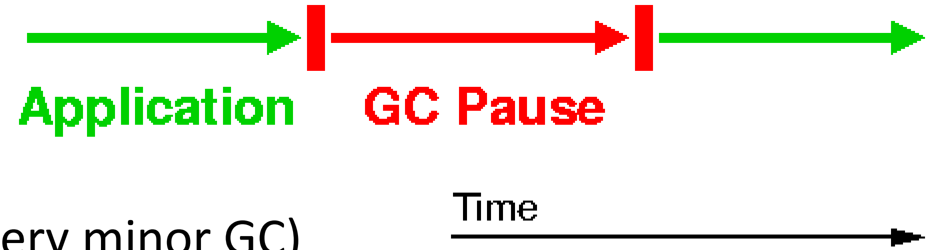
Architecture of software systems

Course 12: Memory management with garbage collector, references

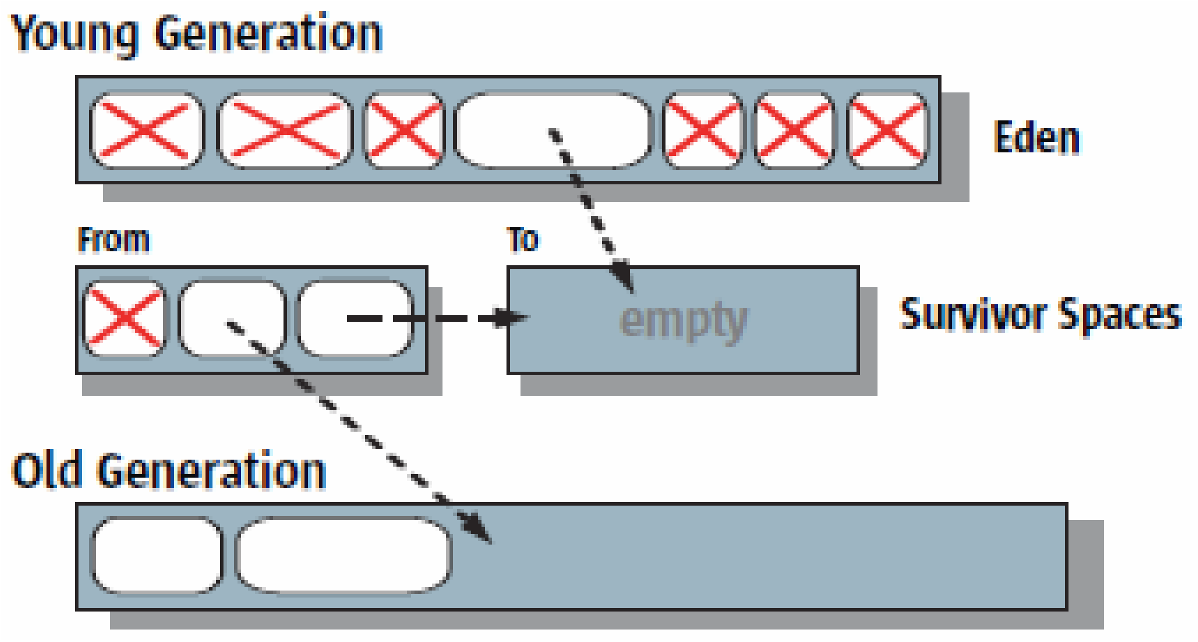
David Šišlák
david.sislak@fel.cvut.cz

Serial collector

- young collection -> old generations collection serially in *stop-the-world* fashion

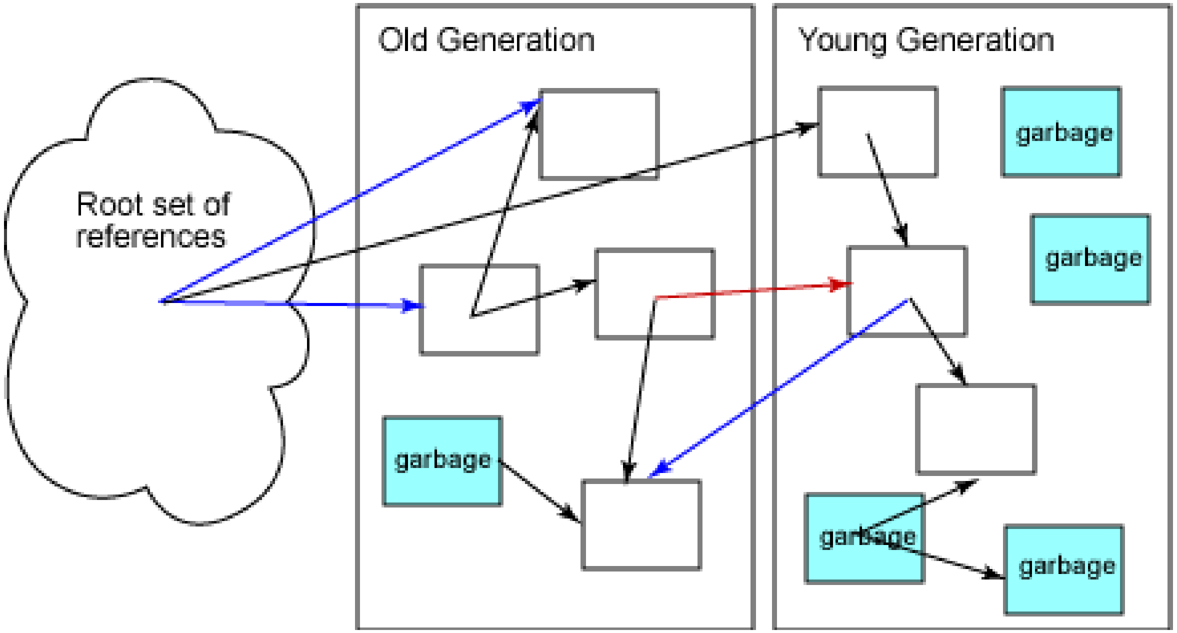


- young generation:
  - age of object (incremented every minor GC)
  - efficiency is proportional to number of copied objects !



Young generation live object detection – IBM version

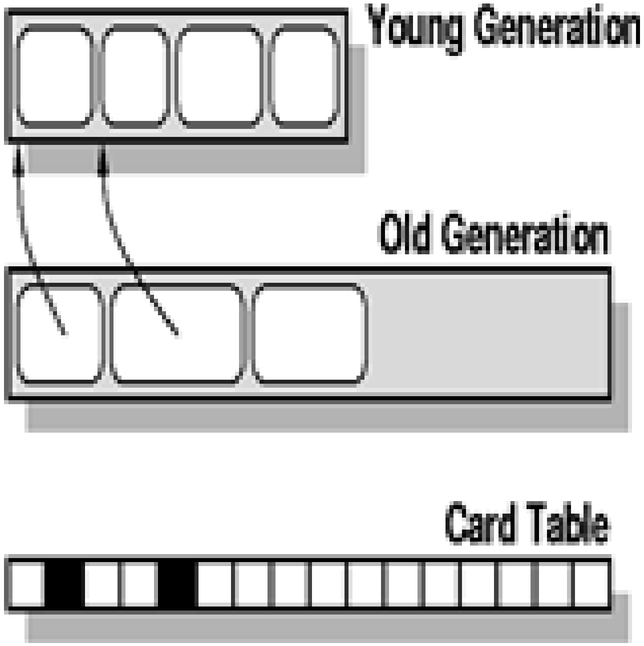
- maintains separate list of old-to-young references as they are created
- maintain the list during object promotion, introduce new, remove old



red – old-to-young, blue – to old (don't need trace during minor collection)

