Architecture of software systems

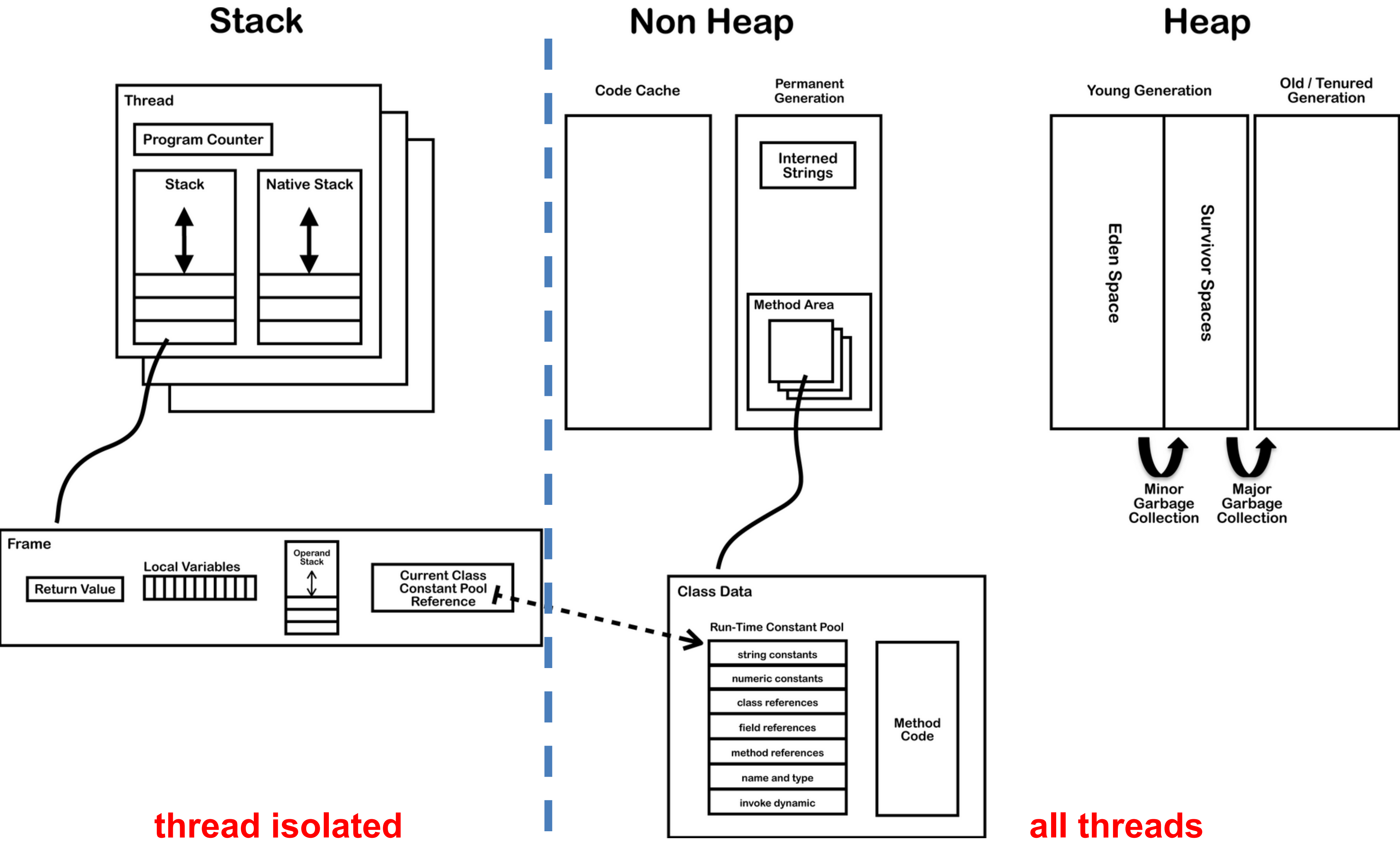
Course 10: Data structures, memory management , garbage collector, references

David Šišlák
david.sislak@fel.cvut.cz

Data structures

- primitives: boolean, byte, char, int, long, float, double
  - without implicit allocation
  - placed in frame in variables or operand stack
- objects (object header structure overhead)
  - every object is descendant of Object by default
    - methods – clone(), equals, getClass(), hashCode(), wait(…), notify (…), finalize()
  - objects for primitives: Boolean, Byte, Character, Integer, Long, Float, Double; can be **null**; all are **immutable** objects (final values)
  - other objects
- arrays
  - special data structure which store a number of items of the same type in linear order; have the defined limit
  - JAVA automatically check limitations
  - allocated on the heap
  - multi-dimensional arrays = arrays of arrays; ragged array

