Garbage Collection

Steven R. Bagley

Reference Counting

- Counts number of pointers to an object
- Object deleted when the count hits zero
- Eager — object deleted as soon as it is finished with
- Problem: Circular references

Reference Counting

- Saw how we can graft reference counting support onto C++
- Programmer must manually retain and release the object
- Can lead to errors
- Isn’t there an automatic method?

Garbage Collection

- Automatically reclaim memory from ‘garbage’ (unused objects)
- Several techniques out there
- Not without its own issues
- Consider the basic techniques

Reference Counting

- Common technique — ActionScript, Python
- Language hides the calls to retain/release object
- Still suffers from circular references and thus memory leaks
- Python tries to find cycles to avoid this

Reference Counting

- Lightweight
- Minimal code required to implement
- Minimal execution time
- Important advantages in real-time environments

Tracing Collectors

- Trace-based Garbage Collector
- 'Trace over every object'
- Look at two types:
  - Mark-sweep collectors
  - Copying collectors

Tracing Collectors

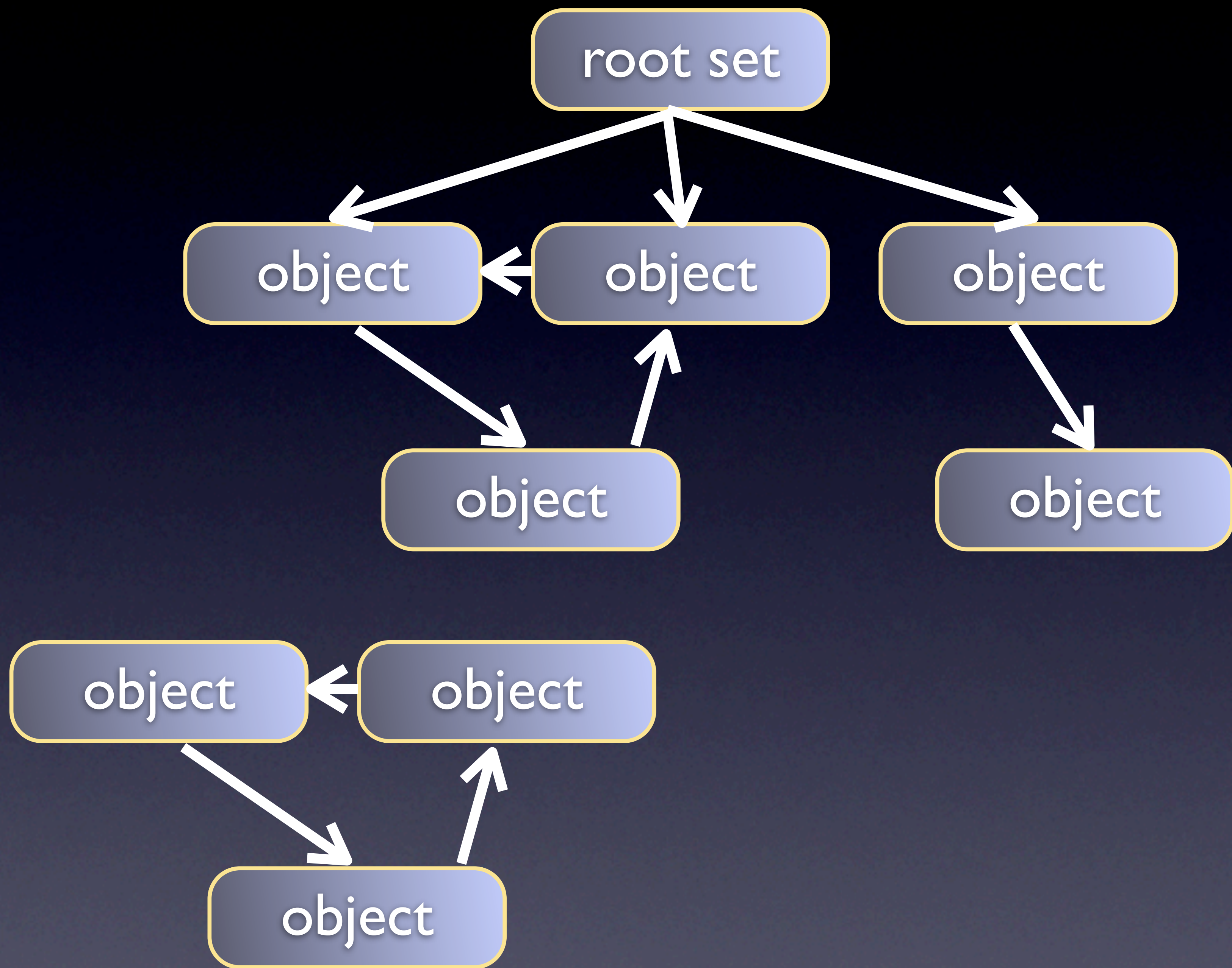
- Stop-the-world
- Check to see which objects are in use
- And, by corollary, those which are not
- Delete the unused ones
- Restart-the-world

