Sustainable Memory Use

Allocation & (Implicit) Deallocation (mostly in Java)

Often called “Garbage Collection”

Copyright 2019, Keith D. Cooper & Zoran Budimlić, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.
Preliminaries

Today’s lecture

Automatic memory management

• Implicit Deallocation of Heap-Allocated Objects
  → Reference Counting
  → Copying Collectors
  → Short-Pause Collectors

• How you should program defensively now that you know

Second part: Java memory model, allocation, & recycling

• How it works
• How it affects your code’s runtime
• How you should program defensively now that you know
Where do objects live in Java?

COMP 140 & COMP 215

- Students encouraged to ignore the issue of where objects, variables, and methods live
- The implementation (Python or Java) takes care of these details
- Fundamentally, abstraction is a good thing — right up to the point where it causes problems
- At some point in your Java career, performance will matter
  - COMP 412 labs
  - At that point, you need to pay attention to details
- Today’s lecture is about details
Where do objects live?

The Java System maps Java World onto Processor Resources

- Processor has finite resources
- Java suggests that you have “enough” resources
- Mapping “enough” onto “what’s there” is the job of the Java compiler and runtime (JVM)

Knowing how that mapping works can help you understand the behavior of your programs, and suggest ways to improve the program’s behavior.
In the example, what needs storage?

- The two classes (Point & C)
- Point’s local members (x, y, & draw)
- C’s local members (s, t, & m)
- m’s local variables (a, b, & p)

A classic example
In the example, what needs storage?
- The two classes (Point & C)
- Point’s local members (x, y, & draw)
- C’s local members (s, t, & m)
- m’s local variables (a, b, & p)

Memory in the Java runtime is divided, broadly speaking, into a **Heap** and a collection of **Stacks**
- One heap per program (large)
- One stack per thread (smaller)
JVM Memory Layout

Conceptually, Java memory is laid out along these lines

- When running code creates a variable, it goes into the thread’s stack
- When running code creates a class or an object (e.g., with a `new`), it goes into the heap
- Code lives off to the right (might consider it part of the heap)

So, can a program run out of heap space? (too many `new`s)
⇒ Yes. Emphatically yes

What happens?
⇒ The runtime system tries to recycle space on the heap
Sustainable Memory Management

When the heap runs out of space, the system copes

• Scours the heap looking for objects that are no longer of interest
  – Technical term is “live”
  – An object is considered live iff it can be reached from the running code

• Start from all the names in the running code
  – Variables are on the stack\(^1\)
  – Global names such as declared or imported classes
  – Each object on the stack has a declaration which reveals its structure
  – You can imagine chasing down chains of references to find all live objects\(^2\)
    
      → That’s how it was done for a long time ...

---

\(^1\) Locals of the current method are on the stack. Locals of the method that called it are below the current method on the stack. Locals of the method that called that method are below ..., and so on. That’s why the runtime uses a stack

Garbage Collectors

- Reference Counting
  - Stop-the-World
    - Mark-and-Sweep
      - Basic
    - Mark-and-Compact
      - Baker’s
      - Basic
      - Cheney’s
  - Trace-Based
    - Short-Pause
      - Incremental
      - Partial
    - Generational
    - Train
Reference Counting

The basic idea behind reference counting is simple.

- At runtime, the code maintains, for each heap-allocated object, a count of the number of copies of its address that exist in program variables.
- The code updates the counts at each assignment to a reference.
- If, after an assignment, one of the pointers involved points to an object with a reference count of zero, that object is freed.
- Adds overhead to assignment, but makes heap maintenance simple.

Pros

- Simple implementation
- Conservative behavior
- Easy to adapt to real-time operation

Cons

- Fragmentation is a problem
- Self-referencing (cyclic) garbage
- Overhead per pointer modification
- Locality effects
- Memory overhead

Python relies on reference counting.
What happens when the running code assigns a different pointer to the root object?

Pros
- Simple implementation
- Conservative behavior
- Easy to adapt to real-time operation

Cons
- Fragmentation is a problem
- Self-referencing (cyclic) garbage
- Overhead per pointer modification
- Locality effects
- Memory overhead
What happens when the running code assigns a different pointer to the root object?

Pros
- Simple implementation
- Conservative behavior
- Easy to adapt to real-time operation

Cons
- Fragmentation is a problem
- Self-referencing (cyclic) garbage
- Overhead per pointer modification
- Locality effects
- Memory overhead
What happens when the running code assigns a different pointer to the root object?

**Pros**
- Simple implementation
- Conservative behavior
- Easy to adapt to real-time operation

**Cons**
- Fragmentation is a problem
- Self-referencing (cyclic) garbage
- Overhead per pointer modification
- Locality effects
- Memory overhead
What happens when the running code assigns a different pointer to the root object?

