CMSC 433 – Programming Language Technologies and Paradigms
Fall 2006

Memory Management

Memory Management in Java

- Local variables live on the stack
  - Allocated at method invocation time
  - Deallocated when method returns

- Other data lives on the heap
  - Memory is allocated with new
  - But never explicitly deallocated
    - Java uses automatic memory management

Memory Mgmt and the JVM

- The JVM Specification doesn't say how to manage the heap
- Simplest valid memory management strategy: never delete any objects
  - Not such a bad idea in some circumstances (when?)
  - Need to consider relevant performance criteria

Performance Criteria

- Throughput
  - How much work does my application complete?
- Latency (promptness)
  - How long might my application pause (due to a memory management)?
- Memory footprint
  - How much memory is required by the application above what's needed for its work?

Garbage Collection (GC)

- At any point during execution, can divide the objects in the heap into two classes:
  - *Live* objects will be used later
  - *Dead* objects will never be used again
    - They are garbage

- Idea: Can reuse memory from dead objects

Many GC Techniques

- We can’t know for sure which objects are really live or dead
  - Undecidable, like solving the halting problem

- Thus we need to make an approximation
  - OK if we decide something is live when it’s not
  - But we’d better not deallocate an object that will be used later on

Reachability

- An object is **reachable** if it can be accessed by chasing pointers from live data
- Safe policy: delete unreachable objects
  - An *unreachable* object can *never* be accessed again by the program (the object is definitely garbage).
  - A *reachable* object *may* be accessed in the future (the object could be garbage but will be retained anyway).
    - Could lead to memory leaks
- How to implement reachability?
  - What data is live?
  - How to perform pointer chasing?

Roots

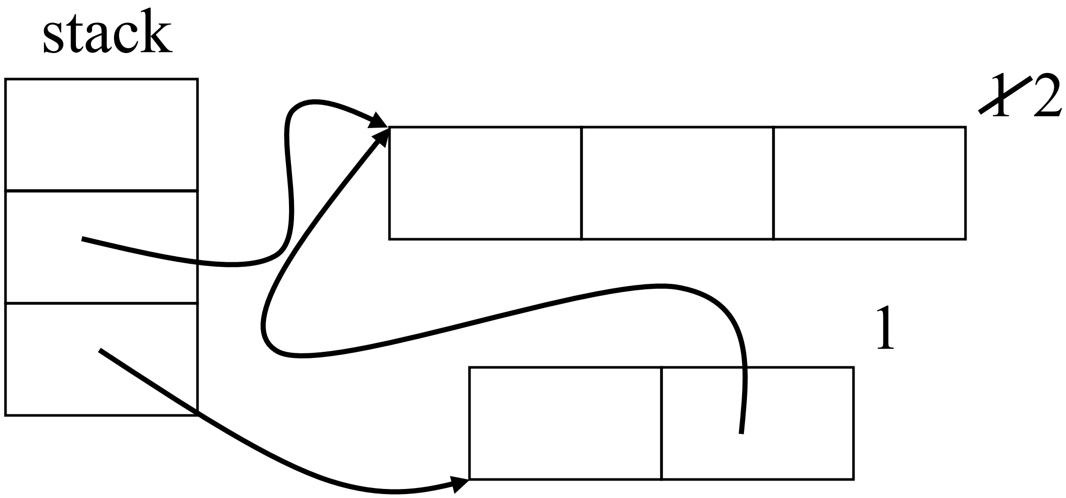
- At a given program point, we define liveness as being data reachable from the root set:
  - Global variables (i.e., static fields)
  - Local variables of all live method activations (i.e., the stack)
- At the machine level, we also consider the register set (usually stores local or global variables)

- Next: techniques for pointer chasing

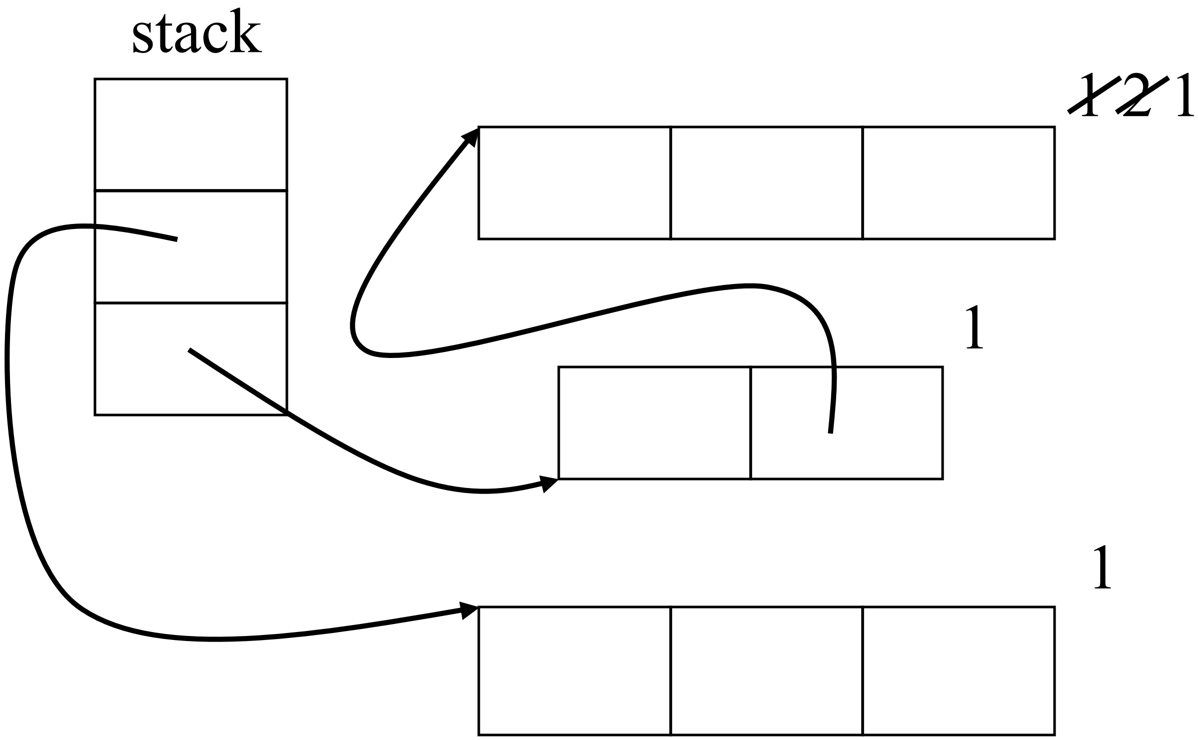
Reference Counting

- Old technique (1960)
- Each object tracks the number of pointers to it from other objects and from the roots.
  - When count reaches 0, object can be deallocated
- Counts incremented/decremented by the compiler (auto) or by hand (manual) when program statements create or remove aliases.

