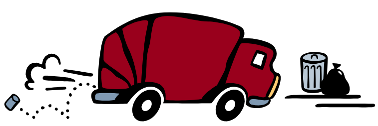
Garbage Collection in Java

this slide usually left blank

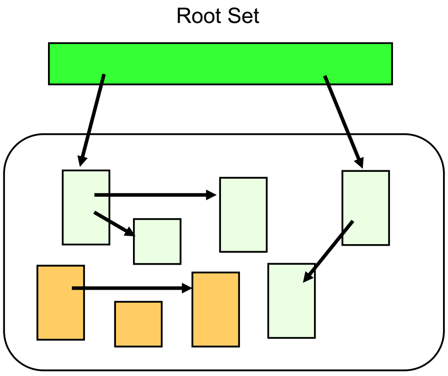
Garbage Collection

- Automatic detection and reclaiming of unused heap memory
- Advantages
  - reduces likelihood of memory leak
  - reduces likelihood of crash due to premature freeing of memory
  - generally simplifies code
- Disadvantages
  - performance overhead
  - usually deals only with memory, not other resources



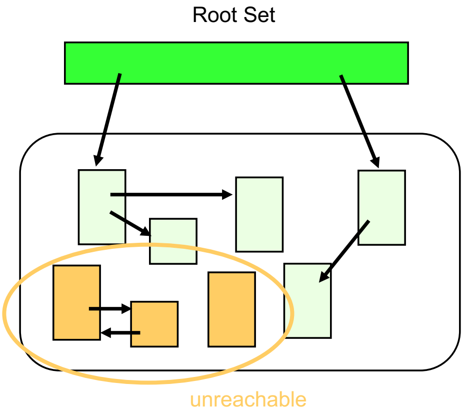
How a Garbage Collector Works

- Start with a set of "root" references
  - perhaps vars in each stack frame
- Determine "live" objects from the root set
  - "reachability"
- Conservative collector may not find all unreachable objects
  - may not be able to detect all object references
  - may be hard to differentiate between ref and int



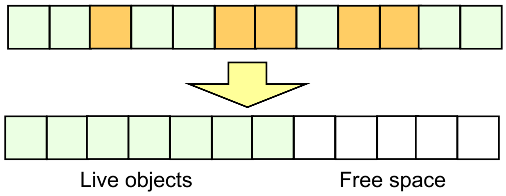
"Mark and Sweep" Algorithms

- Generally accurate
- Trace object graph from root reference
  - mark each object as "reachable"
- All non marked objects may be collected
- More performance overhead
  - two or more passes through the heap
  - application paused while collector runs

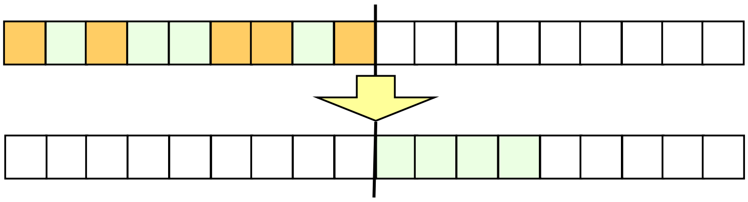


Compacting and Copying

- Compacting moves all free space to end of heap

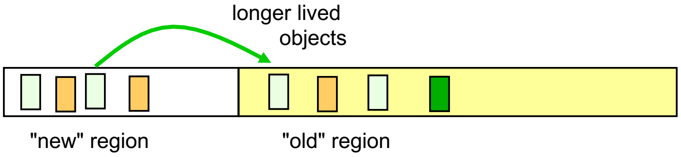


- Copying moves all live object to another area of heap

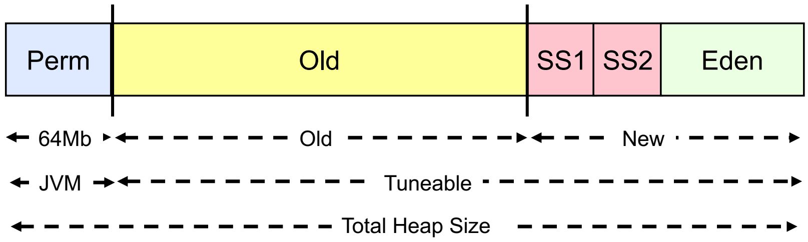


Generational Garbage Collection

- Most objects are short lived
  - "infant mortality"
- Allocate new objects from one region of the heap
  - use fast garbage collector regularly
- Move longer lived objects to another region
  - garbage collector runs less often here
  - can use more effective (or slower) algorithm



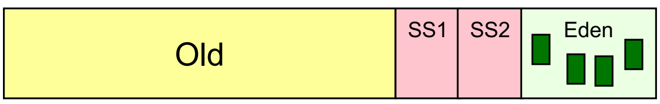
Heap Organisation in the HotSpot JVM



- Permanent section used for reflective data
  - class, method objects
- New Objects allocated from Eden
  - SS1 and SS2 are used in for copying objects
  - "Survival Spaces"
- Different algorithm used for old region