JVM Memory



thread isolated

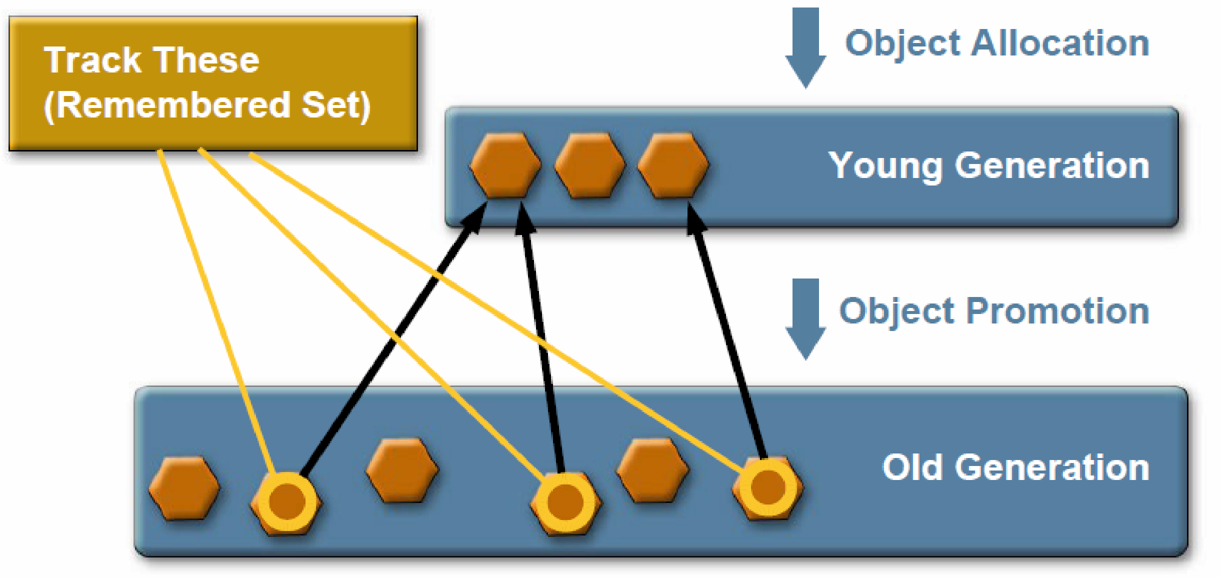
all threads

Memory management

- explicit vs. ***automatic***
  - no crashes due to errors – e.g. usage of de-allocated objects
  - no memory (space) leaks
- garbage collection managed by ***garbage collector***
  - live objects (transiently reachable from **roots** – thread frames, static fields) remain in memory
  - dead are reclaimed
- desired garbage collection characteristics:
  - allocation performance - find a block of unused memory with certain size
  - avoid fragmentation (e.g. by compaction)
  - efficiency without long pauses in application run
  - no bottleneck for multi-threaded (multi-core/multi-CPUs) systems
- design architectures:
  - serial vs. parallel
  - concurrent vs. stop-the-world
  - compacting vs. non-compacting vs. copying

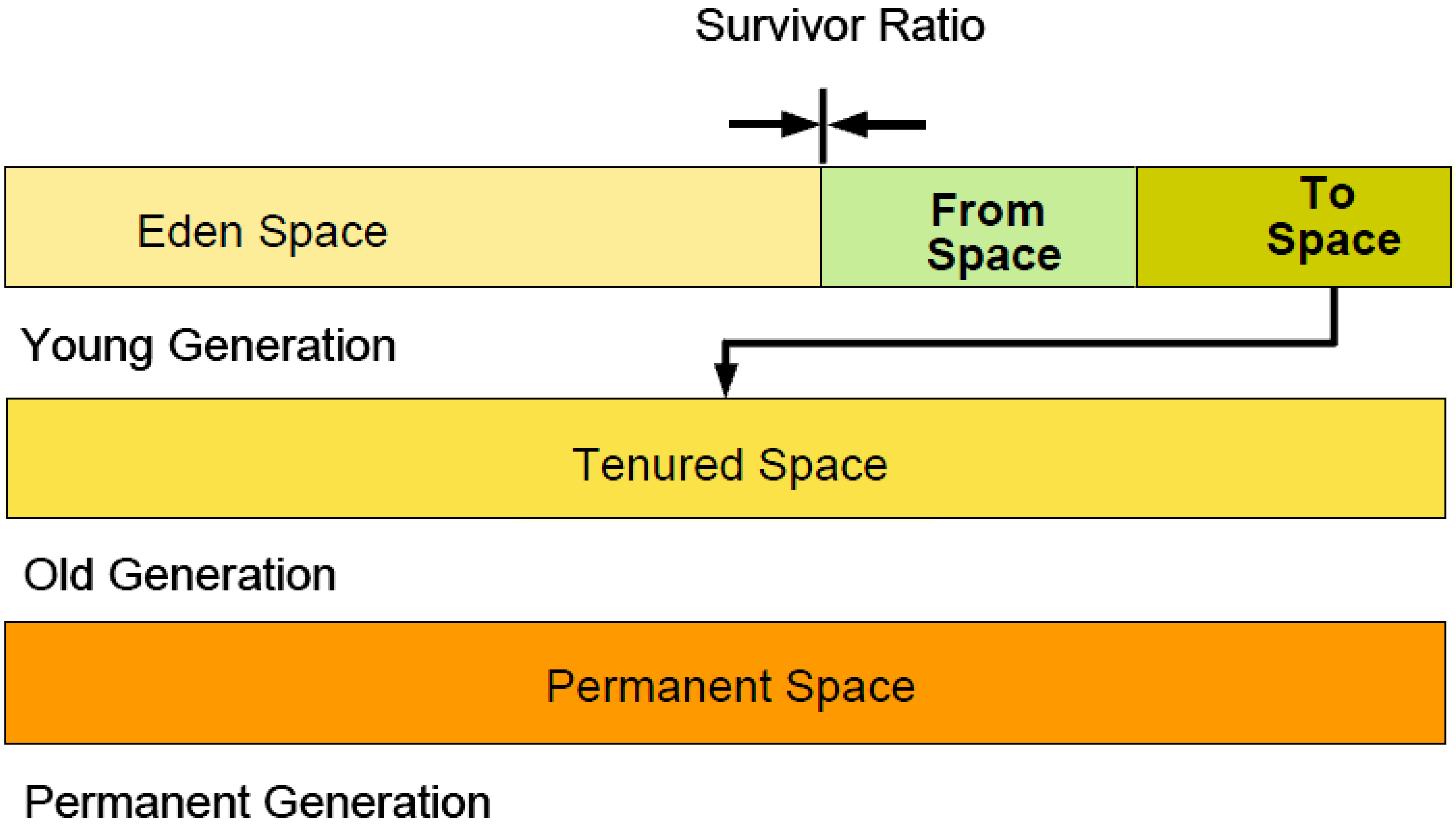
Generational concept

- heap divided into generations based on object ages:
  - young – frequent GC, small size -> fast GC
  - old – rare GC, large size -> slow GC
- promotion (tenuring) objects based on survival of objects during GC
- based on weak **generational hypothesis**:
  - most allocated objects are not referenced for long – they die young
  - few references from older to younger object exist
- need track old-to-young references



