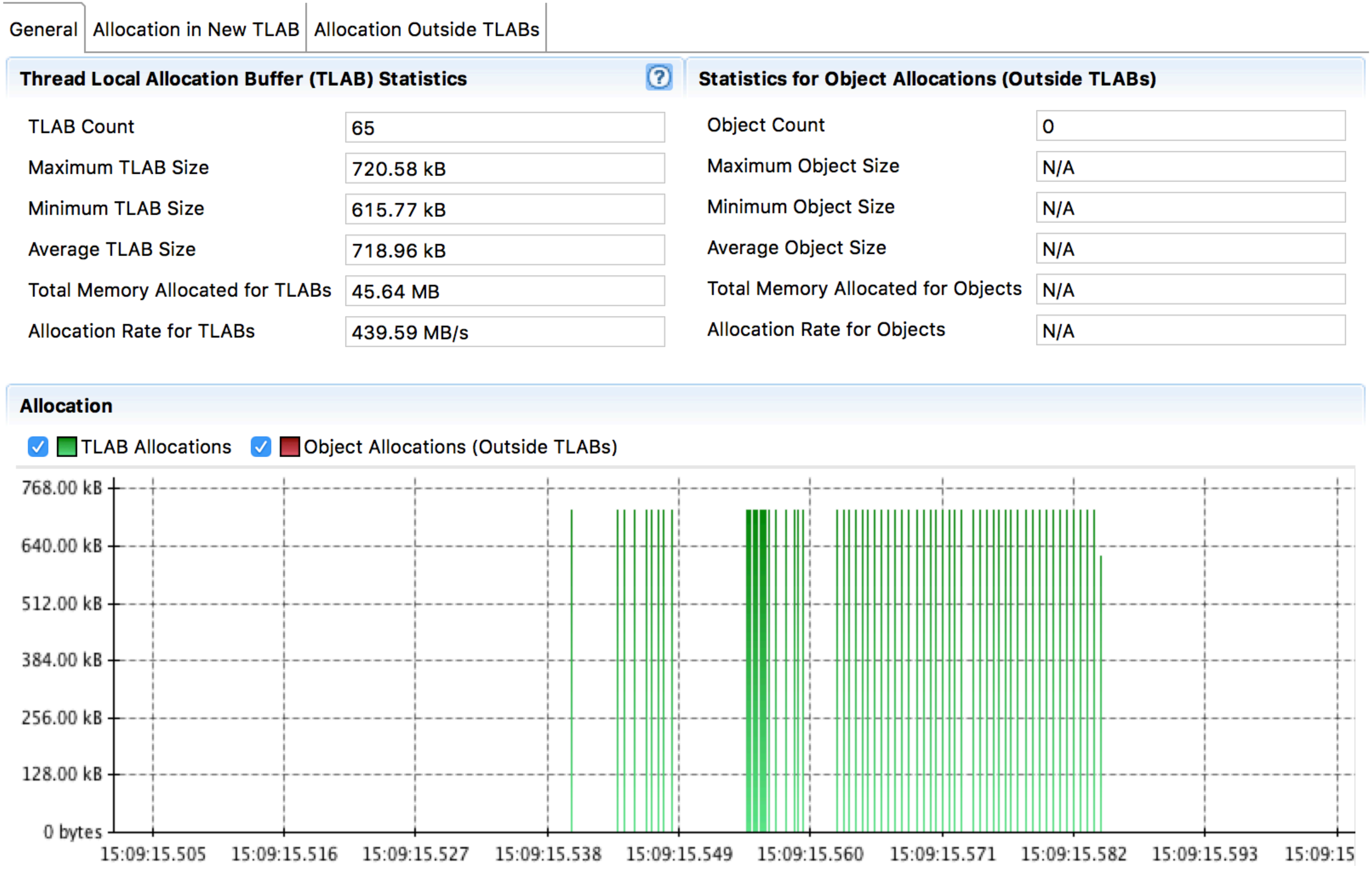
```java
class Structure {
    private boolean boolean1;
    private byte byte1;
    private char char1;
    private short short1;
    private int int1;
    private long long1;
    private float float1;
    private double double1;
    private Object object1;
    private boolean boolean2;
    private byte 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 in JVM 8 64-bit, Intel Haswell CPU, AT&T syntax

Flight Recording to Analyze TLAB

example with million of allocations of Structure



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)

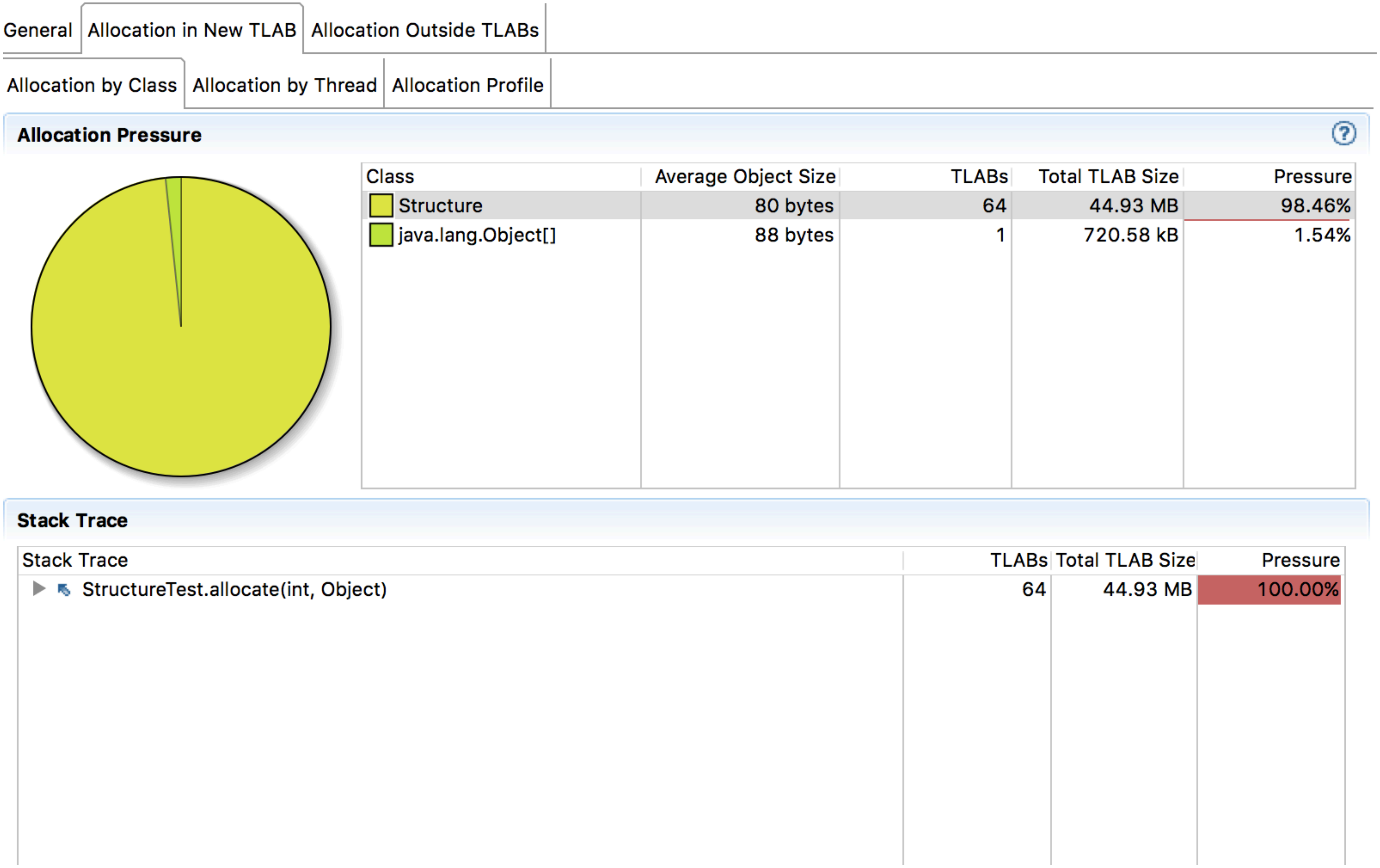
Object Count	0
Maximum Object Size	N/A
Minimum Object Size	N/A
Average Object Size	N/A
Total Memory Allocated for Objects	N/A
Allocation Rate for Objects	N/A

Allocation

☑ 🟩 TLAB Allocations ☑ 🟥 Object Allocations (Outside TLABs)

Flight Recording to Analyze TLAB

example with million of allocations of Structure; compressed OOP used



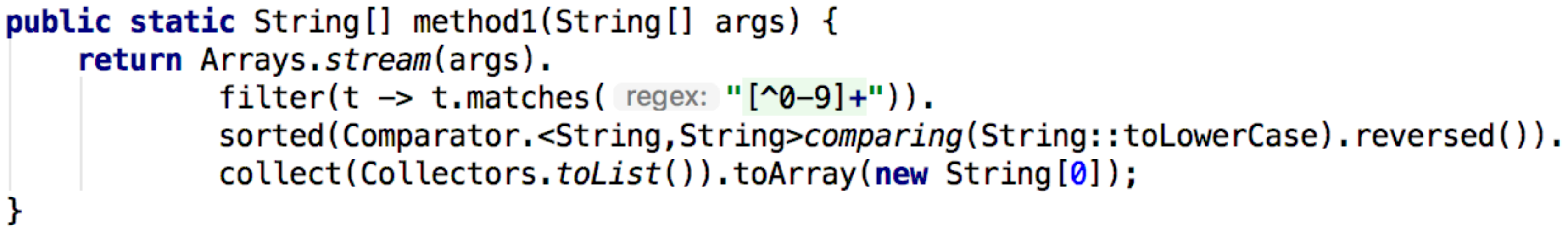
Allocation Pressure

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

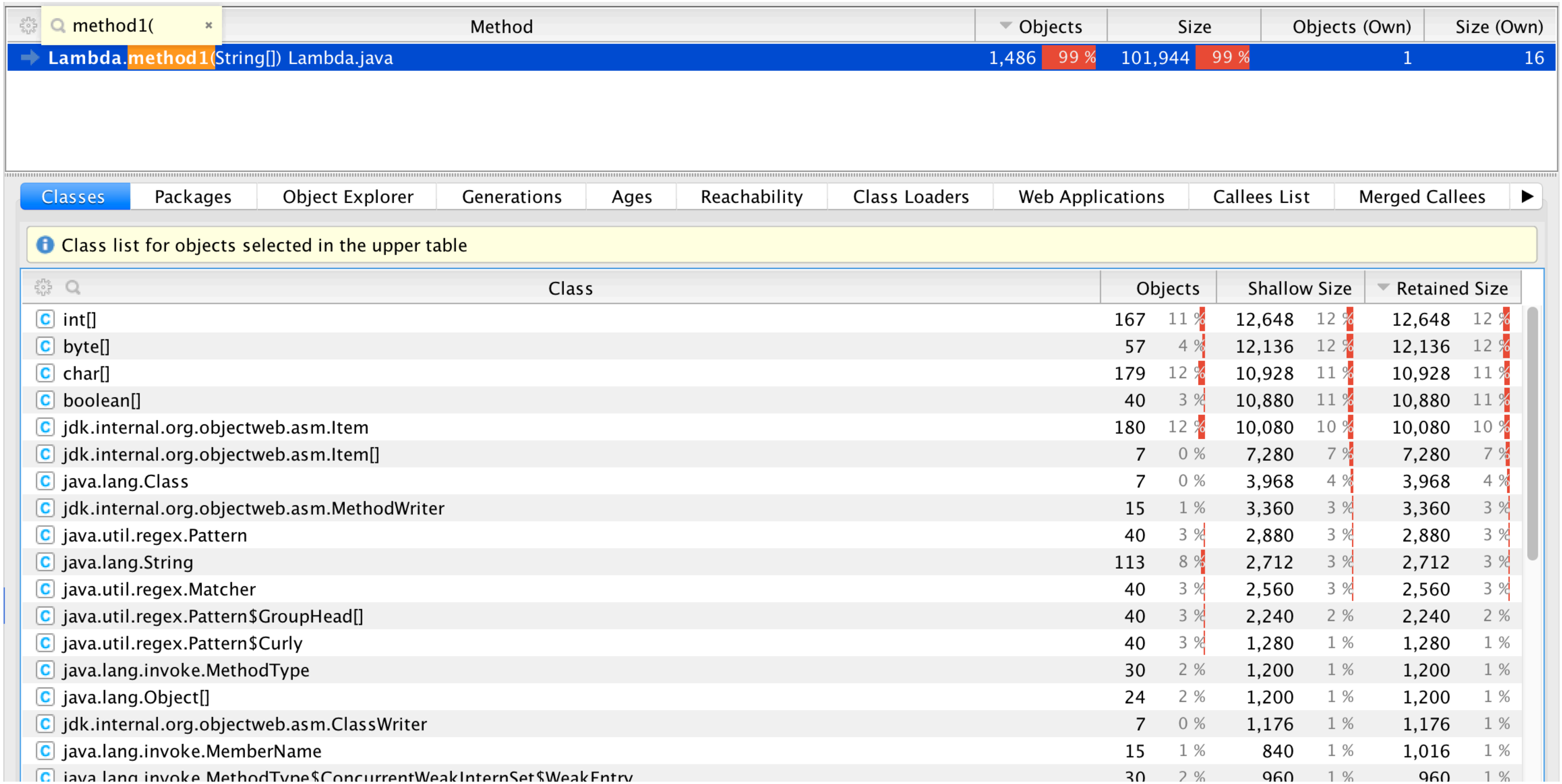


