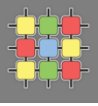


Architecture of software systems

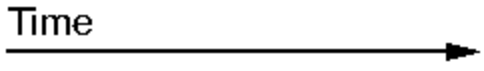
Course 10: Memory management with garbage collector, references

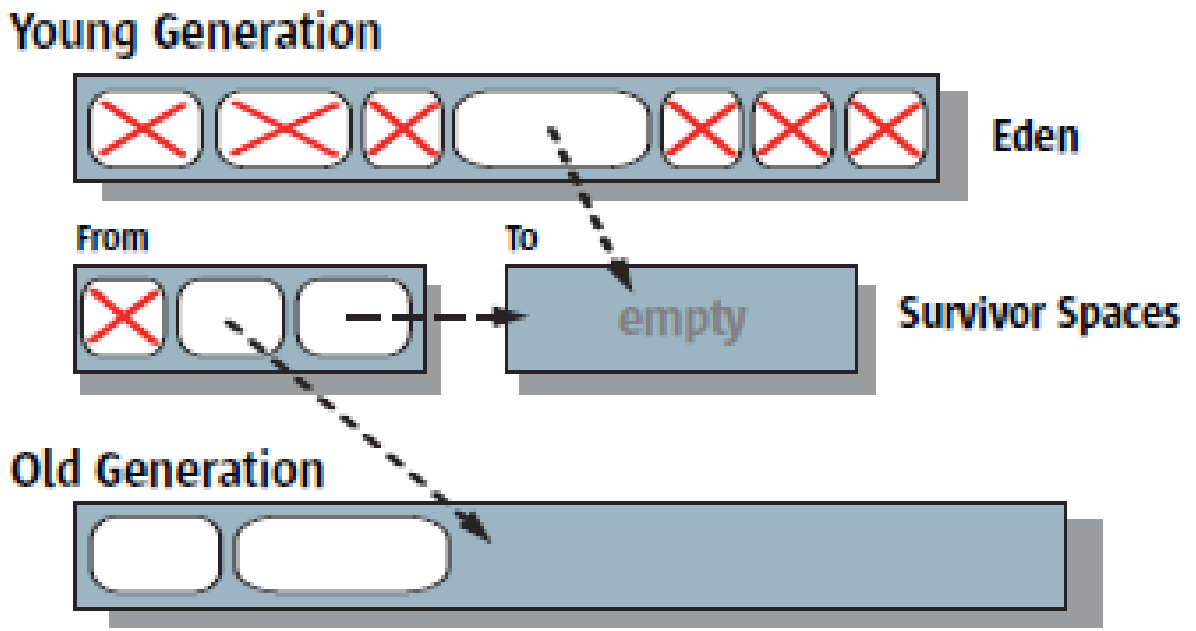
David Šišlák
david.sislak@fel.cvut.cz

Serial minor garbage collector

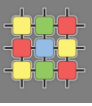
- young collection -> old generations collection serially in *stop-the-world* fashion

- young generation:
  - reported as **Copy** in telemetry
  - age of object (incremented every minor GC)
  - efficiency is proportional to number of copied objects !

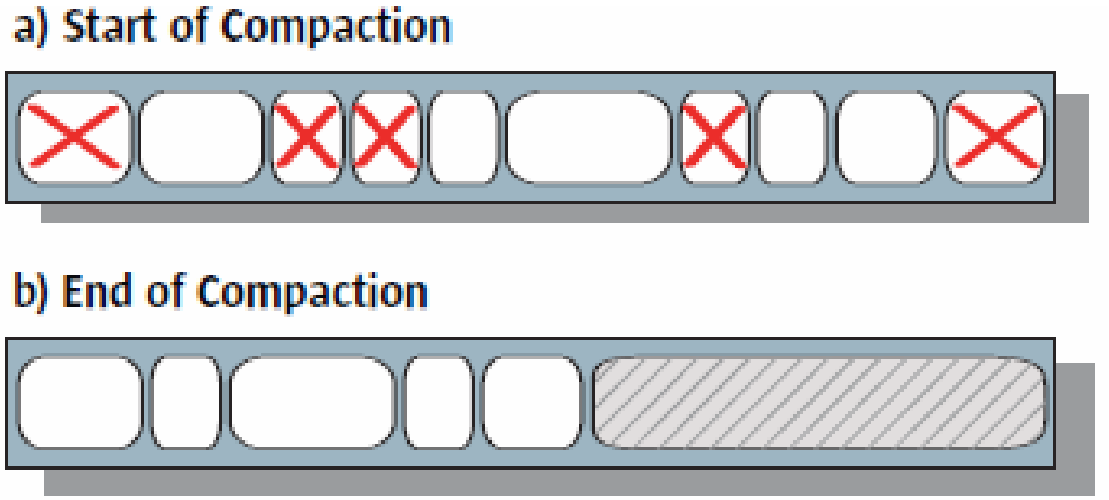




Serial major garbage collector



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



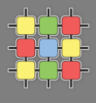
» default **client JVM** from Java 5.0 or when requested by

```
-XX:+UseSerialGC
```

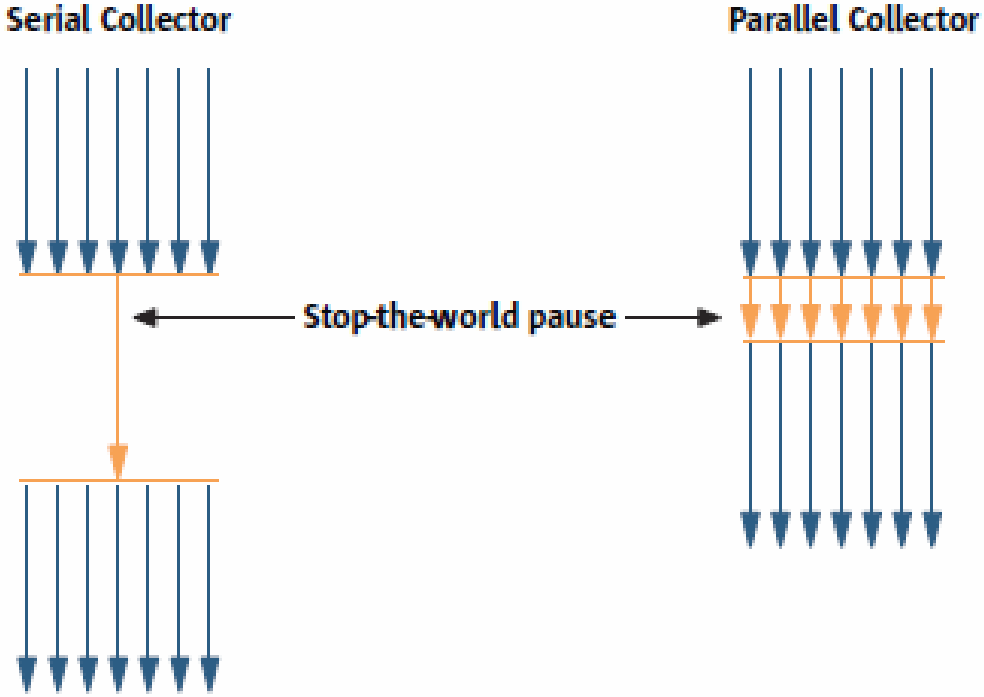
» effectively handles application with 64MB heaps

