JavaOne℠

Sun's 2003 Worldwide Java Developer Conference™

Garbage Collection in the Java HotSpot™ Virtual Machine

John Coomes,
Tony Printezis

Sun Microsystems, Inc.

Presentation Goal

Help you understand garbage collection and how it interacts with Java™ programs, so you can make informed choices when designing, developing, and deploying your applications

Speaker's Qualifications

- **John Coomes** is a Staff Engineer at Sun Microsystems
- Currently works on garbage collection in the HotSpot™ virtual machine
- Has worked on various parts of Java™ 2 Platform, Standard Edition™ for~5 years; 3 years on HotSpot
- Enjoys mountain biking

Speaker's Qualifications

- **Tony Printezis** is a member of the Java™ Technology Research Group at Sun Microsystems Labs
- Previously a faculty member in the Dept. of Computing Science at the University of Glasgow, Scotland
- Has been working on GC for ~5 years; wrote the first version of the mostly-concurrent collector
- Doesn't enjoy mountain biking

What We're Trying to Sell...

Garbage collection is your friend

Finalization is not

Agenda

- Automatic memory management—what, why
- Garbage collection characteristics
- Garbage collection in HotSpot
- GC-friendly programming
- Q&A

What Is Automatic Memory Management?

- Object allocation
  - **`new`** operation
- Garbage collection (GC)
  - Reclaim unused memory
  - Class unloading
  - Weak reference processing, finalization
  - Layout of object heap

Why Automatic Memory Management?

- Makes programs simpler
  - Removes need for explicit deallocation
  - Prevents memory leaks
  - Simplifies interface to data types
  - Enables proper encapsulation
- Enables programming language safety
  - Prevents dangling pointers

Agenda

- Automatic memory management—what, why
- **Garbage collection characteristics**
- Garbage collection in HotSpot
- GC-friendly programming
- Q&A

GC Design Goals

- Fast, fast, fast allocation
- Prompt reclamation of unused memory
- Minimal disruption of running application
  - Small, predictable pauses
- Low overhead (space, time)
- Scalable to large object heaps

GC ‘Facts of Life’

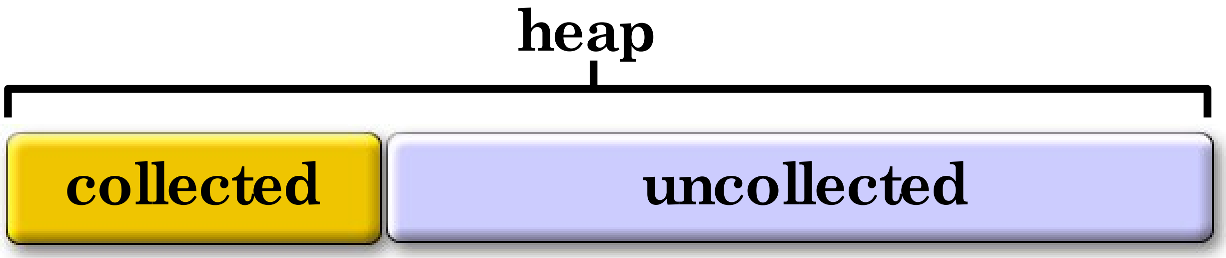
- Algorithm and policy trade-offs
  - Heap size vs. collection time overhead
  - Pause time vs. application throughput
- Since one size doesn’t fit all...
  - Offer choices
  - Use adaptable, self-adjusting algorithms

GC Characteristics

- Partial collection vs. full collection
- Stop-the-world vs. concurrent
- Serial vs. parallel

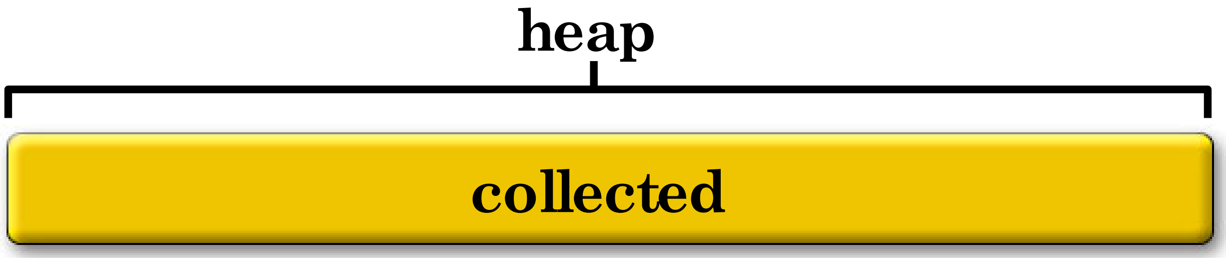
Partial Collection vs. Full Collection

- Partial collection
  Collect sub-regions of heap independently
  - \+ Exploit object lifetimes, garbage densities
    - More memory reclaimed per unit of GC work
  - – Must track references into collected area
    - Write barrier
      - Slight impact on execution time
    - JIT compiler must cooperate



