COMP 412
FALL 2017

Sustainable Memory Use in Java Programs

Allocation & (Implicit) Deallocation in Java

Originally given in Comp 215

Preliminaries

Today's lecture

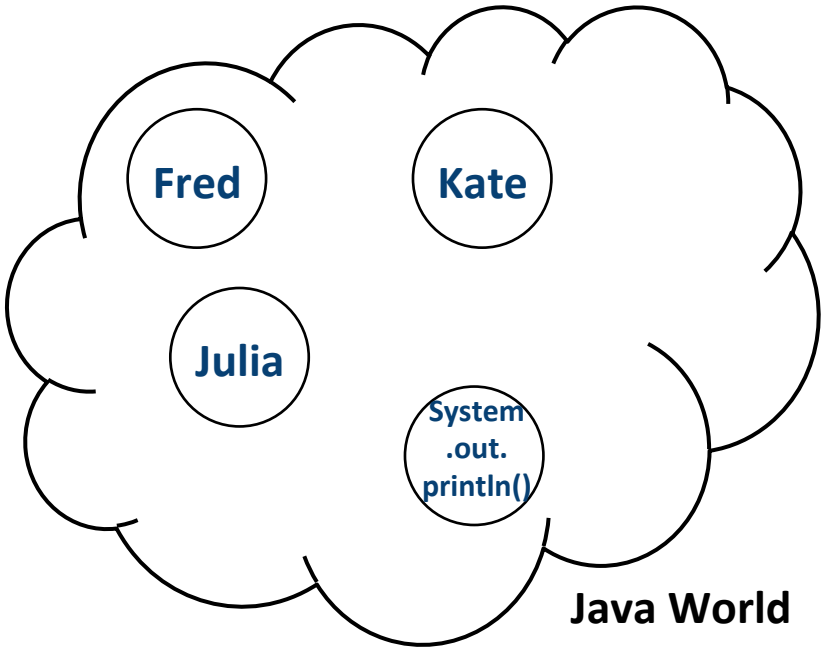
Focus for today is on the Java memory model, allocation, & recycling

→ How it works

→ How it affects your code's runtime

→ How you should program defensively now that you know

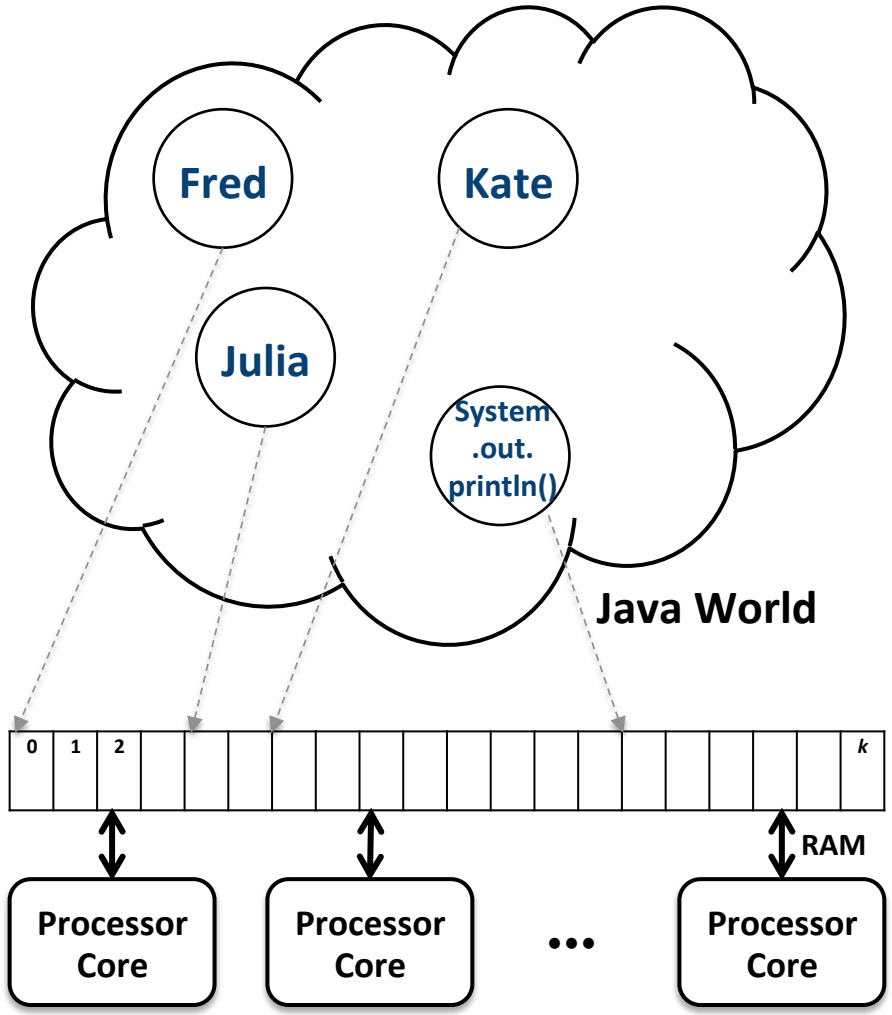
Where do objects live?