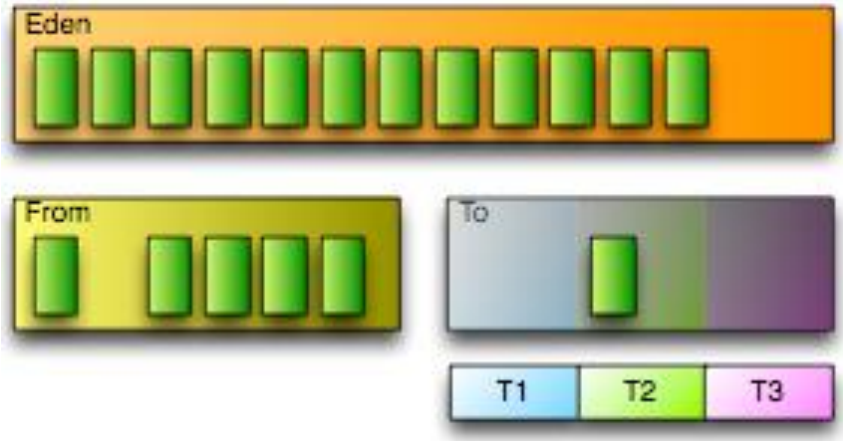
» In memory telemetry reported as **MarkSweepCompact**

Parallel minor garbage collector



- utilize more cores/CPUs, known as *throughput garbage collector*
- *In memory telemetry reported as* **ParNew** or **PS *Scavenge***
- still stop-the-world but in parallel manner for young generation
- fragmentation in survivor area; **no ages** like in serial GC



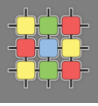


- default for **server JVM** from Java 5.0 or when requested by

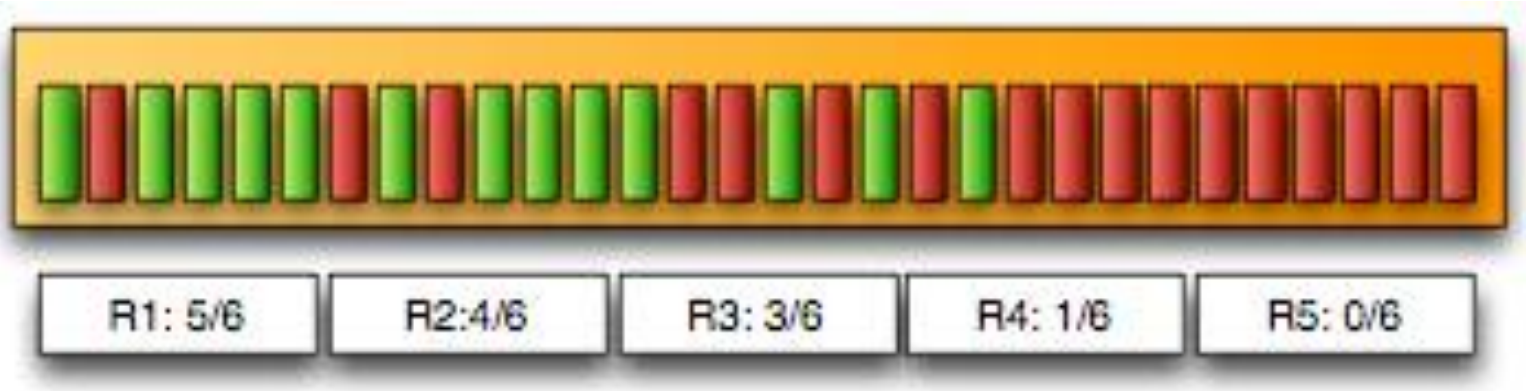
```
-XX:+UseParNewGC   or   -XX:+UseParallelGC
```

- the number of threads controlled by `-XX:ParallelGCThreads=n`

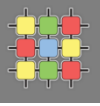
Parallel major compacting collector



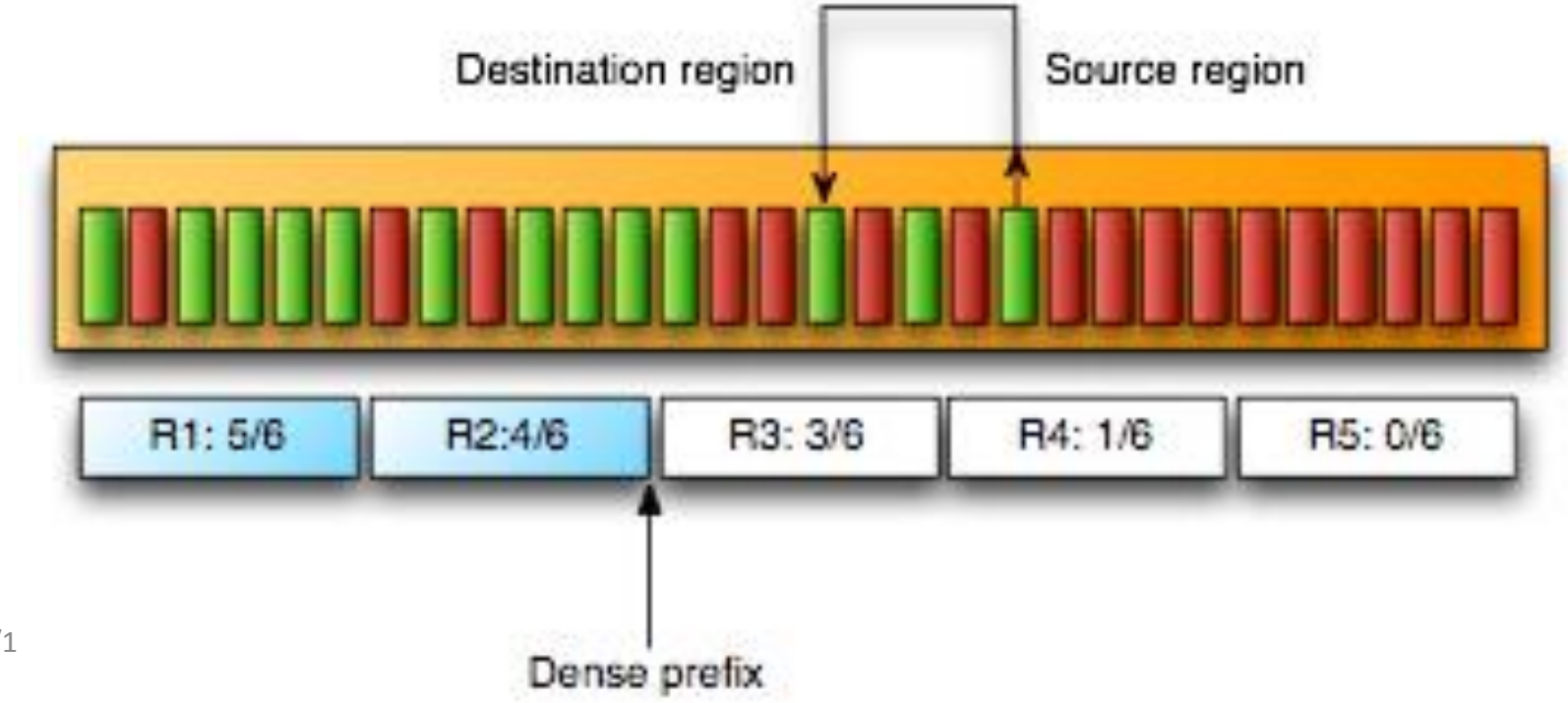
- reported as **PS Mark Sweep**
- can be used only with minor PS Scavenge
- done in stop-the-world manner
- each generation (old/permanent) logically divided into fixed-sized regions
- ***parallel mark*** phase:
  - initiated by divided reachable root objects
  - info about live objects (size & location) are propagated to the corresponding region data