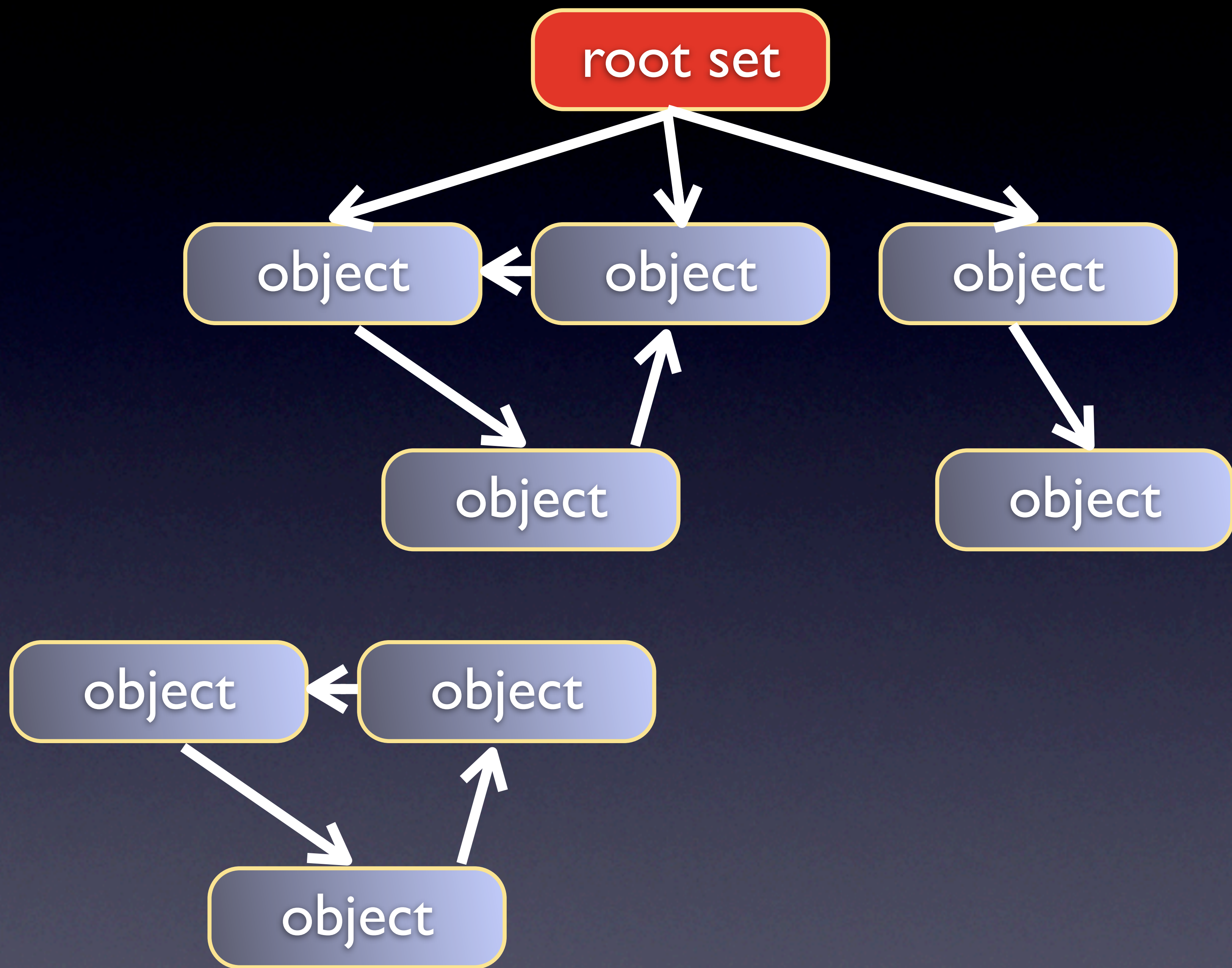
Big Problem

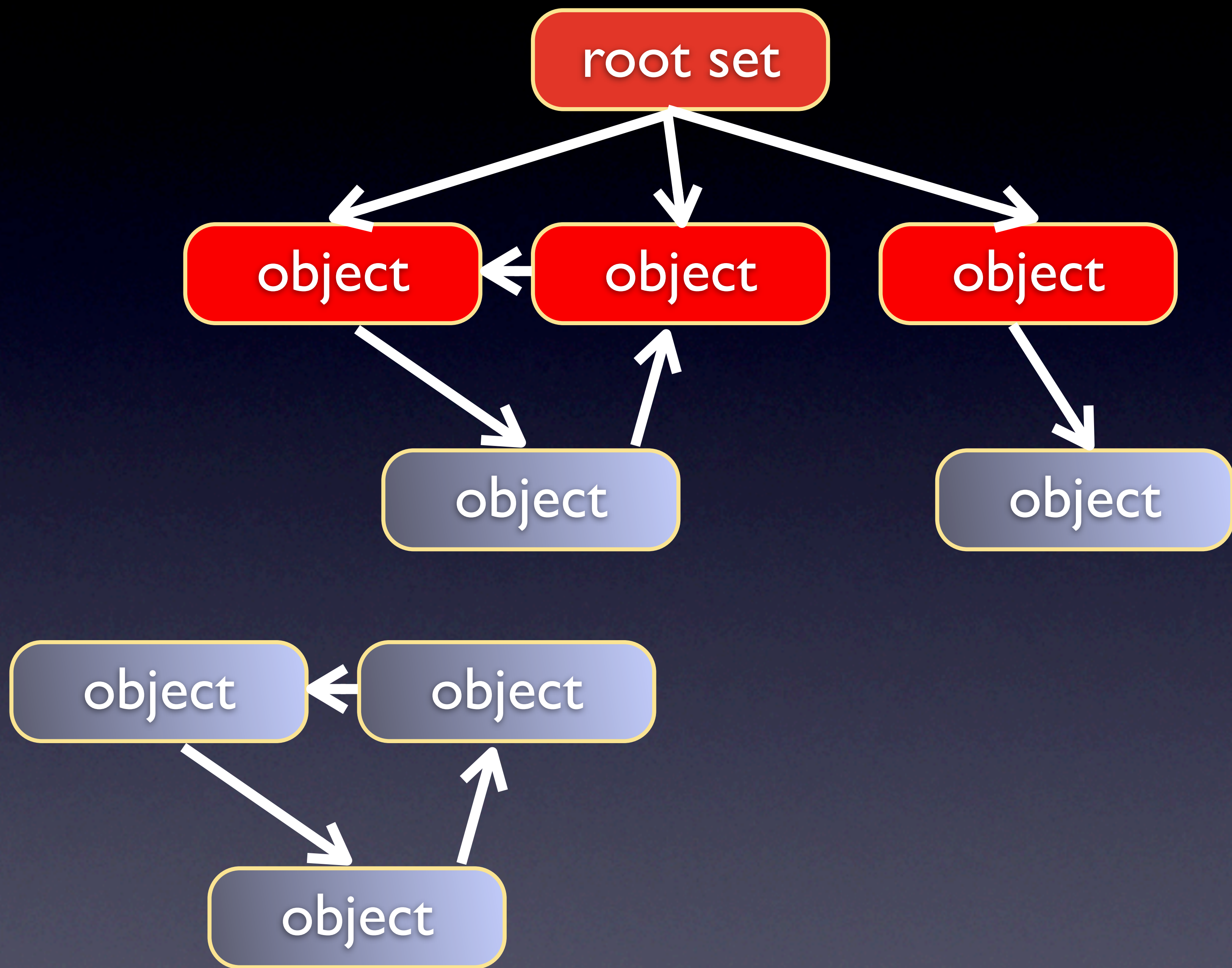
- How do we know:
  - Which objects are in use? (or *reachable*)
  - How many objects there are?
- In fact, isn't this the problem Garbage Collection is meant to solve
- Key is in the name — Trace-based