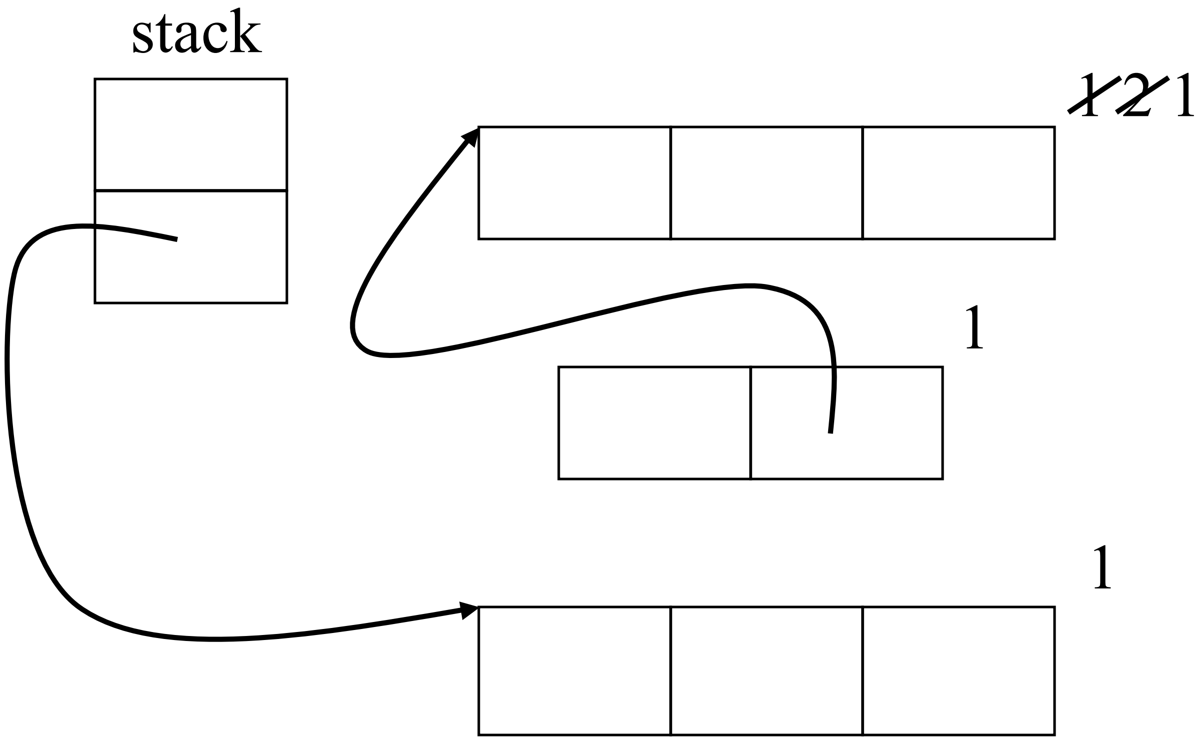
Reference Counting Example



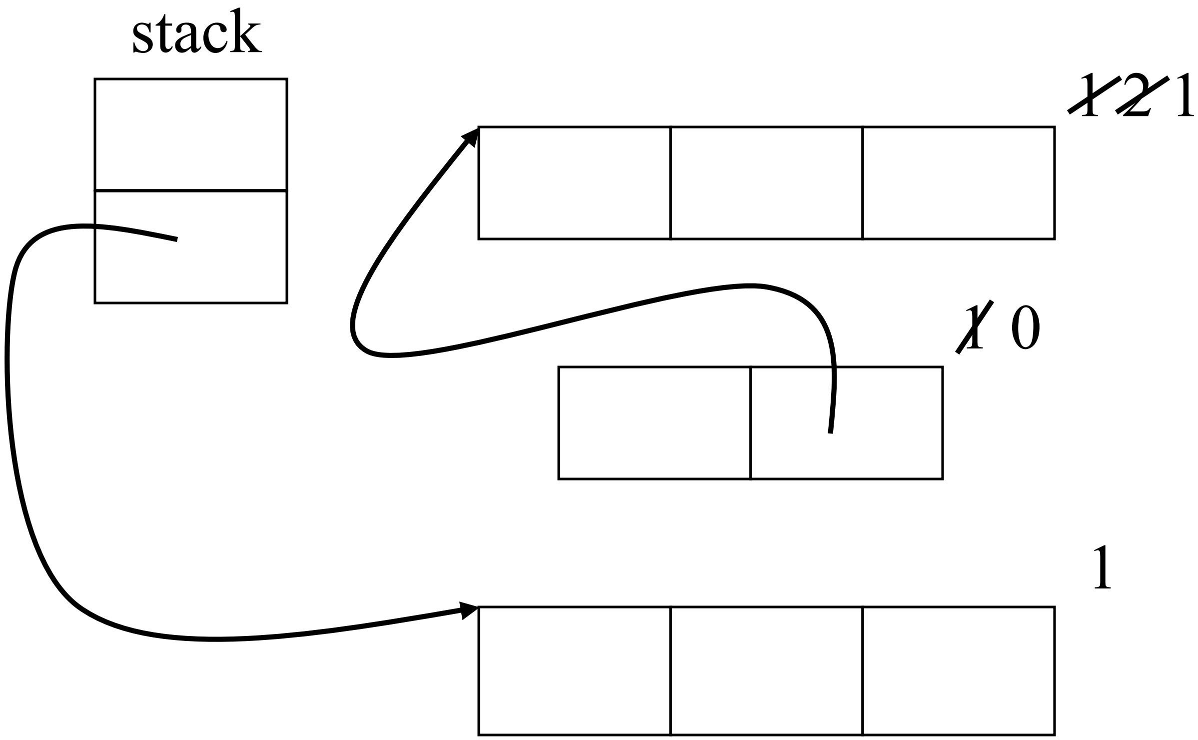
Reference Counting Example



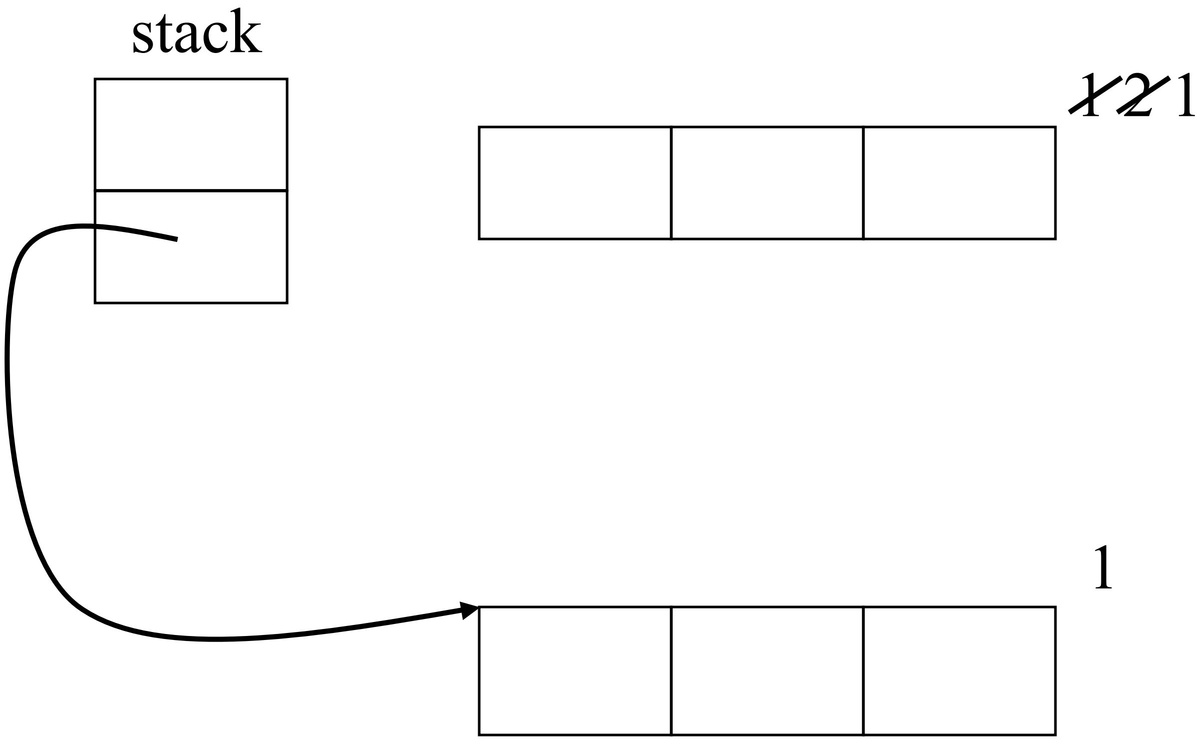
Reference Counting Example



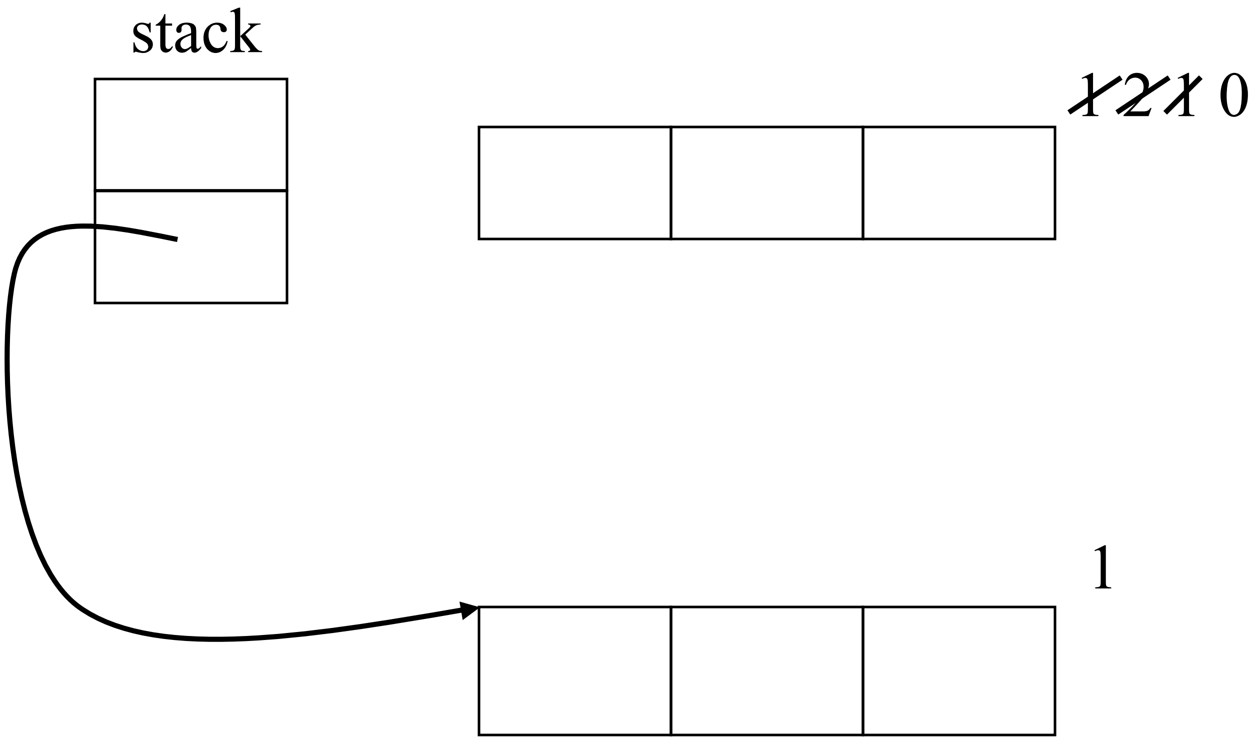
Reference Counting Example



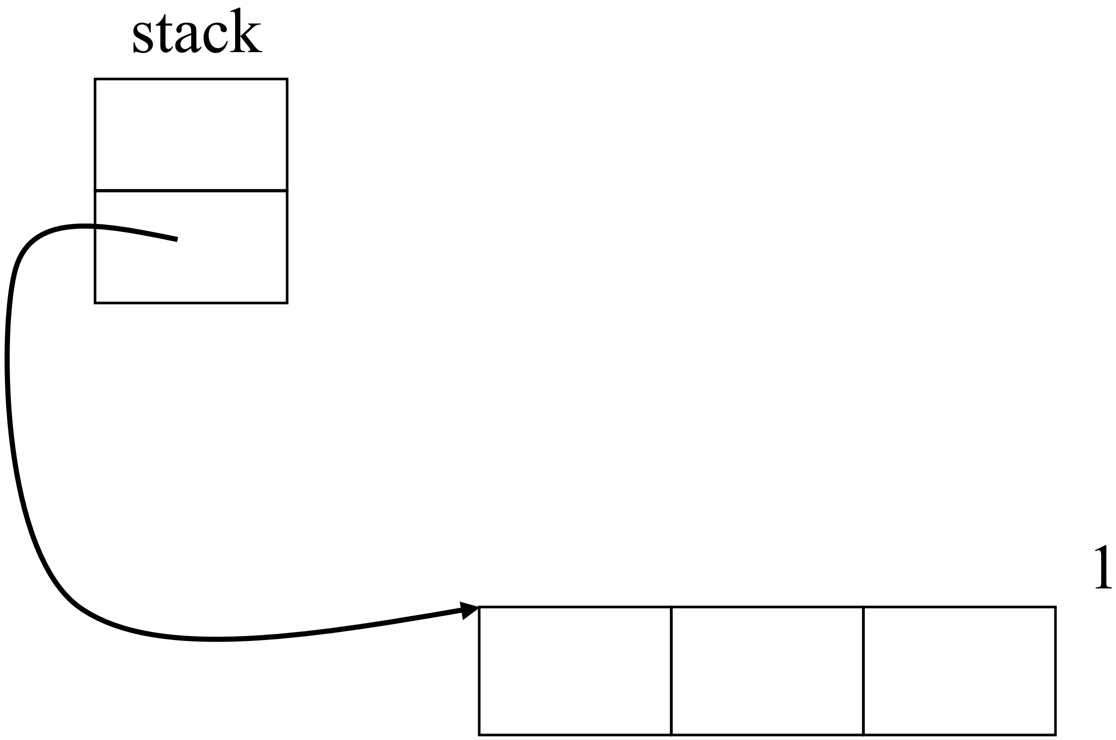
Reference Counting Example



Reference Counting Example



Reference Counting Example



Tradeoffs with Ref Counting

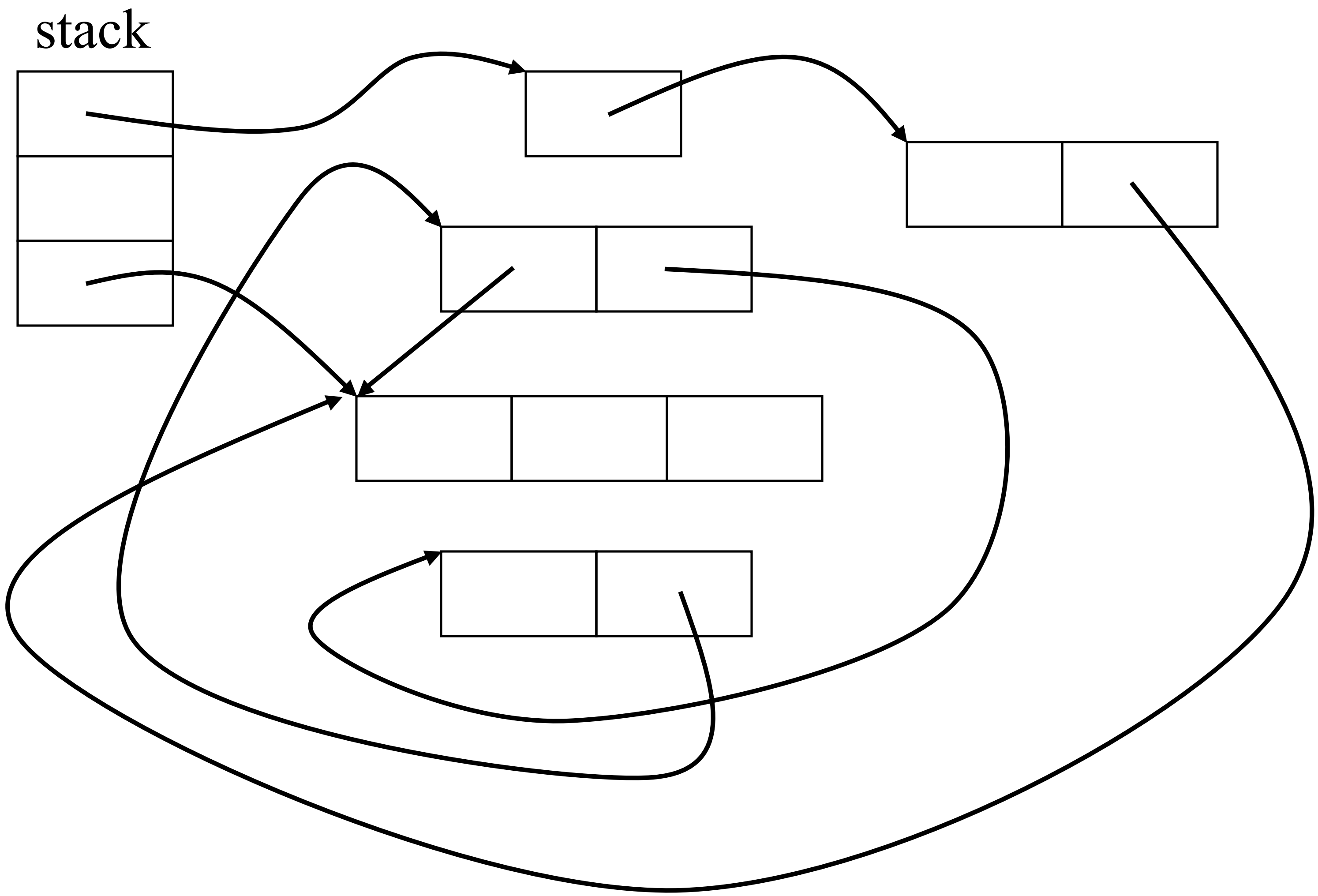
- Advantage: Incremental technique
  - Small amount of work per memory write