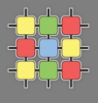
Parallel major compacting collector



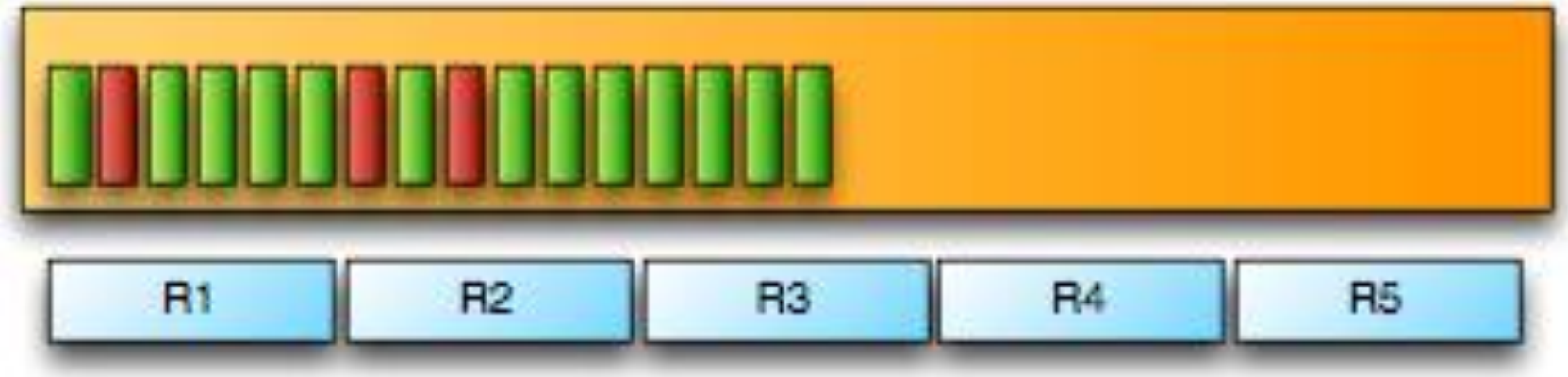
- ***summary phase*** (implemented in serial):
  - identify density of regions (due to previous compactions, older objects should be on the left, younger to right side)
  - find from which region (starting from the left side) it has sense to do compaction regarding recovered from a region:
    - *dense prefix* – left regions which are not collected
  - calculate new location of each live data for each region; most right regions will fill most left ones



Parallel major compacting collector



- ***parallel compaction/sweeping*** phase:
  - divide not moving regions (compacting to themselves), and fully reclaimed regions among threads
  - each thread first compact/copy/clear the region itself and then start filling it by designated right regions
  - *no synchronization* needed, only one thread operate per each region
  - finally heap is packed and large empty block is at the right end

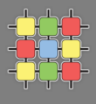


- default for **server JVM** from Java 5.0 or when requested by

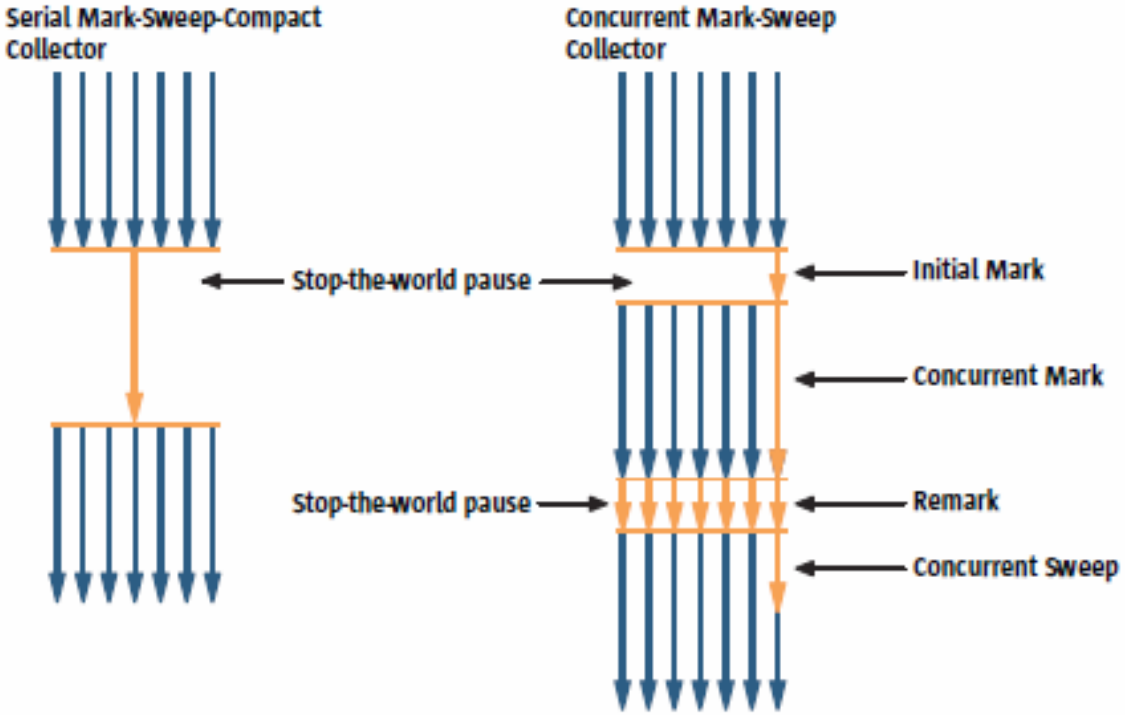
  `-XX:+UseParallelOldGC`

- the number of threads controlled by `-XX:ParallelGCThreads=n`

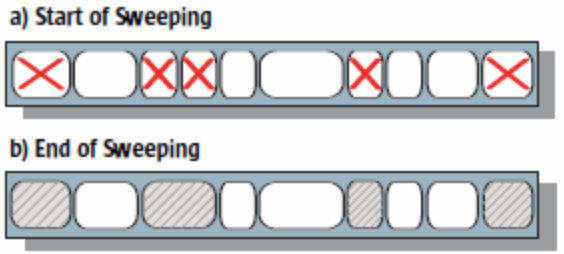
Concurrent mark-sweep (CMS) collector



- low-latency collector
- reported as **ConcurrentMarkSweep** in memory telemetry
- done concurrently with the application execution
- *initial mark* – short pause identifying the initial set of live objects directly reachable from roots; one thread
- *concurrent mark* – traversal of objects; all reference modification are monitored by changed flag
- *remark* – revisiting modified objects (overhead); but parallel
- *concurrent sweep* – no compaction