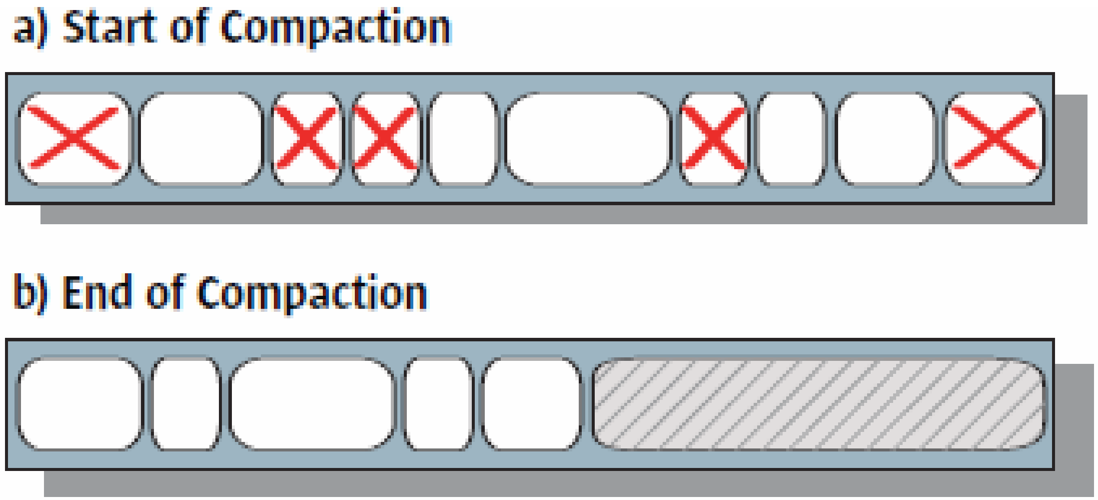
Young generation live object detection – Sun version

- identification of live objects based on card table structure (boolean)
- 512-byte chunks in old generation (smaller than memory page)
- every update to a reference marks dirty
- bytecode interpreter and JIT uses reference write barrier to maintain card table
- only dirty cards are scanned for old-to-young references
- finally marks are cleared



Serial collector

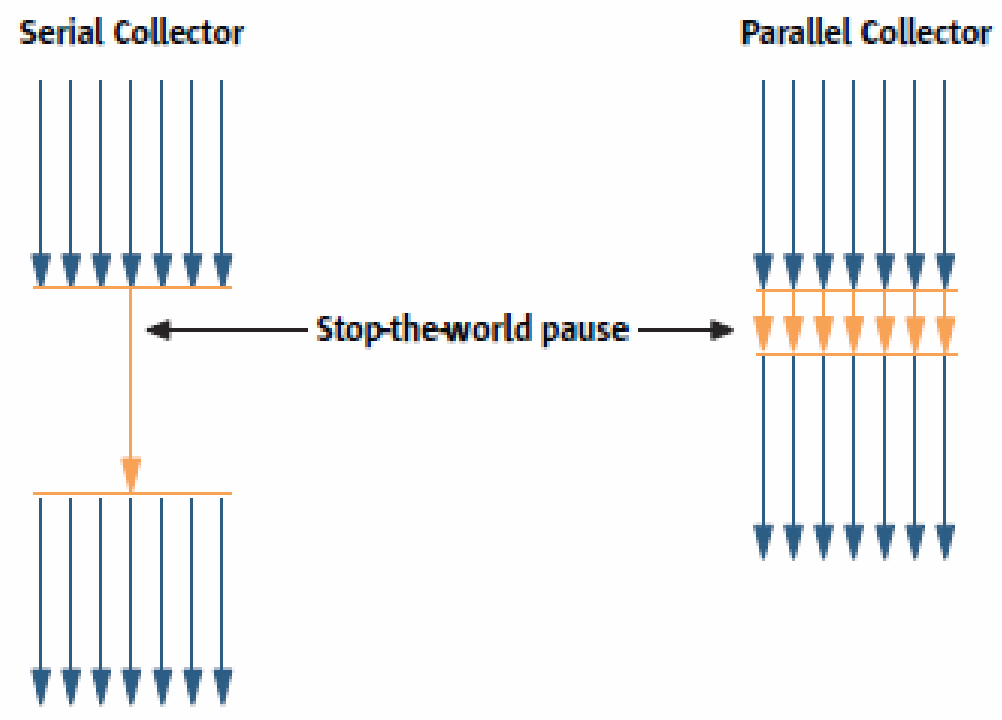
- old and permanent generation:
  - using *mark-sweep-compact* algorithm
  - allocation can use *bump-the-pointer* technique



- default in Java 5.0 for client JVM
- effectively handles application with 64MB heaps
- `-XX:+UseSerialGC`

Parallel collector

- utilize more cores/CPUs
- still stop-the-world but in parallel manner for young generation



- **uses the same serial *mark-sweep-compact* algorithm for old generation**
- default for server JVM from Java 5.0
- `-XX:+UseParallelGC`

Parallel compacting collector

- » introduced in J2SE 5.0 update 6
- » no change in young generation collection – use parallel one
- » old and permanent generations:
  - done in stop-the-world manner
  - each generation logically divided into fixed-sized regions
  - ***parallel mark*** phase:
    - initiated by divided reachable live objects
    - info about live objects (size & location) are propagated to the corresponding region data

Parallel compacting collector

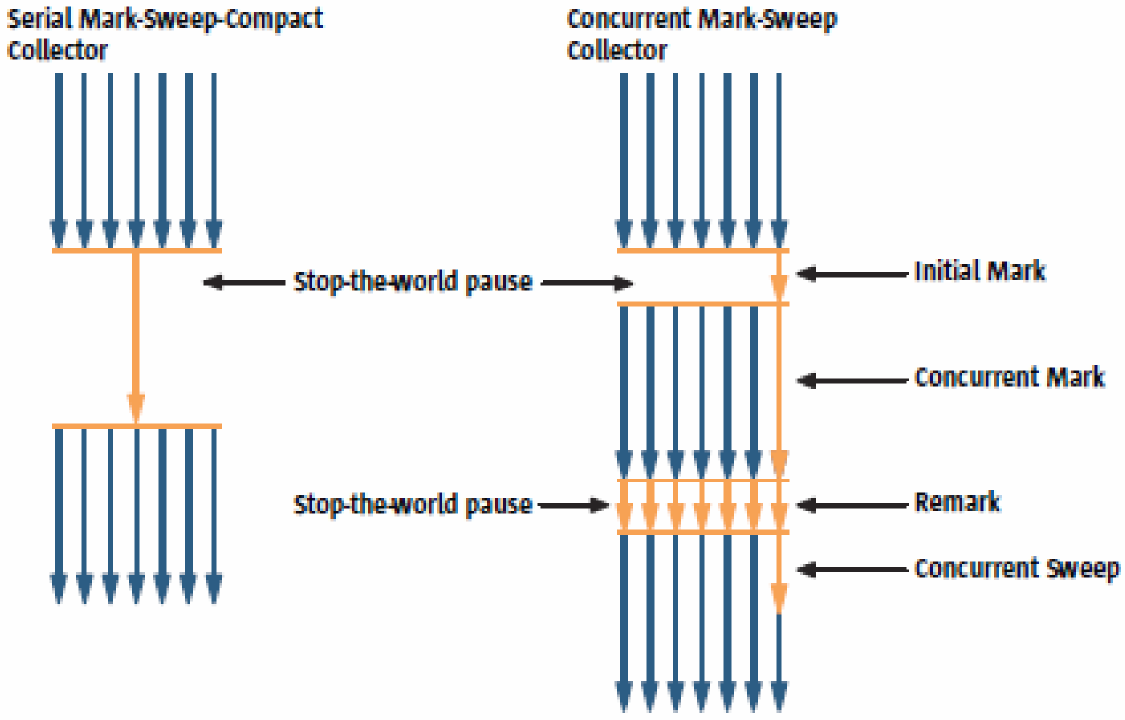
- ***summary phase*** (implemented in serial):
  - identify density of regions (due to previous compactions, more dense are at the beginning)
  - find from which region it has sense to do compaction regarding recovered from a region:
    - *dense prefix* – before, no movement
  - calculate new location of each live data for each region
- ***compaction*** phase:
  - parallel copy of data based on the summary data
  - finally heap is packed and large empty block is at the end