  - With more effort, can bound running time
    - Useful for real-time systems
- Problems:
  - Data on cycles can't be collected; counts never go to 0
  - Cascading decrements can be expensive
    - E.g., when an aggregate data structure that is dropped

Mark and Sweep GC

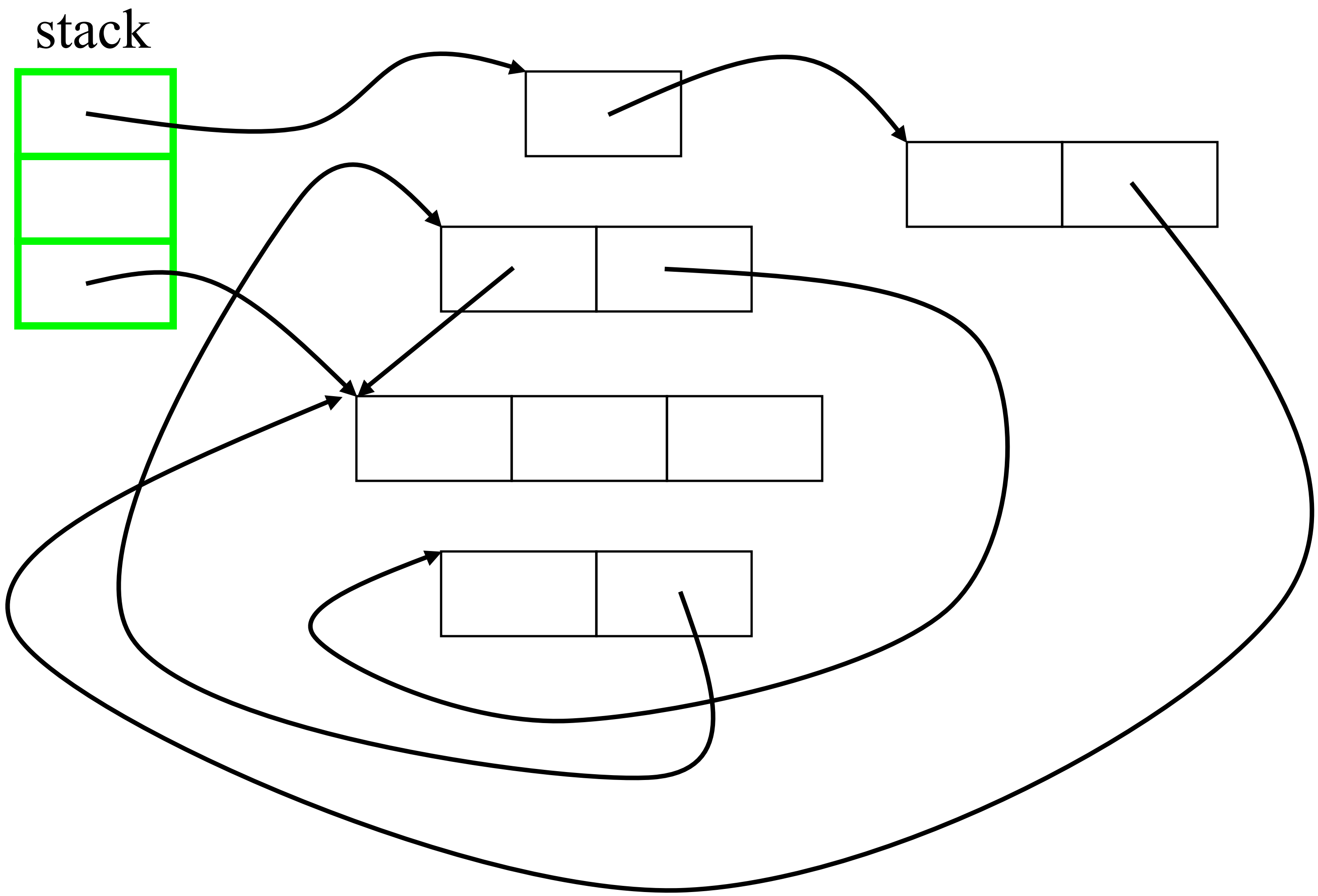
- Idea: Only objects reachable from stack could possibly be live
  - Every so often, stop the world and do GC:
    - Mark all objects on stack as live
    - Until no more reachable objects,
      - Mark object reachable from live object as live
    - Deallocate any non-reachable objects