JAVA heap layout

- *minor (young)* vs. *major (old)* GC – different algorithms
- major GC can be invoked by young GC if there is no space in tenured space

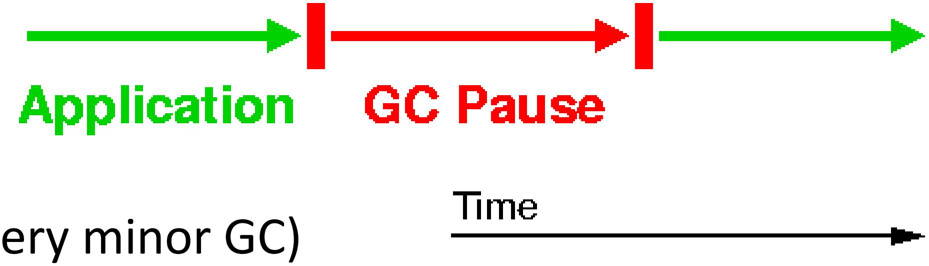


Fast object allocation

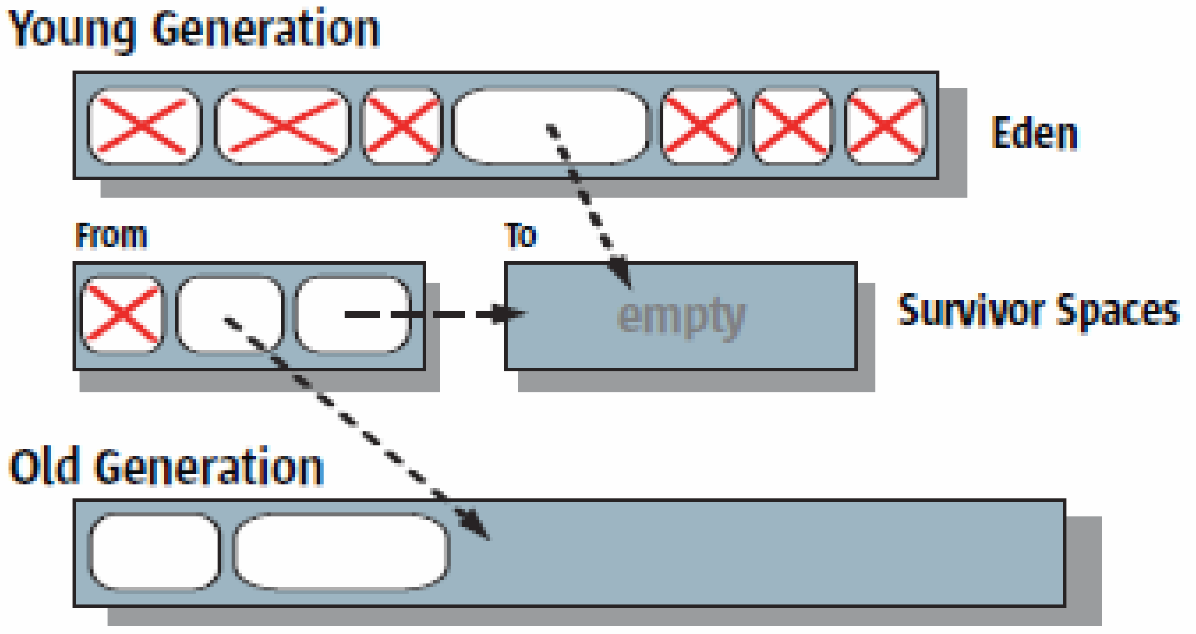
- based on ***bump-the-pointer*** technique
  - track previously allocated object
  - fit new object into remainder of generation end
- **thread-local allocation buffers** (TLABs)
  - remove concurrency bottleneck
  - each thread has very small exclusive area (few % of Eden in total)
  - infrequent full TLABs implies synchronization (based on CAS)
  - exclusive allocation takes about *10 native instructions*

Serial collector

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

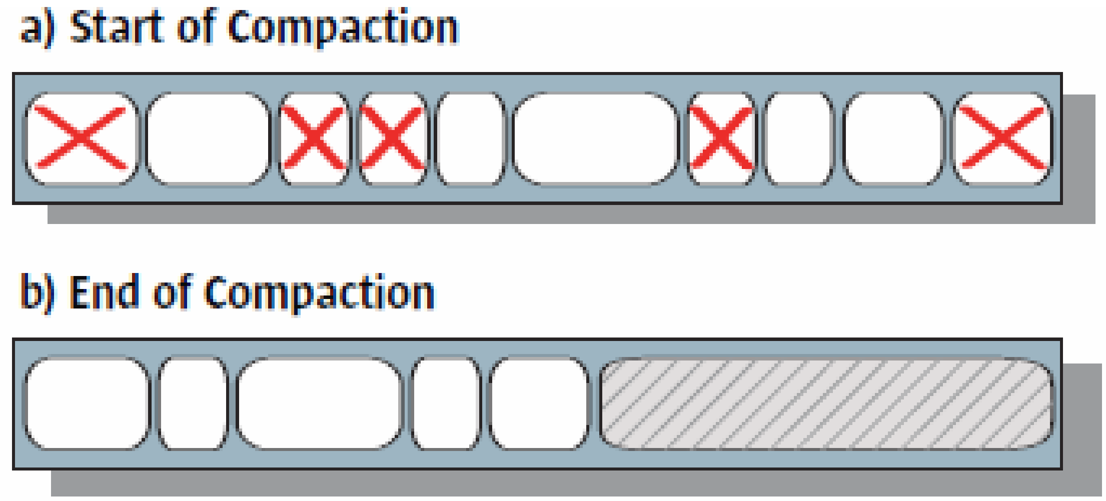


- <u>**young generation**</u>:
  - <u>age of object</u> (incremented every minor GC)
  - efficiency is proportional to number of copied objects !