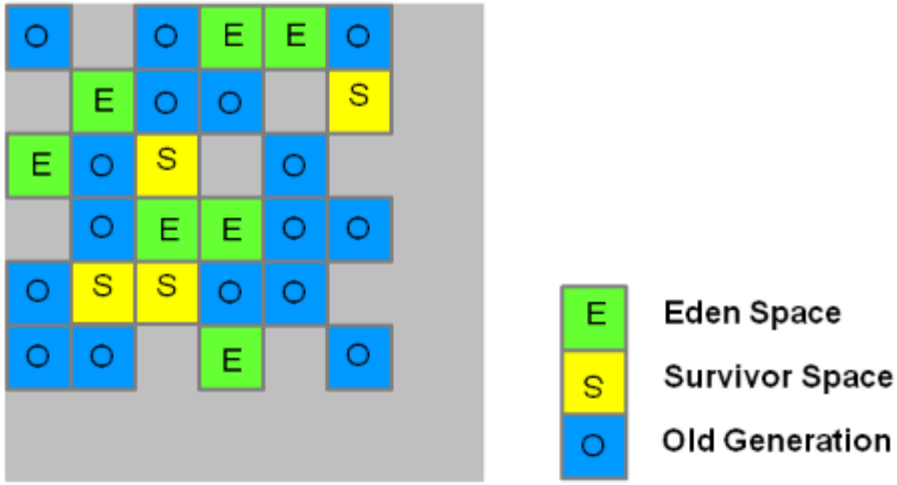
Concurrent mark-sweep (CMS) collector



- **non-compacting**
- cannot use bump-the-pointer allocation
- 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`

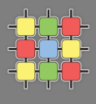
mixed G1 collector

- the latest GC (introduced in Java 6 update 14)
- whole heap divided into regions (by def. about 2000 regions 1-32MB)
- no explicit separation between generations, only regions are mapped to generational spaces (generation is set of regions, changing in time)

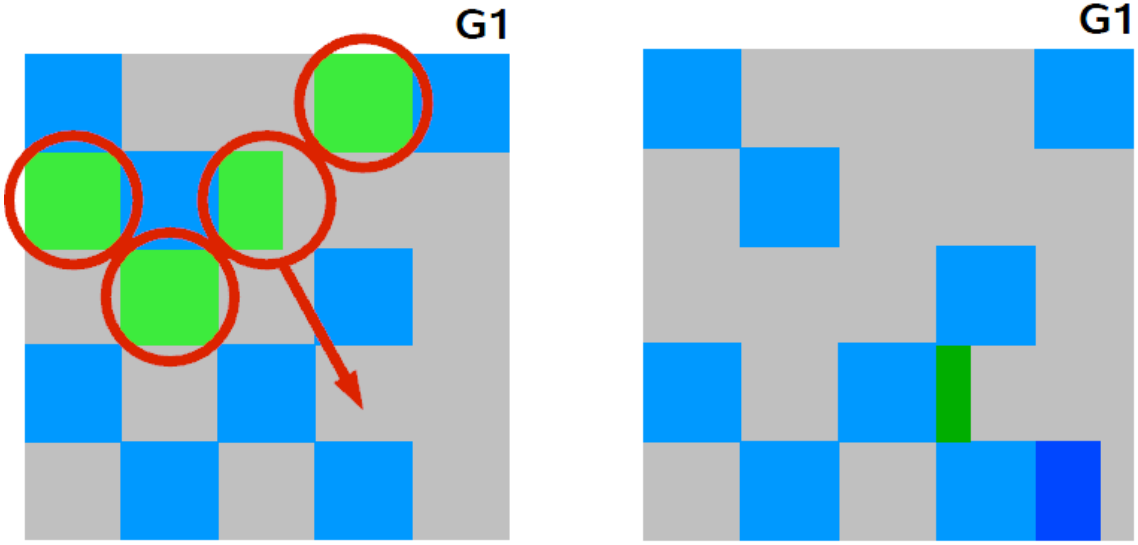


- compacting -> enables bump-the-pointer, TLABs, uses CAS
- compaction = copy live from a region to an empty region
- keep **Humongous regions** (sequence) for objects >=50% regions size
- maintain list of free regions for constant time

mixed G1 collector – minor GC

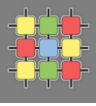


- stop-the-world approach with parallel threads
- live object are copied (from eden and survivor regions) into one or more new survivor regions
- if aging threshold is met => promoted into old generation regions



- G1 uses **Remembered Set** (RS) monitoring cross region references – ignore inter-region and null references
  - mechanism based on memory barrier for modification of object reference
  - 512 bytes cards in each regions with corresponding dirty flag for each region

mixed G1 collector – major GC

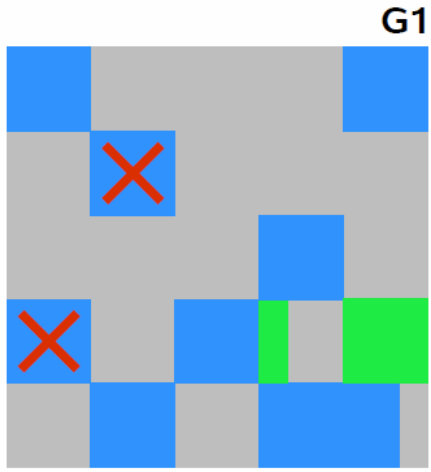


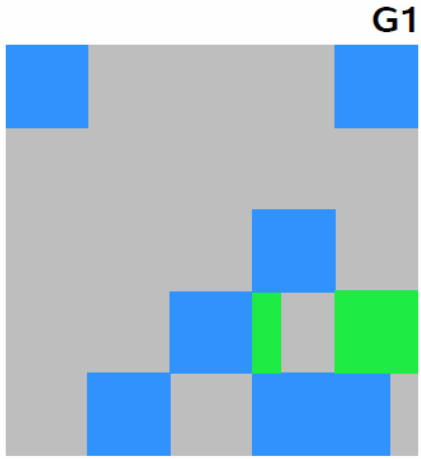
» combination of CMS and parallel compacting collector