- This is a *tracing* garbage collector

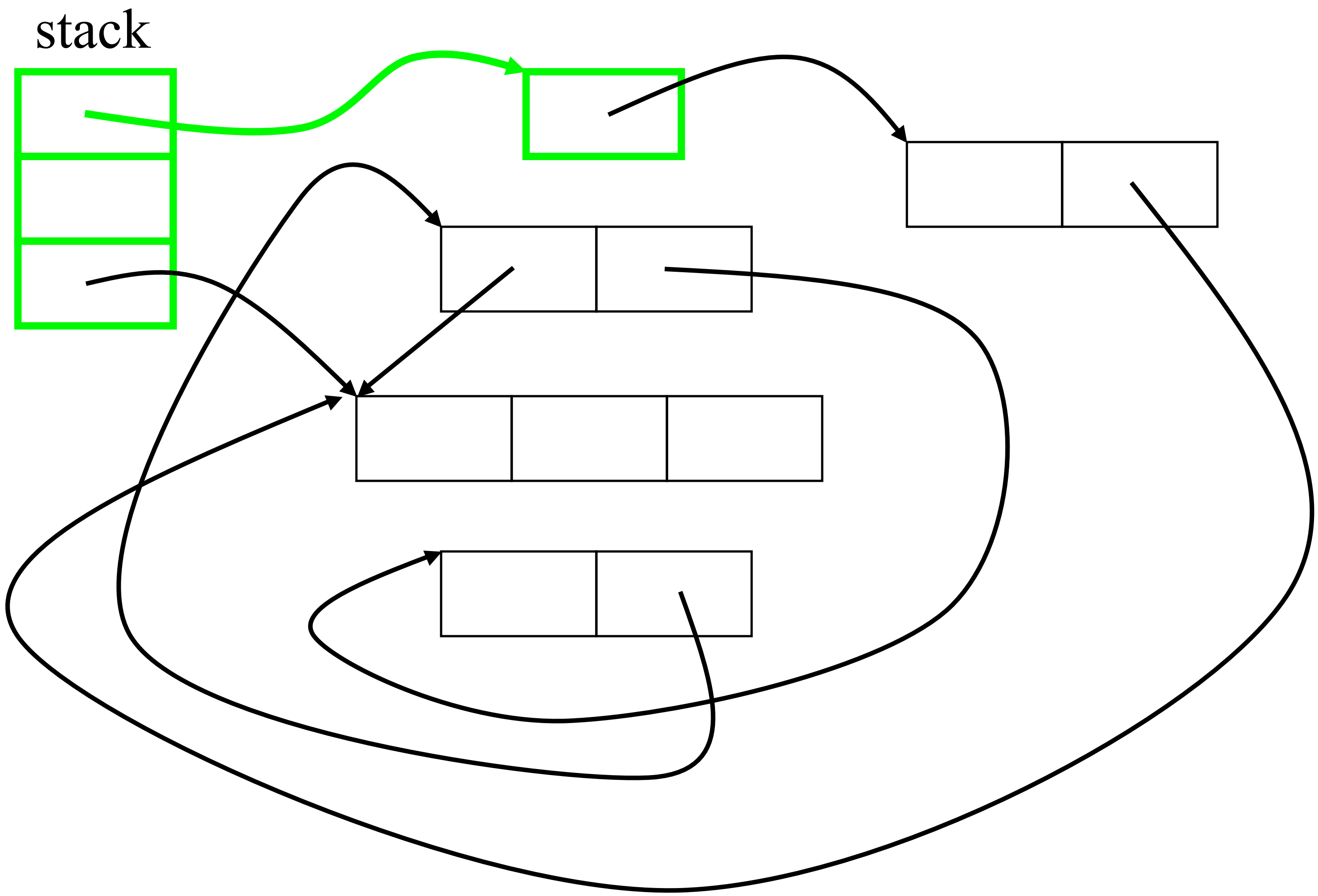
Mark and Sweep Example



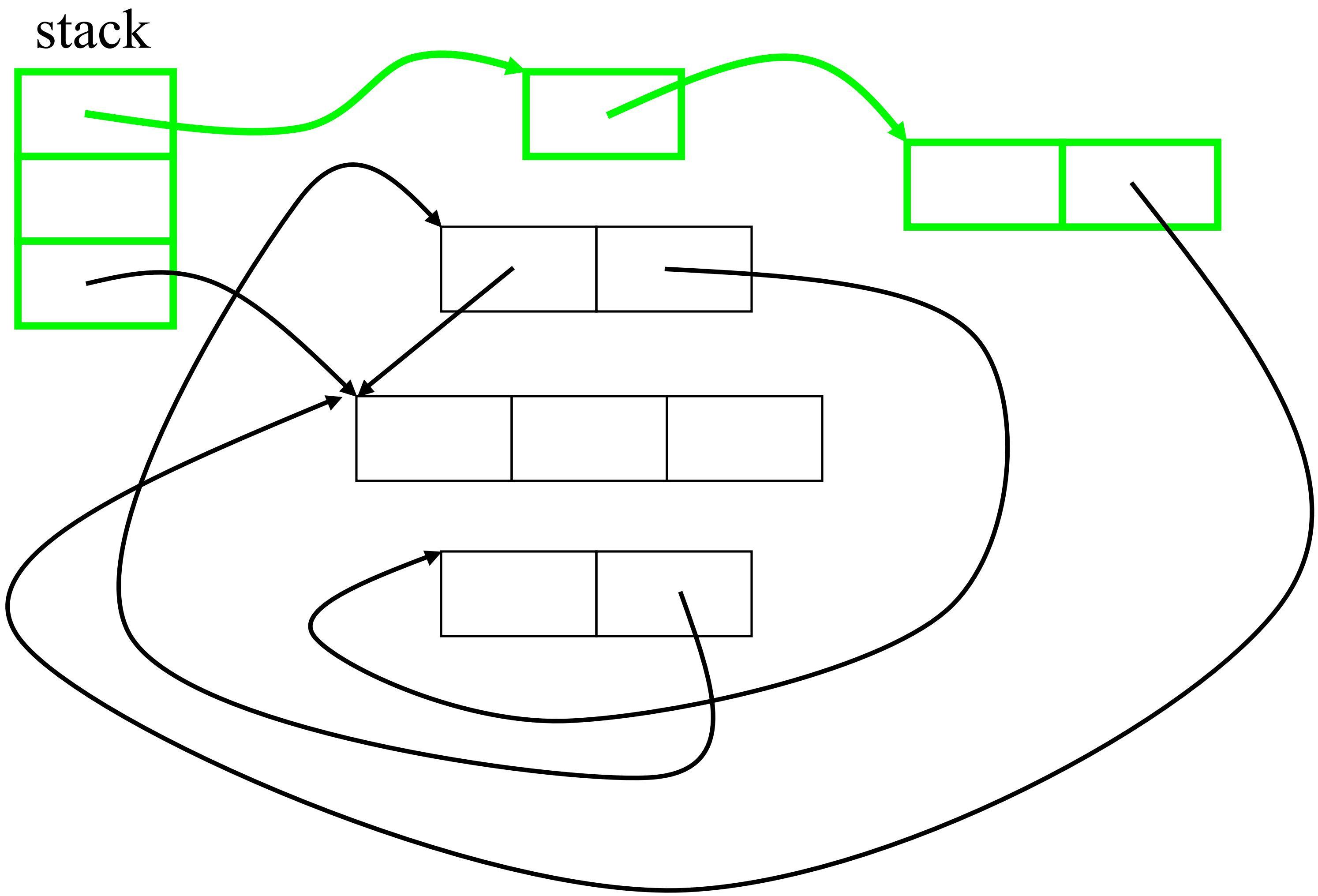
Mark and Sweep Example



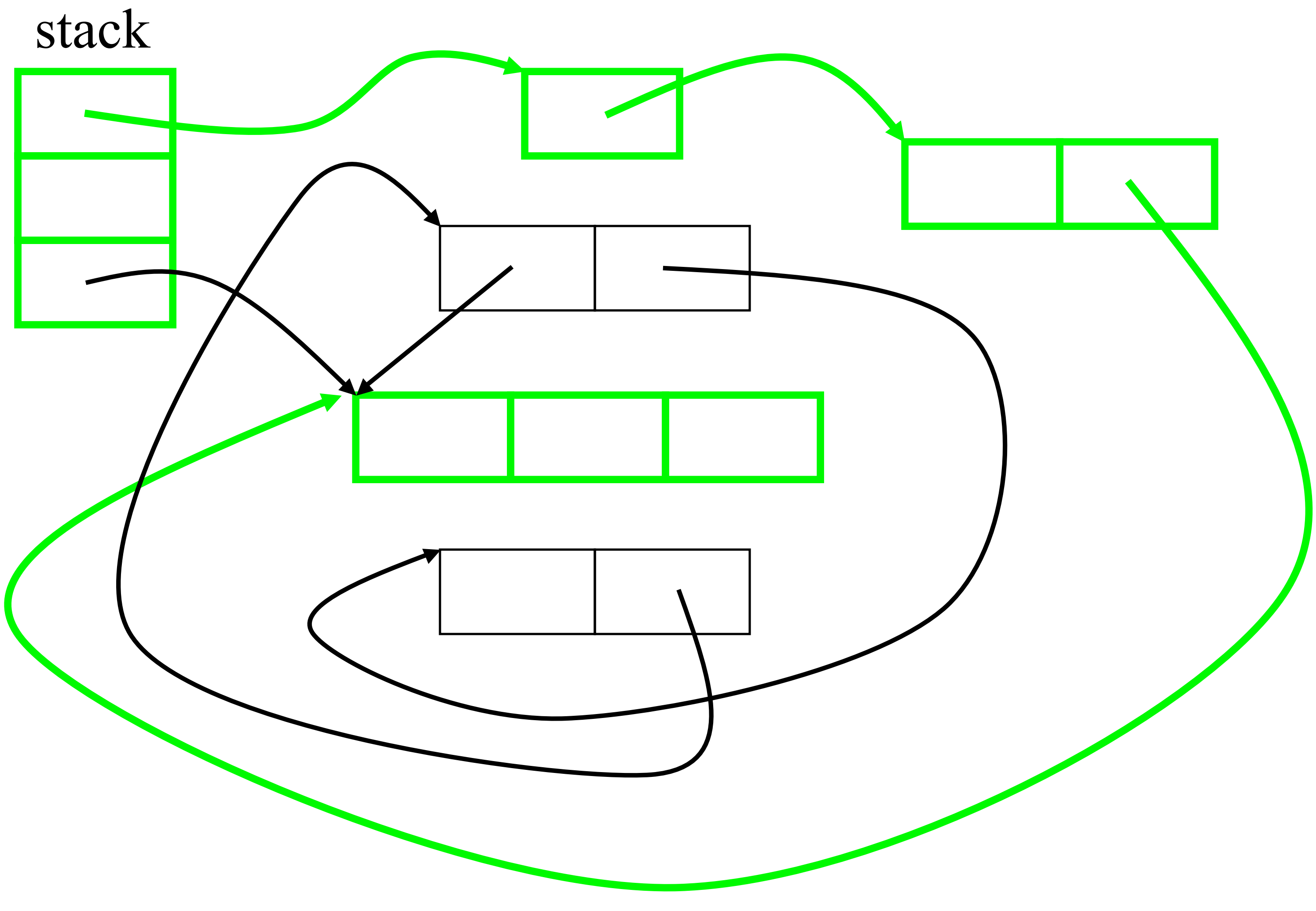
Mark and Sweep Example



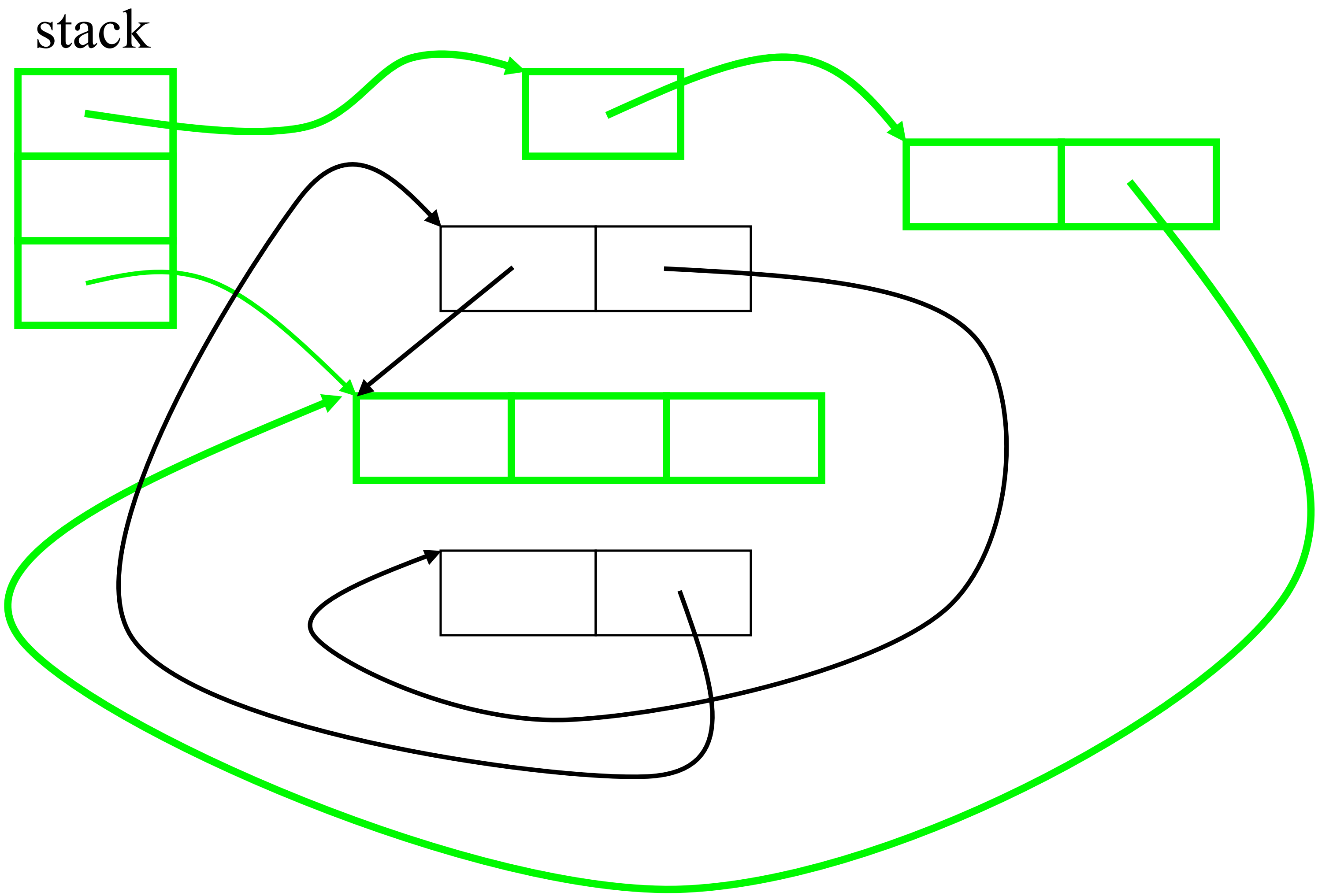
Mark and Sweep Example



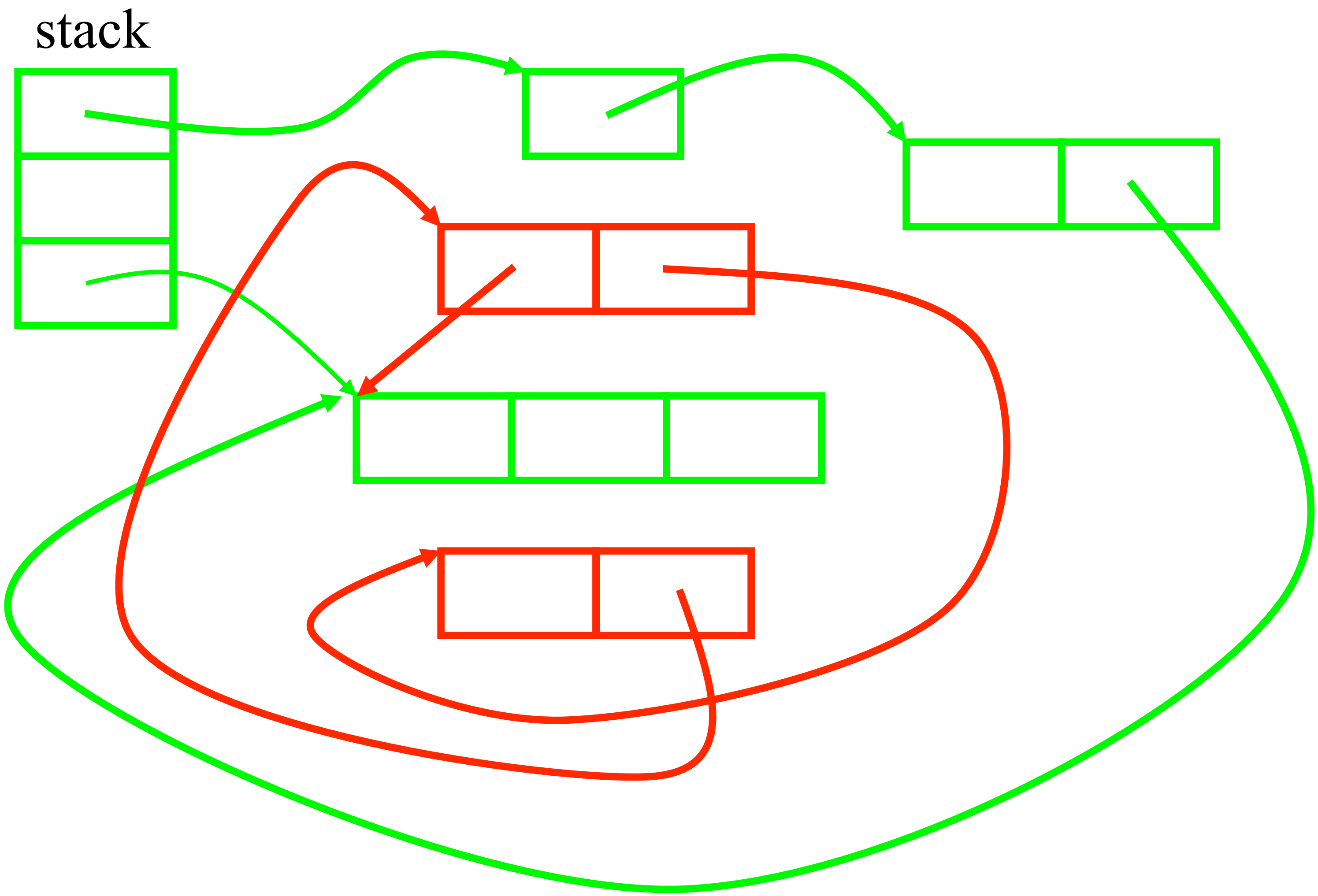