```
public static String[] method1(String[] args) {
    return Arrays.stream(args).
            filter(t -> t.matches( regex: "[^0-9]+")).
            sorted(Comparator.<String,String>comparing(String::toLowerCase).reversed()).
            collect(Collectors.toList()).toArray(new String[0]);
}
```

Example – Dynamic Memory Analysis

```java
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 when called with 40 elements (27 without digits):

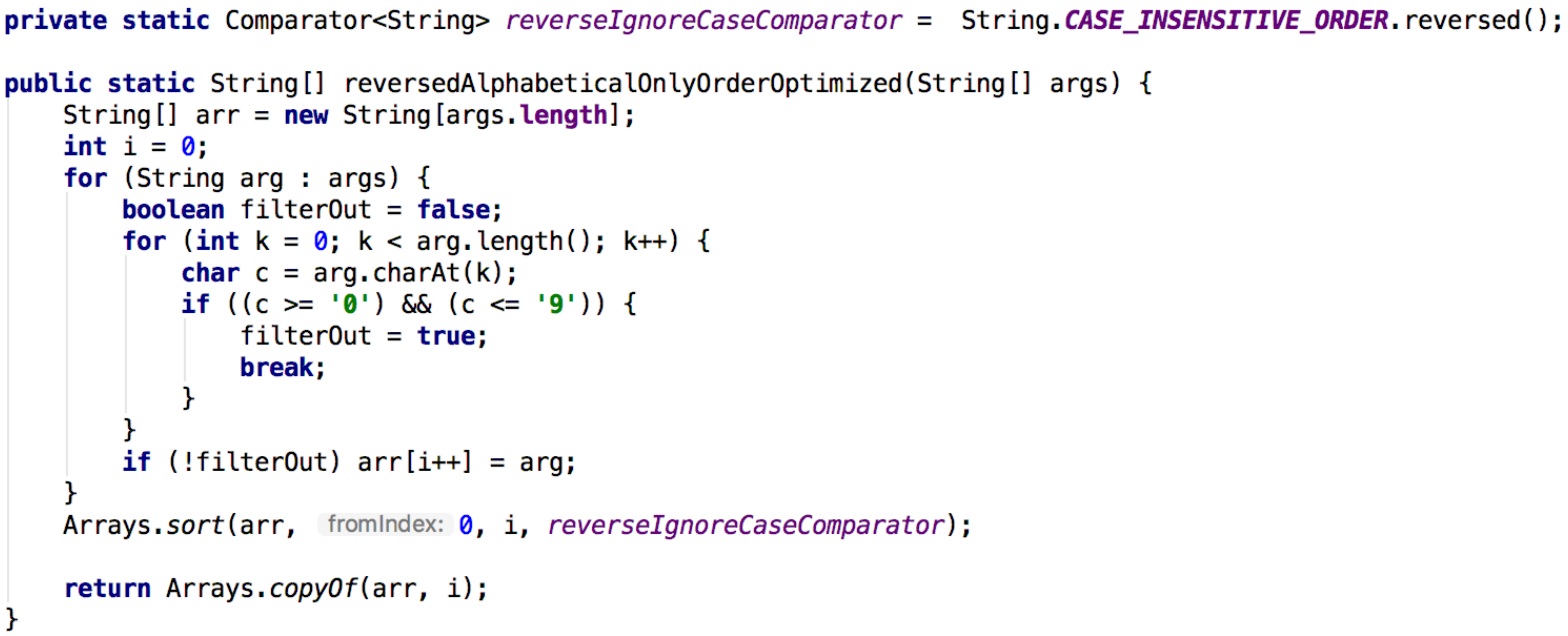


Method	Objects		Size		Objects (Own)	Size (Own)
Lambda.method1(String[]) Lambda.java	1,486	99 %	101,944	99 %	1	16