**Pros**
- Simple implementation
- Conservative behavior
- Easy to adapt to real-time operation

**Cons**
- Fragmentation is a problem
- Self-referencing (cyclic) garbage
- Overhead per pointer modification
- Locality effects
- Memory overhead

At this point, every object has a non-zero reference count, so collection stops.
Taxonomy

Garbage Collectors

Reference-Counters

Trace-Based

Stop-the-World

Mark-and-Sweep

Basic

Baker’s

Mark-and-Compact

Basic

Cheney’s

Short-Pause

Incremental

Partial

Generational

Train
Mark and Sweep

Mark and Sweep is a two pass algorithm
• Pass 1 finds and marks every allocated object
• Pass 2 frees every object that is not marked in pass 1

The Basic Mark Algorithm
1. Mark all objects as **Unreached**.
2. Start with the root set. Put them in state **Unscanned**.
3. **while** **Unscanned** objects remain **do**
   
   examine one of these objects, \( x \);
   
   set \( x \)’s state to **Scanned**;
   
   for each object that \( x \) references;
   
   if it has not already been scanned, add it to **Unscanned**;

   **end**;

States of an object during the algorithm
1. **Free** = not holding an object; available for allocation.
2. **Unreached** = Holds an object, but has not yet been reached from the root set.
3. **Unscanned** = Reached from the root set, but its references not yet followed.
4. **Scanned** = Reached and references followed.

To track each object’s state, we need space in the object for a mark that has “enough” bits.
Mark and Sweep

Mark and Sweep is a two pass algorithm

- Pass 1 finds and marks every allocated object
- Pass 2 frees every object that is not marked in pass 1

The Basic Sweep Algorithm

1. For each object in the **Unreached** state, mark it as **Free** & add it to the free list.

2. For each object in the **Scanned** state, mark it as **Unreached** to prepare for the next mark-and-sweep cycle.

States of an object during the algorithm

1. **Free** = not holding an object; available for allocation.
2. **Unreached** = Holds an object, but has not yet been reached from the root set.
3. **Unscanned** = Reached from the root set, but its references not yet followed.
4. **Scanned** = Reached and references followed.

To track each object’s state, we need space in the object for a mark that has “enough” bits.

This approach handles cyclic structures correctly.
Mark and Sweep

Improving the Performance of Mark and Sweep

• **Problem**: Mark and Sweep takes time proportional to the heap size.
  – It must visit all objects to see if they are *Unreached*.

Baker’s Algorithm

• Baker’s algorithm keeps a list of all allocated chunks of memory, as well as the *Free* list.

• **Key change**: In the sweep, look only at the list of allocated chunks.

• Those that are not marked as *Scanned* are garbage and are moved to the *Free* list.

• Those in the *Scanned* state are put in the *Unreached* state.
  – For the next collection.
Garbage Collectors

Reference-Counters

Stop-the-World

Mark-and-Sweep

Mark-and-Compact

Basic

Baker’s

Cheney’s

Trace-Based

Short-Pause

Incremental

Partial

Generational

Train
Mark-and-Compact Collectors

Why Compact the Heap?

• Reduce fragmentation of the free space
  – Enables allocation of large objects
  – Shorten the free list

• Improve locality
  – Reduce the cache and TLB footprint of allocated portion of the heap

These issues should be familiar from your malloc() lab.
Mark-and-Compact Collectors

A Basic Mark-and-Compact Allocator

1. Mark reachable objects as in mark-sweep
2. Sweep objects from low end of the heap
   - Compute a new location for each object, packing densely
   - Compute a map from old to new location for each reached object \((old \rightarrow free)\)
   - Maintain a counter, \(free\), that shows the cumulative space use of reached objects \((will \ be \ the \ free-space \ pointer \ after \ compaction)\)
3. Sweep objects, again, moving them to new locations and updating all references in the objects to their new locations
   - Use the map of new locations
4. Update each of the root references
   - Use the map of new locations

Computes and records a new location for each heap object. The map translates old location to new location.

Second sweep moves all objects into the predicted locations & updates pointers in the heap.

Last phase rewrites references that are stored outside of the heap.
Example: Mark-and-Compact

The First Sweep

Initial condition:
Both pointers @ low end of the heap
Example: Mark-and-Compact

The First Sweep

First object was not reachable
Traversal pointer moves to next object
Map updated to put 2\textsuperscript{nd} object at free
Example: Mark-and-Compact

The First Sweep

free updated with object length
Traversal pointer moves to next object
Example: Mark-and-Compact

The First Sweep

3rd object not reachable
4th object is reachable
Map updated to put 4th object at free
Example: Mark-and-Compact

The First Sweep

free updated with object length
Traversal pointer moves to next object
Example: Mark-and-Compact

The First Sweep

5th object not reachable
6th object is reachable
Map updated to put 6th object at free
Sweep halts; it is out of objects
Example: Mark-and-Compact

The Second Sweep

Moves 2\textsuperscript{nd} object to new location
Updates internal pointer from map
Example: Mark-and-Compact

The Second Sweep

Moves 4th object to new location
Updates internal pointer from map
Example: Mark-and-Compact

The Second Sweep

Can improve the efficiency of the basic algorithm by having the first sweep operate directly from the reachable set computed in the mark phase.