Mark and Sweep Example



Mark and Sweep Example



Mark and Sweep Example



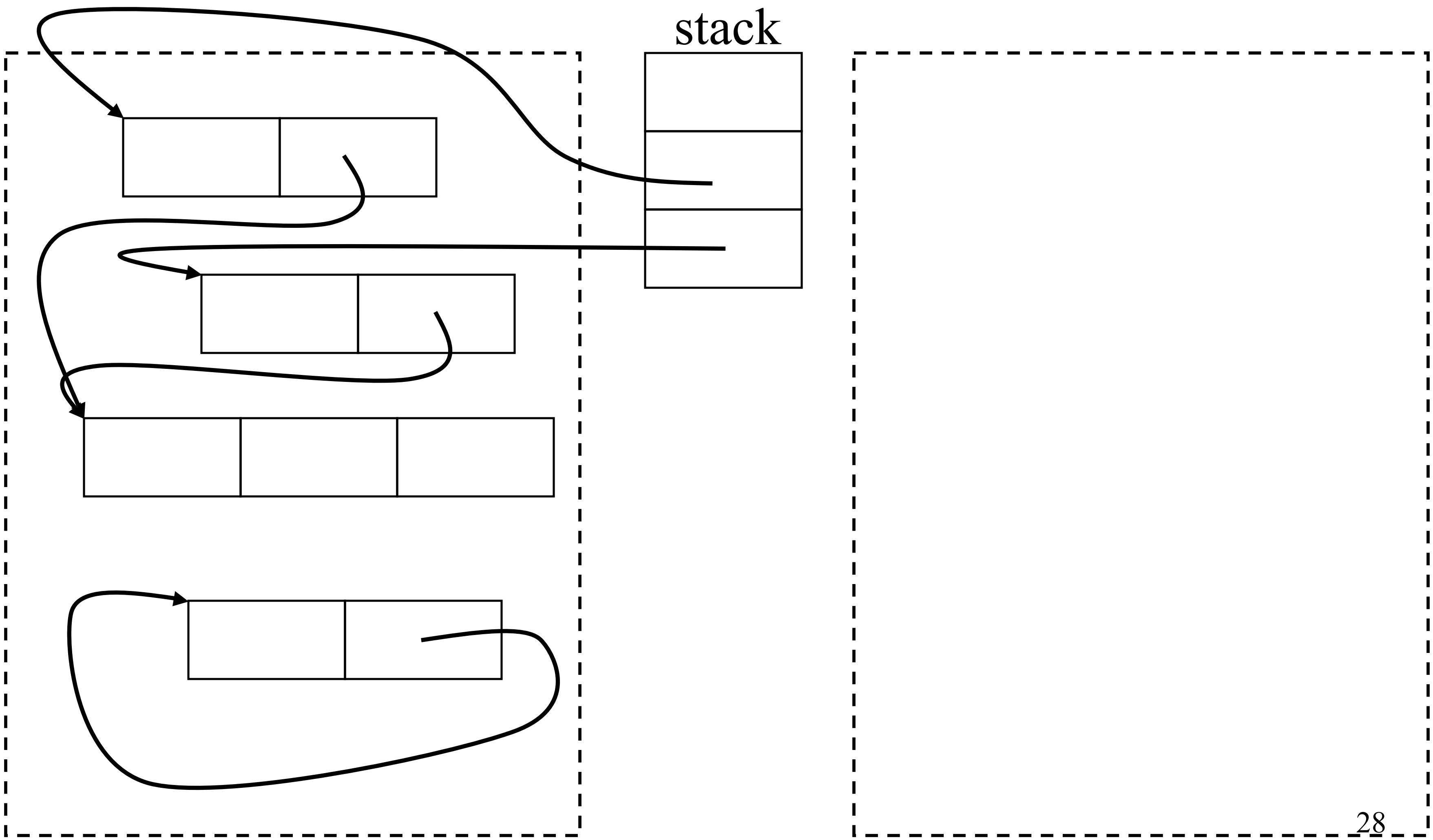
Tradeoffs with Mark and Sweep

- Pros:
  - No problem with cycles
  - Memory writes/dropped aliases have no cost
- Cons:
  - Fragmentation
  - Cost proportional to heap size
    - Sweep phase needs to traverse whole heap