» `-XX:+UseParallelOldGC, -XX:ParallelGCThreads=n`

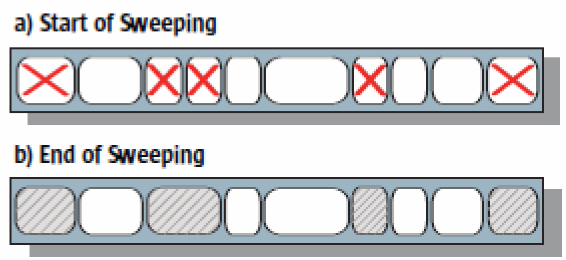
» default in J2SE 6.0 for multi core/CPU systems

Concurrent mark-sweep (CMS) collector

- low-latency collector
- use the same parallelized young generation collector
- old generation:
  - done concurrently with the application execution
  - *initial mark* – short pause identifying the initial set of live objects directly reachable
  - *remark* – revisiting modified objects (overhead)
  - *concurrent sweep*



Concurrent mark-sweep (CMS) collector



- non-compacting
- cannot use bump-the-pointer
- more expensive allocation searching a region
- extra overhead to young generation collection doing promotions
- may split or join free block depending on tracked popular object sizes
- collector started:
  - adaptively based on previous runs (how long it takes, how many is free)
  - initiating occupancy in percentage

    `-XX:CMSInitiatingOccupancyFraction=n`

    default 68
- decreases pauses
- requires larger heap due to concurrent collection
- *incremental mode* – concurrent phases divided into small chunks between young generation collection
- `-XX:+UseConcMarkSweepGC` , `-XX:+CMSIncrementalMode`

Configure garbage collector

- explicit type:
  - `-XX:+UseSerialGC`, `-XX:+UseParallelGC`, `-XX:+UseParallelOldGC`, `-XX:+UseConcMarkSweepGC`
- statistics:
  - `-XX:+PrintGC`, `-XX:+PrintGCDetails`, `-XX:+PrintGCTimeStamps`, `-XX:+PrintTenuringDistribution`
- heap sizing:
  - `-Xmx` – max heap size, default 64MB on client JVM, influence to throughput
  - `-Xms` - initial heap size
  - `-XX:MinHeapFreeRatio=min` – default 40, per generation
  - `-XX:MaxHeapFreeRatio=max` – default 70
  - `-XX:NewSize=n` - initial size of young generation
  - `-XX:MaxNewSize=n`

Configure garbage collector

- heap sizing cont.:
  - `-XX:NewRatio=n` - ratio between young and old gens
    default 2 client JVM, 8 server JVM (young includes survivor),
    n=2 => 1:2 => young is 1/3 of total heap
  - `-XX:SurvivorRatio=n` – ratio between each survivor and Eden
    default 32, n=32 => 1:32 => each survivor is 1/34 of young size
  - `-XX:MaxTenuringThreshold=<threshold>`
  - `-XX:PermSize=n` - initial size of permanent generation
  - `-XX:MaxPermSize=n` - max size of permanent generation
- parallel collector & parallel compacting collector:
  - `-XX:ParallelGCThreads=n` - number of GC threads
  - `-XX:MaxGCPauseMillis=n` - maximum pause time goal
  - `-XX:GCTimeRatio=n` - throughput goal
    1/(1-n) percentage of total time for GC, default n=99 (1%)

Configure garbage collector

» CMS collector:

- `-XX:+CMSIncrementalMode` – default disabled
- `-XX:ParallelGCThreads=n`
- `-XX:CMSInitiatingOccupancyFraction=<percent>`
- `-XX:+UseCMSInitiatingOccupancyOnly` - disable automatic initiating occupancy (auto ergonomics)
- `-XX:+CMSClassUnloadingEnabled` - by default disabled !!!
- `-XX:CMSInitiatingPermOccupancyFraction=<percent>` - unloading has to be enabled !!!
- `-XX:+ExplicitGCInvokesConcurrent`
- `-XX:+ExplicitGCInvokesConcurrentAndUnloadClasses`

  - both useful when want to references / finalizers to be processed

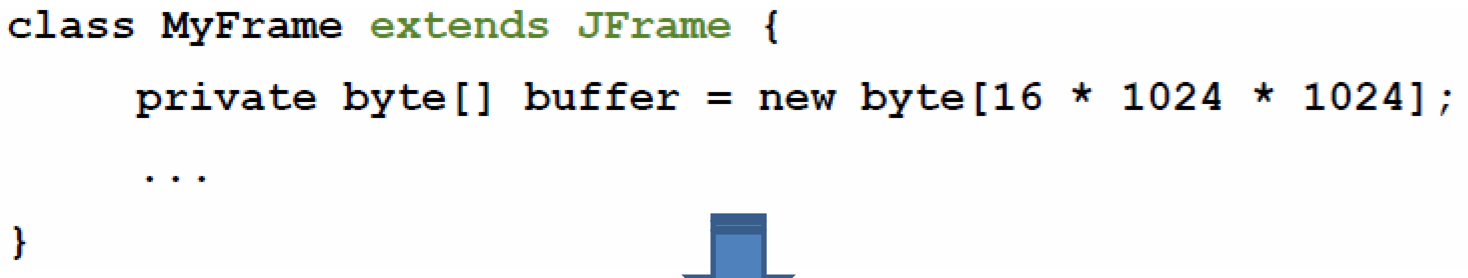
Performance recommendations

- » prefer short-lived immutable objects instead of long-lived mutable objects
- » avoid needless allocations
  - more frequent allocations will cause more frequent GCs
- » large objects:
  - expensive to allocate (not in TLAB, not in young)
  - expensive to initialize (zeroing)
  - can cause performance issues
  - fragmentation for CMS (non-compacting) GC
- » avoid force System.gc() except well-defined application phases
  - can be ignored by `-XX:+DisableExplicitGC`
- » avoid frequent array-based re-sizing
  - several allocations
  - a lot of array copying
  - use:

