Habanero Extreme Scale Software Research Project
Comp215: Garbage Collection

Zoran Budimlić (Rice University)

Adapted from Keith Cooper’s 2014 lecture in COMP 215.
Garbage Collection
“In Beverly Hills... they don't throw their garbage away. They make it into television shows.”

- Woody Allen
Today’s Lecture

Focus for today is on the 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?

So far, we have encouraged you 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.
Where do objects live?

So far, we have encouraged you 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 when it causes problems.

At some point in your Java career, performance will matter.

COMP 215 exercises where your code is timed, or COMP 412.

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 this 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)
Fundamentals

In this 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)

```java
class Point {
    public int x, y;
    public void draw();
}
class C {
    int s, t;
    public void m() {
        int a, b;
        Point p = new Point();
        a =...;
        b =...;
        p.draw();
    }
}
```

A classic example
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.
Why Garbage Collection?

Today’s programs consume storage freely
1GB laptops, 1-4GB desktops, 8-512GB servers
64-bit address spaces (SPARC, Itanium, Opteron)

... and mismanage it
Memory leaks, dangling references, double free, misaligned addresses, null pointer dereference, heap fragmentation
Poor use of reference locality, resulting in high cache miss rates and/or excessive demand paging

Explicit memory management breaks high-level programming abstraction
Luckily, you don’t have to do that in Java
Sustainable Memory Management (AKA “Garbage Collection”)

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 …

Modern garbage collectors are more nuanced
They still start from the beginning: local & global names
Most modern collectors are “copying” collectors

---

\(^1\) 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

Stack on Method Invocation

SP - stack pointer
VP - variable pointer
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 live objects from the current pool to the empty pool.
- swaps the designations current and empty.
- Unreachable objects are not copied, so the new pool has free space.
Copying Collector Tradeoffs

Good: very low cell allocation overhead
   Out-of-space check requires just an addr comparison
   Can efficiently allocate variable-sized cells

Good: compacting
   Eliminates fragmentation, good locality of reference

Bad: twice the memory footprint
   Probably Ok for 64-bit architectures (except for paging)
      When copying, pages of both spaces need to be swapped in. For programs with
      large memory footprints, this could lead to lots of page faults for very little garbage
      collected
      Large physical memory helps
“Youth is such a wonderful thing. What a crime to waste it on children.”

- George Bernard Shaw
Generational Garbage Collector

Java uses a slightly more complex copying collector

All new objects are allocated into Eden

Eden is copied, when full, into one of Stable$_0$ or Stable$_1$

When Stable is too full, it is added to the Long Term Pool

This is an example of a Generational Garbage Collector

Key insight: most objects that die, die young

Divide the heap into generations, and GC the younger object more frequently

Don't have to trace all objects during a GC cycle

Periodically reap the “older generations”

“Promote” young objects to “older” pool if they survive several garbage collections

Amortize the cost across generations
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, make sure you’re not referencing it anymore

Leaving a reference to the heap object will keep it live

**Storage can “leak”, or become un-recyclable**

Leaving a reference 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!
Does this stuff matter?

If performance really matters, pay attention to size of the pool

COMP 412 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
Does this stuff matter?

Performance of One Student’s Java Code, COMP 412, Lab 1

- Standard
- Big Heap
- Small Heap

way too many collections

major collection (2x in speed)
Live Coding

Find garbage in our code!