Class list for objects selected in the upper table

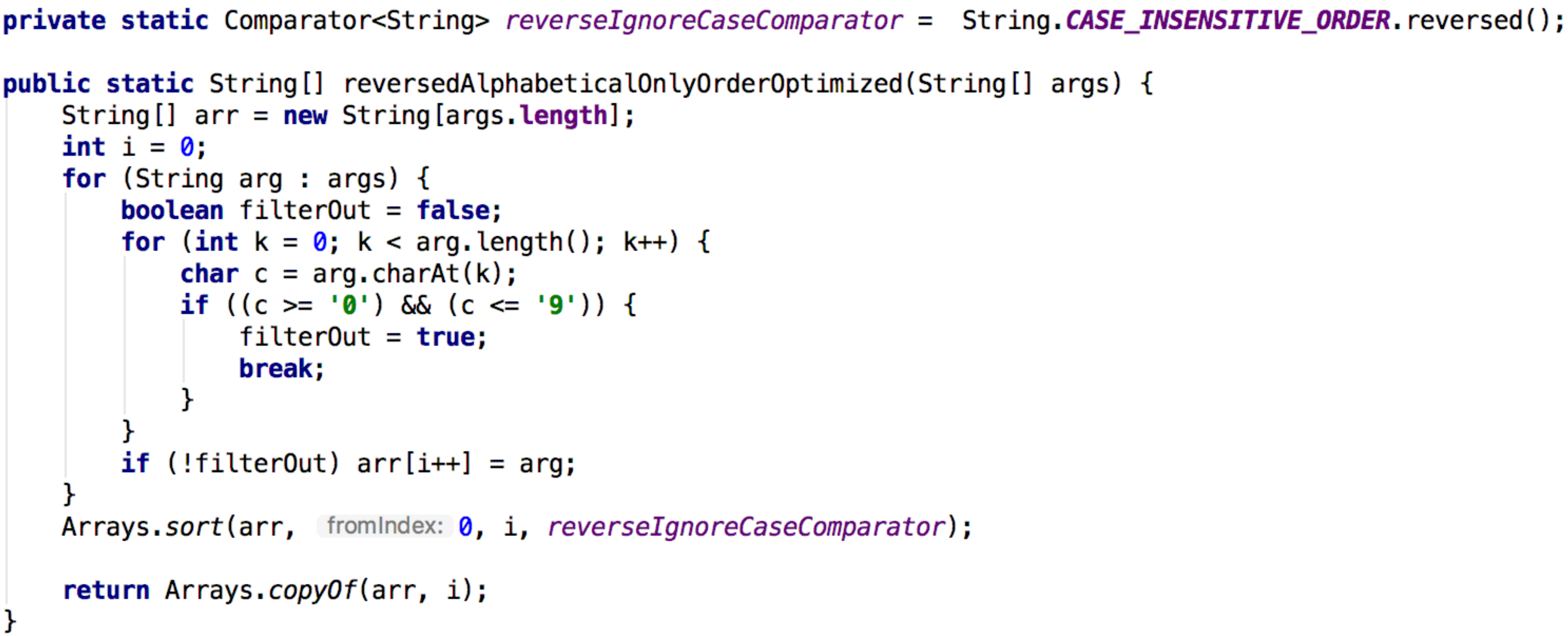
Class	Objects		Shallow Size		Retained Size	
int[]	167	11 %	12,648	12 %	12,648	12 %
byte[]	57	4 %	12,136	12 %	12,136	12 %
char[]	179	12 %	10,928	11 %	10,928	11 %
boolean[]	40	3 %	10,880	11 %	10,880	11 %
jdk.internal.org.objectweb.asm.Item	180	12 %	10,080	10 %	10,080	10 %
jdk.internal.org.objectweb.asm.Item[]	7	0 %	7,280	7 %	7,280	7 %
java.lang.Class	7	0 %	3,968	4 %	3,968	4 %
jdk.internal.org.objectweb.asm.MethodWriter	15	1 %	3,360	3 %	3,360	3 %
java.util.regex.Pattern	40	3 %	2,880	3 %	2,880	3 %
java.lang.String	113	8 %	2,712	3 %	2,712	3 %
java.util.regex.Matcher	40	3 %	2,560	3 %	2,560	3 %
java.util.regex.Pattern$GroupHead[]	40	3 %	2,240	2 %	2,240	2 %
java.util.regex.Pattern$Curly	40	3 %	1,280	1 %	1,280	1 %
java.lang.invoke.MethodType	30	2 %	1,200	1 %	1,200	1 %
java.lang.Object[]	24	2 %	1,200	1 %	1,200	1 %
jdk.internal.org.objectweb.asm.ClassWriter	7	0 %	1,176	1 %	1,176	1 %
java.lang.invoke.MemberName	15	1 %	840	1 %	1,016	1 %
java.lang.invoke.MethodType$ConcurrentWeakInternSet$WeakEntry	30	2 %	960	1 %	960	1 %

Example – Optimized – Dynamic Memory Analysis



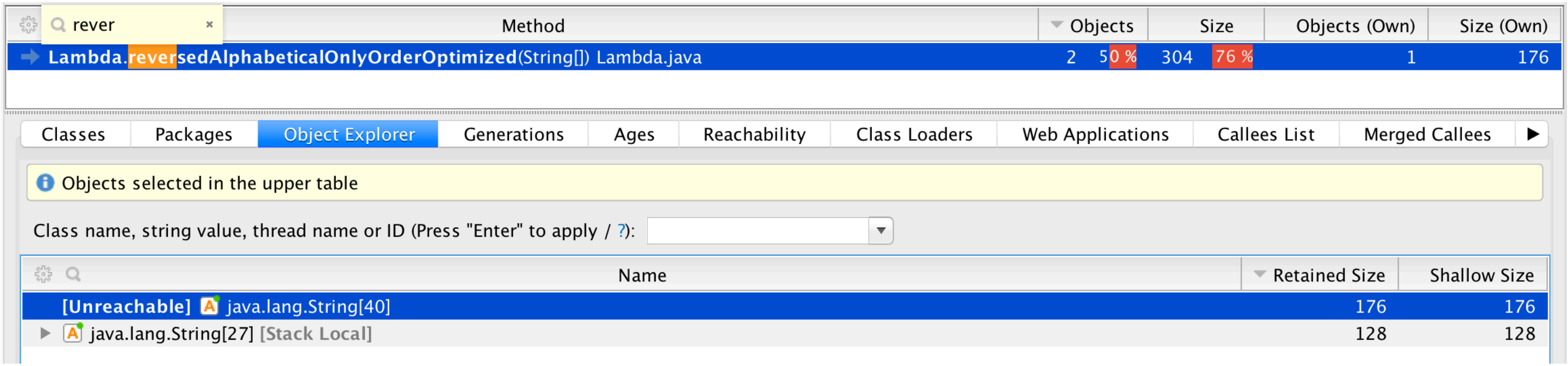
```
private static Comparator<String> reverseIgnoreCaseComparator =  String.CASE_INSENSITIVE_ORDER.reversed();

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);
}
```