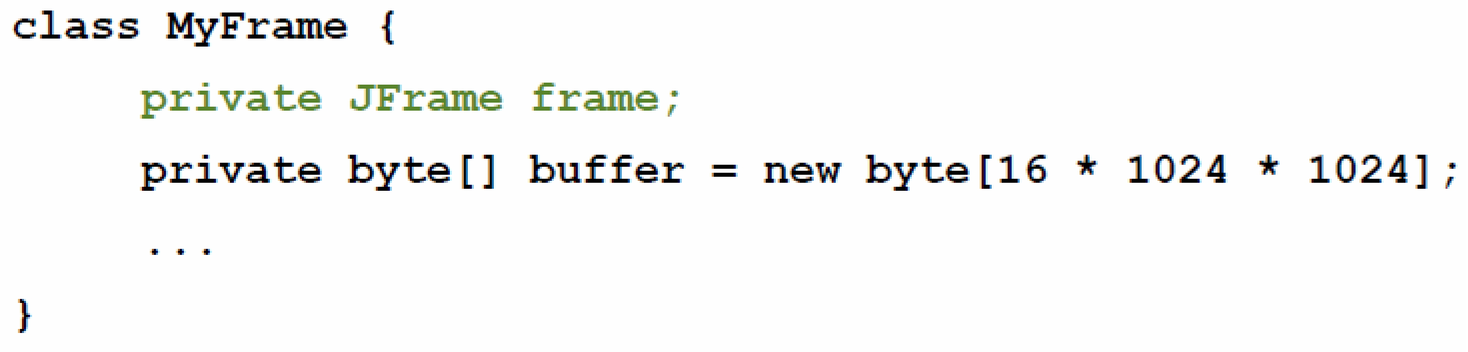
    ```
    ArrayList<String> list = new ArrayList<String>(1024);
    ```

Performance recommendations

» avoid finalizable objects (non-trivial finalize() method)
  - slower allocation due to their tracking
  - require at least two GC cycles:
    - enqueues object on finalization queue
    - reclaims space after finalize() completes
  - beware of extending objects which define finalizers

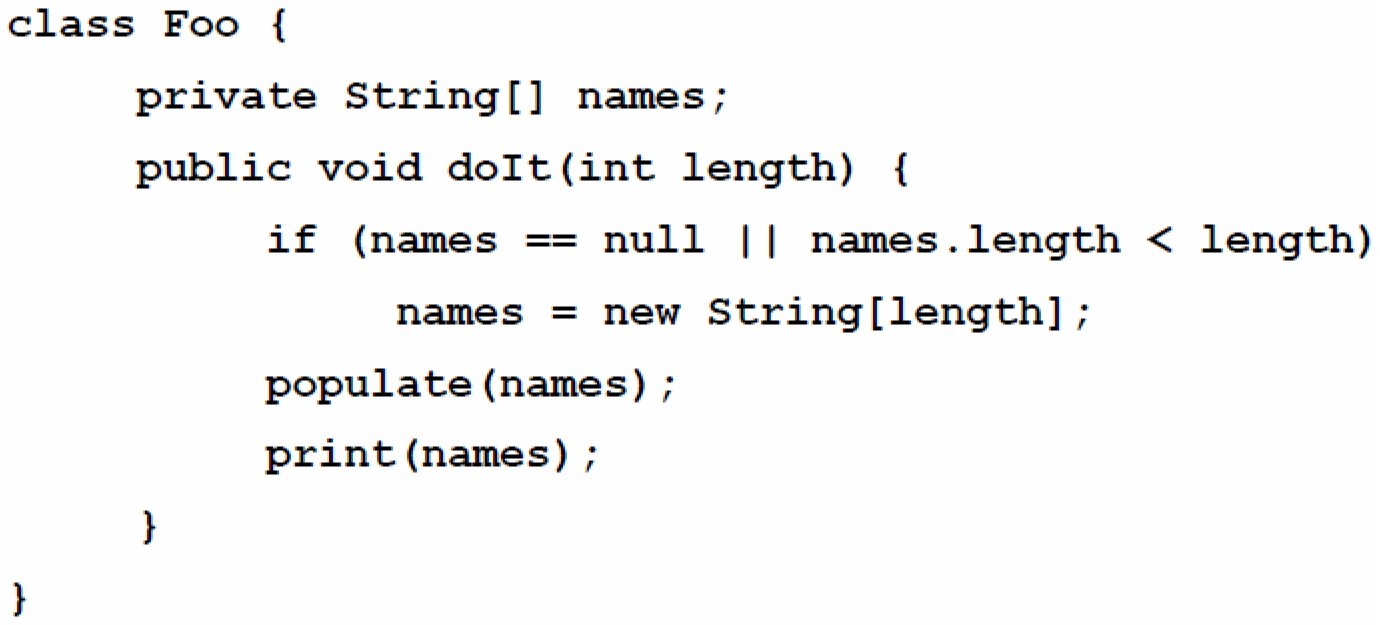


```
class MyFrame extends JFrame {
     private byte[] buffer = new byte[16 * 1024 * 1024];
     ...
}
```



```
class MyFrame {
     private JFrame frame;
     private byte[] buffer = new byte[16 * 1024 * 1024];
     ...
}
```