Partial Collection vs. Full Collection

- Full collection
  Collect entire heap each time
  – Pause times longer
    - Proportional to heap size
  + No write barrier, simpler implementation



Stop-the-World vs. Concurrent

- Stop-the-world GC
  All GC work done while application stopped
  - \+ Object graph frozen during GC
  - – Longer pauses
- Concurrent GC
  Most GC work done while application running
  - – Object graph changing during GC
    - Synchronization required, more complex
  - – Footprint may be larger ("floating garbage")
  - \+ Shorter pauses

Serial vs. Parallel

- Serial
  - \+ Simpler algorithms
  - – Extra CPUs (> 1) idle during GC
  - – GC longer in wall-clock time
- Parallel
  - – Synchronization required, more complex
  - \+ Uses multiple CPUs for GC
  - \+ GC shorter in wall-clock time

Agenda

- Automatic memory management—what, why
- Garbage collection characteristics
- **Garbage collection in HotSpot**
- GC-friendly programming
- Q&A

GC in HotSpot

- Fast allocation
- Partial collections
    - Generational
- Stop-the-world, or mostly concurrent[1]
- Serial, or parallel[1]

[1] Available, but not enabled by default

Fast Allocation

- New objects allocated in “nursery”
  - Nurseries are often thread-local
  - Large objects may be allocated elsewhere
- Allocation: update a single pointer
  - Usually inlined by compilers
- **`new java.lang.Object()`**
  is about 10 native instructions
- Fast allocation enabled by GC

Stop-the-World GC in HotSpot

- VM tracks thread state
  - Running compiled bytecodes
  - Interpreting bytecodes
  - Running native code
  - Waiting on lock
- Only threads executing bytecodes are stopped
  - Threads in native code continue to run
- Some collection work can be concurrent

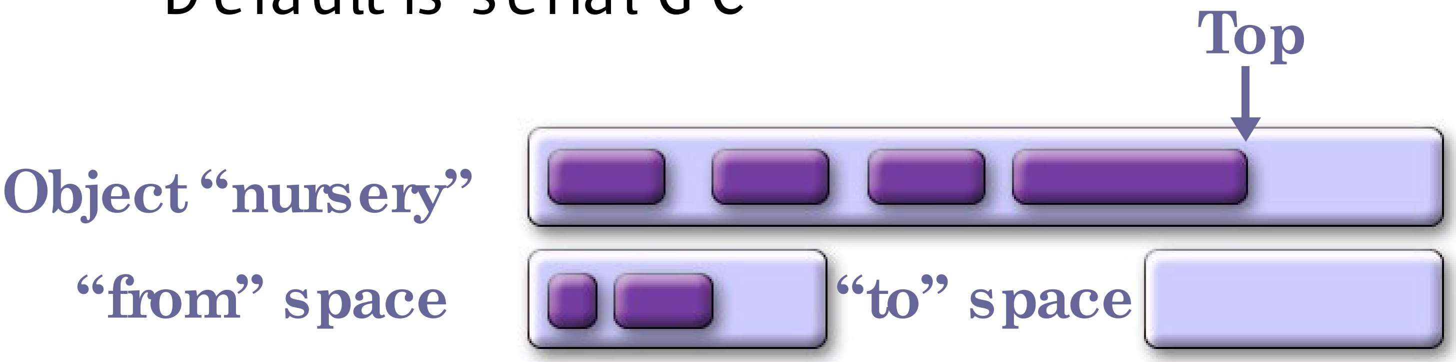
Partial GC in HotSpot

Heap divided into “generations”

- Weak generational hypothesis:
  - Most objects are short-lived
  - Few references from old to young objects
- Young generation: nursery
  - Frequent collections
  - Shorter duration
- Old generation: objects that survive one or more GCs
  - Infrequent collections
  - Longer duration

Young Generation in HotSpot

- Most objects are short-lived
  - Born, live, die in young generation
- Copying collector
  - Most efficient when garbage ratio is high
- Serial and parallel collection policies
  - Default is serial GC



Top

Object “nursery”