Example – Optimized – Dynamic Memory Analysis



```
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 when called with 40 elements (27 without digits):



rever	Method	Objects		Size		Objects (Own)	Size (Own)
	Lambda.reversedAlphabeticalOnlyOrderOptimized(String[]) Lambda.java	2	50 %	304	76 %	1	176

Objects selected in the upper table

Class name, string value, thread name or ID (Press "Enter" to apply / ?):

Name	Retained Size	Shallow Size
[Unreachable] java.lang.String[40]	176	176
java.lang.String[27] [Stack Local]	128	128

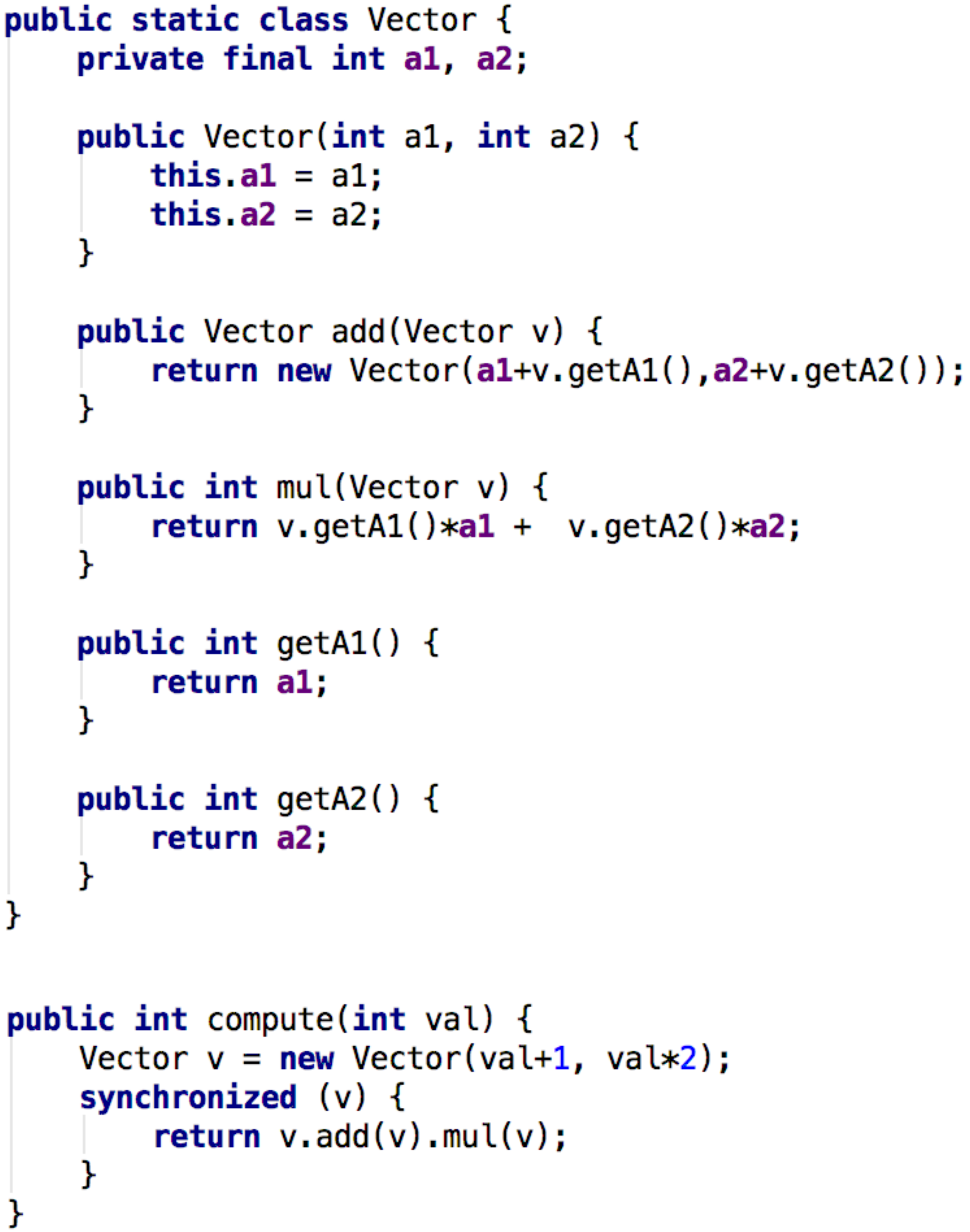
Know Your Application Behavior

- simple code could be very inefficient – **know what you are using**
- a lot of **small short-lived objects** still slow down your application
  - allocations in TLAB are quite fast but not as fast as no allocation
    - check ***escape analysis*** or change your code
  - objects in TLAB fulfill cache **data locality** and are **NUMA** aligned
  - **no false sharing** between cores (data in cache line are just used by one CPU core)
  - increase pressure on young generation and thus minor GC
    - other objects are promoted earlier to old generation
    - increase number of major GC
- a lot of **long-lived objects** slow your application even more
  - each time all live objects have to be traversed
  - compacting GC have to copy objects
    - breaks original data locality
    - can imply false sharing between cores

Escape Analysis – Not All Objects Are Allocated

- **C2 compiler** perform **escape analysis** of new object *after inline of hot methods*
- each new object allocation is classified into one of the following types:
  - ***NoEscape*** – object does not escape method in which it is created
    - all its usages are inlined
    - never assigned to static or object field, just to local variables
    - at any point must be JIT-time determinable and not depending on any unpredictable control flow
    - if the object is an **array**, indexing into it must be JIT-time constant
  - ***ArgEscape*** – object is passed as, or referenced from, an argument to a method but does not escape the current thread
  - ***GlobalEscape*** – object is accessed by different method and thread
- *NoEscape* objects are **not allocated at all** but JIT does **scalar replacement**
  - object deconstructed into its constituent fields (stack allocated)
  - disappear automatically after stack frame pop (return from the method)
  - no GC impact at all + do not need track references (write comp. barrier)
- *ArgEscape* objects are allocated on the heap but all monitors are eliminated

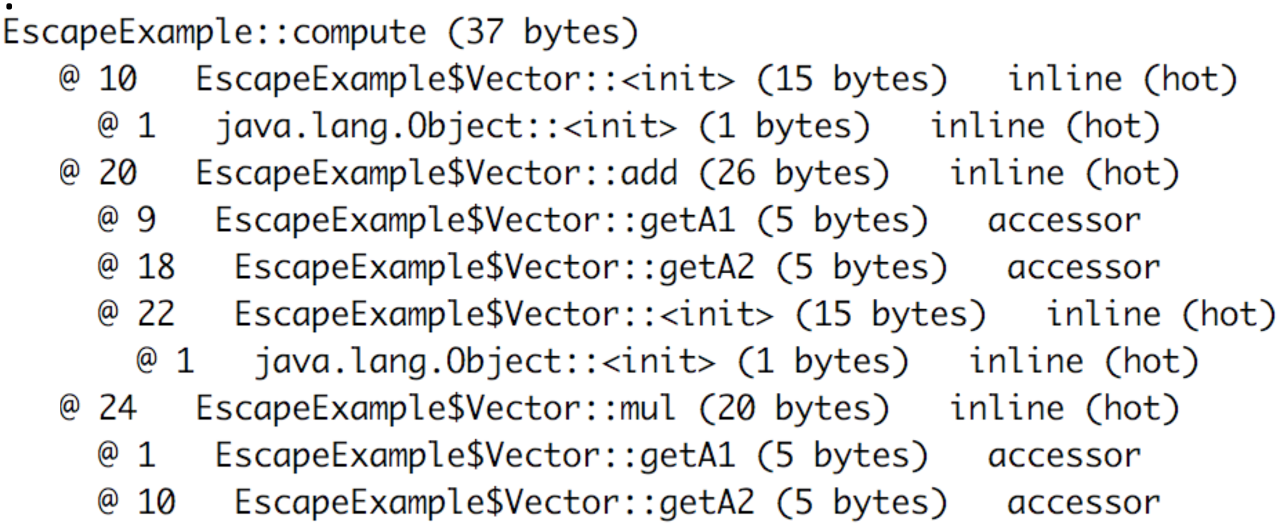
Escape Analysis Example



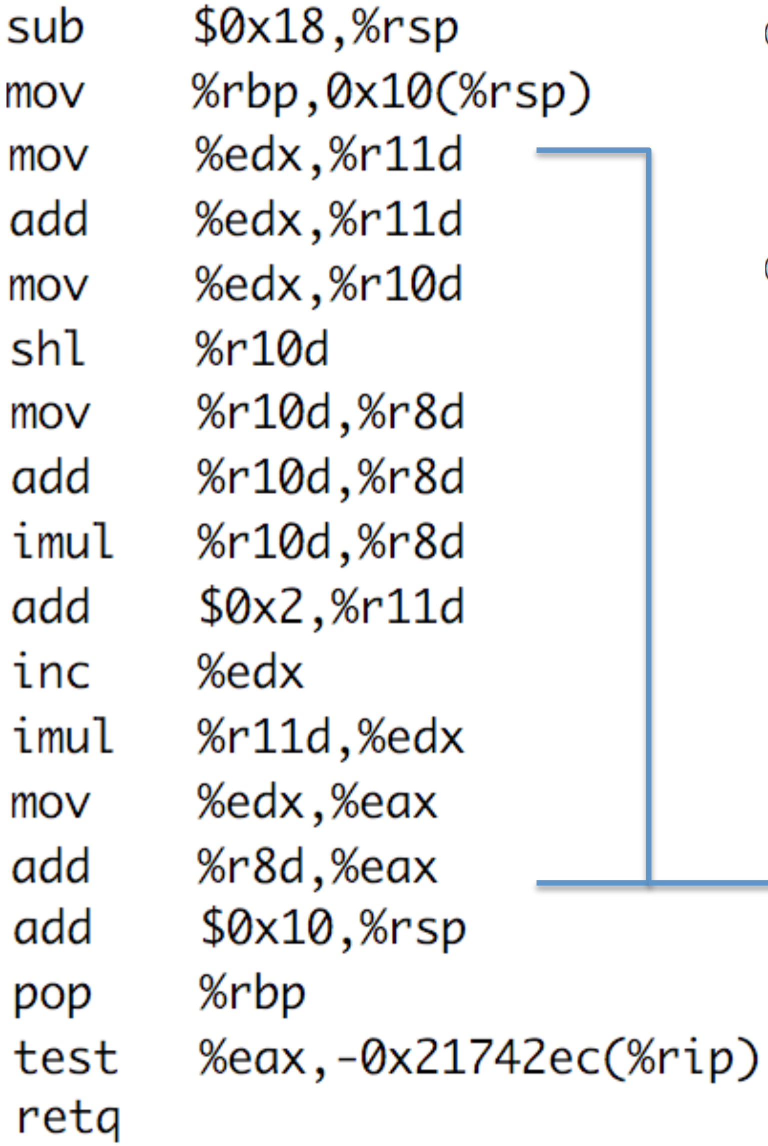
```
public static class Vector {
    private final int a1, a2;

    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

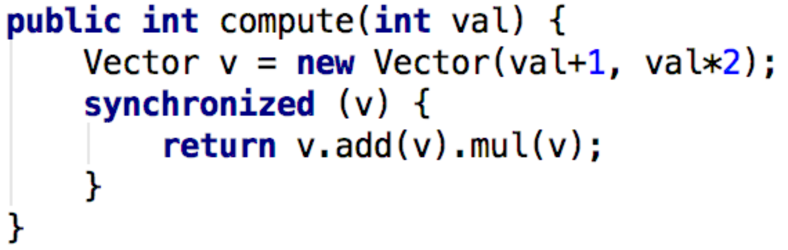
C2 compilation with inline:



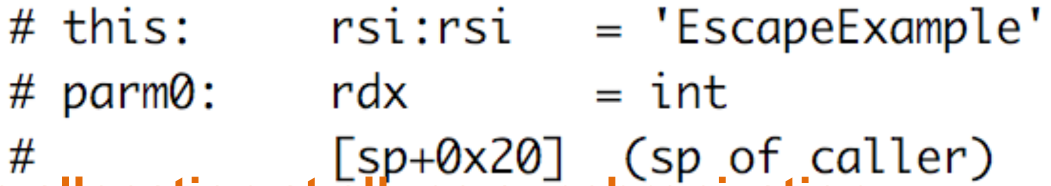
```
     74   22   !   4       EscapeExample::compute (37 bytes)
                             @ 10   EscapeExample$Vector::<init> (15 bytes)   inline (hot)
                               @ 1   java.lang.Object::<init> (1 bytes)   inline (hot)
                             @ 20   EscapeExample$Vector::add (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
```



```java
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)
```