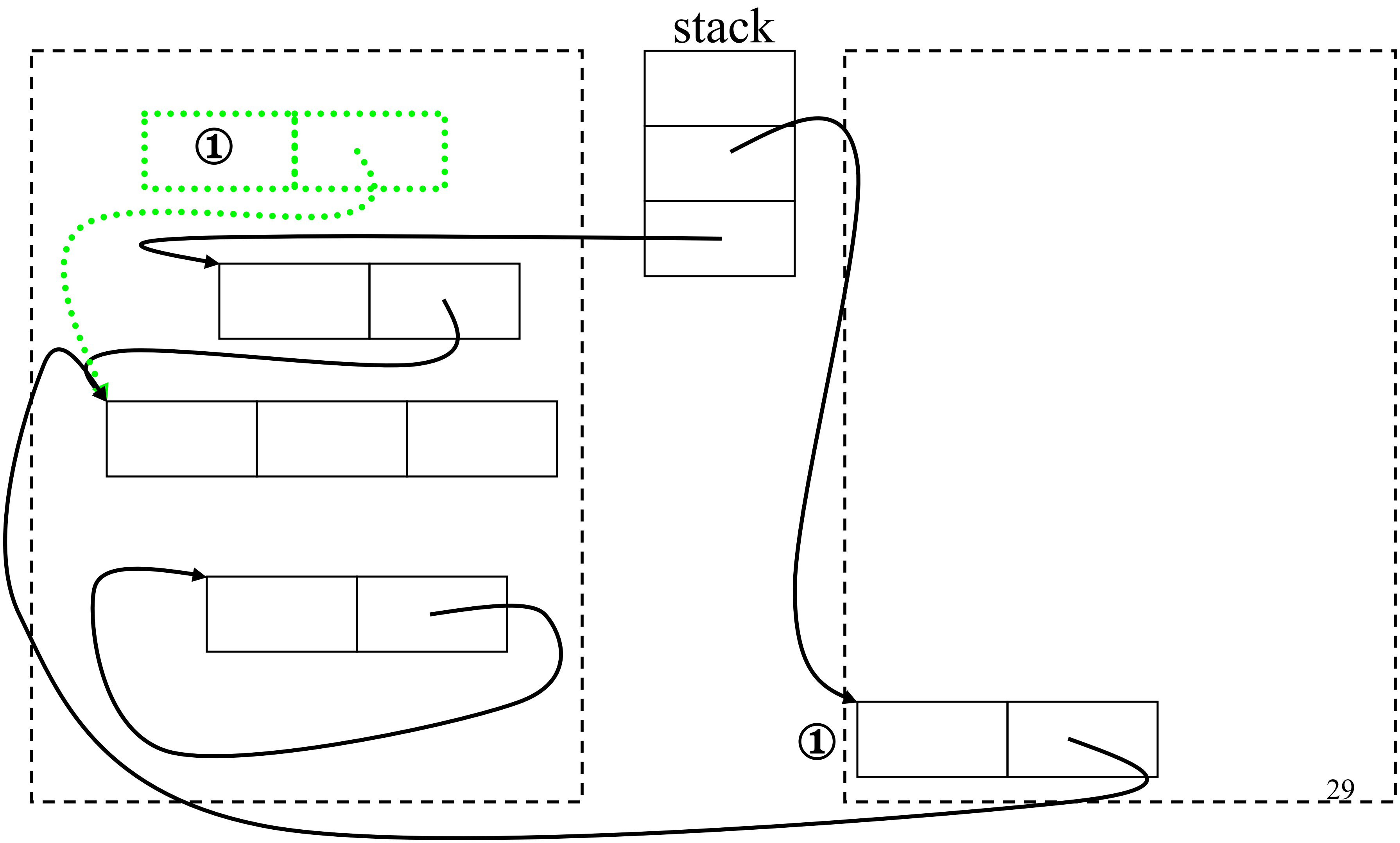
Copying GC

- Like mark and sweep, but only touches live objects
  - Divide heap into two equal parts (semispaces)
  - Only one semispace active at a time
  - GC copies data from one to the other
    - Trace the live data starting from the stack
    - Copy live data into other semispace
    - Declare everything in current semispace dead; switch to other semispace