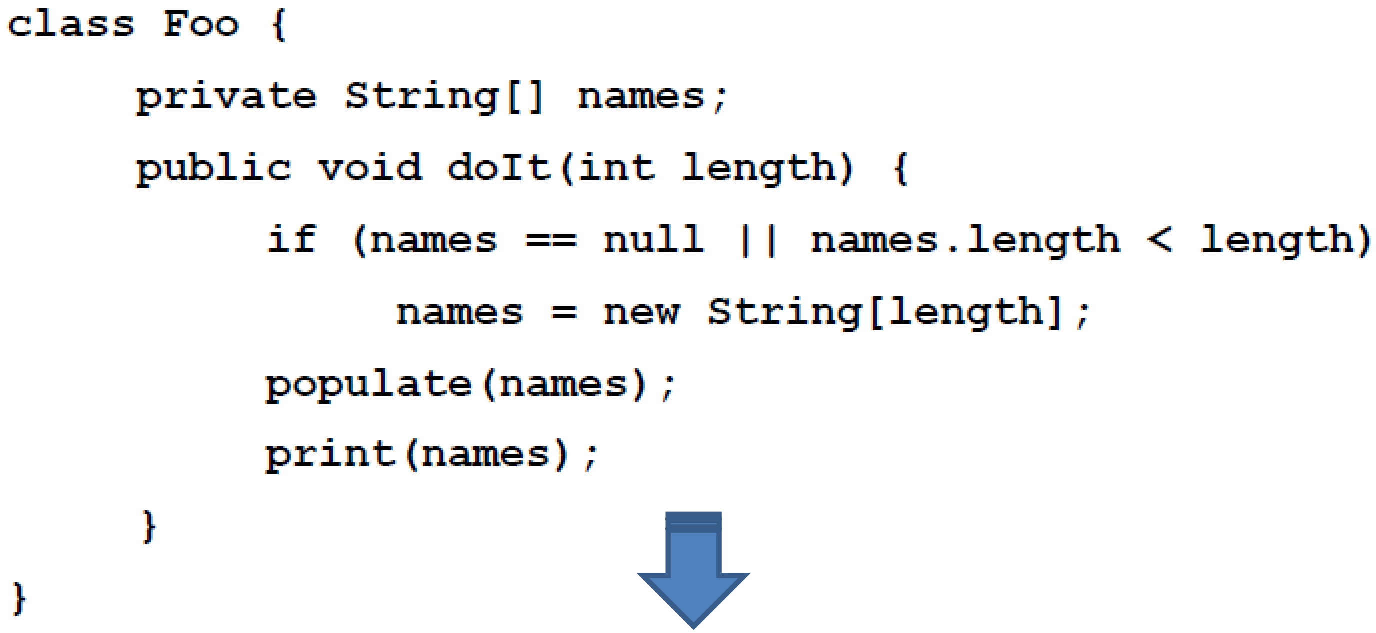
Performance recommendations



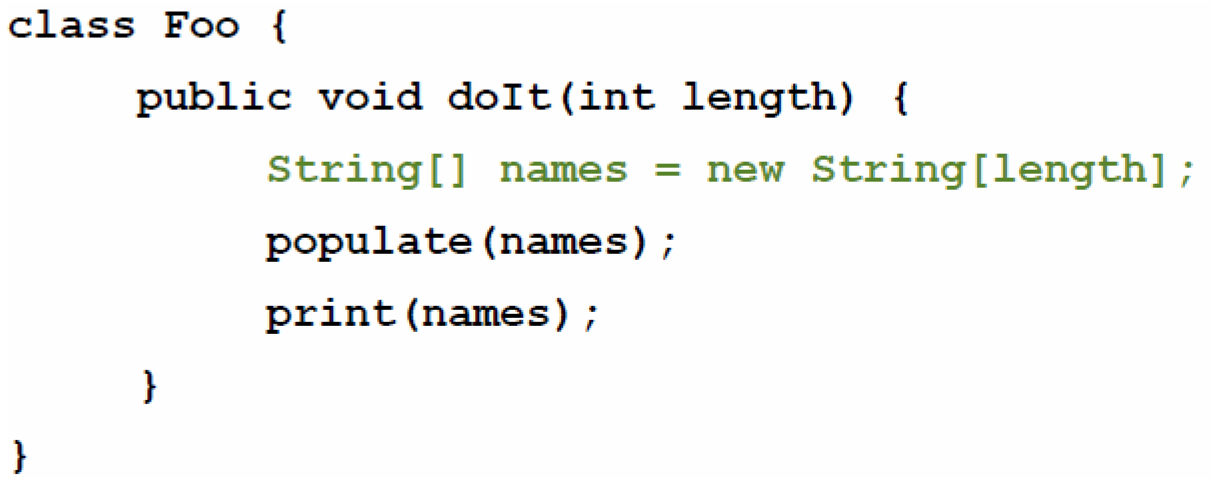
```
class Foo {
     private String[] names;
     public void doIt(int length) {
          if (names == null || names.length < length)
               names = new String[length];
          populate(names);
          print(names);
     }
}
```

Performance recommendations

- objects in the wrong scope



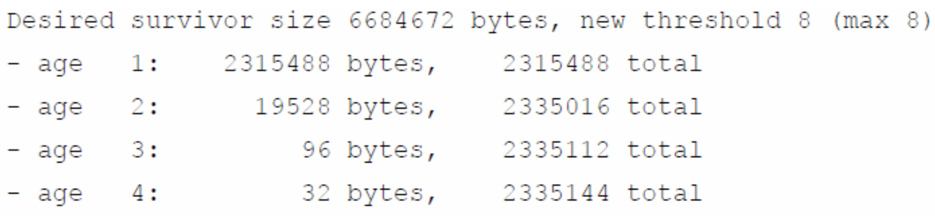
```
class Foo {
    private String[] names;
    public void doIt(int length) {
        if (names == null || names.length < length)
            names = new String[length];
        populate(names);
        print(names);
    }
}
```



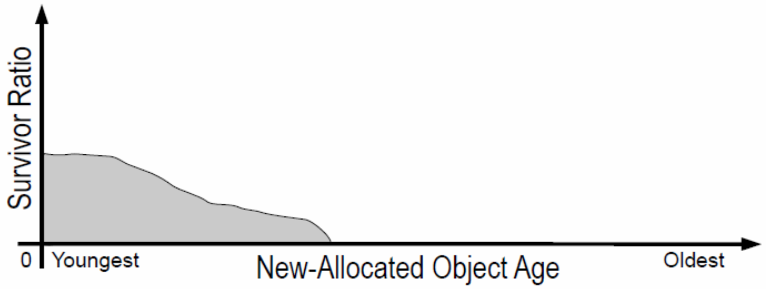
```
class Foo {
    public void doIt(int length) {
        String[] names = new String[length];
        populate(names);
        print(names);
    }
}
```

Performance recommendations

- instances of inner classes have an implicit reference to the outer instance
- larger heap space for both generations -> less frequent GCs, lower GC overhead, objects more likely to become dead (smaller heap -> fast collection)
- tune size of young generation -> implies frequency of minor GCs, maximize the number of objects released in young generation, it is better to copy more than promote more
- tune tenuring distribution (-XX:+PrintTenuringDistribution),



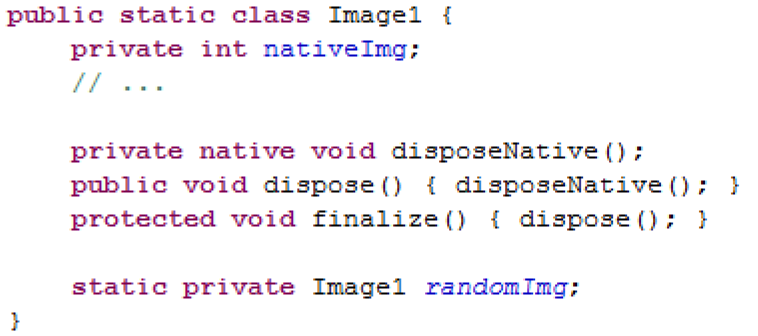
```
Desired survivor size 6684672 bytes, new threshold 8 (max 8)
- age   1:    2315488 bytes,    2315488 total
- age   2:      19528 bytes,    2335016 total
- age   3:         96 bytes,    2335112 total
- age   4:         32 bytes,    2335144 total
```



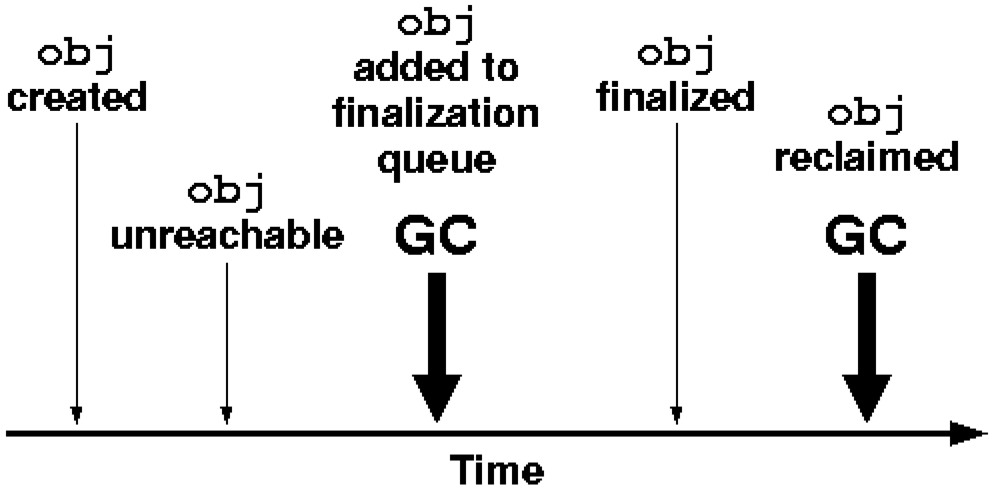
- overall application footprint should not exceed physical memory !
- different Xms and Xmx implies full GC during resizing (consider Xms=Xmx)