Serial collector

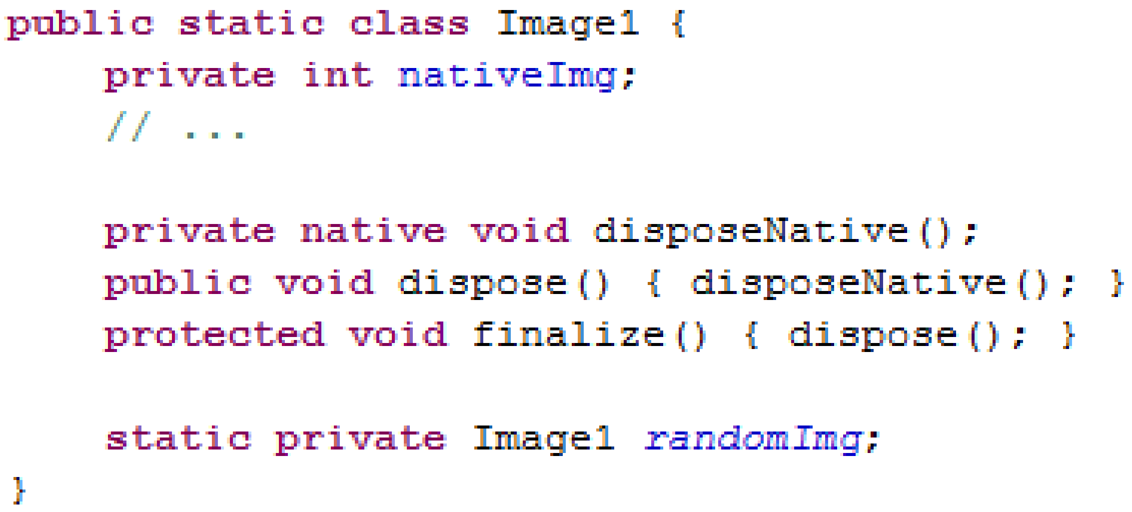
- **old and permanent generation**:
  - using ***mark-sweep-compact*** algorithm
  - allocation can use *bump-the-pointer* technique



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

Finalizable objects

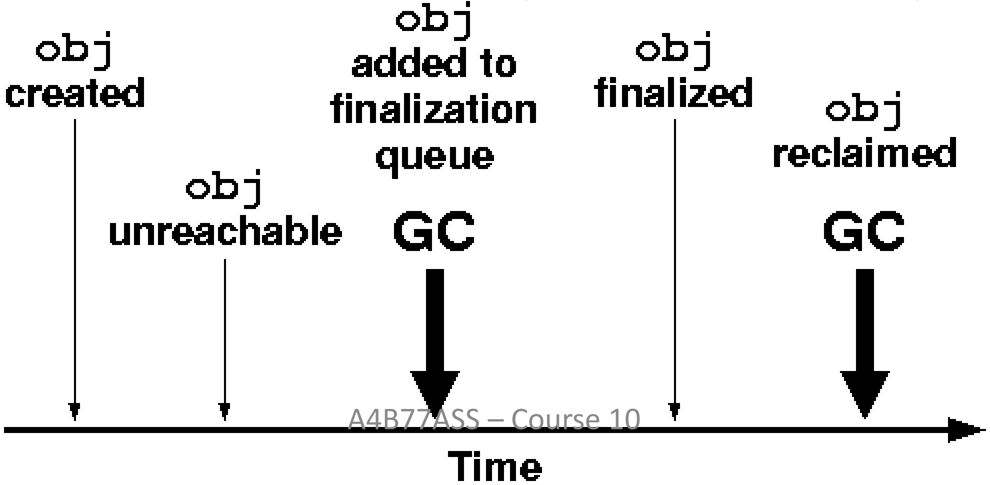
- » have a non-trivial **finalize()** method
- » finalize hook
- » used for clean-up for **unreachable object**, typically reclaim native resources:
  - GUI components
  - file
  - socket



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

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

    static private Image1 randomImg;
}
```

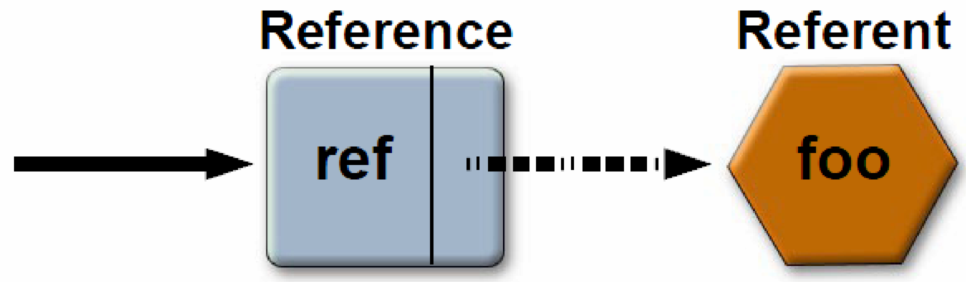
Finalizable objects

- finalizable object **allocation**:
  - slightly 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 no control of priority (one queue sequence of all finalizable objects)
- finalizable objects occupy memory longer along with ***everything reachable from them*** !!!
- implementation based on references (see Finalizer class)



Reference objects

- » mortem hooks
- » are more **flexible** than finalization
- » <u>reference types </u>(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);
```