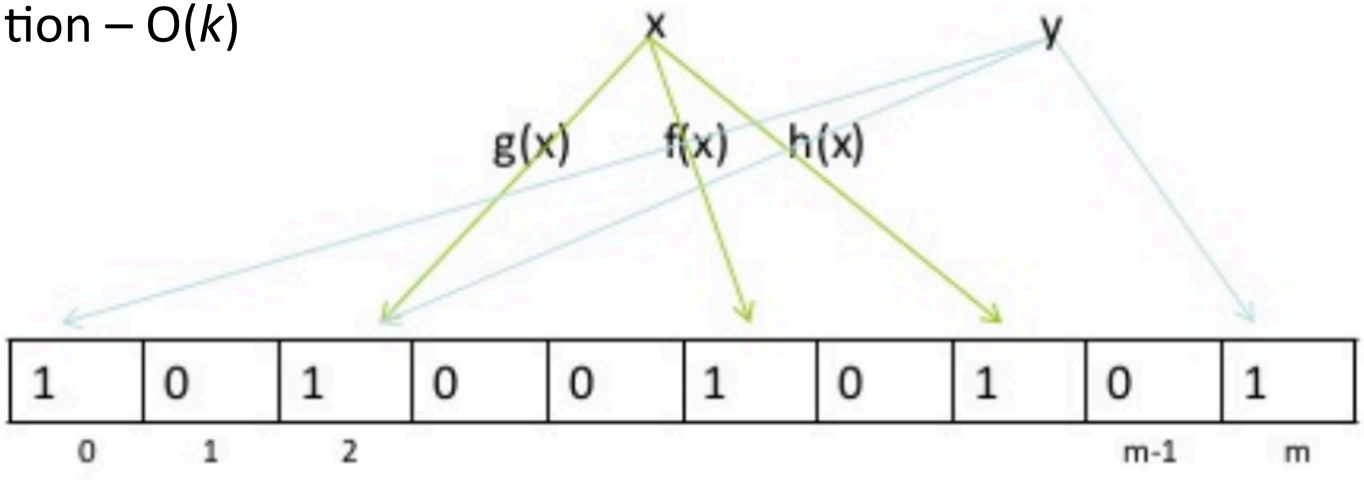
no allocation at all, no synchronization
all done out of stack in registers only

Bloom Filter

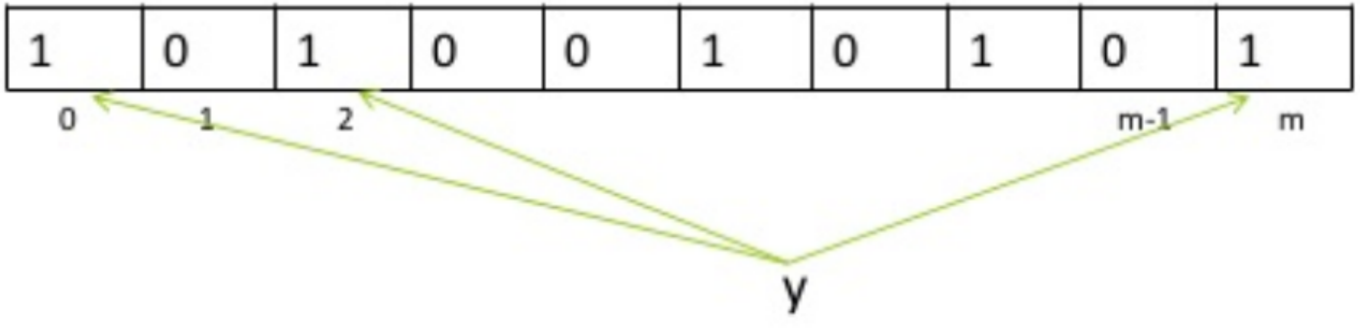
- **bloom filter** operations
    - add a new object to the set
    - test whether a given object is a member of the set
    - **no deletion** is possible
- **strong memory reduction** (few bits per element) compared to other collections
    - compensated by **small false positive** rate (usually 1%)
    - guaranteed **no false negative**
    - not storing object itself (where *all standard collections must store objects*)
- always **constant** add and test/query **complexity** (even for collisions)
- very useful in **big data** processing and other applications
    - used to test that the **object is certainly not present**
    - e.g. reduce a lot of I/O operations reading full collections in a particular file where bloom filters are kept in RAM or read quickly from disk

Bloom Filter

- use bit array with a *m* bits
- use *k* independent hash functions
- **add** operation – O(*k*)



- **query** operation



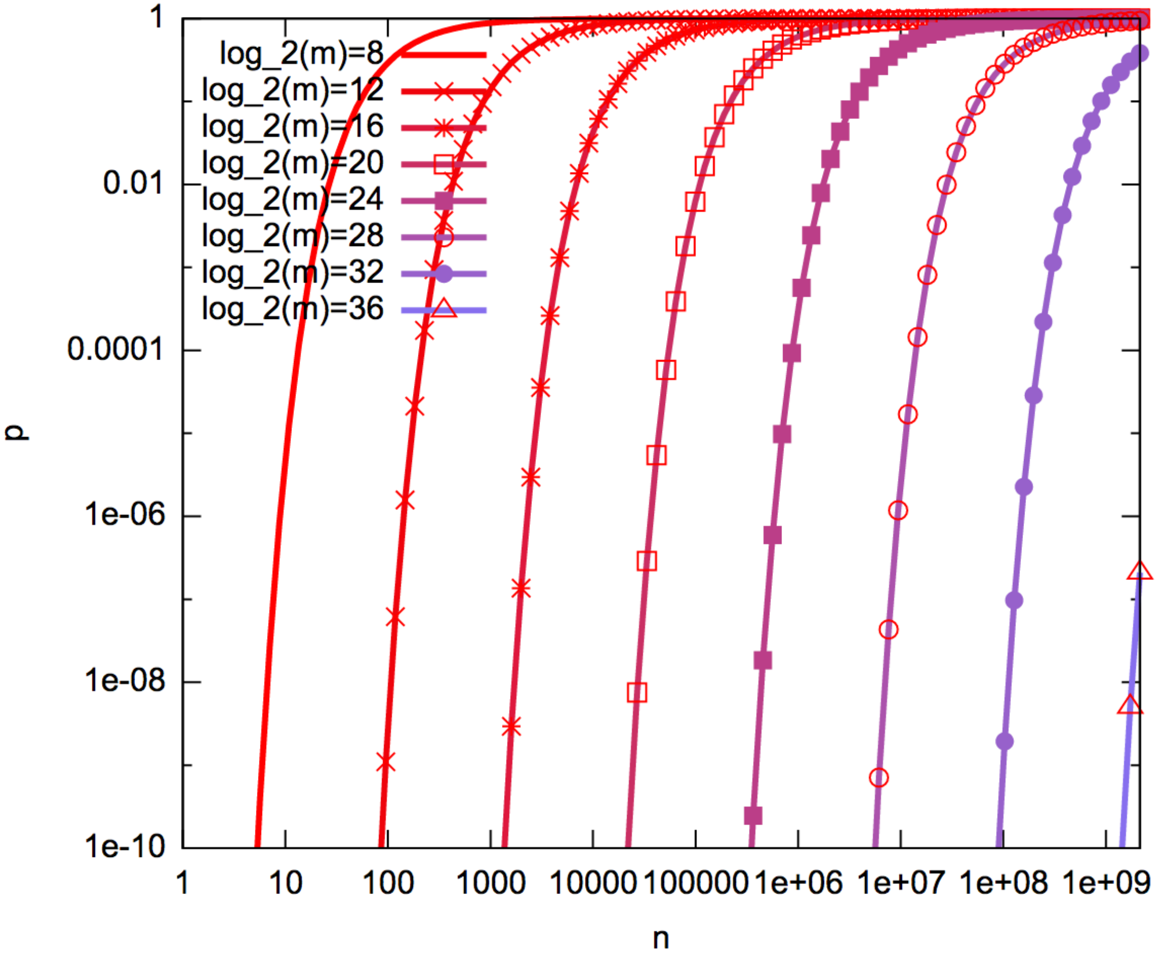
Bloom Filter

- 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

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

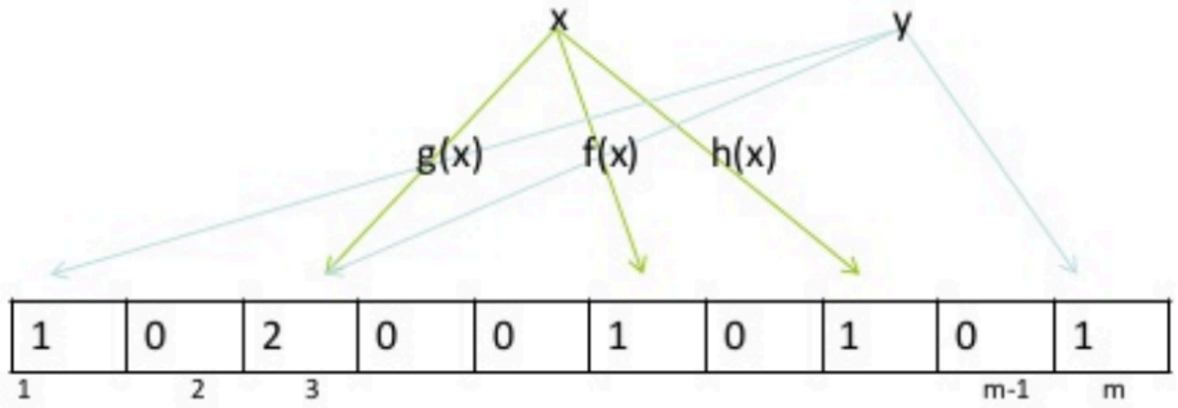


- example – store 1 million of Strings with total size 25 MB
  - Set<String> requires >50 MB retained size
  - Bloom Filter with FP rate 1% requires 1.13 MB and 7 hash functions
    - more than 44 times smaller and in 99% cases query is TP