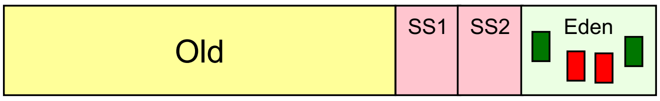
Managing the New Region

- Objects allocated in Eden space



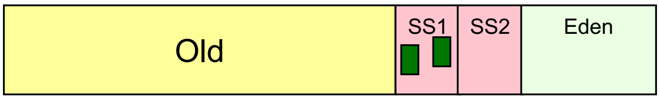
Managing the New Region

- Objects become unreachable



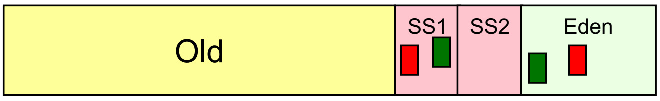
Managing the New Region

- Minor collection copies live objects from Eden to SS1
  - Eden space now emptied



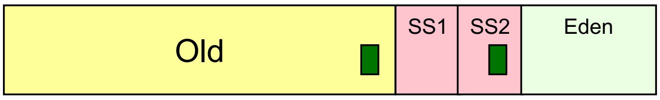
Managing the New Region

- Each new object allocated in Eden Space
  - may become unreachable
  - object(s) in SS1 may become unreachable



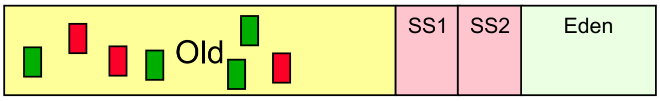
Managing the New Region

- Surviving objects moved to SS2
  - from Eden space and SS1
  - SS1 and Eden space cleared



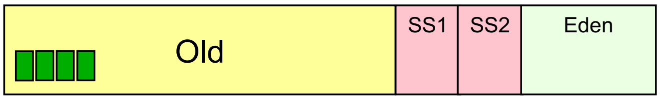
Managing the Old Region

- Major collection
  - mark and compact



Managing the Old Region

- Major collection
  - mark and compact
  - much slower than minor collection

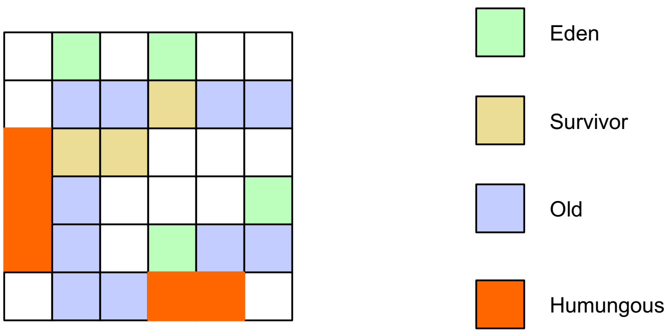


G1

- Garbage First Collector
  - experimentally available in Java 6
  - fully available in Java 7
- Architecture aimed to minimise GC pauses
  - tunable threshold to manage this
  - default "max pause" is 200ms
- Partly concurrent
  - stop the world phases kept small
- Generational

G1

- Generations still supported



- Humungous region for large object allocation
- Enable using `–XX:+UseG1GC` runtime flag

Monitoring Garbage Collector Performance

- Applications have different object usage patterns
  - garbage collector may require tuning
- Use the `-verbose:gc` flag when running program
  - reports statistics on each run of the collector

Minor Collection

```
[GC 707K->432K(1984K), 0.0045157 secs]
[GC 944K->943K(1984K), 0.0081382 secs]
[GC 1455K->1423K(1984K), 0.0078742 secs]
[GC 1935K->1871K(2496K), 0.0068408 secs]
[Full GC 1871K->600K(2496K), 0.0254038 secs]
[GC 1111K->1111K(1984K), 0.0064123 secs]
[GC 1623K->1592K(2112K), 0.0070688 secs]
[Full GC 1592K->686K(2112K), 0.0261748 secs]
...
```

Major Collection

Object space before collection

Object space after collection

Total size of available heap

Time to collect

Tuning Garbage Collection

- Many aspects of garbage collection can be tuned
  - heap size (initial and maximum)
  - algorithms used
  - "ergonomics"
- Different application types will have different overall requirements
- Sensible defaults available through "client" and "server" settings

```
$ java -server MyBigServerApp
```

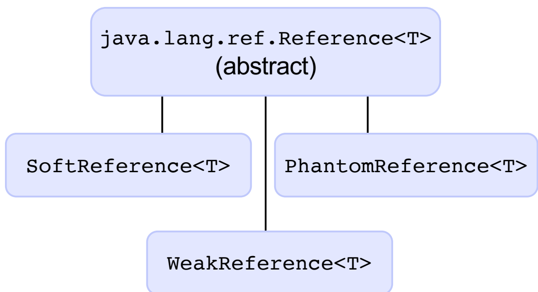
Object Finalization

- Object can have `finalize()` method
  - defined in `java.lang.Object`
  - override for specific behaviour
- Called by garbage collector
  - after object marked for reclamation
- Originally designed for resourcecleanup
  - but non-deterministic
  - generally not recommended