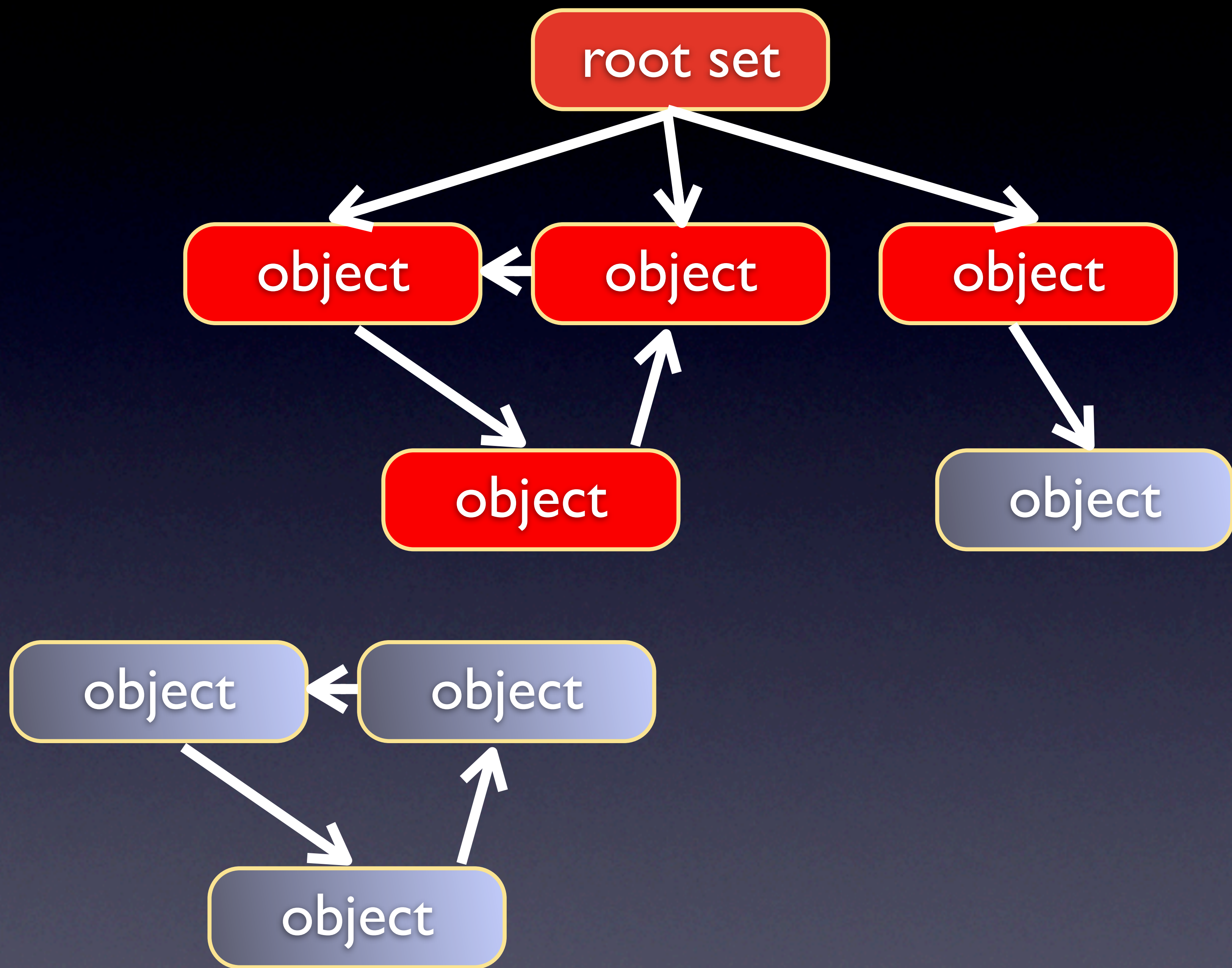
In-use objects

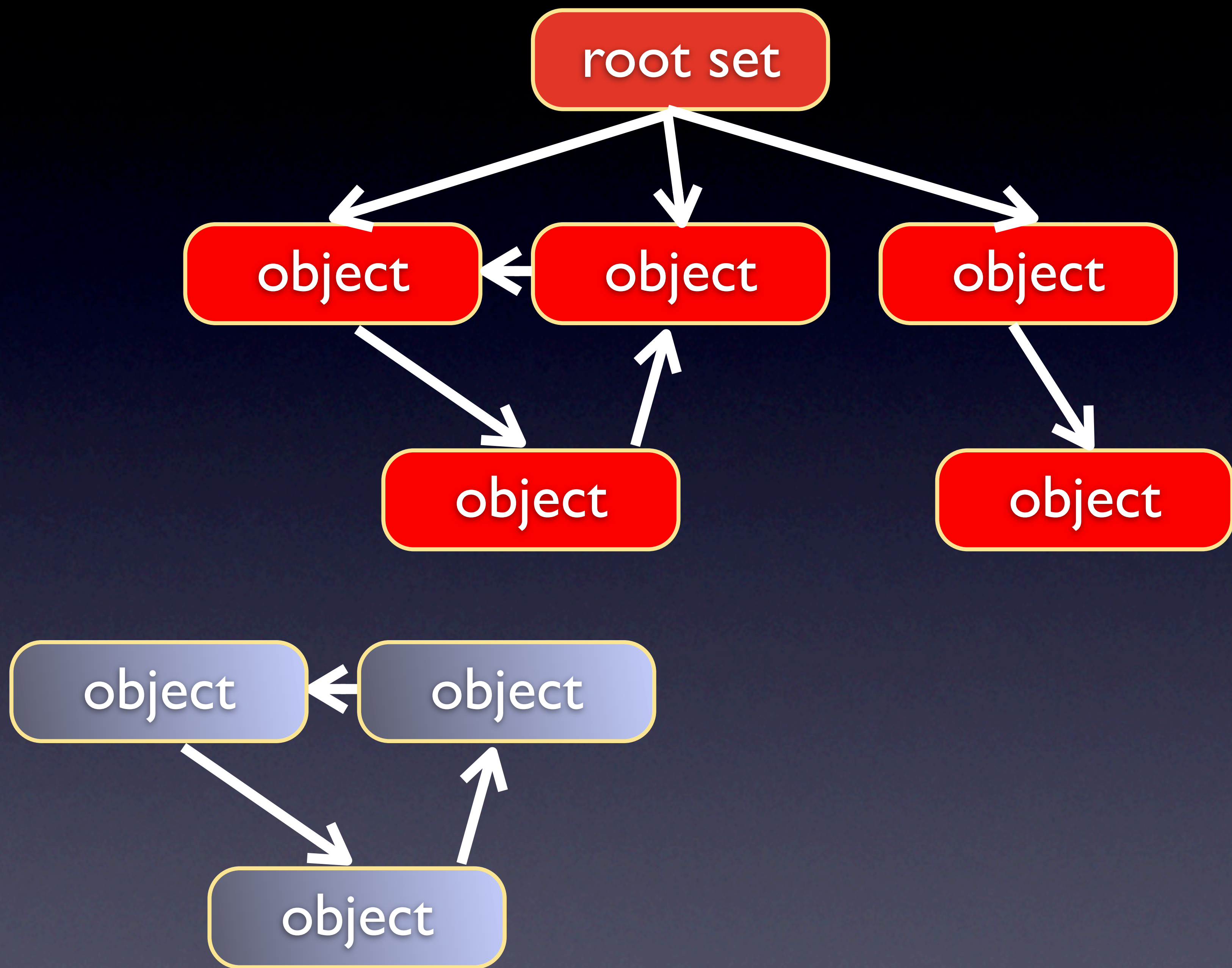
- Assume there is a *root set* of known objects
- Reachable object is the *root set* and any objects referenced by the *root set*
- And any objects referenced by objects referenced by the *root set*…
- Until there are no new objects to add to the set of reachable objects