Extensions of Bloom Filter

» **counting bloom filter**

- support **delete** and **count estimate** operation



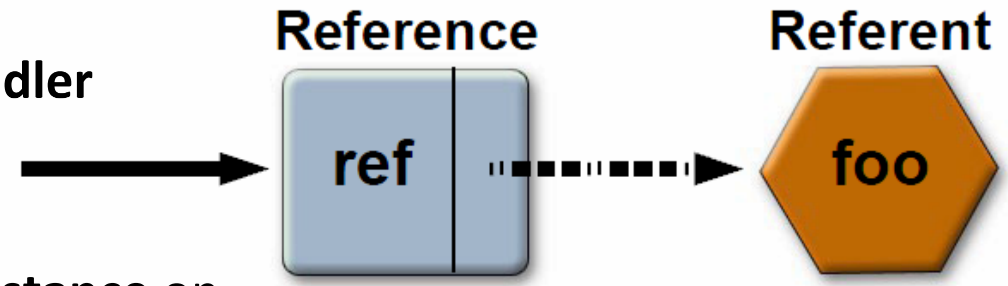
- each position in filter is buckets (e.g. 3 bits) working as counter
  - add – increment
  - delete – decrement; count is min value
  - query – test non-zero
- bucket overflow problem
  - no more increments when there is max counter value
  - increasing FN errors by deletions of elements

» **bitwise bloom filter**

- multiple counting (dynamically added) filters to address issues above

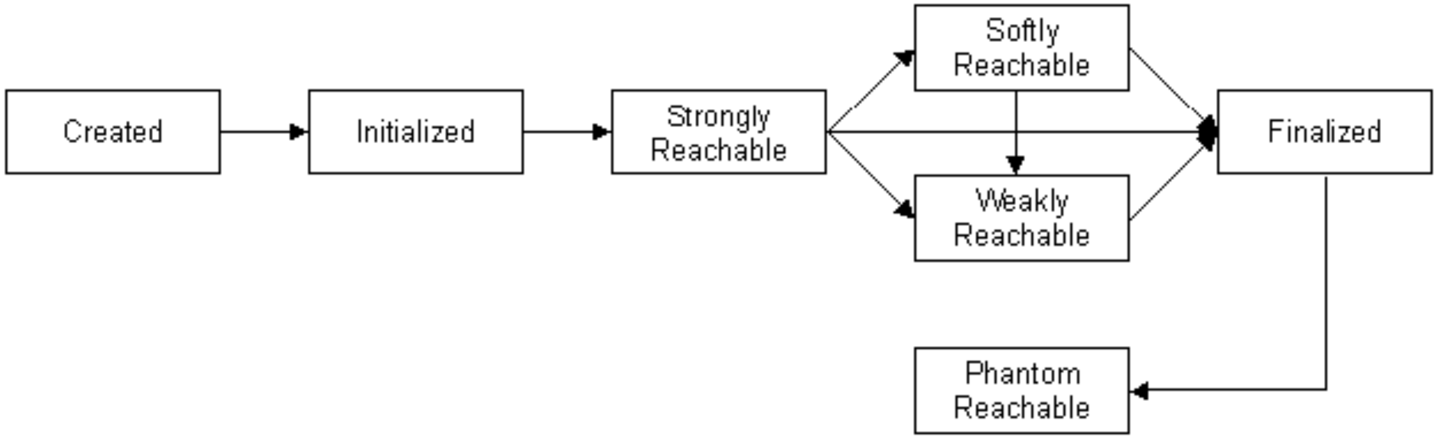
Reference Objects

- mortem hooks more **flexible** than finalization
- reference types (ordered from strongest one):
  - {strong reference}
  - soft reference – optional reference queue
  - weak reference – optional reference queue
  - {final reference} – mandatory reference queue
  - phantom references – mandatory reference queue
- can enqueue the reference object on a designated **reference queue** when GC finds its referent to be less reachable, referent is released
- references are enqueued **only if you have strong reference to REFERENCE**
- GC has to run to pass them to **Reference Handler** to enqueue them into **reference queue**
- Reference is **another instance** on the heap – 48 Bytes for standard OOP, 64-bit JVM



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

Reachability of Object



- **strongly reachable** – from GC roots without any Reference object
- **softly reachable** – not strongly, but can be reached via soft reference
- **weakly reachable** – not strongly, not softly, but can be reached via weak reference; clear referent link and become eligible for finalization
- **eligible for finalization** – not strongly, not softly, not weakly and have non-trivial finalize method
- **phantom reachable** – not strongly, not softly, not weakly, already finalized or no finalize method, but can be reached via phantom reference
- **unreachable** – none of above; eligible for reclamation

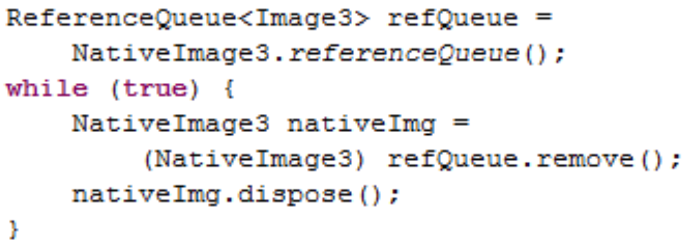
Weak Reference

- pre-finalization processing
- usage:
  - **do not retain this object because of this reference**
  - don't own target, e.g. listeners
  - canonicalizing map – e.g. ObjectOutputStream
  - implement **flexible version of finalization**:
    - prioritize
    - decide when to run finalization
- get() returns
  - referent if not cleared
  - null, otherwise
- **referent is cleared** by GC (cleared when passed to Reference Handler) and **can be reclaimed**
- need **copy referent to strong reference and check that it is not null before using it**
- WeakHashMap<K,V> - uses weak keys; cleanup during all standard operations