COMP 140 & COMP 215 (*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 where 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.

Fundamentals

```
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

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)

Fundamentals

```
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();
      }
}
```

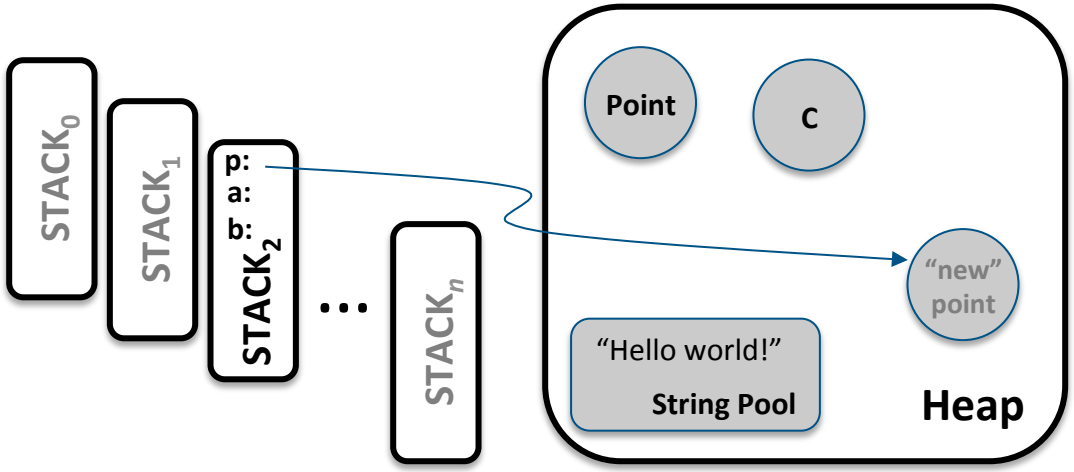
A classic example

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)

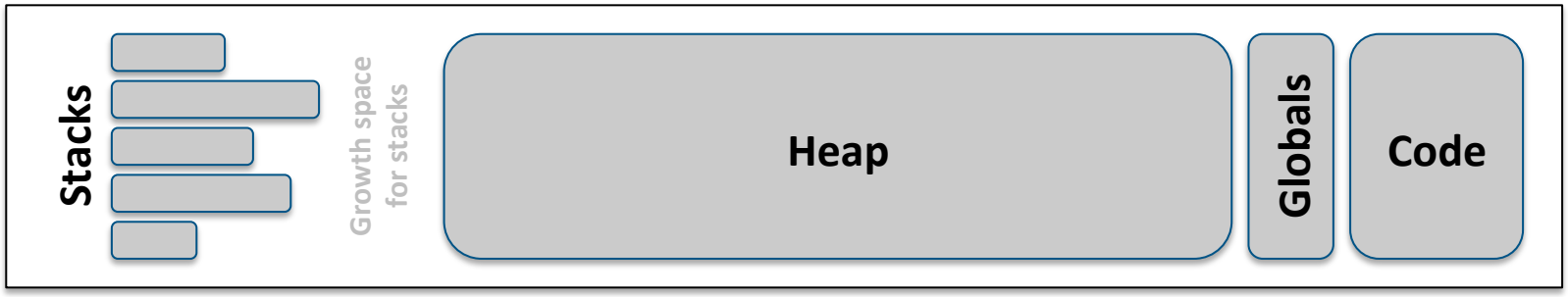
Memory in the Java runtime is divided, broadly speaking, into a **Heap** and a collection of **Stacks**

- 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 left (*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