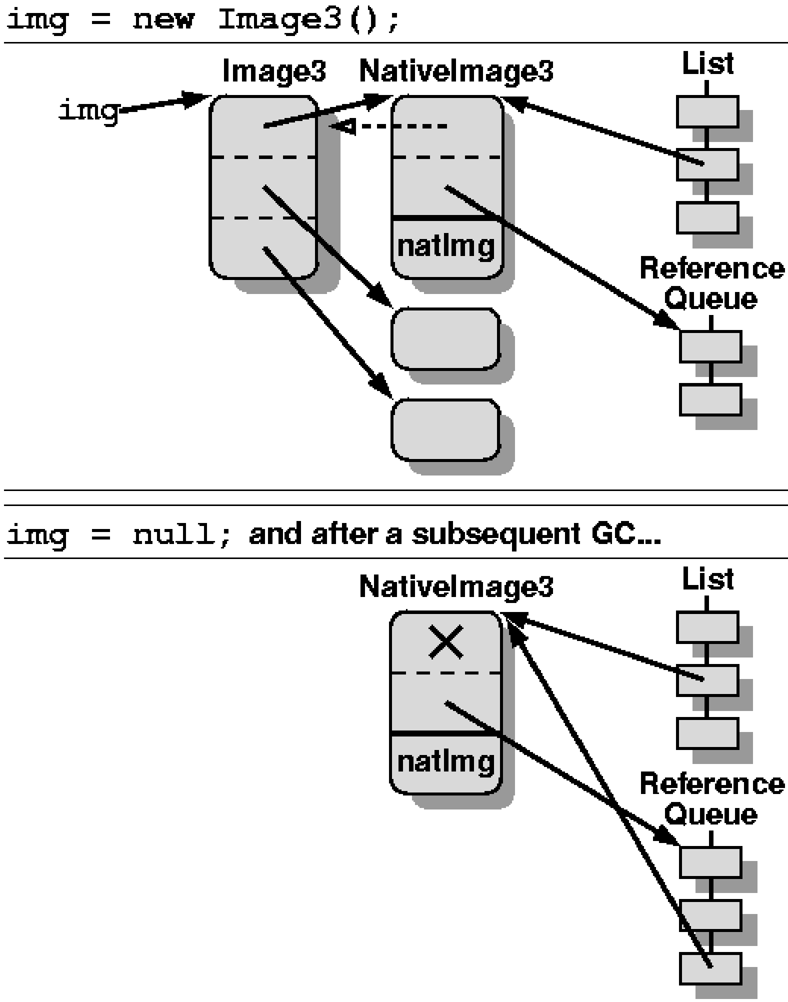
Weak reference

- 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

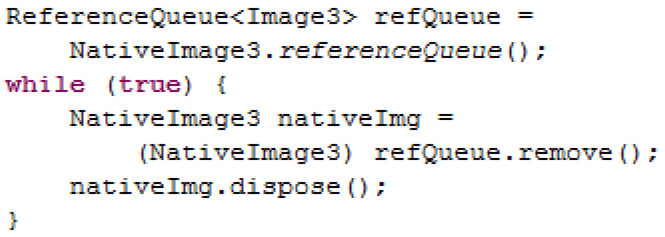
» NativeImage3 cannot be inner non-static class (due to strong ref)

```
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();
}
```