» runs immediately after minor GC if heap occupancy threshold is met

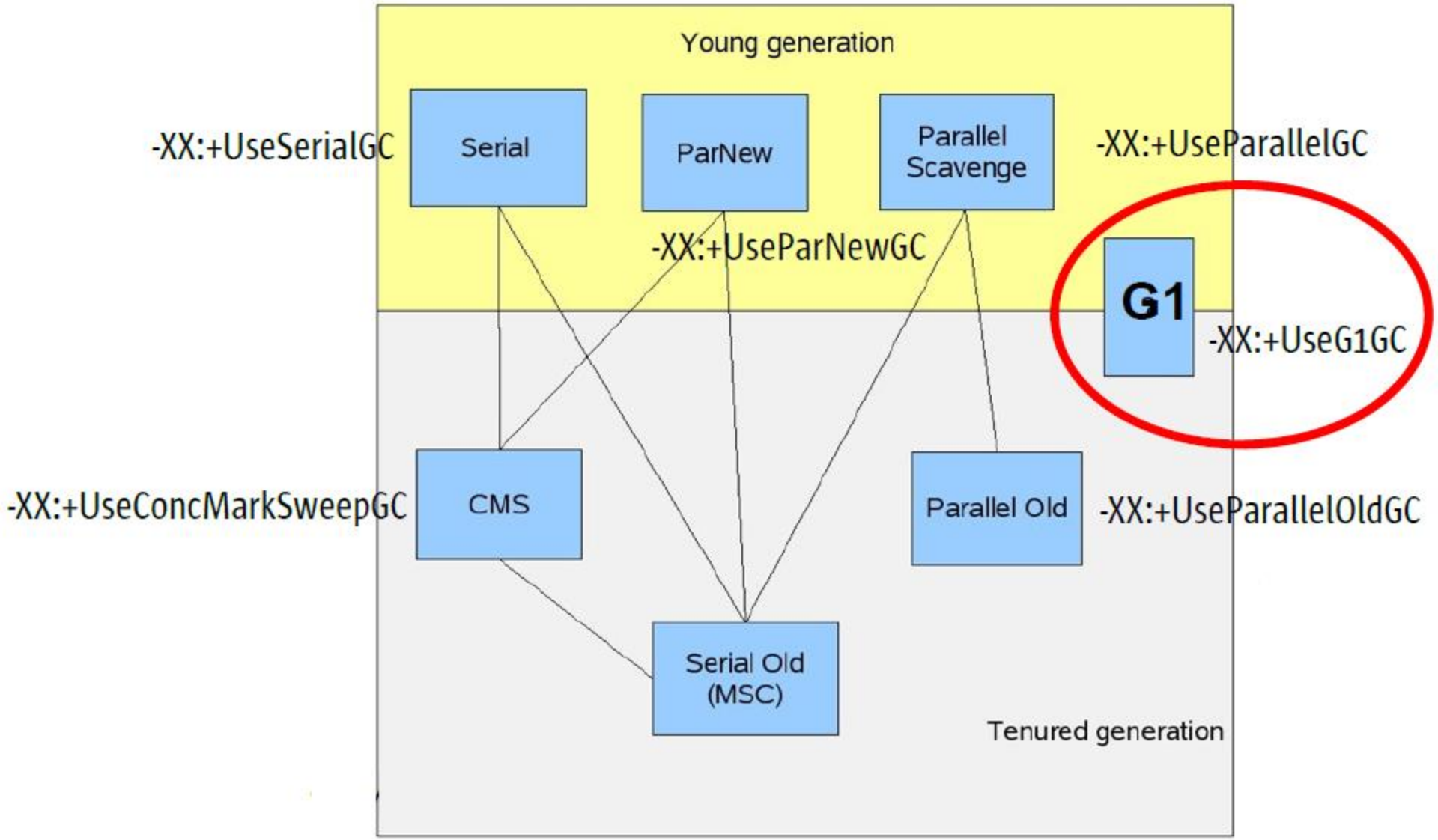
-XX:InitiatingHeapOccupancyPercent=n (defualt 45%)

- initial mark based on SATB (snapshot-at-the-beginning)
  - stop-the-world
- concurrent marking and region-based stats generation
- remark
  - stop-the-world
  - reclaim empty regions
- reclaim old regions (no sweeping using regions)
  - pick regions with low live ratio
  - only few are collected per such GC based on -XX:MaxGCPauseMillis=n (default 200ms)
  - leave garbage in regions with high live ratio

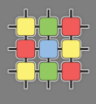




Garbage collectors relation

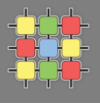


Configure garbage collector



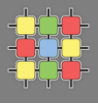
- explicit type:
  - `-XX:+UseSerialGC, -XX:+UseParallelGC, -XX:+UseParallelOldGC, -XX:+UseConcMarkSweepGC -XX:+UseG1GC`
- statistics:
  - `-XX:+PrintGC, -XX:+PrintGCDetails, -XX:+PrintGCTimeStamps, -XX:+PrintTenuringDistribution`
- heap sizing:
  - `-Xmx` – max heap size, default 64MB on client JVM, 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



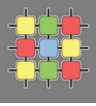
- heap sizing cont.:
  - `-XX:NewRatio=n` - ratio between young and old gens
    default 2 client JVM (young includes survivor),
    n=2 => 1:2 => young is 1/3 of total heap
  - `-XX:SurvivorRatio=n` – ratio between each survivor and Eden
    default 32, n=32 => 1:32 => each survivor is 1/34 of young 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

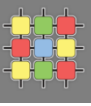
Finalizable objects



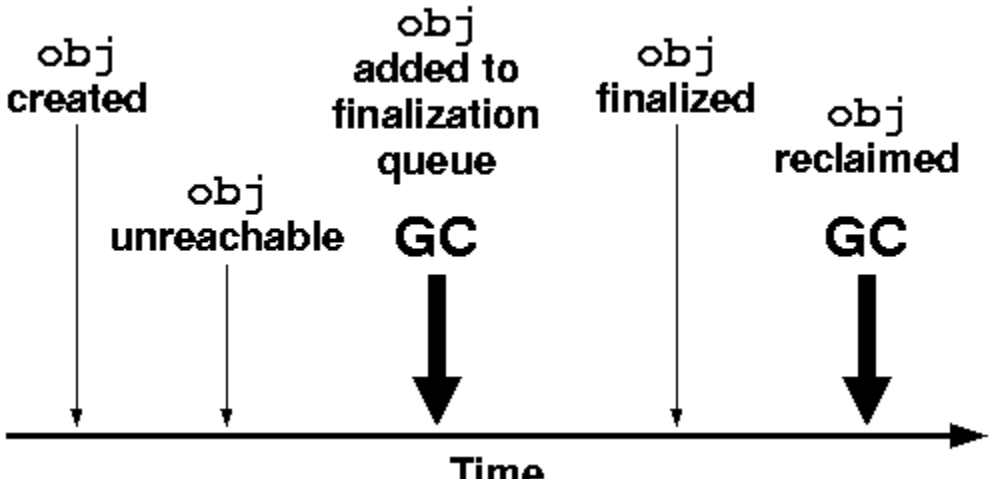
- have a non-trivial **finalize()** method
- postmortem hook
- used for clean-up for **unreachable object**, typically reclaim native resources:
  - GUI components
  - file
  - socket