Weak Reference – External Resource Clean-up

» **clean-up** approach for ReferenceQueue<T>

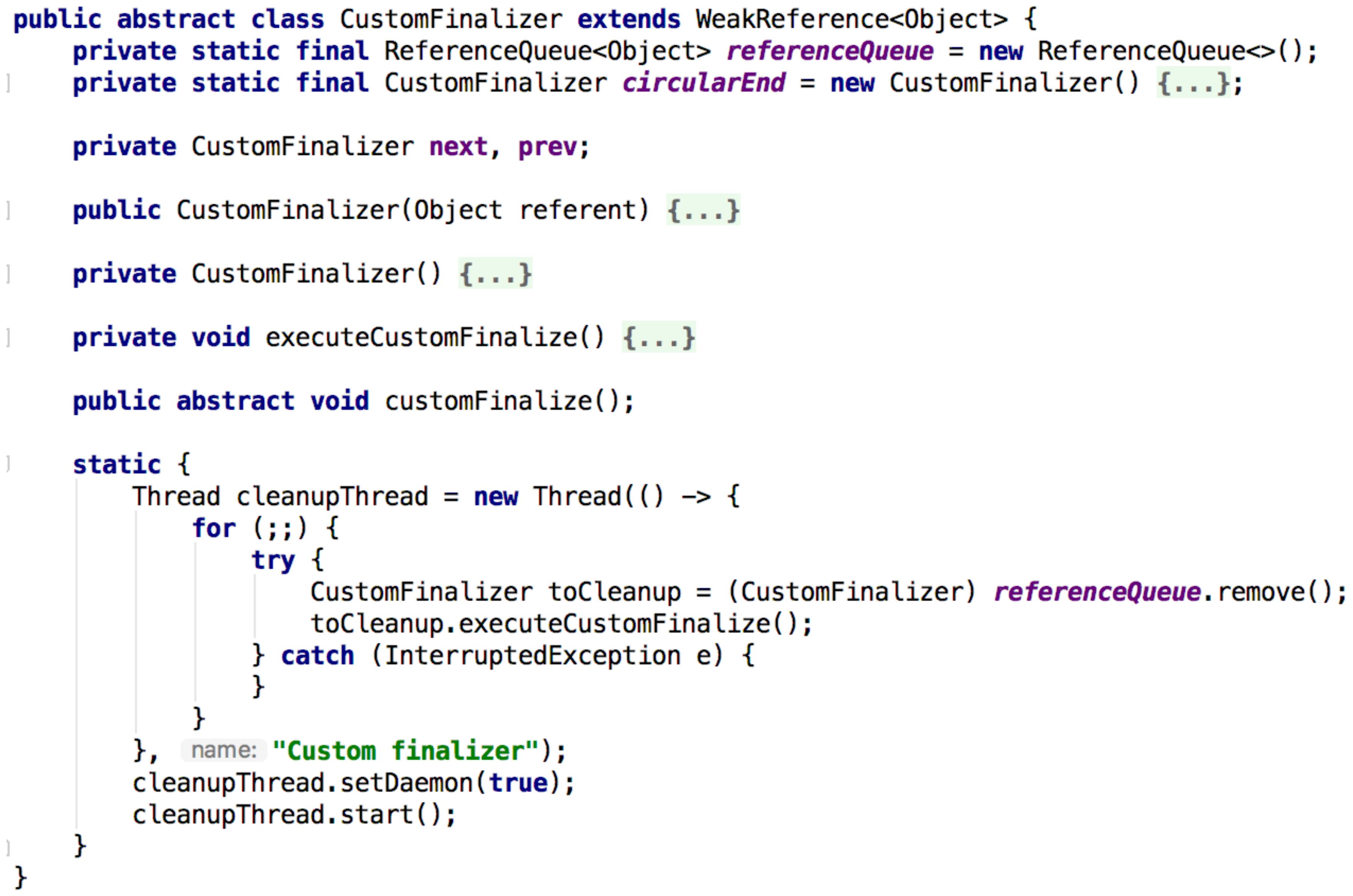
- own **dedicated thread**



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

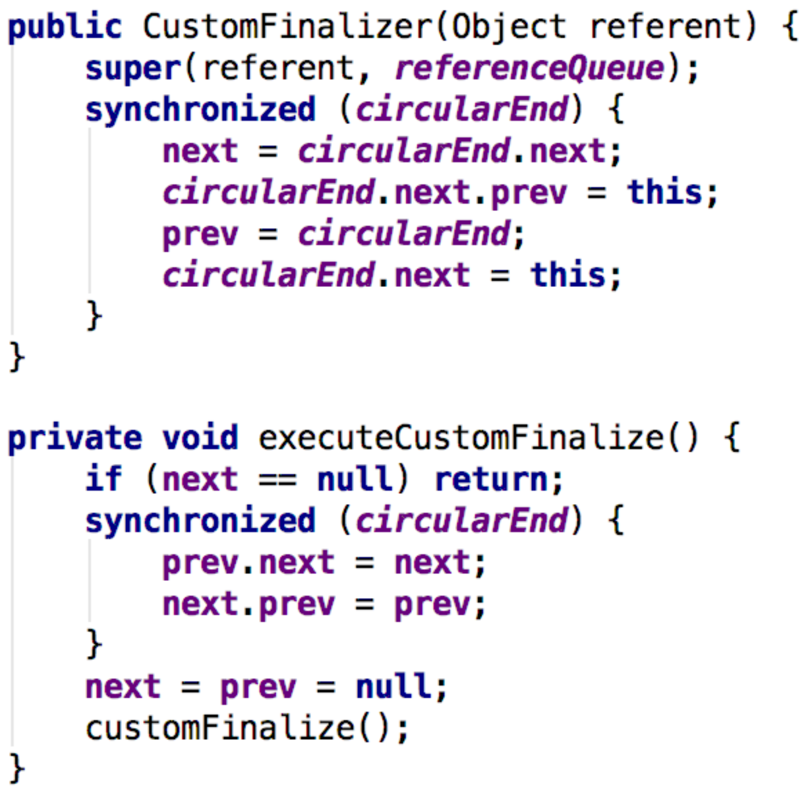
- clean-up **before creation of new** objects
  - limited clean-up processing to mitigate long processing
  - use poll() – non-blocking fetch of first