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

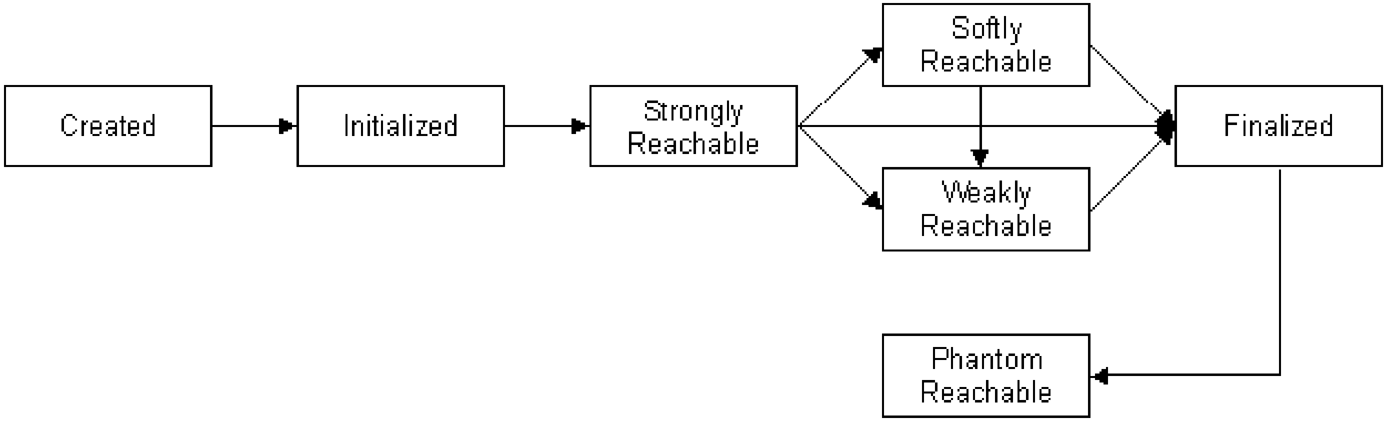
Soft reference

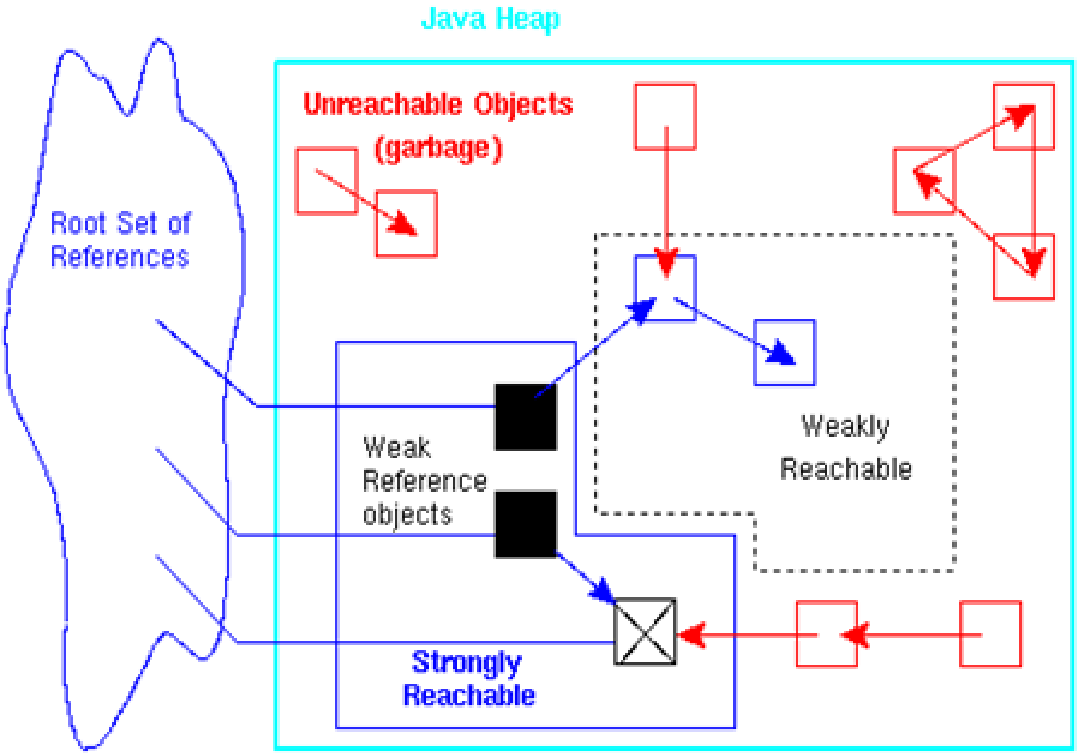
- pre-finalization processing
- usage:
  - **would like to keep referent, but can loose it**
  - reclaim only if there is “memory pressure” based on heap usage
  - suitable for caches – create strong reference to data required to keep, best for large objects
  - all are cleared before OutOfMemoryError
- 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

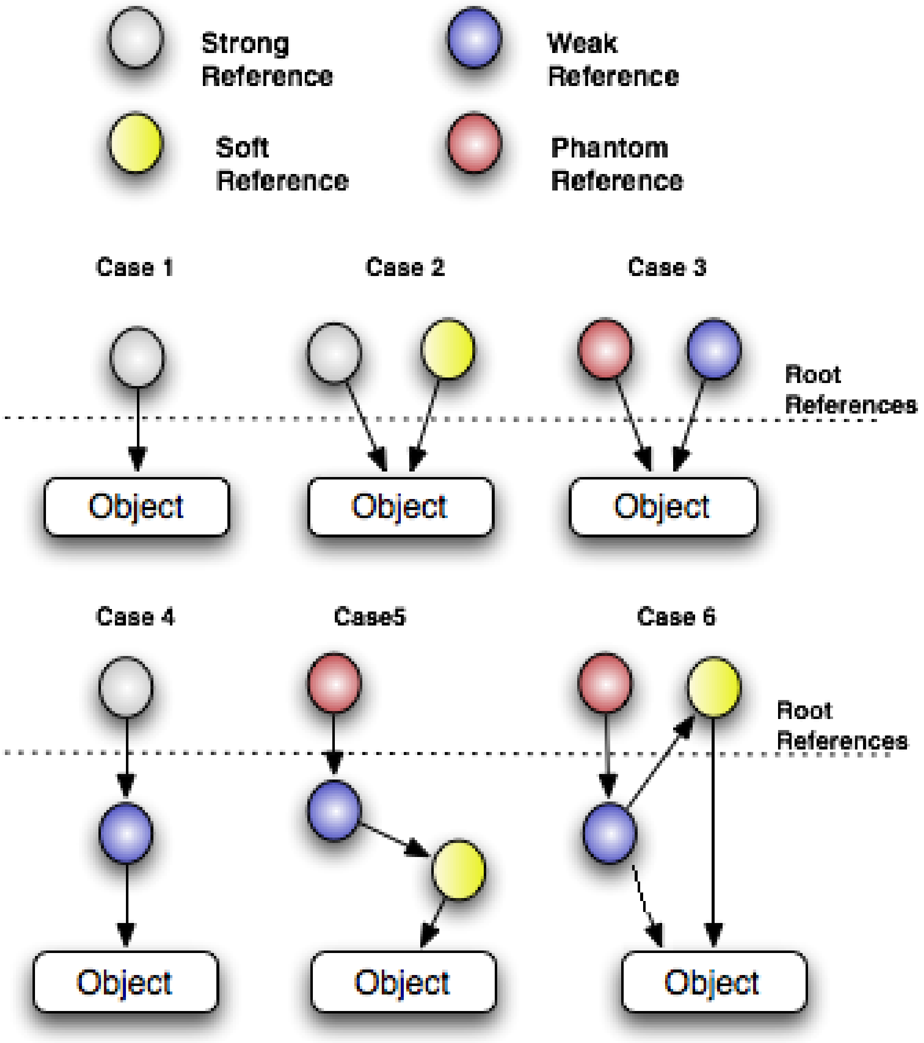
- post-finalization processing
- usage:
  - **notifies that the object is no longer used**
  - keep some data after the object becomes finalized
- get() returns:
  - null always
- **have to** specify reference queue for constructor
- **referent is not collected** until all phantom references are not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