“from” space

“to” space

Parallel GC in HotSpot

- Parallel young generation collection
  - Takes advantage of multiple CPUs
  - Improves throughput, pause times
  - Scales to large heap sizes
  - Load balancing done via work stealing
  - Customer quote:
    "The best VM enhancement I've seen in years"
- Enabled with -XX:+UseParallelGC
- Old generation collection done serially

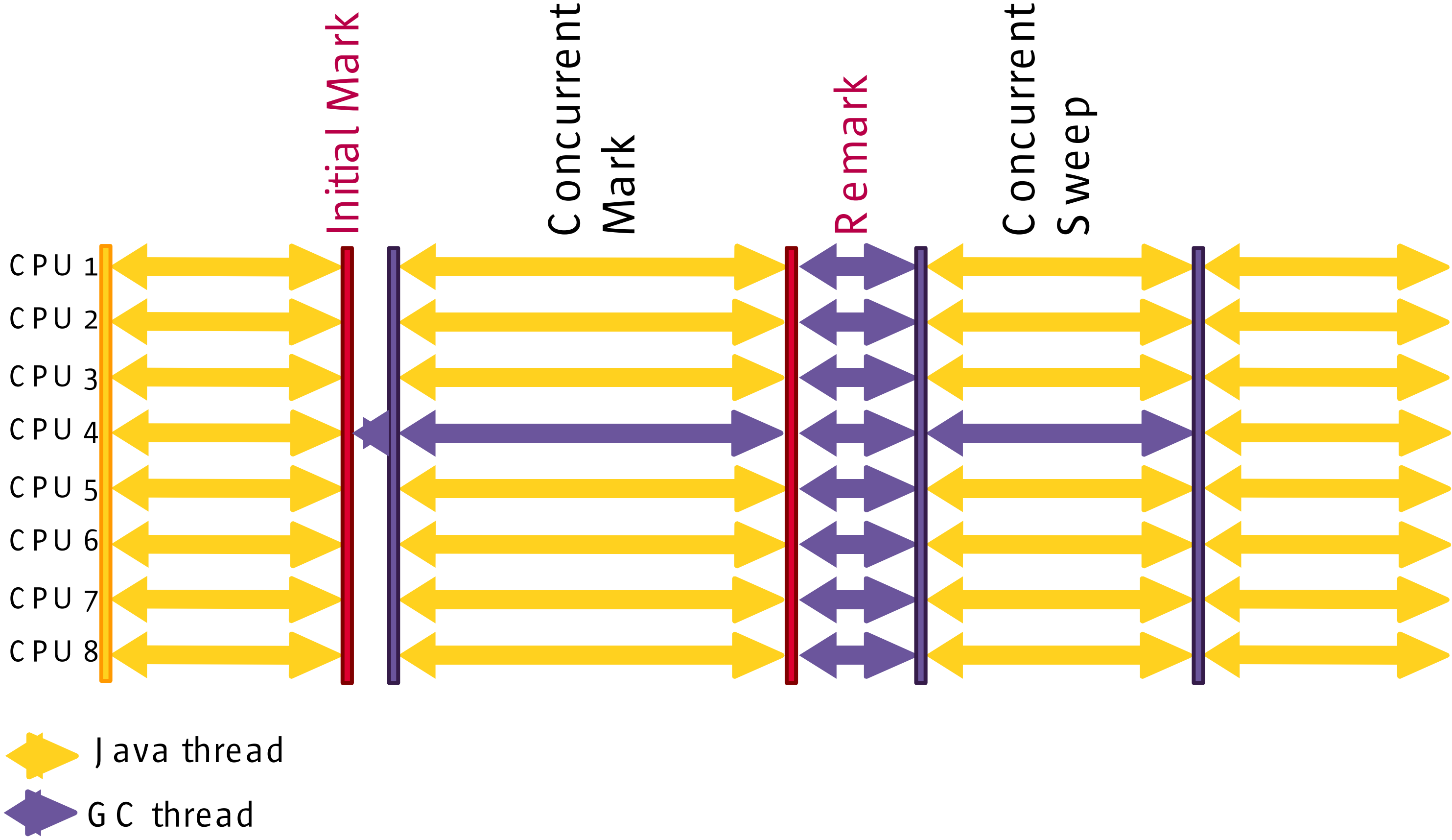
Demo

Concurrent Collection in HotSpot

- Mostly-concurrent old generation collector
  - Concurrent Mark-Sweep, or CMS
  - Bulk of GC done while application running
    - Short pause times
  - Old generation objects not moved
    - Allocation more than bumping a pointer
- Enabled with -XX:+UseConcMarkSweepGC
- Young generation GC
  - Parallel used by default if CPUs available