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 …*

[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

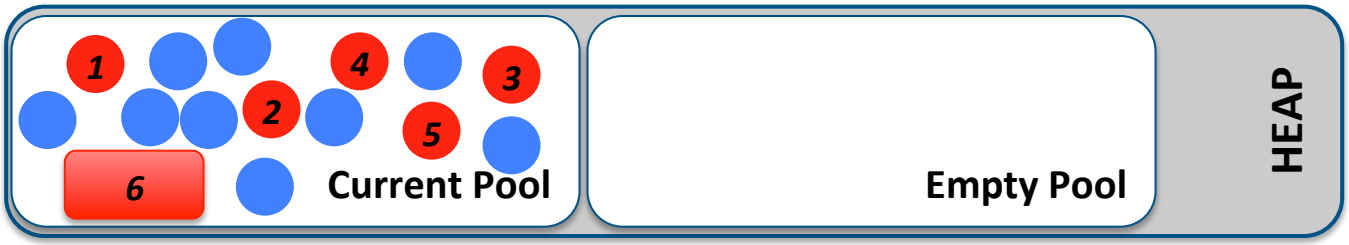
[2] H. Schorr and W.M Waite, " An efficient machine-independent procedure for garbage collection in various list structures, Comm. ACM, 10 (8), 1967, pages 501—506

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 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

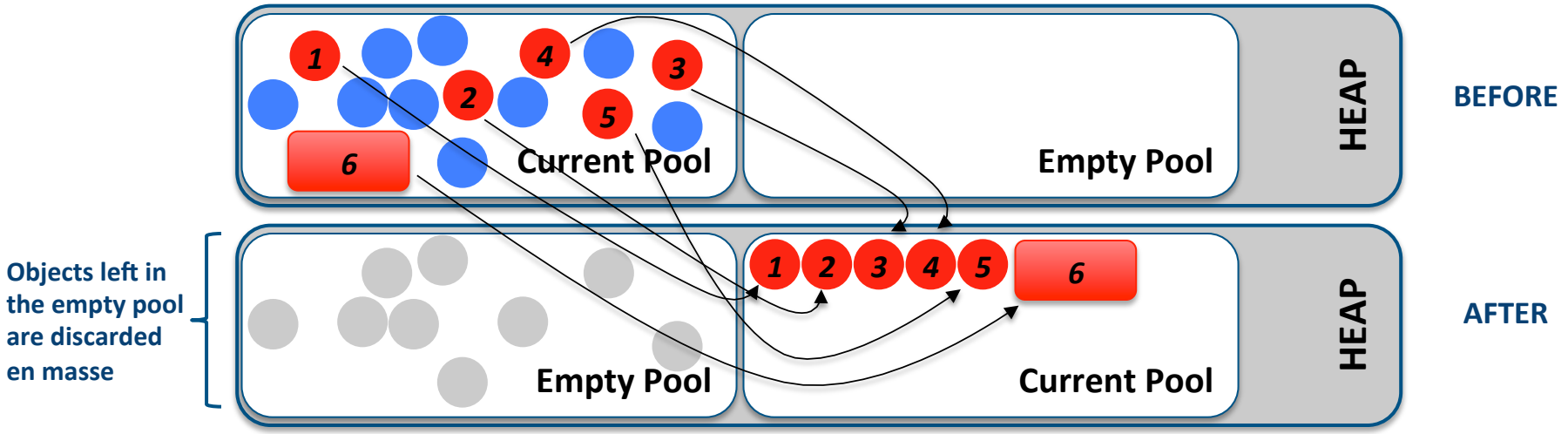


Garbage Collection via Copying

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 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



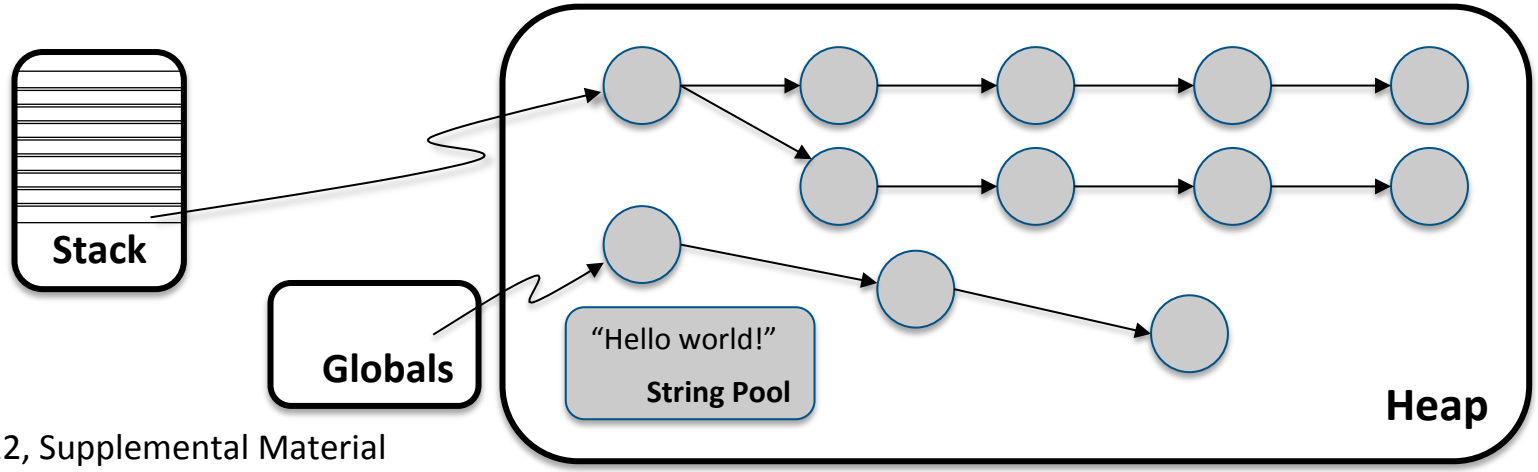
Implications for Programming