Finalizable objects

- have a non-trivial finalize() method
- postmortem hook
- used for clean-up for unreachable object, typically reclaim native resources:
  - GUI components
  - file
  - socket



```
public static class Image1 {
    private int nativeImg;
    // ...

    private native void disposeNative();
    public void dispose() { disposeNative(); }
    protected void finalize() { dispose(); }

    static private Image1 randomImg;
}
```

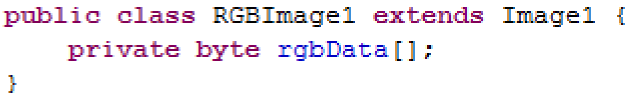


Finalizable objects

- finalizable object allocation:
  - much slower
  - VM must track finalizable objects
- finalizable object reclamation
  - at least two GC cycles:
    - identification and enqueue object on finalization queue
    - reclaim space after finalize()
- not guaranteed when finalize() is called, whether is called (can exit earlier) and the order in which it is called
- finalizable objects occupy memory longer along with ***everything reachable from them*** !!!
- implementation based on references (see Finalizer class)

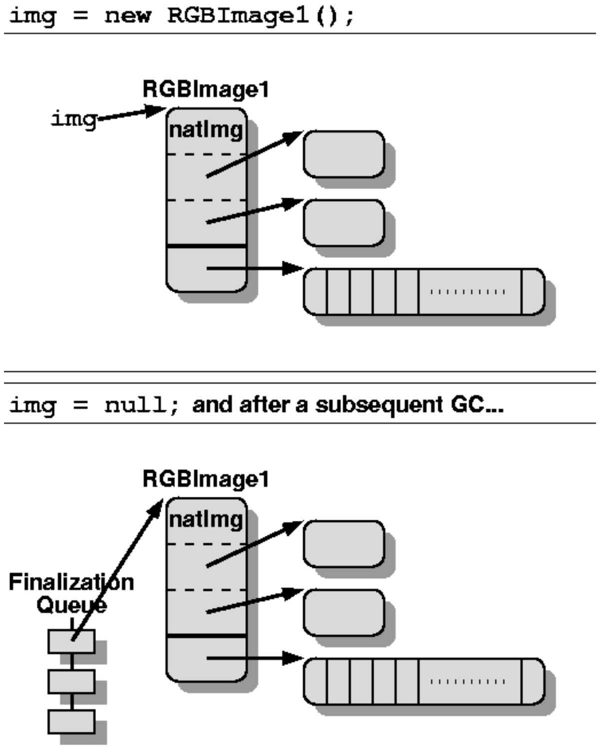
Finalizable objects - example

- subclassing issue
  - delay reclamation of resources not explicitly used



```
public class RGBImage1 extends Image1 {
    private byte rgbData[];
}
```

  - RGBImage1 inherit finalize() method

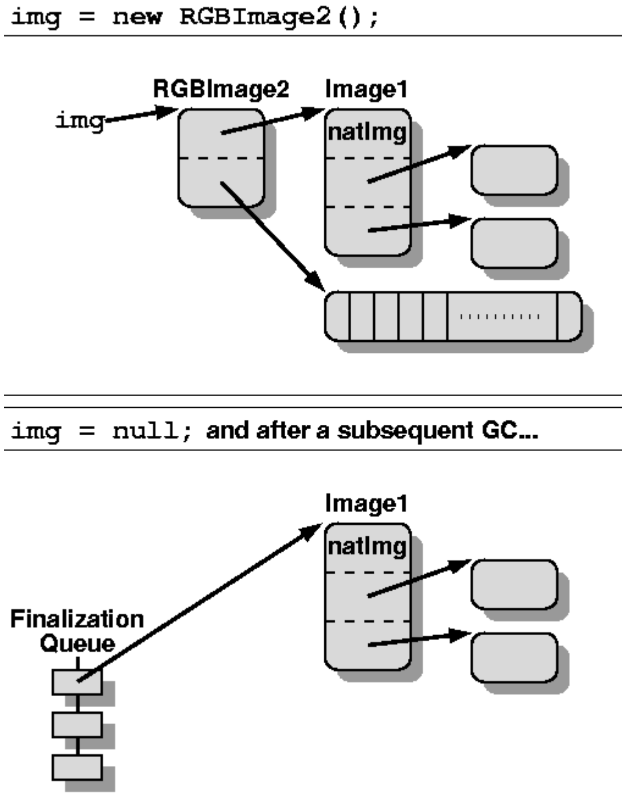


Finalizable objects – example solution 1

- contains reference instead of extends

```
public class RGBImage2 {
    private Image1 img;
    private byte rgbData[];

    public void dispose() {
        img.dispose();
    }
}
```