Concurrent Collection in HotSpot

Concurrent Mark Sweep Phases



Demo

Future Enhancements

- GC tuning needed for best performance
  - Moving toward self-tuning VM
  - Tuning guide available
    - http://java.sun.com/docs/hotspot/
- Better observability
  - JSR 174 Monitoring and Management
- Concurrent collector
  - More parallelism, more concurrency
  - Shorter pauses

Agenda

- Automatic memory management—what, why
- Garbage collection characteristics
- Garbage collection in HotSpot
- **GC-friendly programming**
- Q&A

GC-Friendly Programming

- Finalization
- Object pools
- Other things to consider

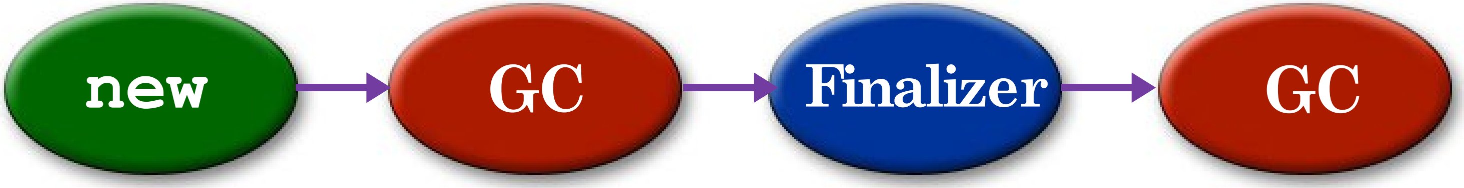
What Is Finalization?

- A cleanup hook for external resources
  - File descriptors
  - Native GUI state
- A class overrides
    - `protected void finalize() { ... }`
  - At some *unspecified* time after object becomes unreachable
  - **`finalize()`** *may* be invoked

How Does Finalization Work?

Each instance

1. Is registered when allocated
2. Is enqueued when it becomes unreachable
3. Has the `finalize()` method invoked
4. Becomes unreachable again
   (if resurrected in step 3)
5. Has the storage reclaimed



Demo

Impact of Finalization

- Execution speed
  - Slower allocation
  - Finalizer thread affects scheduling
- Heap size
  - Memory retained longer
- Collection pauses
  - Discovery and queuing

GC-Friendly Finalization

- Use for cleanup of external resources
- Suggestions
  - Limit the number of finalizable objects
  - Reorganize classes so finalizable object holds no extra data
  - Beware when extending finalizable objects in standard libraries
  - GUI elements, nio buffers
- Alternatives
  - Use **`java.lang.ref.WeakReference`** without finalizers

Object Pools

- Manual memory management
  - Allocation serialized
  - Current collectors support parallel allocation
- Data is kept artificially alive
  - Adds pressure on garbage collector
- Breaks down abstract data types
  - Who is responsible for instances?
- Use only if allocation or initialization is expensive

Object Pool Example

-
  ```
  class Node {
    private static Node head = null;
    private Node next;

    public static synchronized Node allocate() {
      if (head == null) return new Node();
      Node result = head;
      head = head.next;
      return result;
    }
    public static synchronized void free(Node n){
      n.next = head;
      head = n;
    }
    ...
  }
  ```

Real-World Problem

- Object pools never truncated
- Peak live data ~300MB
- Average live data ~100MB
- Problem
  - Other garbage generated from libraries
  - GCs less frequent, but dealt with 300MB
- Solution
  - Removed object pools; application ran faster

Other Things to Consider

- Size heap appropriately
  - Maximum should be larger than working set … but smaller than available physical memory
  - Leave room for the system to adapt
- Avoid **`java.lang.System.gc()`**
- Try different collection algorithms
  - -XX:+UseParallelGC
  - -XX:+UseConcMarkSweepGC

Summary

- GC simplifies Java™ programs
- Several types of GC
  - Serial, parallel, concurrent, ...
  - Each suited to a subset of applications
  - HotSpot provides choices
- Avoid finalization, object pooling
  - Use only as a last resort

What We Hope You Bought …

- Garbage collection is your friend
  - Or at least it can be

What We Hope You Bought …

- Garbage collection is your friend
  - Or at least it can be

- Finalization is not
  - Or at least it may not be (you just don’t know!)

Q&A

JavaOne℠

Sun's 2003 Worldwide Java Developer Conference

Java™

java.sun.com/javaone/sf