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

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

    static private Image1 randomImg;
}
```

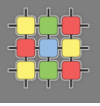
Finalizable objects



- » finalizable object allocation:
  - slower because VM must track finalizable objects
- » finalizable object reclamation
  - at least two GC cycles:
    - identification and enqueue object on finalization queue (**only one !**)
    - 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

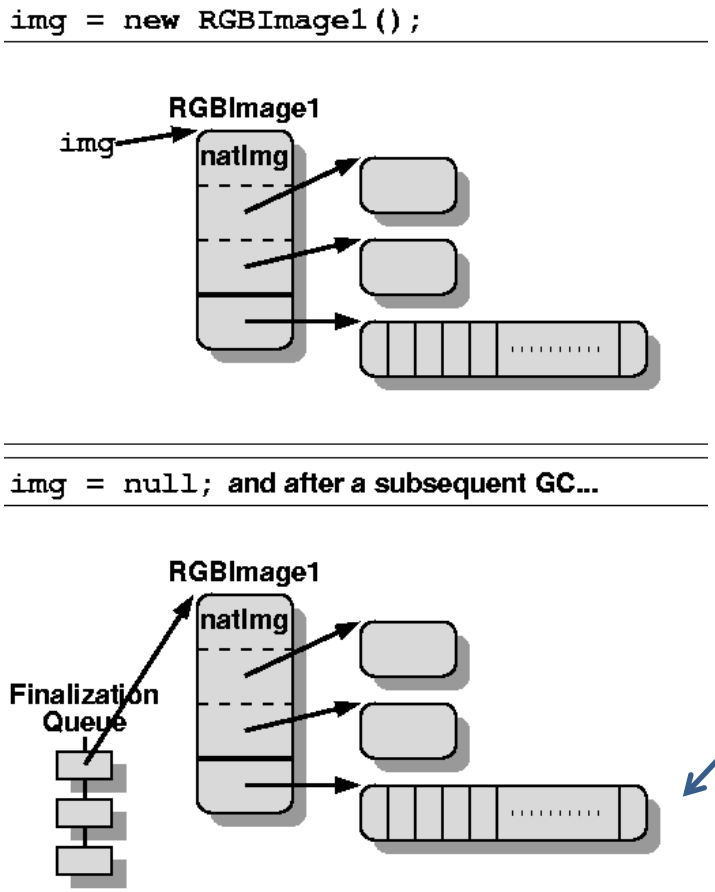


» subclassing issue

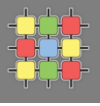
- **delayed reclamation** of resources not explicitly used

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

- RGBImage1 inherit finalize() method



Finalizable objects – example solution 1

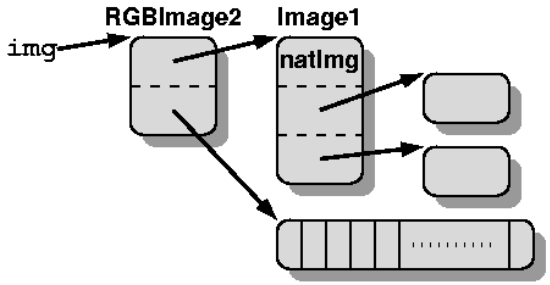


» contains **reference instead of extends**

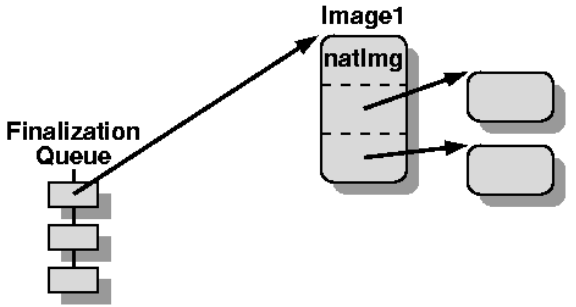
```
public class RGBImage2 {
    private Image1 img;
    private byte rgbData[];

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

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

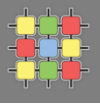
`img = new RGBImage2();`



`img = null;` **and after a subsequent GC...**



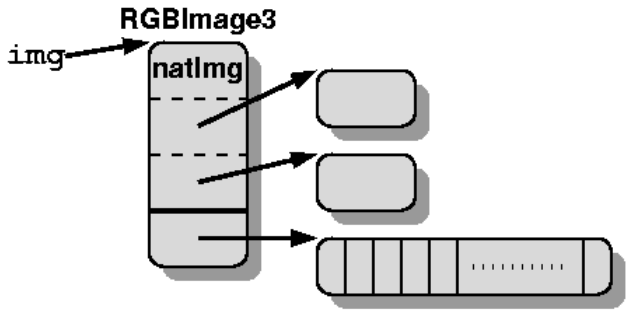
Finalizable objects – example solution 2



» **manual nulling**

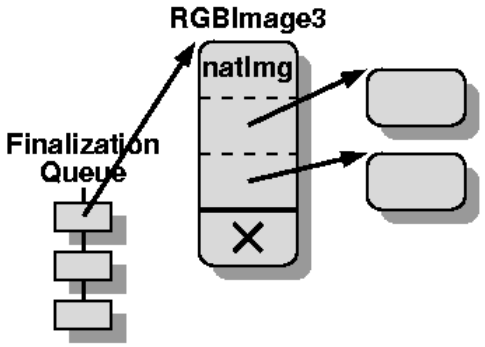
```
public class RGBImage3 extends Image1 {
    private byte rgbData[];

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

`img = new RGBImage3();`

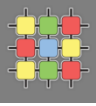


» BUT requires explicit disposal

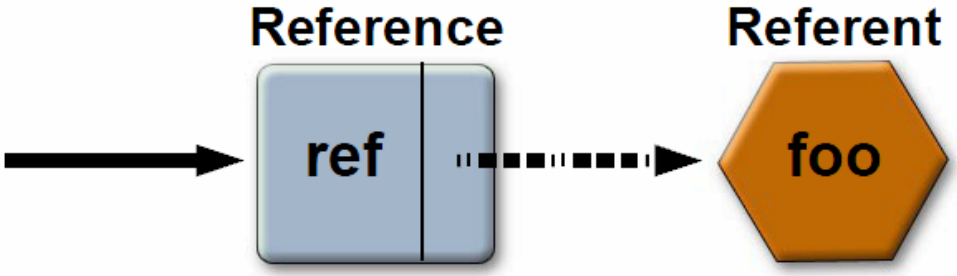
`img = null;` **and after a subsequent GC...**



Reference objects

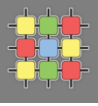


- mortem hooks
- are more **flexible** than finalization
- types (ordered from strongest one):
  - {strong reference}
  - soft reference
  - weak reference
  - phantom references
- can enqueue the reference object on a designated reference queue when GC finds its referent to be unreachable, referent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



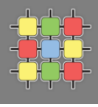