root set
object
object
object
object
object
object
object
object

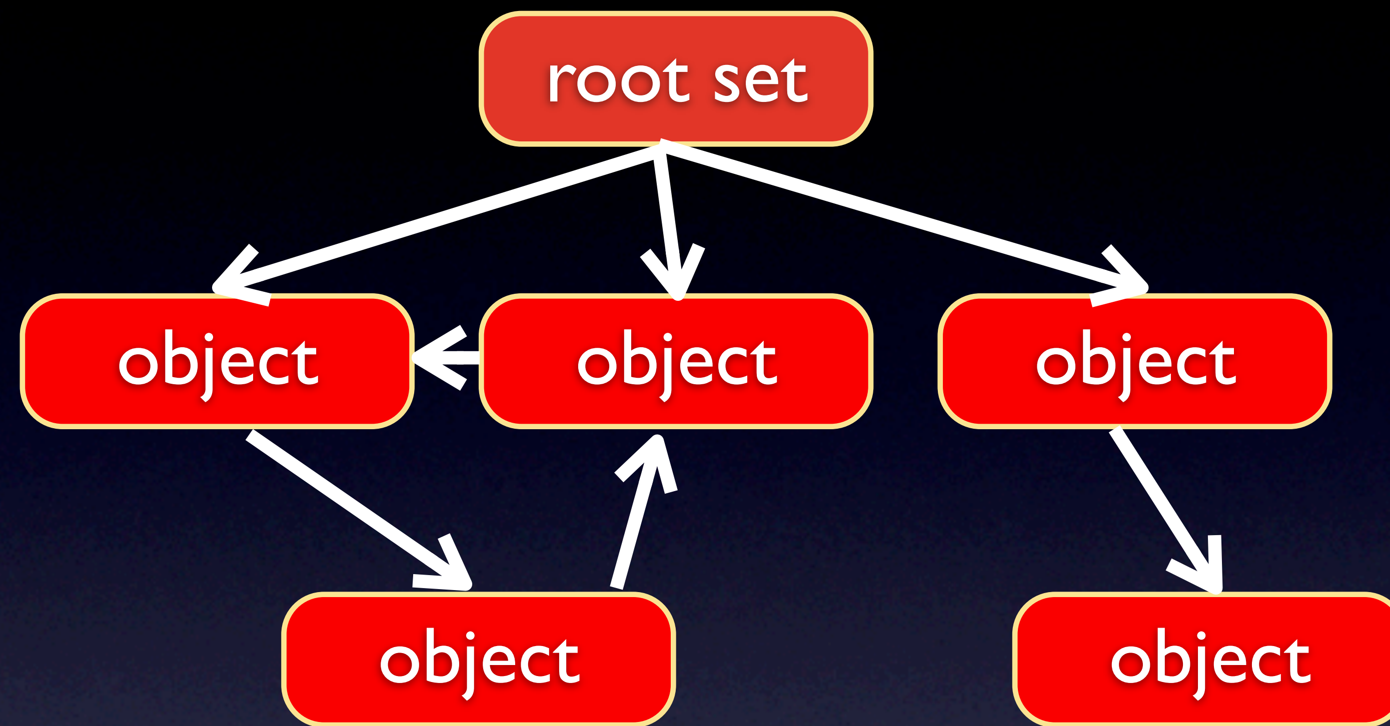

root set
object
object
object
object
object
object
object
object


root set
object
object
object
object
object
object
object
object


root set
object
object
object
object
object
object
object
object



Objects in bottom-left unreachable from root set



Objects in bottom-left unreachable from root set

Root Set

- All objects pointed to by global variables
- All objects pointed to by static class variables
- All objects pointed to by local variables
- All objects pointed to by CPU registers

Local variables of this method and all methods up the call stack

All objects

- Can find the objects still in use
- Objects not in use = All object minus in use objects
- How do we find all objects?
- Think about how objects are created…
- Every object created by calling `new`

All objects…

- Could store pointers returned from `new`
- But there is a simpler way
- Memory Allocator maintains a list of allocated memory blocks and of free memory
- Garbage Collector can just use the allocator’s list

Allocated memory blocks will be the objects

Garbage Collector is part of the memory allocator anyway so this is not a problem

Stop The World

- Important to stop execution of the program
- If not, program will change state as collector runs
- Objects may by marked as in-use when they are no longer used

Stop the world

- Suppose the GC builds the root set
- Program then moves an object pointer from an object (a) into a local variable
- GC traces over reachable objects
- Object (a) never reached and so freed
- Program accesses (a) and crashes

Mark-Sweep collector

- Mark-Sweep is a basic form of garbage collector
- First pass, marks every object that is accessible as 'in-use' (by visiting every reachable object)
- Second pass, visit every object allocated and destroy those not marked as in-use

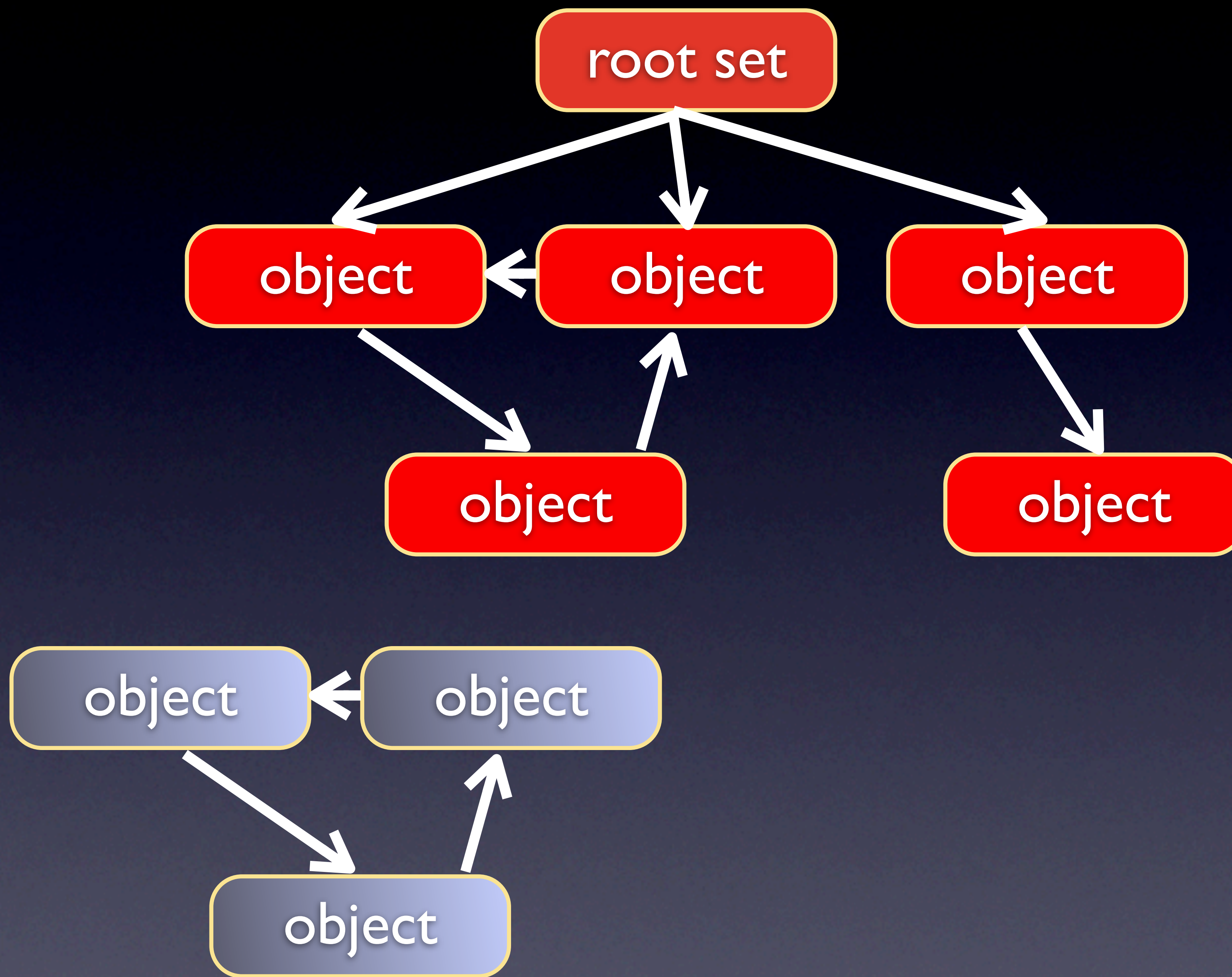
Pass 1 – Marking

- Marks all objects that are ‘in-use’
- Already seen how this can be done
- Start off with objects in the root-set and recursively follow pointers to other objects
- Mark object as ‘in-use’ as you visit it