Reachability of an object



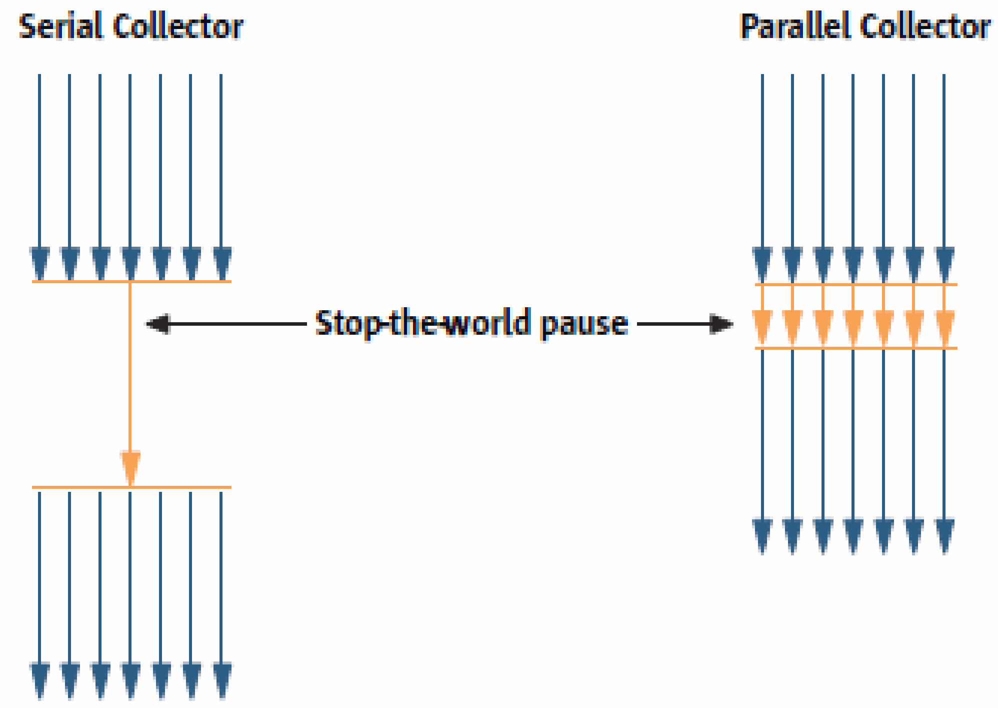


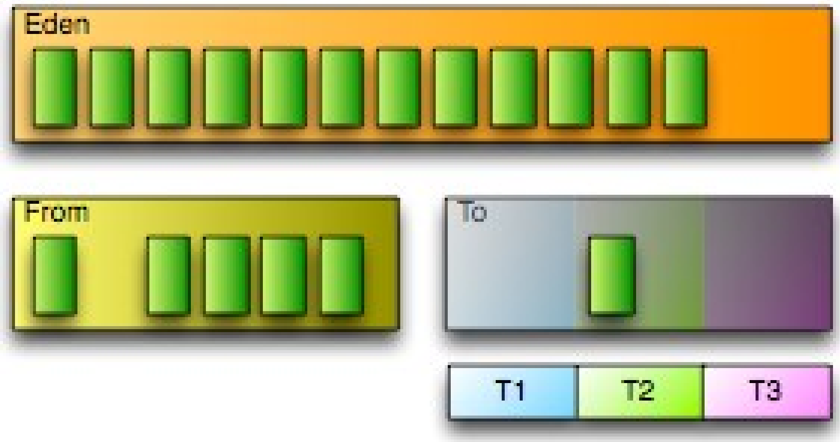
Reachability of an object



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



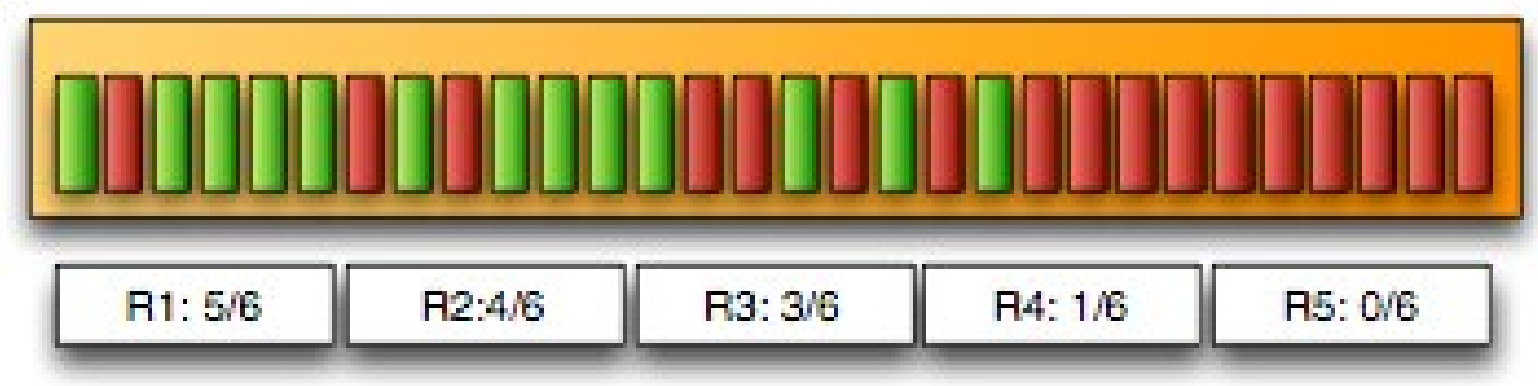


- default for **server JVM** from Java 5.0 or when requested by

  `-XX:+UseParNewGC` or `-XX:+UseParallelGC`
- the number of threads controlled by `-XX:ParallelGCThreads=n`