- BUT no access to non-public, non-package members

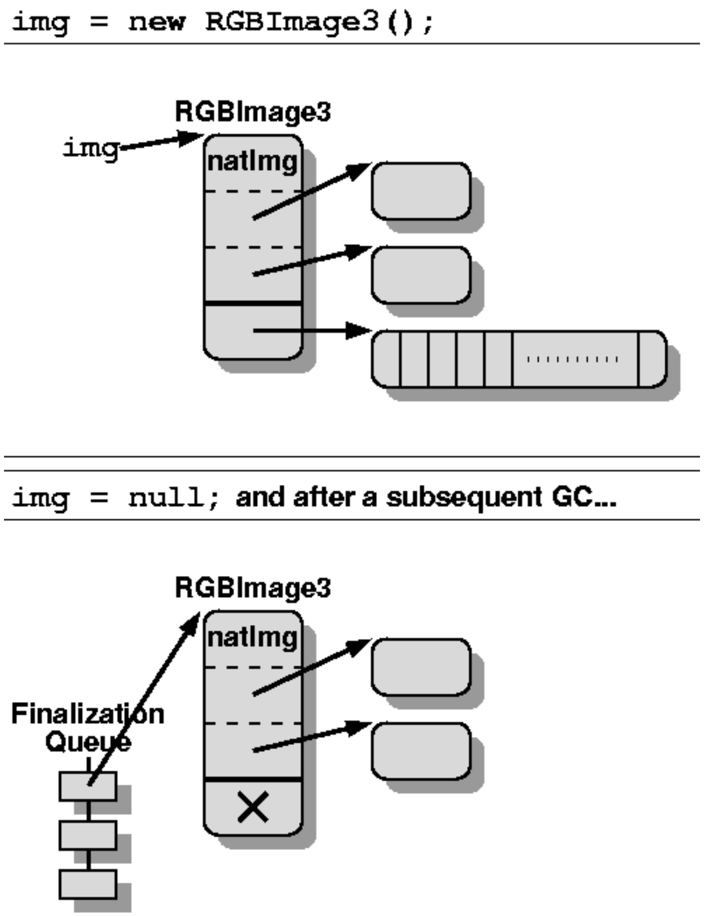


Finalizable objects – example solution 2

- manual nulling

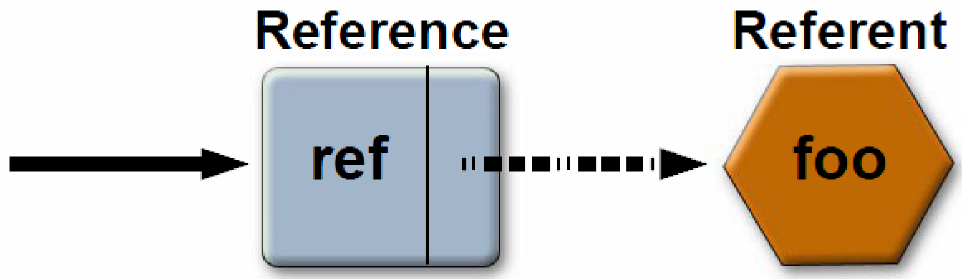
```
public class RGBImage3 extends Image1 {
    private byte rgbData[];

    public void dispose() {
        super.dispose();
        rgbData = null;
    }
}
```

- BUT requires explicit disposal



Reference objects

- mortem hooks
- are more flexible than finalization
- types (ordered from strongest one):
  - {strong reference}
  - soft reference
  - weak reference
  - phantom references
- can enqueue the reference object on a designated reference queue when GC finds its referent to be unreachable, referent is released
- references are added only if you have strong reference to REFERENCE !
- GC has to run !



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

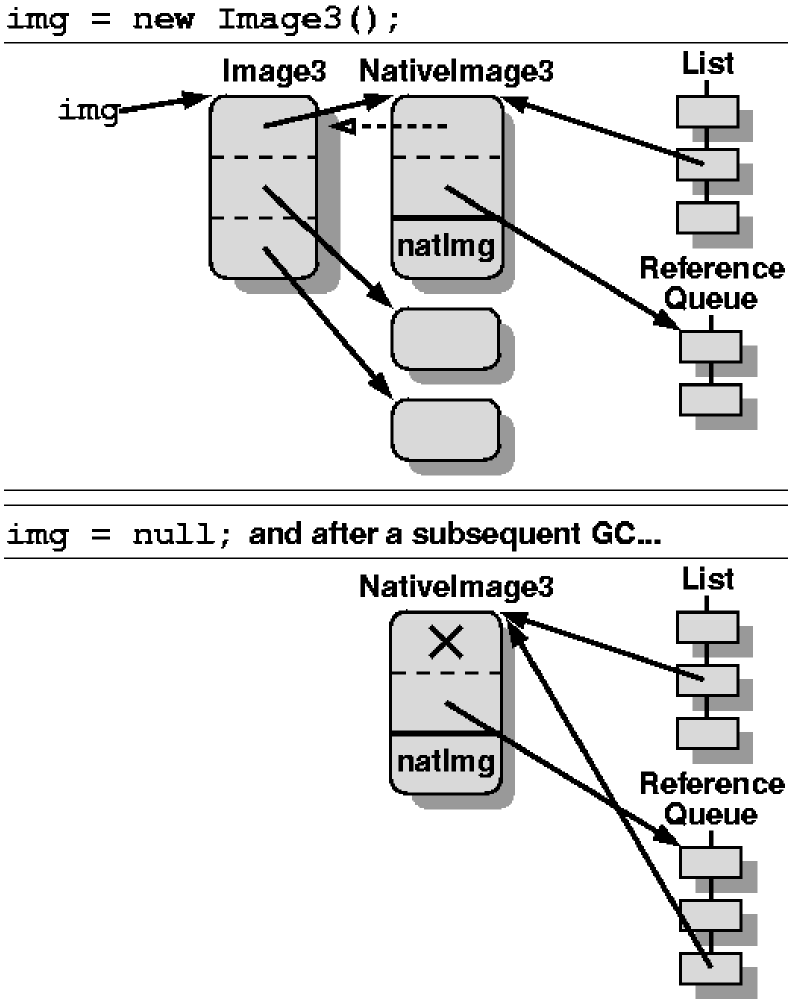
Weak reference

- pre-mortal hook
- usage:
  - do not retain this object because of this reference
  - canonicalizing map – e.g. ObjectOutputStream
  - don't own target, e.g. listeners
  - implement flexible version of finalization:
    - prioritize
    - decide when to run finalization
- get() returns
  - referent if not reclaimed
  - null, otherwise
- referent is cleared by GC (cleared before enqueued) -> need copy referent to strong reference and check that it is not null !!!
- WeakHashMap<K,V> - uses weak keys

Weak reference example

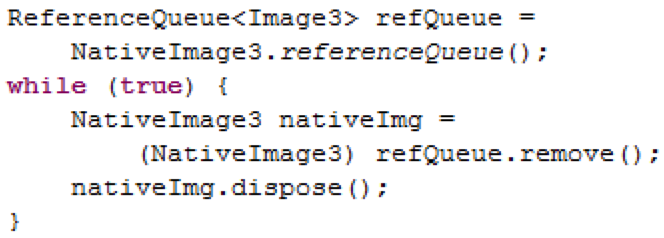
- NativeImage3 cannot be inner class again

```
final static class NativeImage3 extends WeakReference<Image3> {
    private int nativeImg;

    private native void disposeNative();
    void dispose() {
        disposeNative();
        refList.remove(this);
    }

    static private ReferenceQueue<Image3> refQueue;
    static private List<NativeImage3> refList;
    static ReferenceQueue<Image3> referenceQueue() {
        return refQueue;
    }

    NativeImage3(Image3 img) {
        super(img, refQueue);
        refList.add(this);
    }
}

public class Image3 {
    private NativeImage3 nativeImg;
    // ...

    public void dispose() { nativeImg.dispose(); }
}
```