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

This is the takeaway message!

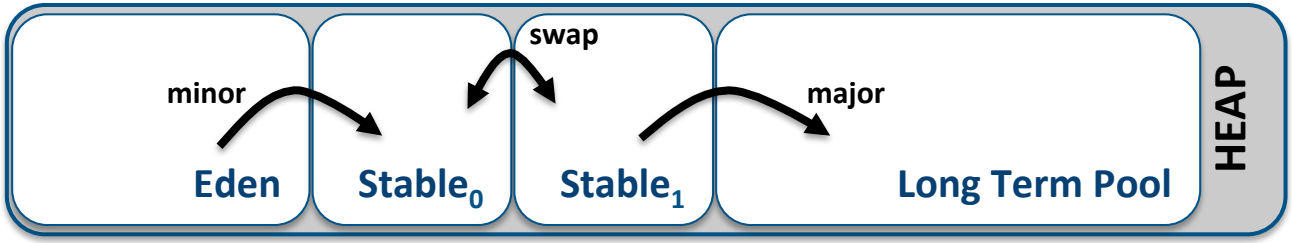
- Storage can "leak", or become un-recyclable
  - Leave a pointer to a large data structure on the stack, or in a global, …
  - or forgotten in another object, that happens to be live
  - Leads to extra collections and, eventually, an out of memory error



Implications for Programming

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

- 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 $Stable_1$ — **Minor collection**
- When Stable is too full, it is added to the Long Term Pool — **Major collection**



Does this stuff matter?

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