Pass 2 – Sweep

- Stepping through the allocator’s list of allocated memory blocks
- If the object is not marked ‘in-use’ it is destroyed
- The ‘in-use’ flag is cleared

if it wasn’t cleared we’d never know if an object went out-of-scope



Illustrate with laser pointer how objects are reached

Mark/Sweep

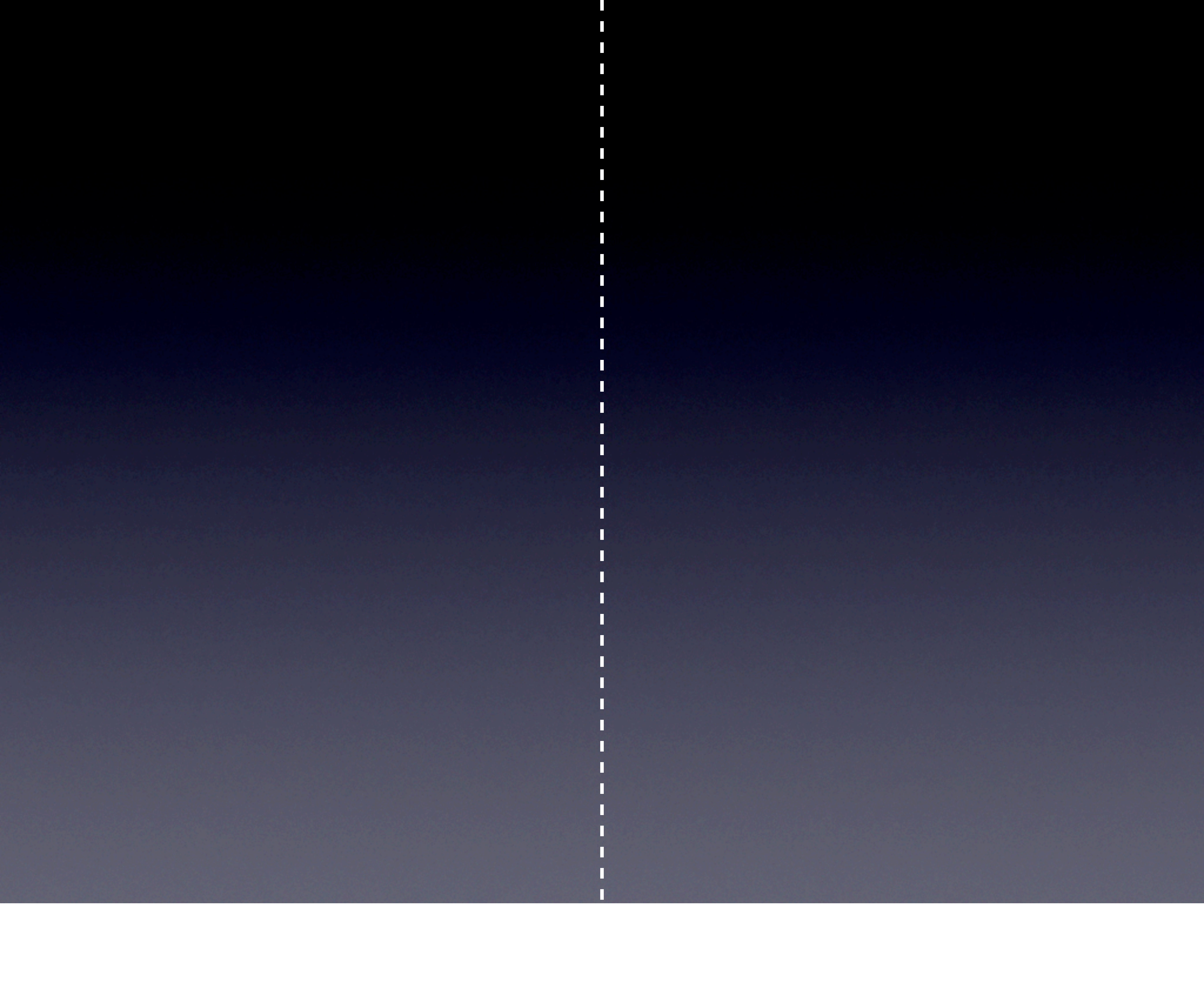
- Visits every allocated object — potentially twice
- Garbage collection time is proportional to number of objects created

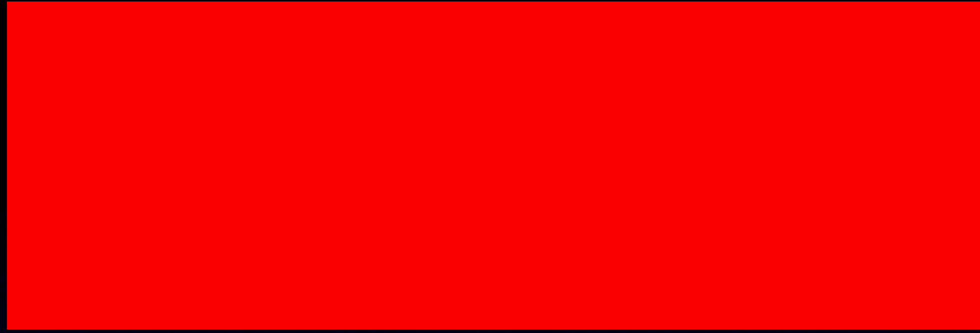
Copy Collector

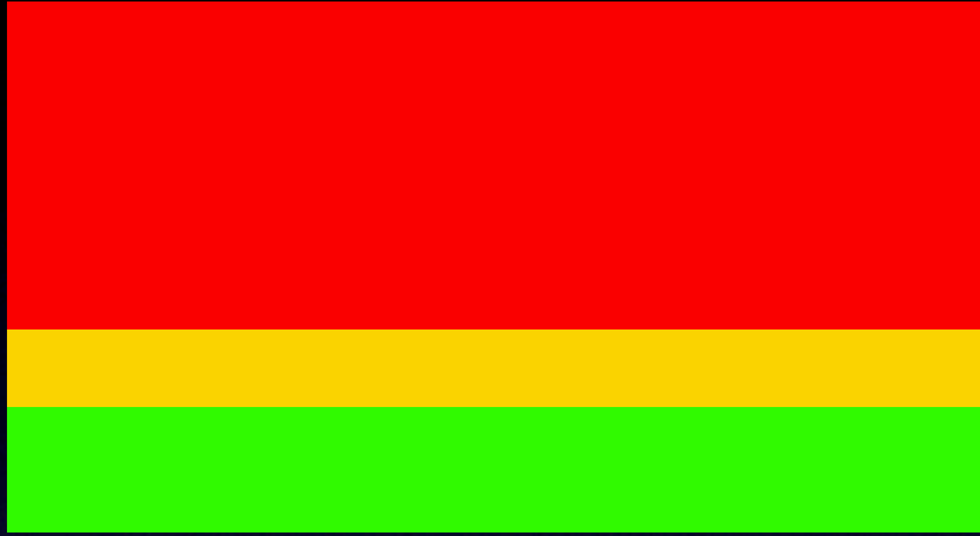
- Very different approach
- Splits the memory into two halves or banks
- All memory allocations occur from one of the banks, the other half is unused

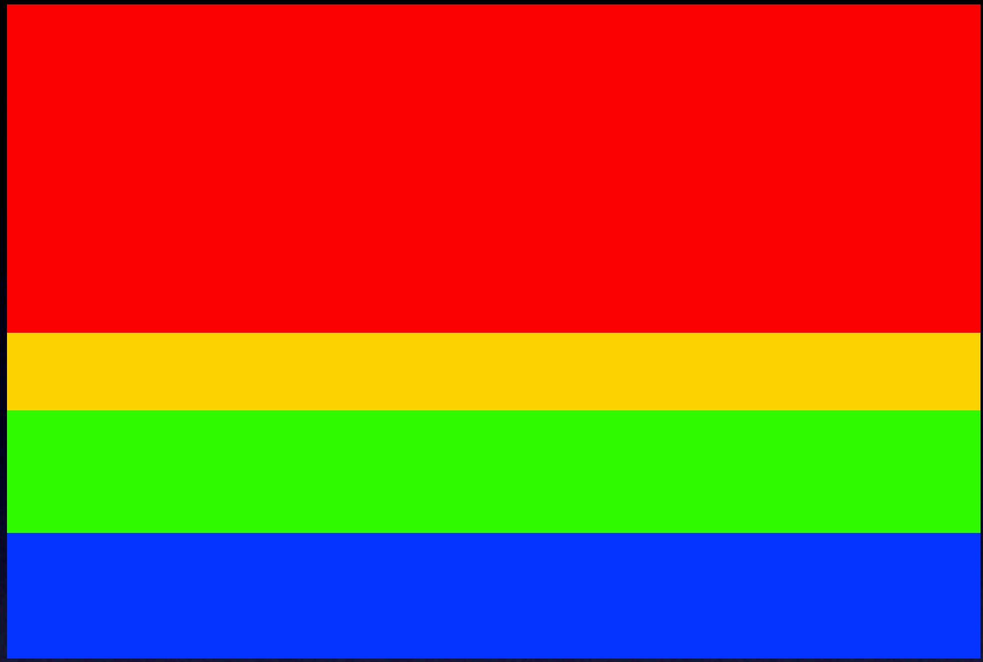
Copy Collector

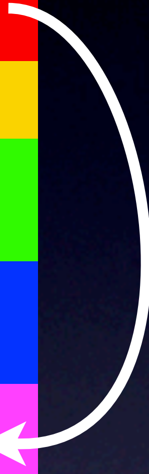
- Collector traces from the root set as per usual
- Rather than marking as ‘in-use’, it copies reachable objects from one bank of memory to the other
- Result is that all reachable objects are now in the new bank