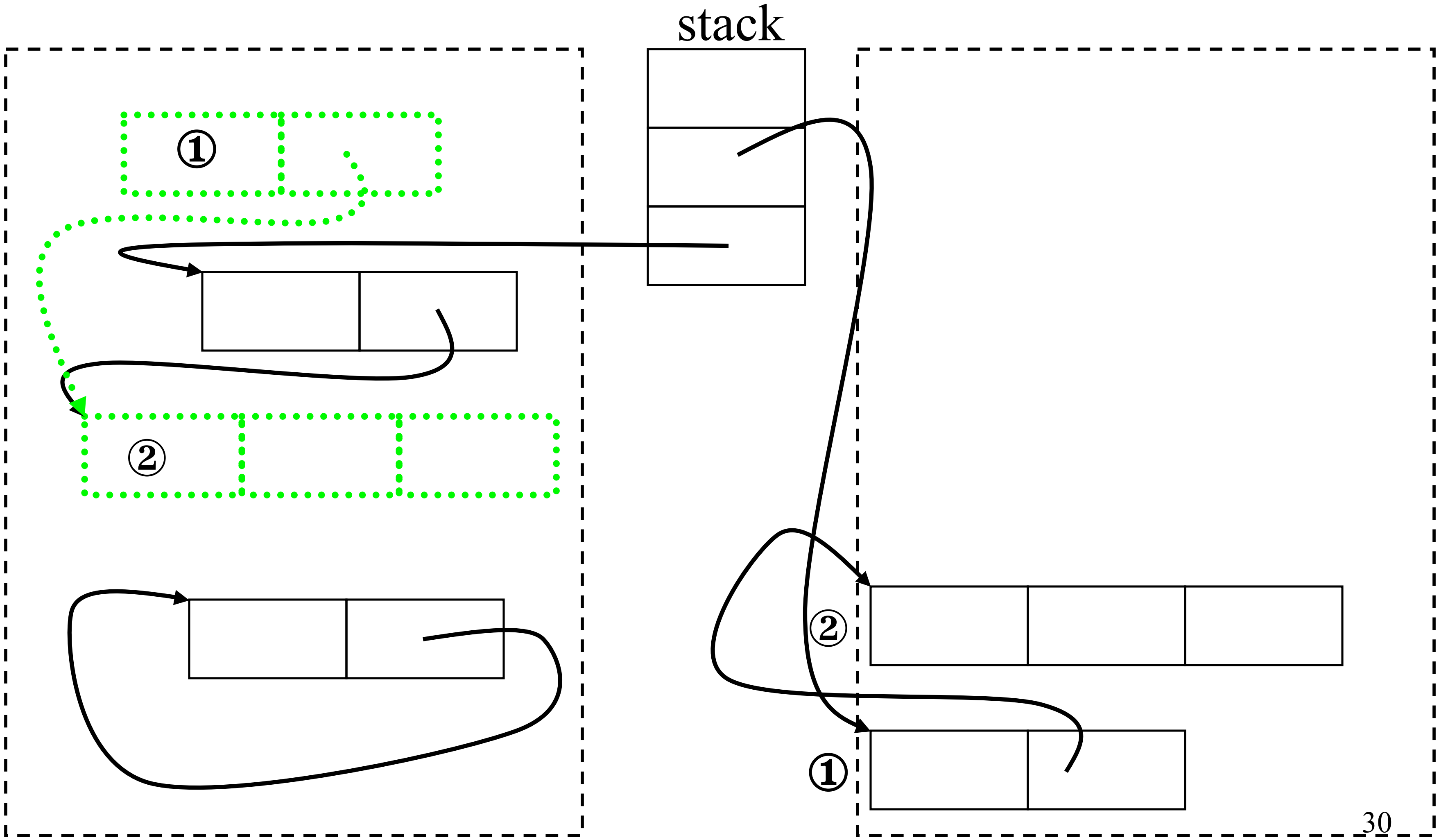
Copying GC Example



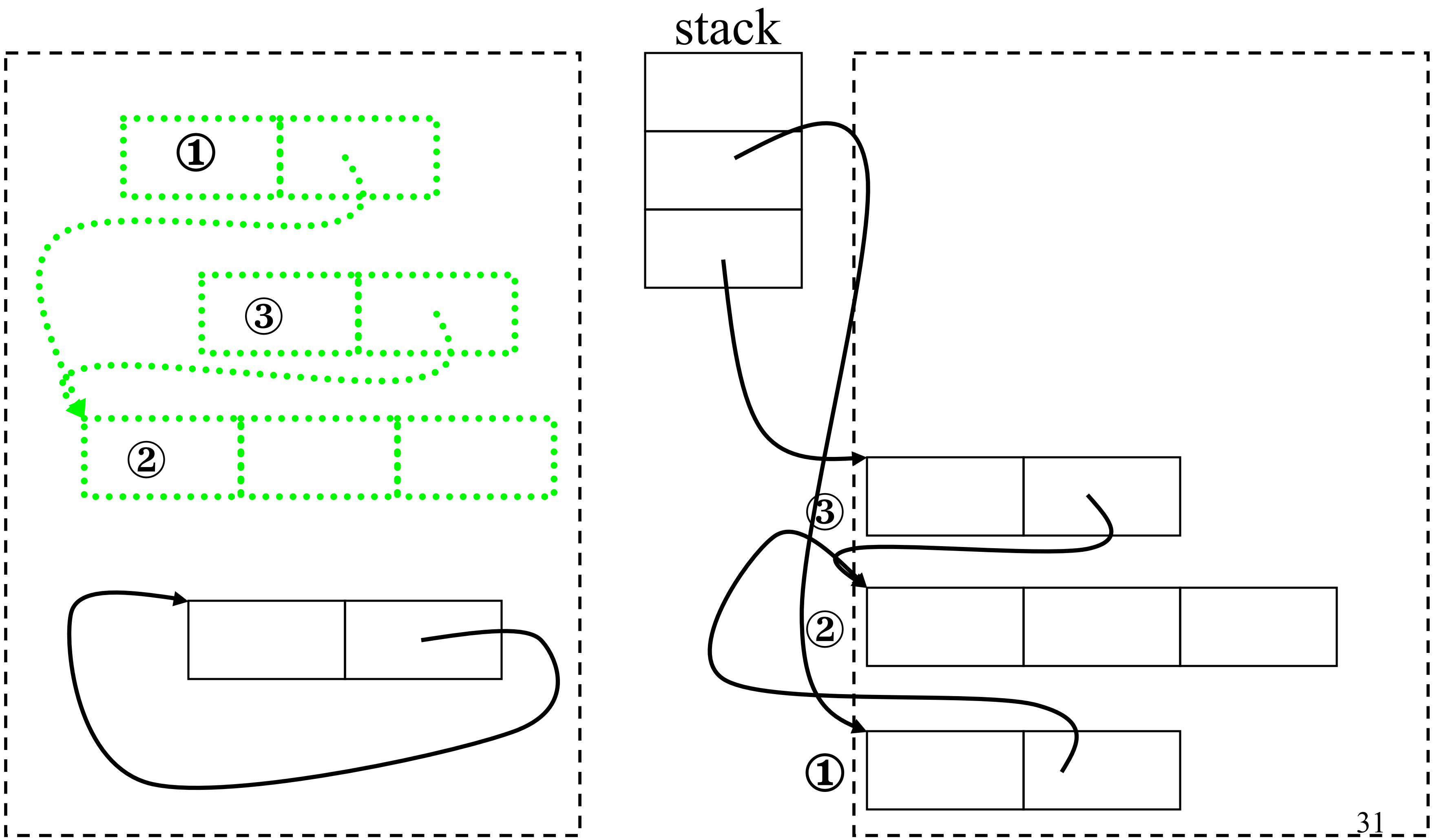
Copying GC Example



Copying GC Example



Copying GC Example



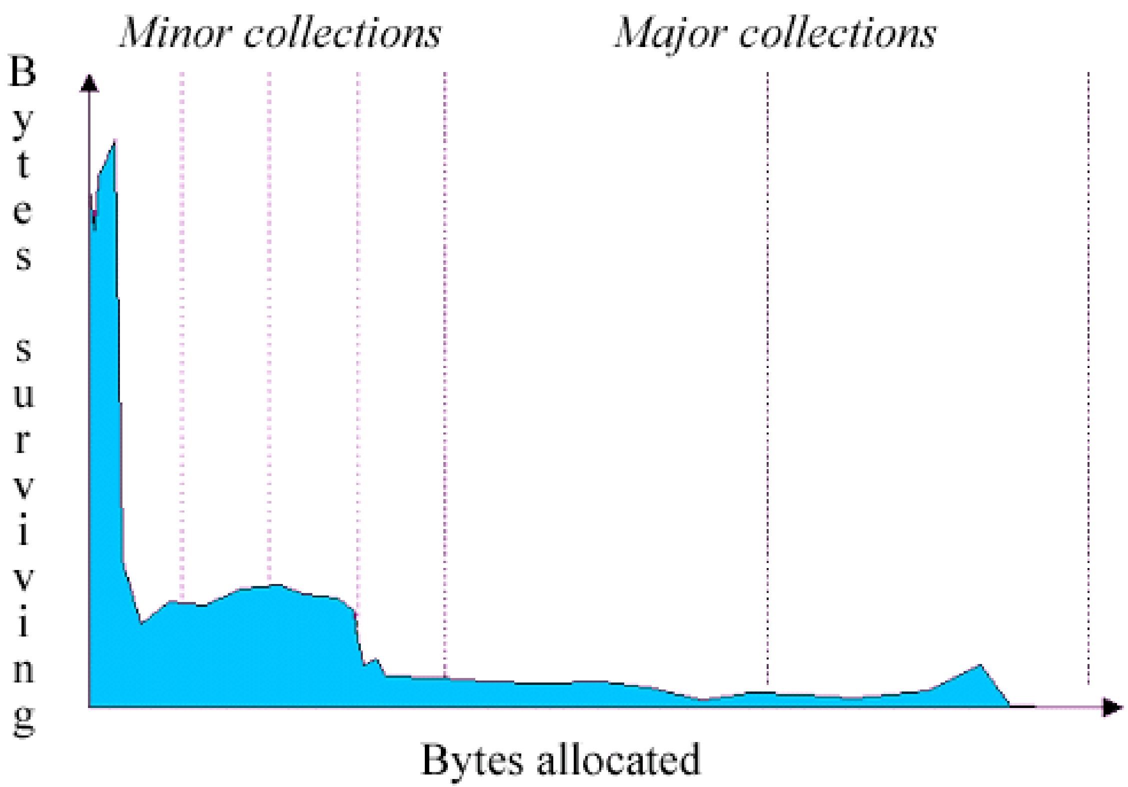
Copying GC Tradeoffs

- Pros:
  - Only touches live data
  - No fragmentation; automatically compacts
    - Will probably increase locality

- Cons:
  - Requires twice the memory space
  - Like mark and sweep, need to stop the world

The Generational Principle

More objects live ⇑



Object lifetime increases ⇒

"Young objects die quickly; old objects keep living"

Generational Collection

- Divide heap into *generations*
- Objects that survive many collections get pushed into older generations
  - Older generations collected less often
  - Need to track pointers from old to young generations to use as roots for young generation collection
    - Usually implemented with a "write barrier"

HotSpot SDK 1.4.2 Collector

- Multi-generational, hybrid collector
  - Young gen : stop and copy collector
  - Tenured gen : mark and sweep collector
  - Permanent gen : no collection
- Why does using a copy collector for the youngest generation make sense?
- What apps will be penalized by this setup?

Stopping the World

- What if the program (the "mutator") is modifying the heap while the GC is running?
- Simplest thing: stop the world during a GC
- Problem: big performance impact for parallel programs.

Scaling up GC

- Incremental collection
  - GC runs in mutator thread, but runs only for a short time and then resumes
- Concurrent collection
  - GC thread runs in parallel with the mutator
- Parallel collection
  - Many GC threads, all running in parallel

- All techniques require read/write barriers, which can be expensive

HotSpot VM

- Parallel collection of young space
  - Easy: each thread has its own young generation
- Concurrent collection of tenured space
- Incremental collection of tenured space
  - Does a little work with each minor collection

- All enabled by separate flags

Metronome

- GC developed by IBM research, targetting real-time systems.
- Uses parallel, concurrent collection
- Can provide strong guarantees of pause time and throughput.

What Does GC Mean to You?

- Ideally, nothing
  - It should make your life easier
  - And shouldn't affect performance too much
    - May even give better performance than you'd have with explicit deallocation
- If GC becomes a problem, hard to solve
  - You can set parameters of the GC
  - You can modify your program

Dealing with GC Problems

- Best idea: Measure where your problems are coming from
- For HotSpot VM, try running with
  - **-verbose:gc**
  - Prints out messages with statistics when a GC occurs

```
[GC 325407K->83000K(776768K), 0.2300771 secs]
[GC 325816K->83372K(776768K), 0.2454258 secs]
[Full GC 267628K->83769K(776768K), 1.8479984 secs]
```

GC Parameters

- Can resize the generations
  - How much to use initially, plus max growth
- Change the total heap size
  - In terms of an absolute measure
  - In terms of ratio of free/allocated data
- For server applications, two common tweaks:
  - Make the total heap as big as possible
  - Make the young generation half the total heap