Weak reference example

- » own “clean-up” thread



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

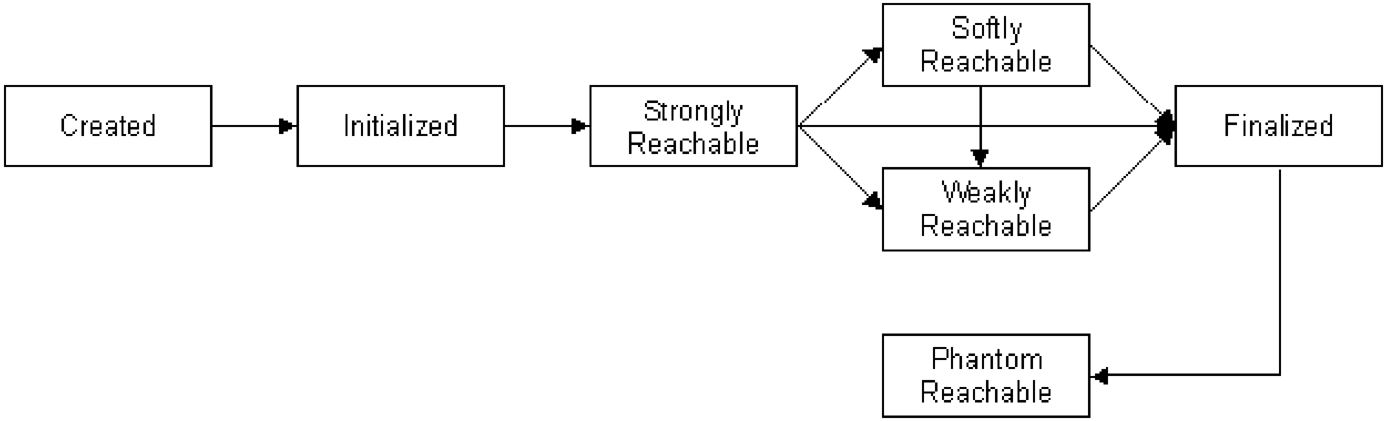
Soft reference

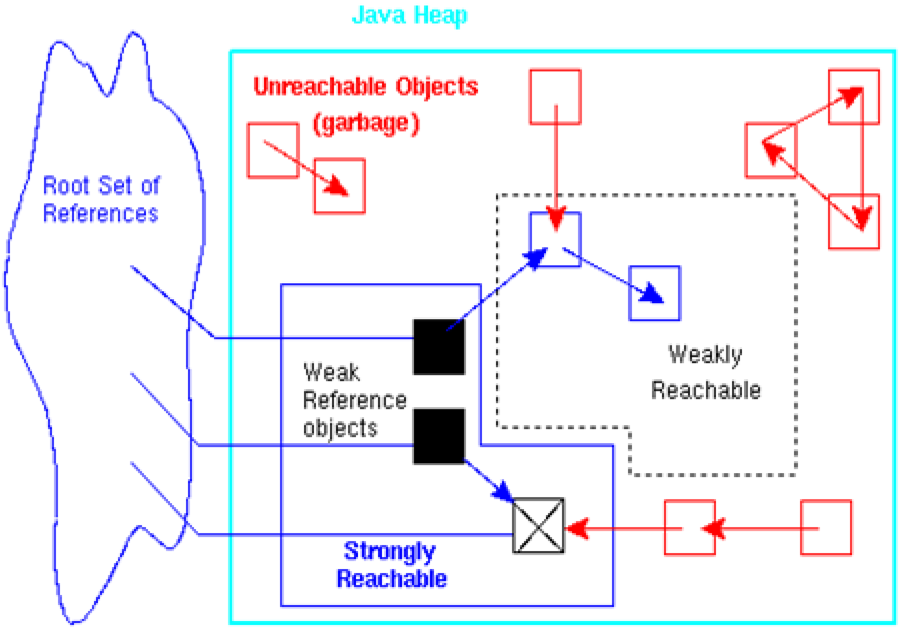
- pre-mortal hook
- usage:
  - reclaim only if there is “memory pressure” based on
  - suitable for caches – create strong reference to data required to keep, best for large objects
  - would like to keep referent, but can loose it
- get() returns:
  - referent if not reclaimed
  - null, otherwise
- referent is cleared by GC (cleared before enqueued)

Phantom reference

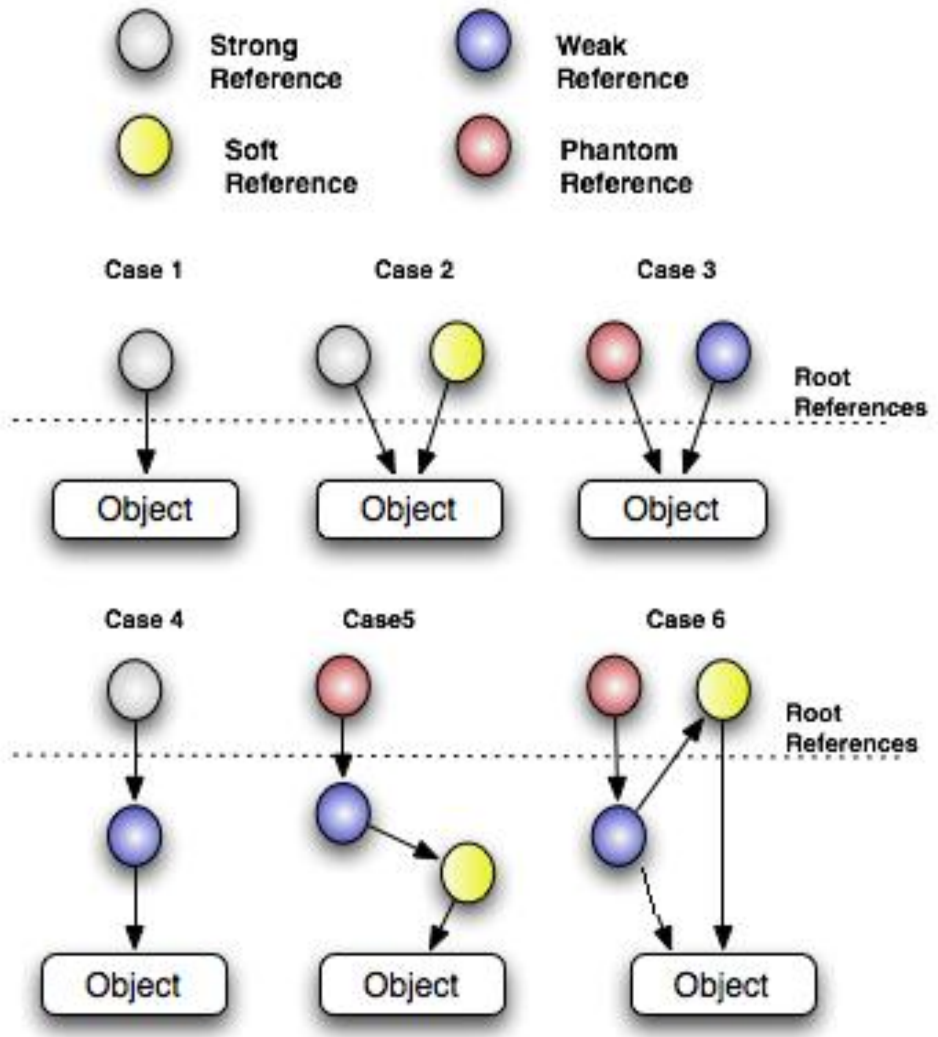
- » post-mortal hook, post-finalization processing
- » designed to be safer than finalizer as the object cannot be resurrected -> ***not true*** !
- » usage:
  - notifies that the object is no longer used
  - keep some data after the object becomes collected
- » get() returns:
  - null always
- » have to specify reference queue for constructor
- » referent is not collected until all references are not become unreachable
- » referent is not cleared automatically, referent can be cleared by method clear()

Reachability of an object





Reachability of an object



Phantom reference example

```
public static class GhostReference extends PhantomReference {
    private static final Collection currentRefs = new HashSet();
    private static final Field referent;

    static {
        try {
            referent = Reference.class.getDeclaredField("referent");
            referent.setAccessible(true);
        } catch (NoSuchFieldException e) {
            throw new RuntimeException("Field \"referent\" not found");
        }
    }

    public GhostReference(Object referent, ReferenceQueue queue) {
        super(referent, queue);
        currentRefs.add(this);
    }

    public void clear() {
        currentRefs.remove(this);
        super.clear();
    }

    public Object getReferent() {
        try {
            return referent.get(this);
        } catch (IllegalAccessException e) {
            throw new IllegalStateException("referent should be accessible!");
        }
    }
}
```