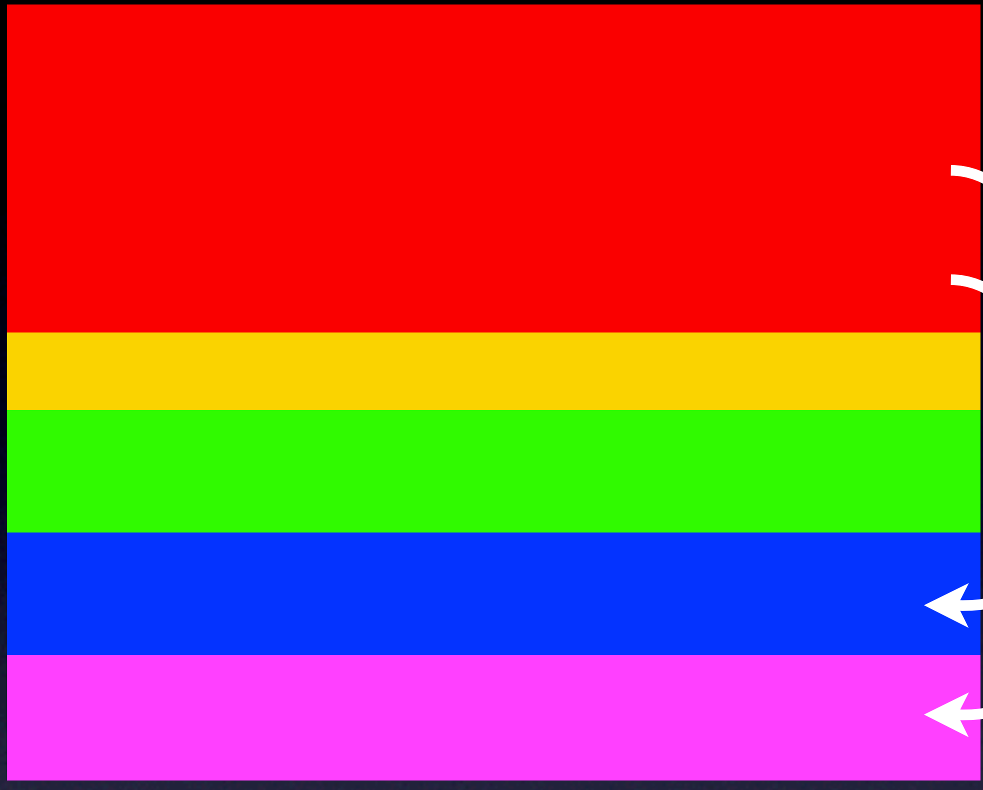


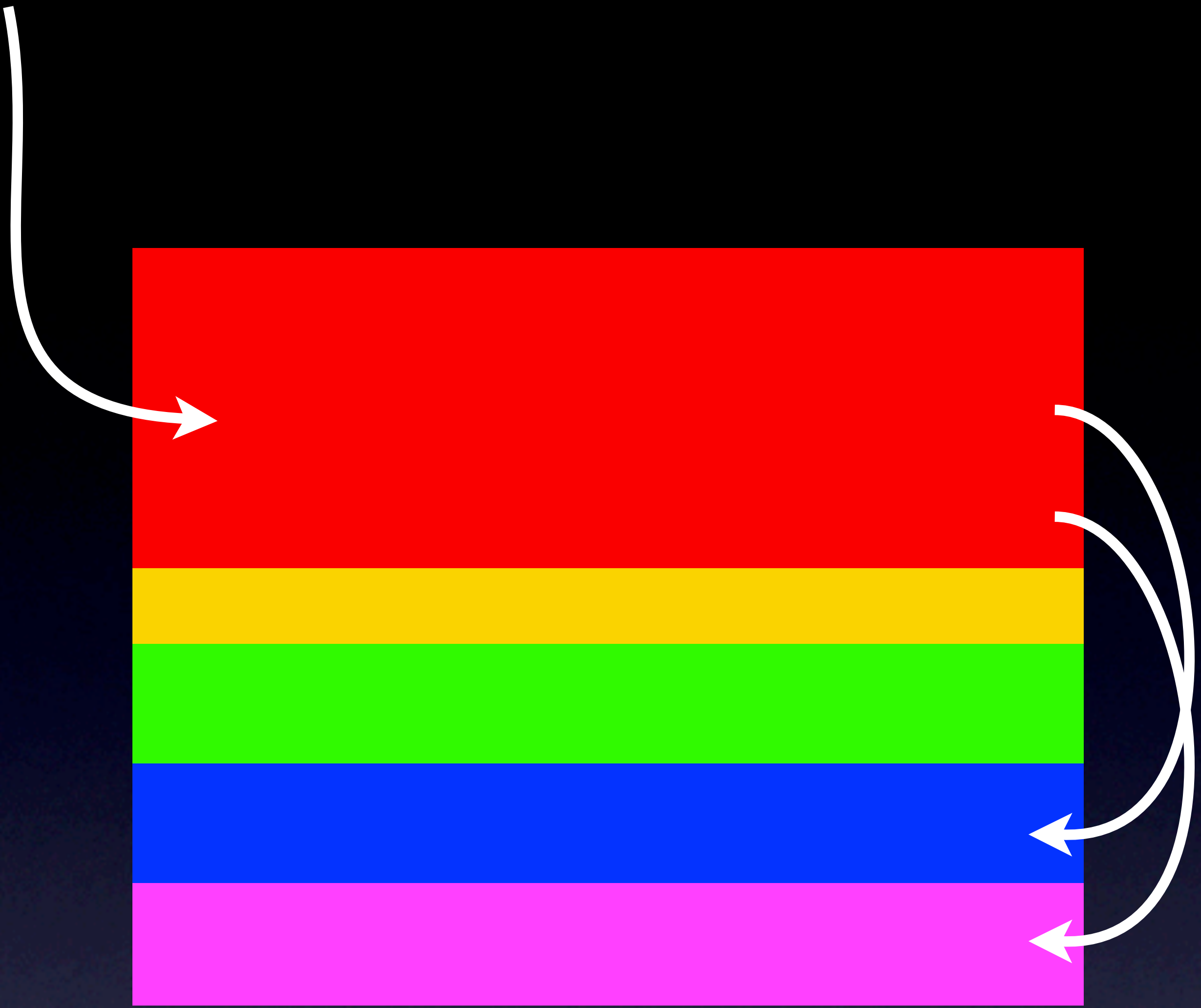


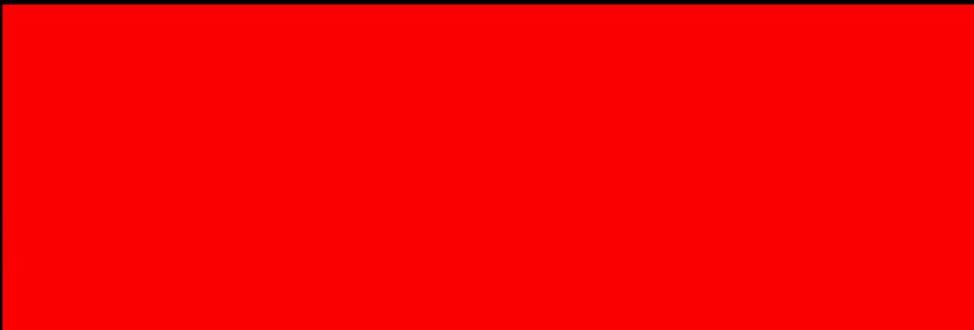


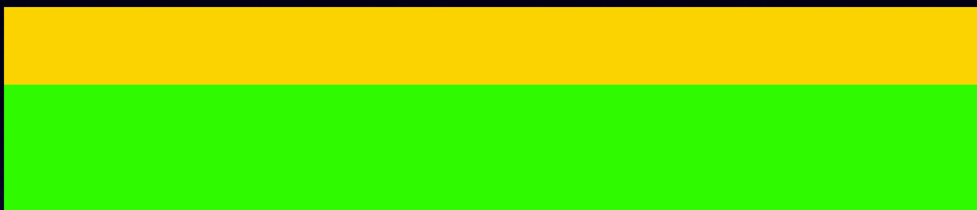


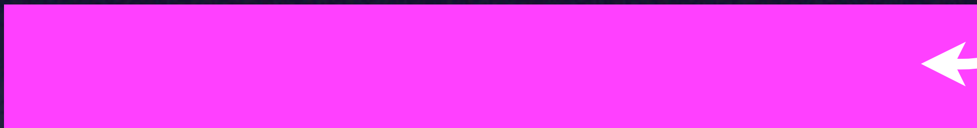


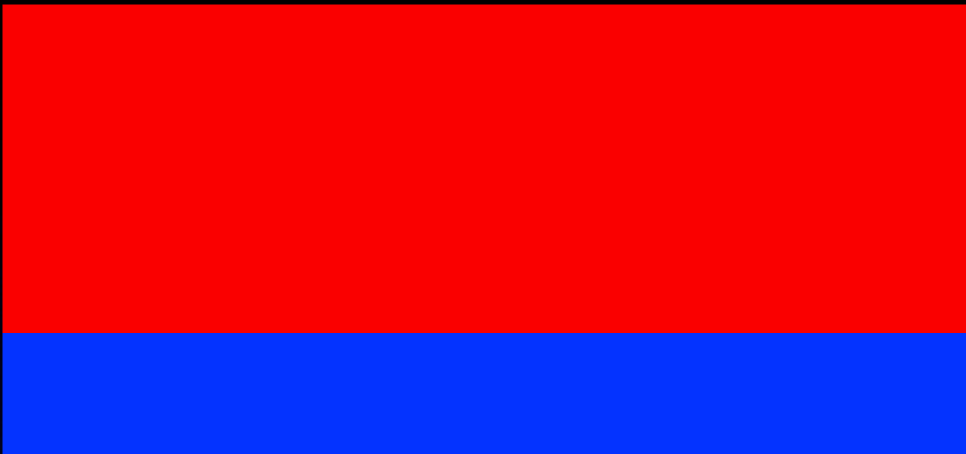


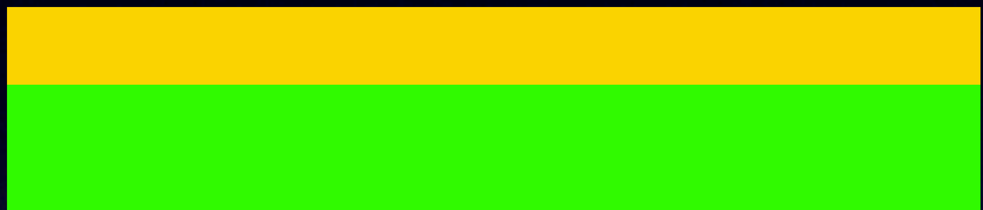


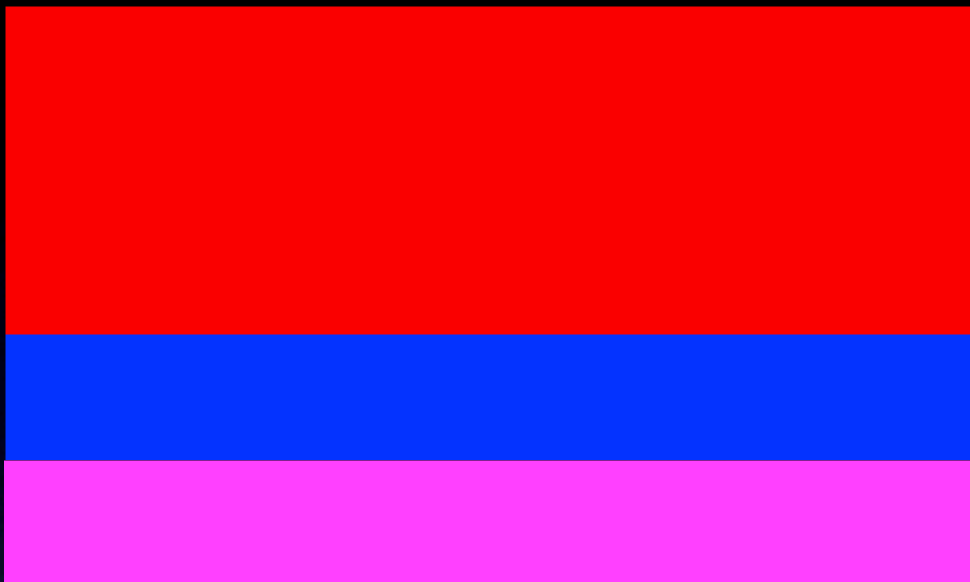


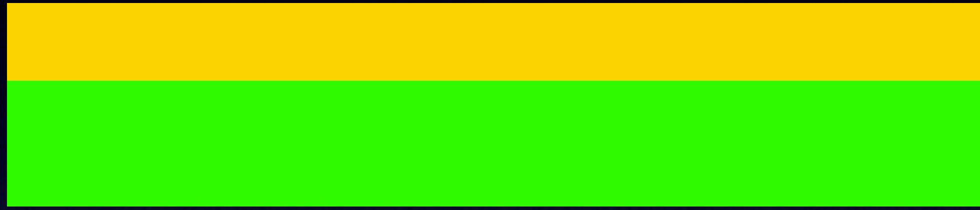


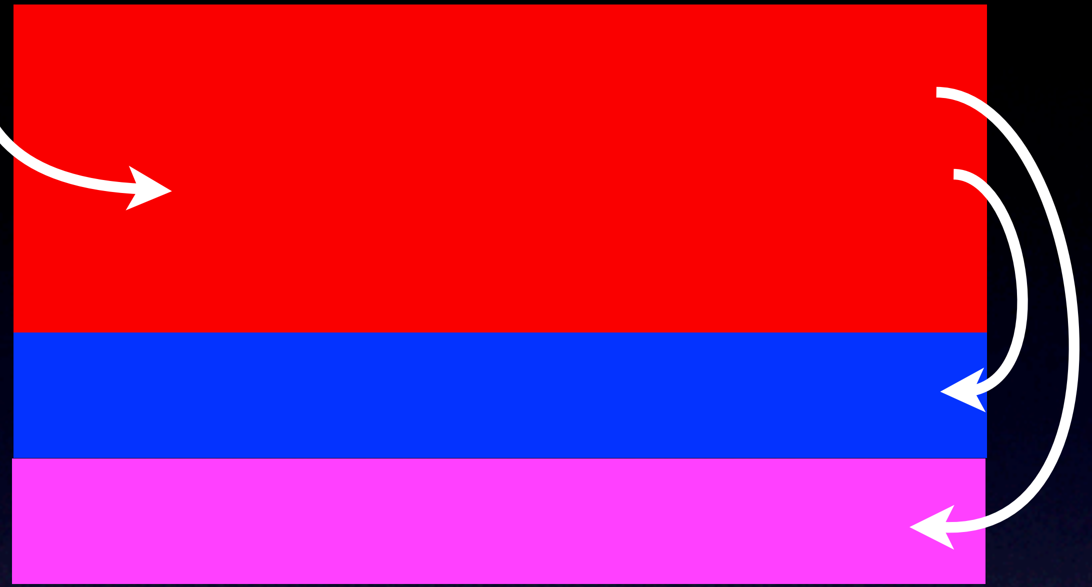


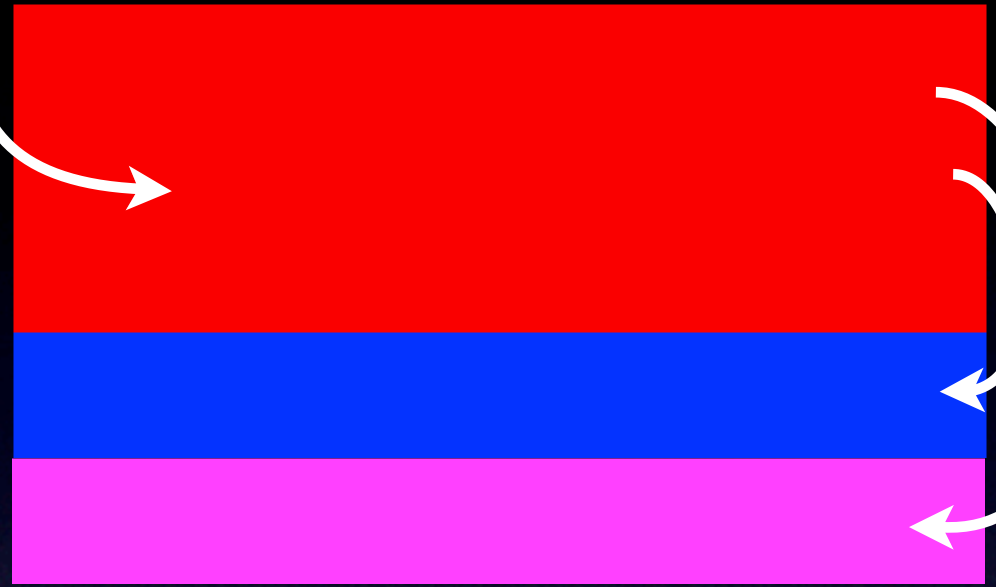


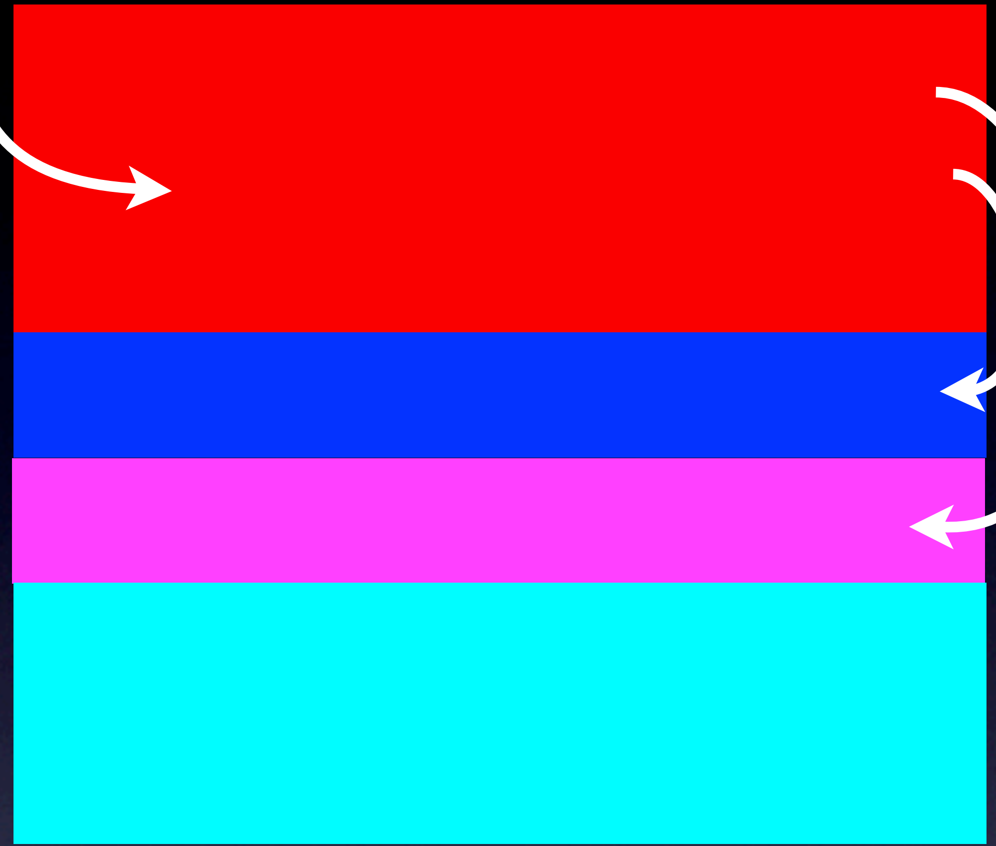












Copy Collector

- Collector fixes up object pointers to point to new location in memory
- Allocation is switched to happen from the new bank
- Next time GC happens it switches back to the previous bank

1st point definitely requires the world stopped or it'll go horrible wrong!

Copy Collector