```
ref  = new WeakReference(foo, rq);
```

Weak reference



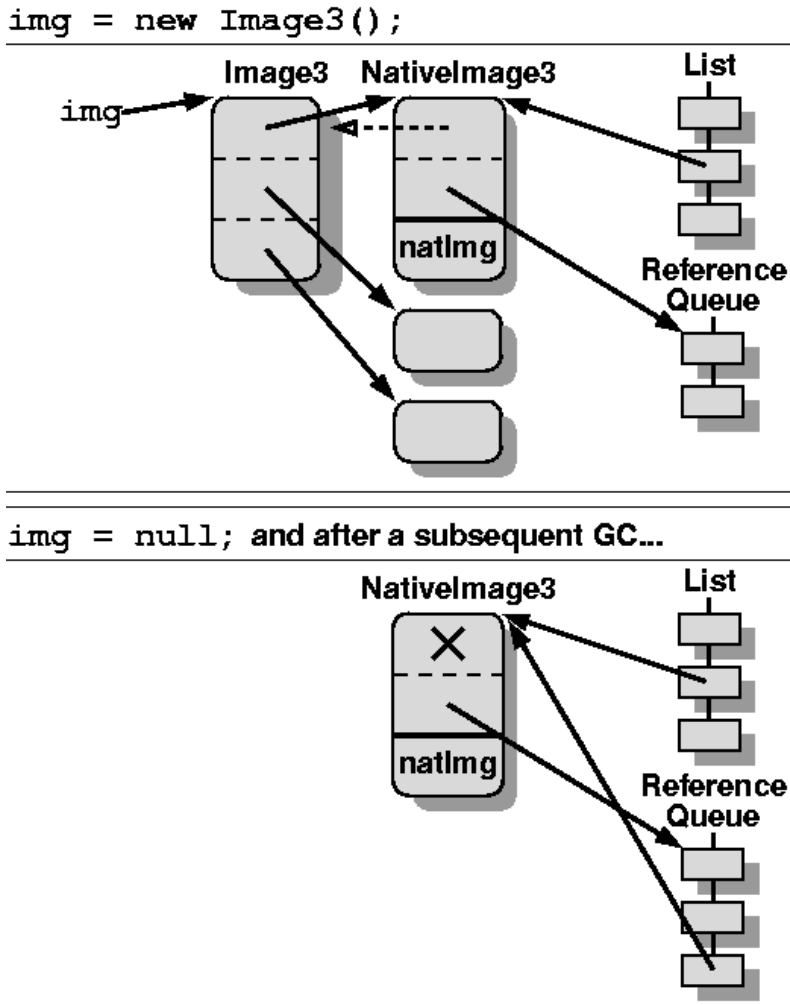
- pre-finalization processing
- usage:
  - **do not retain this object because of this reference**
  - canonicalizing map – e.g. ObjectOutputStream
  - don't own target, e.g. listeners
  - implement flexible version of finalization:
    - prioritize
    - decide when to run finalization
- get() returns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

Weak reference example

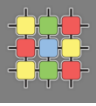


» NativeImage3 cannot be inner non-static class (due to strong ref)