Increasing Memory Performance

- Don’t allocate as much memory
  - Less work for your application
  - Less work for the garbage collector
  - Should improve performance
    - (Why only “should”?)
- Don’t hold on to references
  - Null out pointers in data structures
  - Or use weak references

Find the Memory Leak

```
class Stack {
  private Object[] stack;
  private int index;
  public Stack(int size) {
    stack = new Object[size];
  }
  public void push(Object o) {
    stack[index++] = o;
  }
  public void pop() {
    return stack[index--];
  }
 }
```

– From Haggar, Garbage Collection and the Java Platform Memory Model

Bad Ideas (Usually)

- Calling System.gc()
  - This is probably a bad idea
  - You have no idea what the GC will do
  - And it will take a while

- Managing memory yourself
  - Object pools, free lists, object recycling
  - GC's have been heavily tuned to be efficient

java.lang.ref

- Package that lets you interact with GC
  - If there's a *strong* (normal) reference to an object, GC will consider it live
  - Can use java.lang.ref to make references to object that "don't count"

```
public abstract class Reference {
        void clear();
        public Object get();
        ...
}
```

SoftReference

- Constructor SoftReference(Object o)
  - Make a soft reference to object o
  - Access using get() method
- GC *may* free object if
  - Only soft, weak, or phantom refs to it
    - No strong refs
- GC invokes clear() on SoftRef if freed
  - Like a finalizer

WeakReference

- Constructor WeakReference(Object o)
- GC *will* free object if
  - Only weak or phantom refs to it
    - No strong or soft refs
- GC invokes clear() on WeakRef if freed

PhantomReference

- Can't ever get a reference from a phantom reference
  - only useful to get notification when an object is GC'd

ReferenceQueue

- When you create a reference, you can specify a reference queue
  - when the reference is cleared, it is enqueued in the reference queue

WeakHashMap

- Uses weak references to hold keys
- If the only reference to an object is as the key of a weak hash map
  - the object is GC'd and removed from the WeakHashMap
- Particularly useful for objects that use the default equals
  - If the only reference to an object is from a map, you are unlikely to use it as a key

Uses of Soft and Weak Refs

- Use soft references for caches
  - Data can be recomputed/reconstructed
  - GC flushes cache (clears soft refs) if memory full
- Use weak references for mappings
  - E.g., object reuse (if you're going to do it...)
  - GC frees object once strong refs are gone

Caveat for Weak References

- Which is correct?

```
WeakReference wr = …;
obj = wr.get();
if (obj == null) {
  obj = new A();
  wr = new WeakRef(obj);
}
```

```
WeakReference wr = …;
obj = wr.get();
if (obj == null) {
  wr = new WeakRef(new A());
  obj = wr.get();
}
```

- From Haggar, Garbage Collection and the Java Platform Memory Model

Caveat for Weak Refs (cont'd)

- Left side is correct
  - Consider `new WeakRef(new A())`
  - Object A allocated
  - Put into WeakReference
  - Suppose GC happens right after
  - Then A may be deallocated before `wr.get()`