- Copy Collector much faster than mark-sweep
- Only visits the reachable objects
- Memory allocation becomes trivial
- Java traditionally uses a Copy Collector
- Half of memory unused

Conservative GC

- GC relies on being able to find pointers to objects
- Easy when language designed for GC
  - Structure of object (i.e. what is a pointer and what isn't) is known by GC
  - GC can then find pointers in objects

Conservative GC

- Some languages (C++) don’t distinguish pointers from other bytes in memory
- GC then has to be conservative
- If these bytes could be an object pointer, then I’ll assume they are
- Means that objects may stay around longer than necessary

Is it an Object Pointer?

- Get potential pointer from memory
- Does it ‘point’ into a block of memory that has been allocated?
- If so, it is possibly a pointer so mark that block of memory (object) as ‘in-use’
- If not, not a valid pointer

GC Issues

- Stop-the-world
  - Program pauses periodically
  - What if something interesting happens during a pause?
- Modern GC algorithms that allow collection to happen in parallel with program execution

GC Issues

- Garbage Collection visits every object
- Research has shown that objects tend to have short lifetimes
- Some modern GC algorithms partition the heap into young and old objects

GC issues

- If an object exists for a predetermined period, it is moved from the young partition to the old partition
- Young partition is GC'd more regularly than the old partition
- Less objects to visit so quicker GC

Summary

- Garbage collection
- Tool for automatically destroying objects
- Two basic types: mark-sweep and Copy collectors
- Language support
- GC issues