```
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(); }
}
```



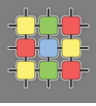
Weak reference example



» own “clean-up” thread

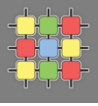
```
ReferenceQueue<Image3> refQueue =
    NativeImage3.referenceQueue();
while (true) {
    NativeImage3 nativeImg =
        (NativeImage3) refQueue.remove();
    nativeImg.dispose();
}
```

Soft reference



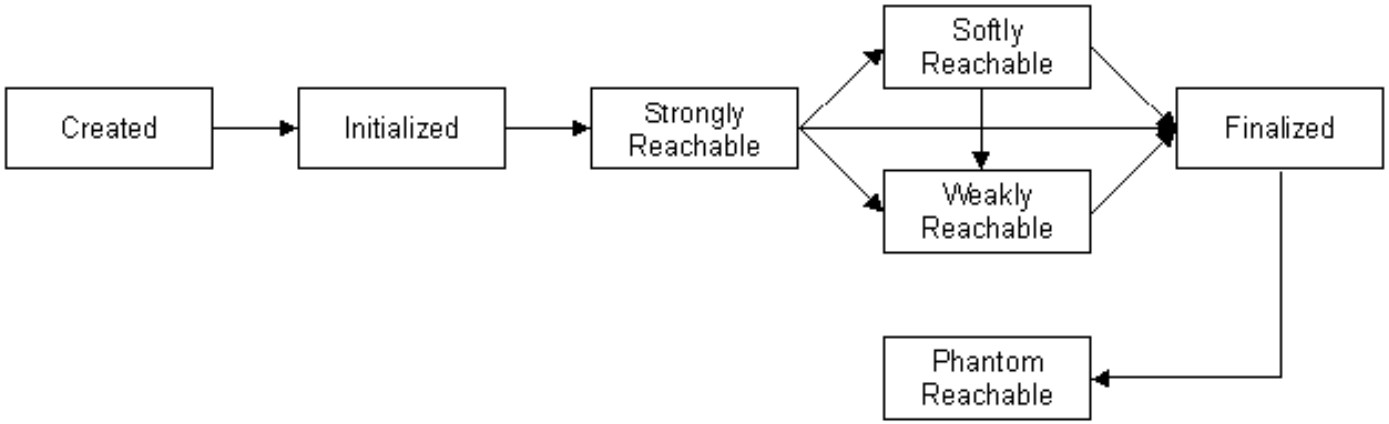
- pre-finalization processing
- usage:
  - **would like to keep referent, but can loose it**
  - reclaim only if there is “memory pressure” based on heap usage
  - suitable for caches – create strong reference to data required to keep, best for large objects
  - all are cleared before OutOfMemoryError
- get() returns:
  - referent if not reclaimed
  - null, otherwise
  - **updates timestamp** of usage (can keep recently used longer)
- referent is cleared by GC (cleared before enqueued) and **can be collected**

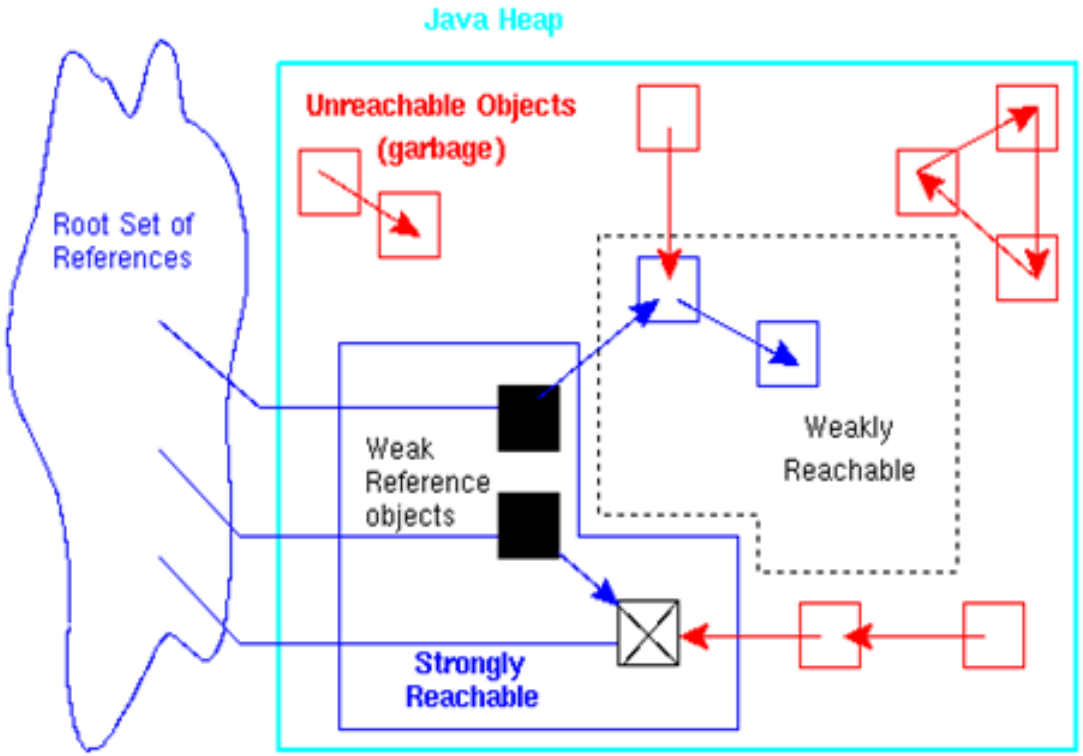
Phantom reference



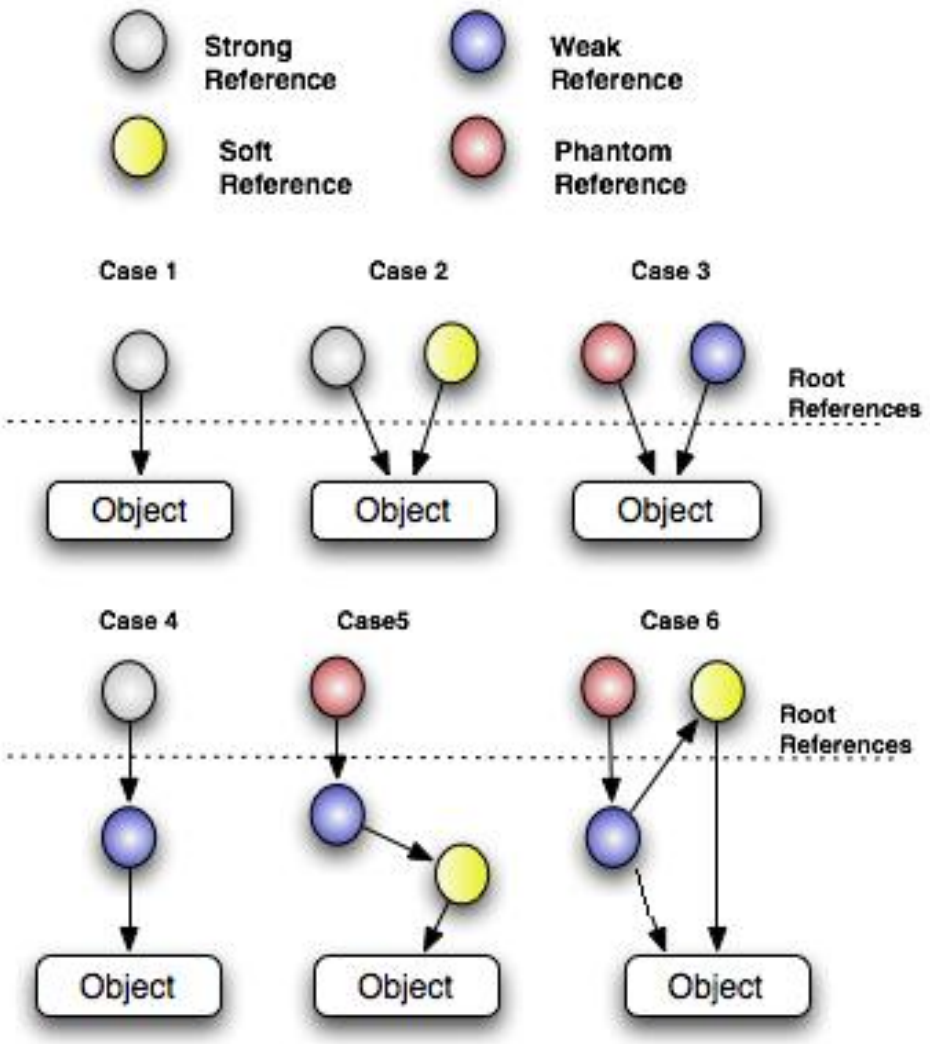
- post-finalization processing
- usage:
  - **notifies that the object is no longer used**
  - keep some data after the object becomes finalized
- get() returns:
  - null always
- **have to** specify reference queue for constructor
- **referent is not collected** until all references are not become unreachable or manually cleared
- internal referent reference is not cleared automatically, it can be cleared by method clear()

Reachability of an object

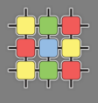




Reachability of an object

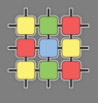


Phantom reference example



```
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!");
        }
    }
}
```

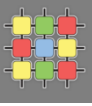
Performance recommendations



- 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:

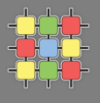
```
ArrayList<String> list = new ArrayList<String>(1024);
```

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
    - use reference instead of extending
    - manual nulling

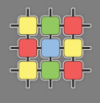
Performance recommendations



» use lazy initialization

```
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

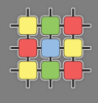


» objects in the wrong scope

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

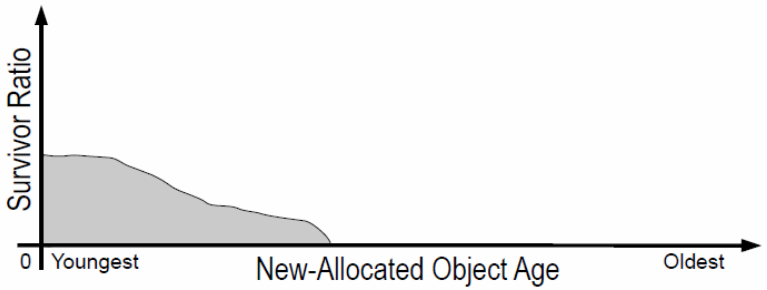
```
class Foo {
     public void doIt(int length) {
          String[] names = new String[length];
          populate(names);
          print(names);
     }
}
```

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
- 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)