Moves 6th object to new location
Updates internal pointer from map
Note that free now points to the heap’s contiguous block of free space.
Garbage Collectors

Reference-Counters

Stop-the-World

Mark-and-Sweep

Basic

Baker’s

Mark-and-Compact

Basic

Cheney’s

Trace-Based

Short-Pause

Incremental

Partial

Generational

Train
Garbage Collection via Copying

**Copying Collectors**

- A **copying collector** divides the heap into two or more pools
- New objects are allocated in the current pool
- When the current pool is full, execution pauses and the collector:
  - Copies all reachable objects from the current pool to the empty pool
  - Updates pointers and swaps the designations current and empty

Unreachable objects are not copied, so the new pool has free space

A copying collector trades heap space for a single sweep
- Lower overhead and better locality
Garbage Collection via Copying

Copying Collectors

- A **copying collector** divides the heap into two or more pools
- New objects are allocated in the current pool
- When the current pool is full, execution pauses and the collector:
  - Copies all reachable objects from the current pool to the empty pool
  - Updates pointers and swaps the designations current and empty

Unreachable objects are not copied, so the new pool has free space
Garbage Collection via Copying

Copying Collectors

• A **copying collector** divides the heap into two or more pools
• Key issue is the **copy** phase
  – Find objects as in the mark phase
  – Build a table of new location as objects are encountered and moved
  – Adjust references in the copy in the “new” heap

Advantages

• Sweep complexity is \( \mathcal{O}(|\text{live set}|) \)
• No issues with unreachable garbage
• Subsequent allocation is fast

Issues

• Can only use 50% of the heap
• Collection must stop execution (“stop-and-copy”)
• Objects must be implemented so as to be “moveable”
Taxonomy

Garbage Collectors

Reference-Counters

Stop-the-World

Mark-and-Sweep

Basic

Baker’s

Mark-and-Compact

Basic

Cheney’s

Trace-Based

Short-Pause

Incremental

Partial

Generational

Train
Generational Garbage Collectors

Generational Collectors capitalize on several observations

• Most objects have short lifetimes
  – Those that survive one collection are likely to survive many
• The mark phase has cost proportional to the size of the storage pool
  – Smaller pools produce faster collections

These collectors are tailored to focus their effort on regions of the heap where new objects have been recently created

• Goal is to produce a better ratio of reclaimed space to collection time
• Introduce notion of multiple-pool heap
  – Specific pools, called nurseries, for new object allocation
Generational Garbage Collectors

**Generational Collectors**

- Divide heap into partitions $P_0$, $P_1$, ..., $P_n$
  - $P_i$ holds objects older than $P_{i-1}$
- Create new objects in $P_0$ until it is full
- Collect $P_0$ with a copy collector that adds to $P_1$
- When $P_1$ is full, collect $P_0$ and $P_1$ into $P_2$
  - *This action should free space in both $P_0$ and $P_1*
- **In general**: When $P_i$ is full, collect $P_0$, $P_1$, ..., $P_i$ into $P_{i+1}$

Generational collectors have proven quite effective in practice.
Implications for Programming

If you want performance, pay attention to garbage

• Collector locates live objects by walking out from variables
  – When you are done with an object, set the variable to **NULL**
  – Leaving the reference to the heap object will keep it live

• Storage can “leak”, or become un-recyclable
  – Leave a pointer to a large data structure on the stack, or in a global, ...
    or forgotten in another object, that happens to be live
  – Leads to extra collections and, eventually, an out of memory error

This is the takeaway message!
If performance really matters, pay attention to size of the pool

- Java uses a variant of a generational copying collector
- All `new` objects are allocated into a nursery, often called Eden
- Eden is copied, when full, into one of Stable\textsubscript{0} or Stable\textsubscript{1}
- When Stable is too full, it is copied to the Long Term Pool

In COMP 412, we offered a 5% bonus for the fastest lab 1 in a language

• In the Java labs, the top three or four were separated by the behavior of the garbage collector
  • The fastest lab had no major collections, & fewer minor collections

Implications for Programming

COMP 412, Fall 2019
Does this stuff matter?

Performance of One Student’s Java Code for the Register Allocator

- Standard
- Big Heap
- Small Heap

way too many collections

major collection (2x in speed)