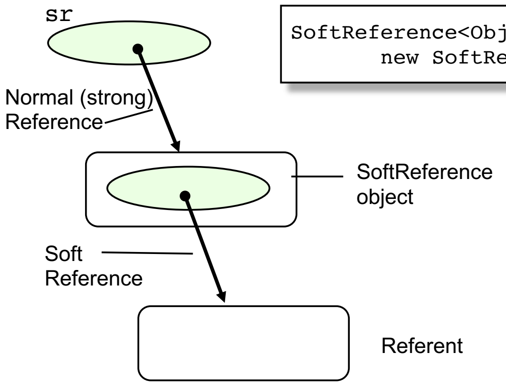
```
public class MyClass
{
  …
  public void finalize()
              throws Throwable
  {
    // Do cleanup things…
    super.finalize();
  }
}
```

Reference Objects

- Allows object reachability to be specified in stages
- Three levels of reference object
  - soft
  - weak
  - phantom
- Each carries reference to actual object
  - referent



Creating a Reference Object




```
SoftReference<Object> sr =
       new SoftReference<>( new Object() );
```

- Pass referent to constructor
  - retrieve referent using Reference Object's `get()` method
  - clear reference to referent using `clear()` method

Garbage Collection and Reachability

- Softly reachable
  - no strong references to object
  - one or more soft reference objects
  - may be reclaimed by garbage collector
- Weakly reachable
  - no strong or soft references
  - one or more weak references
  - must be reclaimed by garbage collector
- Phantomly reachable
  - no strong, soft or weak references
  - one or more phantom references
  - not resurrectable
  - last stage before object is reclaimed

Reference Queues

- Used to notify reachability state changes
  - pass queue reference to constructor for reference object

```
…
PhantomReference<Object> pr = null;
ReferenceQueue<Object> rq = new ReferenceQueue<>();
…
pr = new PhantomReference<>(obj, rq);
…
```

obj will be added to queue rq when it becomes phantomly reachable

- Monitor queue using `poll()` method
  - non blocking
- Blocking `remove()` method also available

Soft References

- Use to implement caches of objects in memory

```
…
SoftReference<MyObject> sr = null;
MyObject obj = null;
…
obj = (sr == null) ? null : sr.get();
if ( obj == null ) {
  // Load (or reload) obj…
  obj = fetchObject(…);
  sr = new SoftReference<>(obj);
}

// Work with obj

// Free the strong reference to obj
obj = null;

// obj is now Softly Reachable
```

Using Weak References

- Used to maintain "canonicalized" mappings

```
…
ReferenceQueue<Object> refQ = new ReferenceQueue<>();
Object obj = new Object();   // A strong reference
WeakReference<Object> ref = new WeakReference<>(obj, refQ);
Map<WeakReference<Object>, String> theMap = new HashMap<>();
// Associate some data with weak reference in theMap
String data = new String("Extra Data"); // A strong reference
theMap.put(ref, data);
// Check that a reference to an object was created
System.out.println("Ref object: " + ref
                                  + " with referent "
                                  + ref.get());
// Check if the Reference Object is enqueued (should be false)
System.out.println("ref.isEnqueued(): " + ref.isEnqueued());
```

```
Ref object: java.lang.ref.WeakReference@7f31245a
            with referent java.lang.Object@6d6f6e28
ref.isEnqueued(): false
```

Using Weak References

- When key is cleared, remove from map

```
// Clear the strong references to obj and data
obj = null;
if (obj == null) data = null;
// Run the garbage collector, wait for things to settle,
// and check the reference object's referent
System.out.println("*** running gc...");
System.gc(); Thread.sleep(3000);
System.out.println("contents of ref: " + ref.get());
// Check if reference object is enqueued (should be true)
System.out.println("ref.isEnqueued(): " + ref.isEnqueued());
// Retrieve reference object from refQ
Reference<?> refFromQ = refQ.poll();
System.out.println("From refQ: " + refFromQ );
// Retrieve reference object from refQ
theMap.remove(refFromQ);
```

```
*** running gc...
contents of ref: null
ref.isEnqueued(): true
From refQ: java.lang.ref.WeakReference@7f31245a
```

Phantom References

- Must have reference queue
- Indicates object is about to be reclaimed
  - specialised cleanup may be carried out

```
…
Object obj = new Object();
ReferenceQueue<Object> rq = new ReferenceQueue<>();
PhantomReference<Object> pr = new PhantomReference<>(obj, rq);
…
obj = null
…
Reference<?> r;
while ( (r = q.poll()) == null ) {
 …
}
…
// obj is now Phantomly Reachable, do cleanup…
// Now allow referent to be reclaimed and clear ref
pr.clear();
pr = null;
```