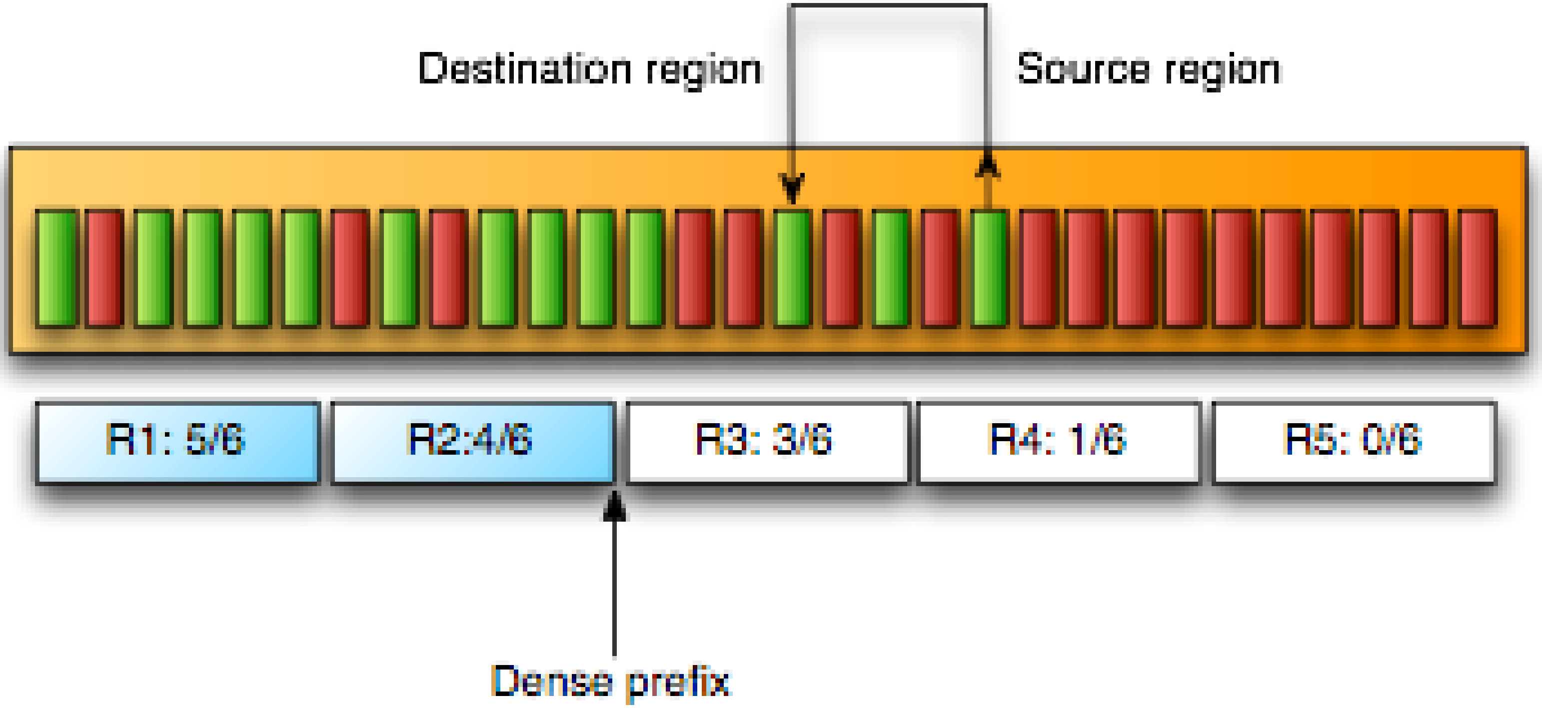
Parallel major compacting collector

- 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



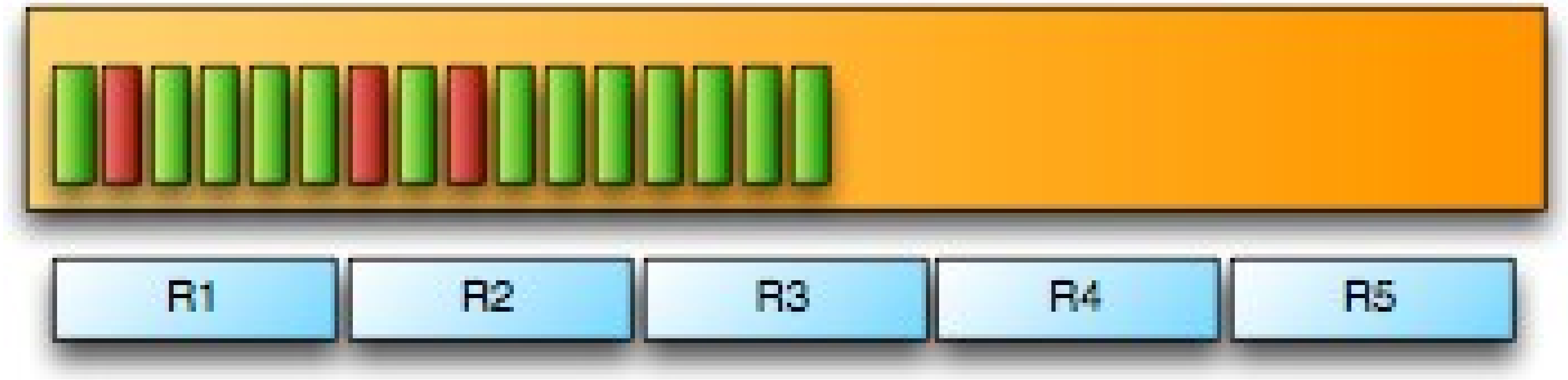
Parallel major compacting collector

- ***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`