Custom Finalizer Example



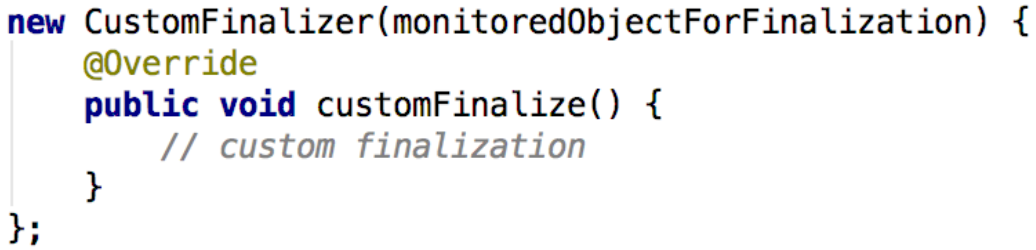
```
public abstract class CustomFinalizer extends WeakReference<Object> {
    private static final ReferenceQueue<Object> referenceQueue = new ReferenceQueue<>();
    private static final CustomFinalizer circularEnd = new CustomFinalizer() {...};

    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 implicit this strong reference** in instance context



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

Soft Reference

- pre-finalization processing
- usage:
  - **would like to keep referent, but can loose it**
  - suitable for **caches** – create strong reference to data to keep them
    - objects with long initialization
    - frequently used information
  - reclaim only if there is “**memory pressure**” based on heap usage

now – timestamp > (SoftRefLRUPolicyMSPerMB * amountOfFreeMemoryInMB)

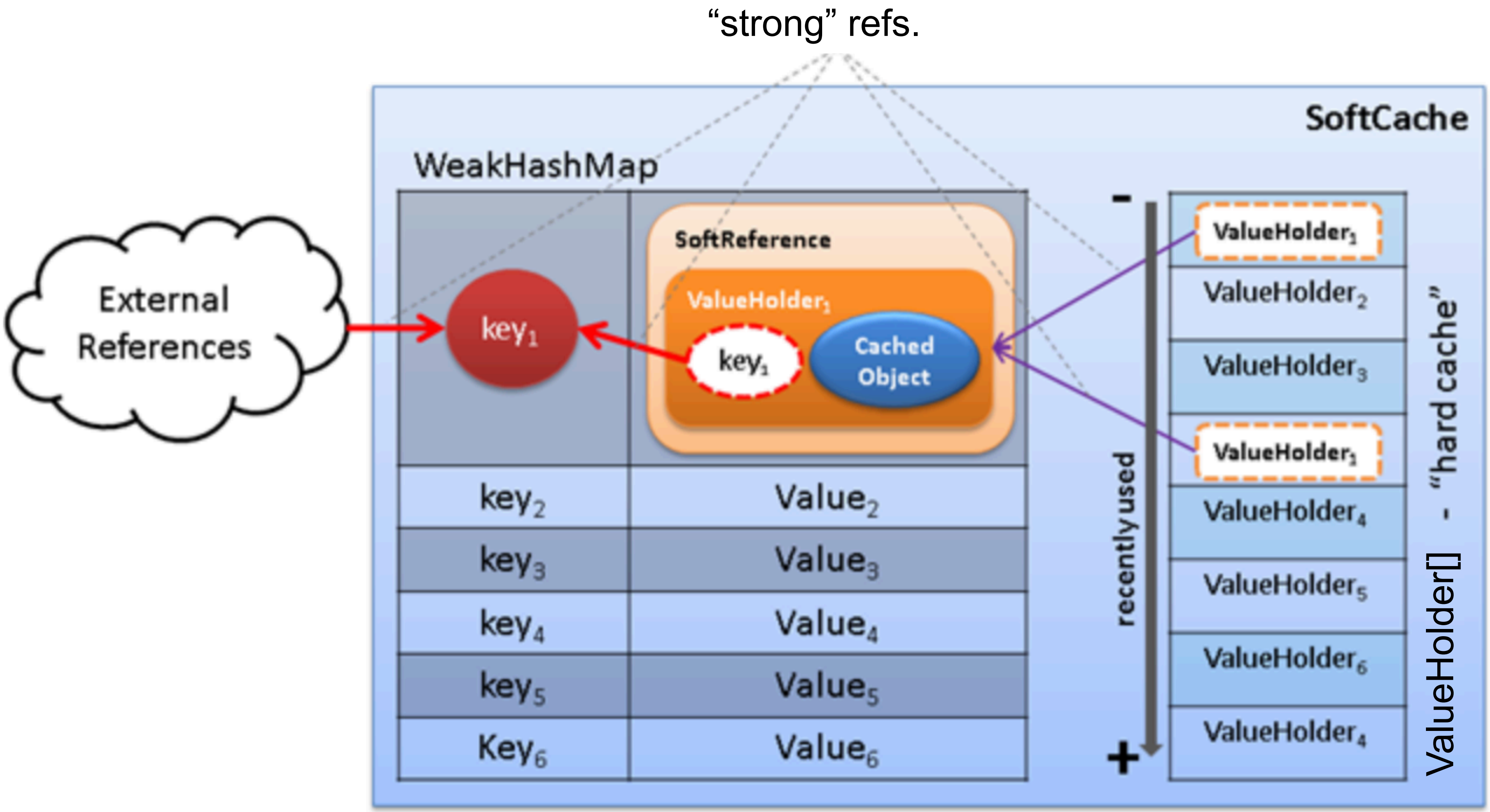
`-XX:SoftRefLRUPolicyMSPerMB=N` (default 1000)

  - all are cleared before OutOfMemoryError
- get() returns:
  - referent if not cleared; null, otherwise
  - **updates timestamp** of usage (can keep recently used longer)

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

Efficient Cache Example

efficient **LRU** tracking in combination with memory pressure for older

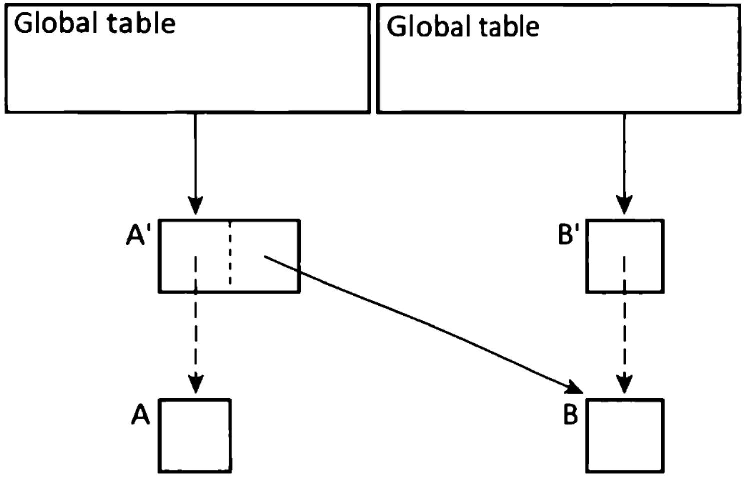


Final Reference – Object with Non-Trivial Finalize

- finalize hook cannot be used directly (package limited)
- instance allocation of object with **non-trivial finalize method**
  - slower allocation than standard objects
  - run time call of **Finalizer.register** with possible global safe point
    - not inlined, all references saved in stack with OopMap
  - allocates **FinalReference** instance and do synchronized tracking
- **referent is not cleared and reclaimed** before finalization
  - **all referenced objects** cannot be reclaimed as well
- only one **Finalizer** thread for all Final references of all types
  - call **finalize** method and **clear** referent
    - **issue** when finalize creates **strong reference again**
    - no priority control between multiple finalize methods
    - long running finalize delays all other finalization
  - daemon thread and JVM can terminate before finalization of all
- finalized objects can be reclaimed during **subsequent GC cycle**

Phantom Reference

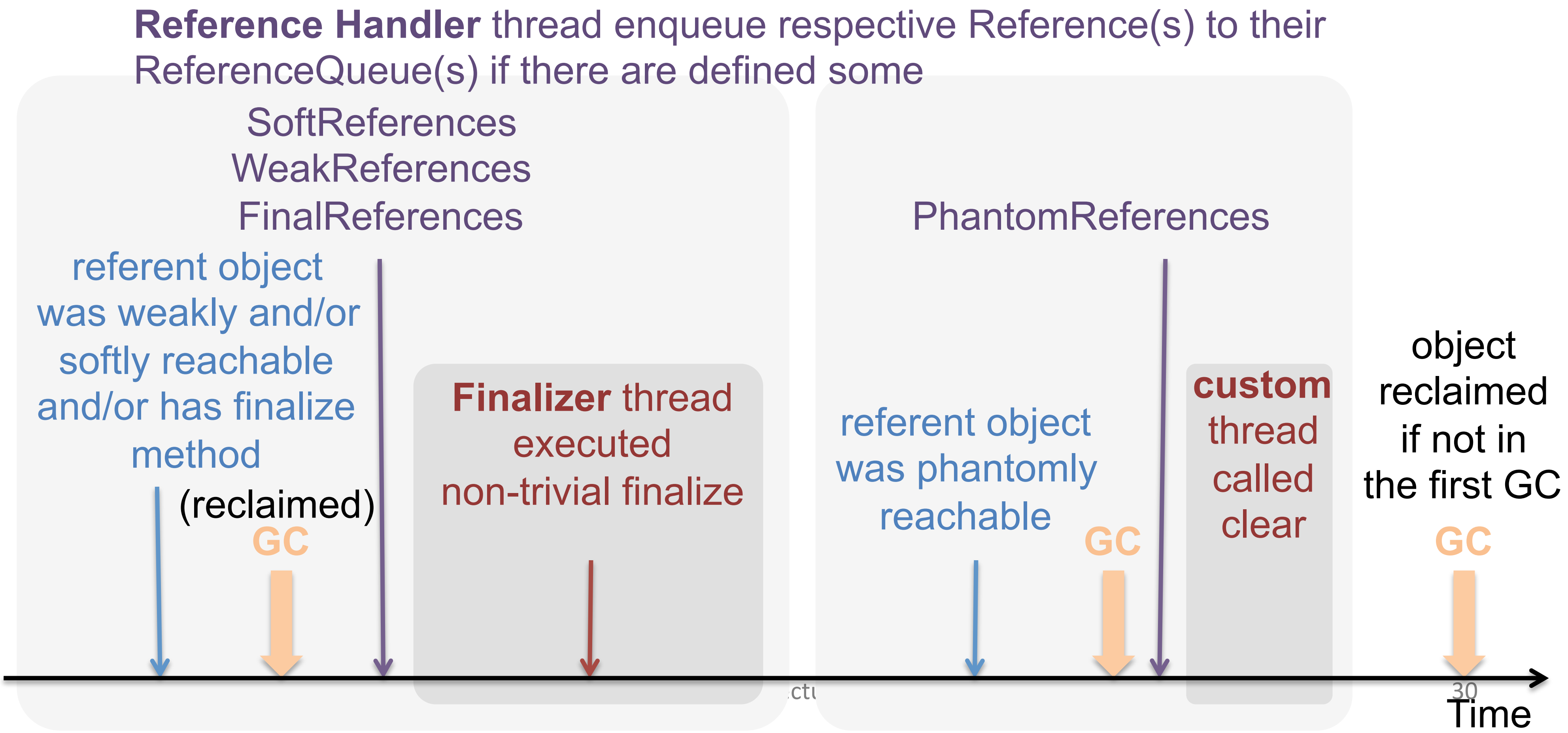
- » post-finalization processing, pre-mortem hook
- » usage:
  - **notifies that the object is not used** – before its reclamation
  - used to guarantee given order of finalization of objects (not possible with Weak references)
    - A, B – finalizable objects (e.g. Weakly)
    - A', B' - PhantomReferences



- » get() returns:
  - null always
  - referent can be read using reflection
    - avoid making strong reference again
- » **have to** specify reference queue for constructor (can be cleared)
- » **referent is not cleared and reclaimed** until all phantom references are not become unreachable or manually cleared using method clear()
  - » **all referenced objects** cannot be reclaimed as well

Reference Object

- only **one GC cycle** needed to reclaim *referent object* if there is only soft references or weak references to the same object
- **multiple GC cycles** needed for *referent objects* with multiple reference types or have at least one final or phantom reference



Reference Handler thread enqueue respective Reference(s) to their ReferenceQueue(s) if there are defined some

SoftReferences
WeakReferences
FinalReferences

referent object was weakly and/or softly reachable and/or has finalize method

(reclaimed)

GC

Finalizer thread executed non-trivial finalize

PhantomReferences

referent object was phantomly reachable

GC

custom thread called clear

object reclaimed if not in the first GC

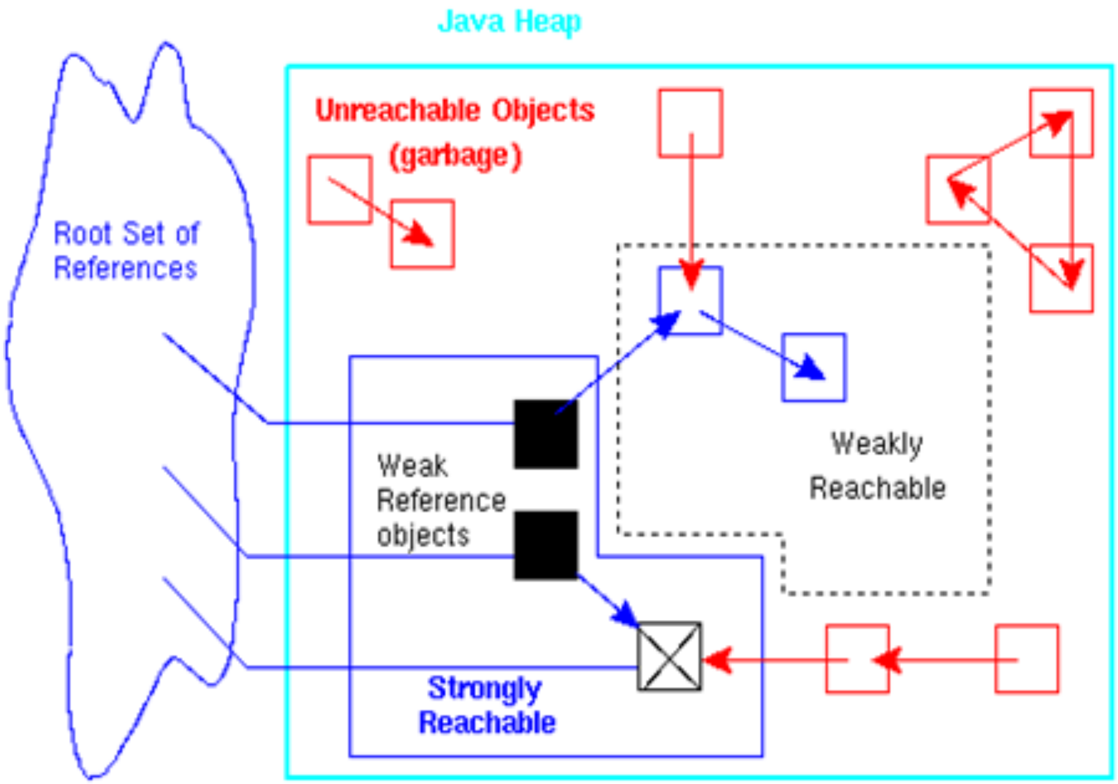
GC

Time

Performance Cost for References

- **creation** cost
  - allocation instance
  - synchronization with tracking of Reference (strong references)
- **garbage collection** cost (-XX:+PrintReferenceGC –XX:+PrintGCDetails)
  - tracking live not follow referents
  - construct list of live References each GC cycle
    - discovered field in Reference
  - per-reference traversal overhead regardless referent is collected or not
    - softly, weakly + finalizable, phantomly
  - Reference Object itself are subject for garbage collection
- **enqueue** cost
  - reference handler enqueue with synchronization
- **reference queue processing** cost
  - synchronized queue consumption

Reachability of Object



Reachability of Object