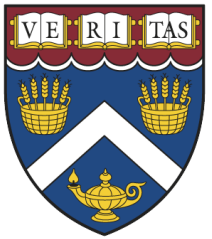
In the Last Week

- Some questions around the final project
  - What's going on with the GC footprint?
  - What are the parameters to `onPointerCreate`?
- Final exam logistics
  - Class tonight and next Monday
  - Final exam released early morning on the 15th
  - Exam due at midnight on the 19th
  - No late days for the final exam

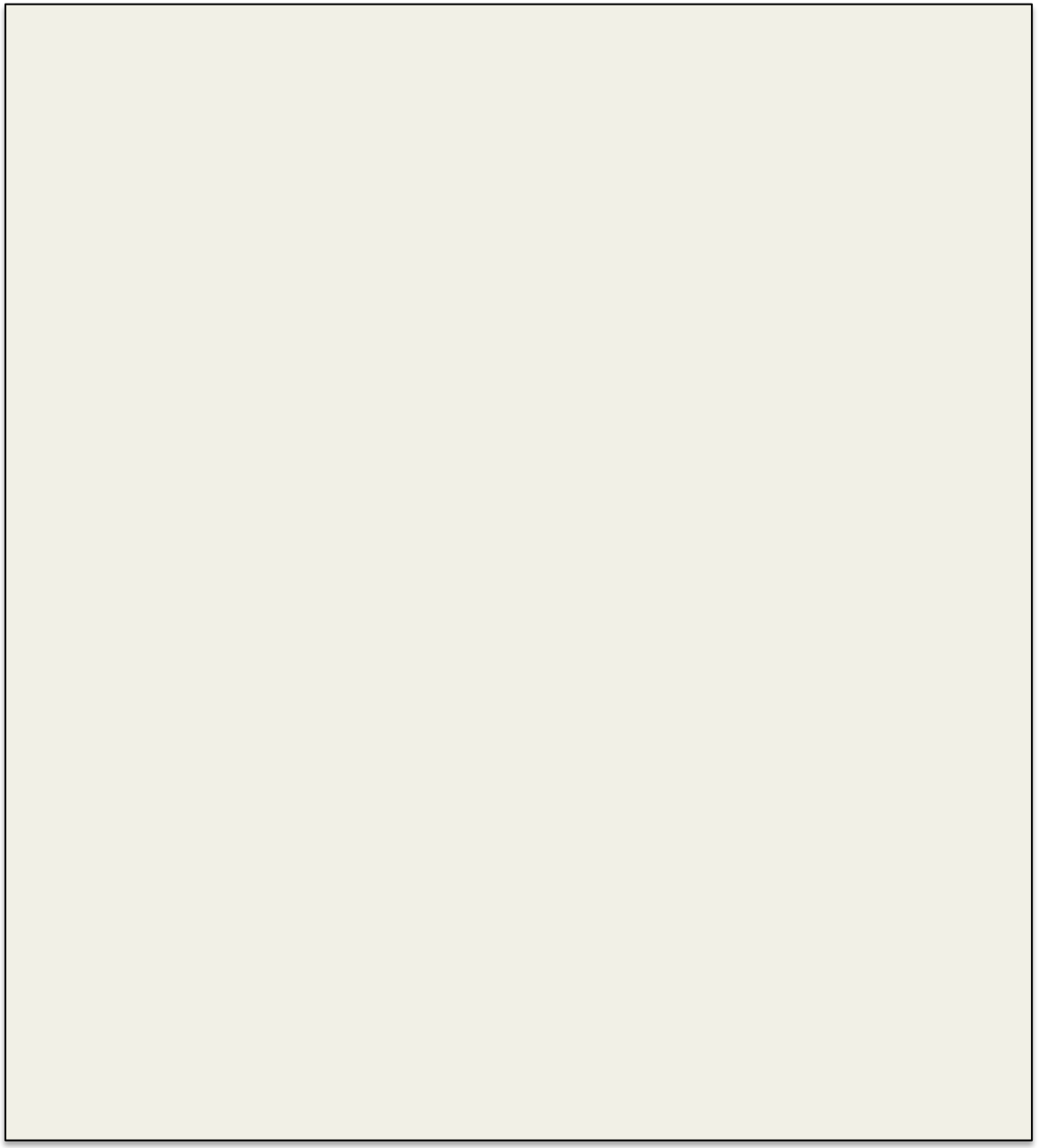
Advanced Topics

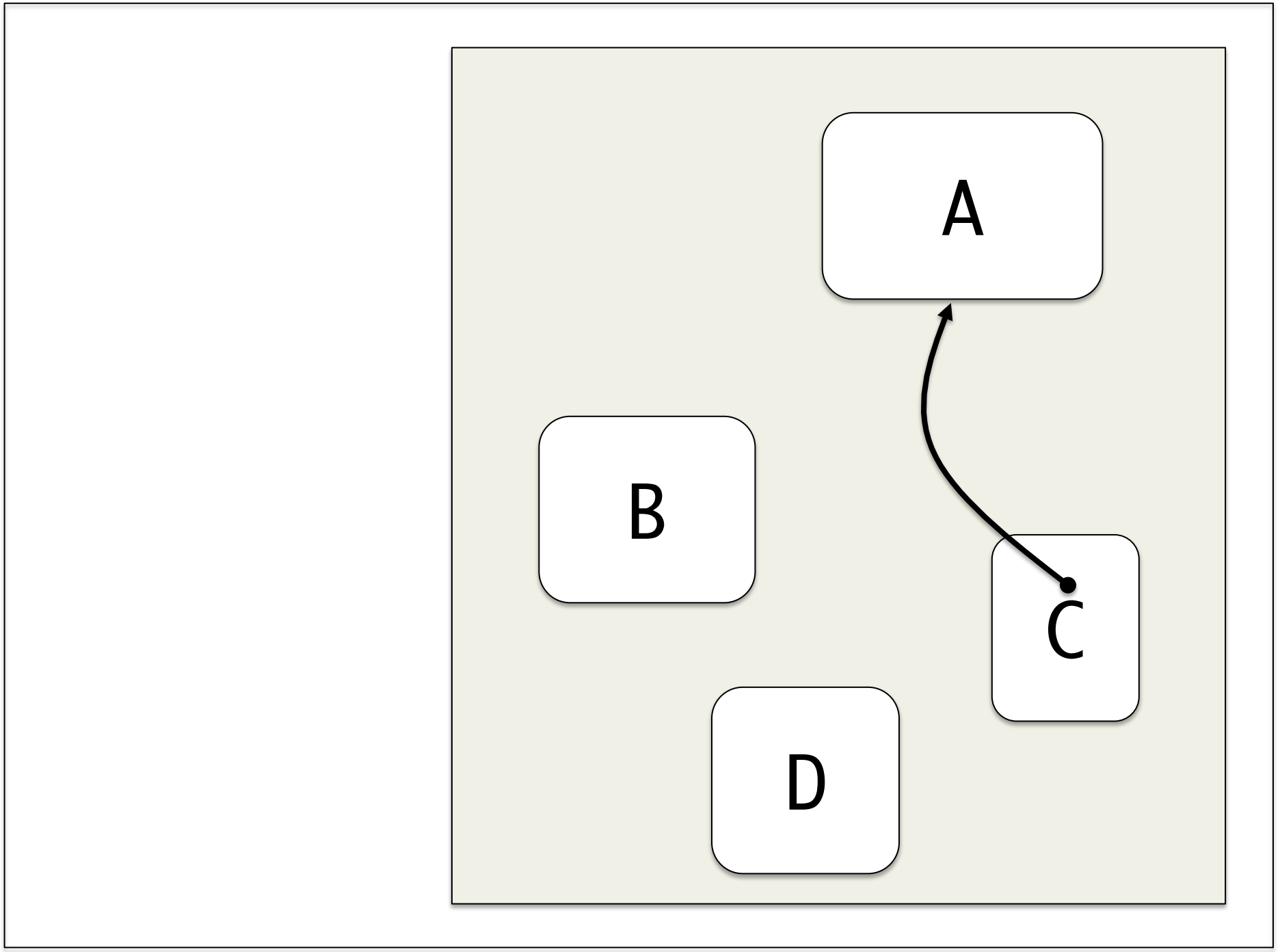


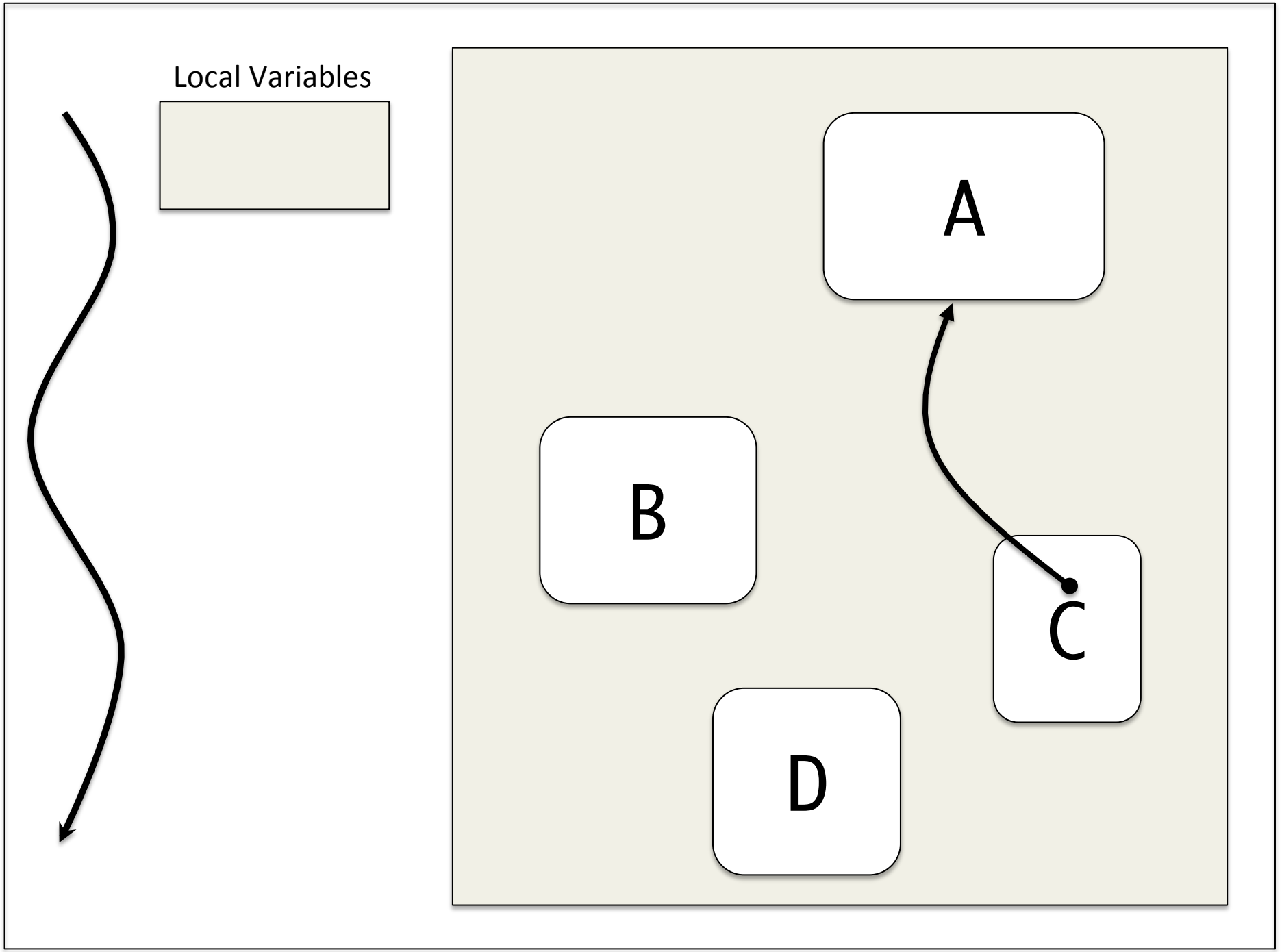
WEAK AND SOFT REFERENCES

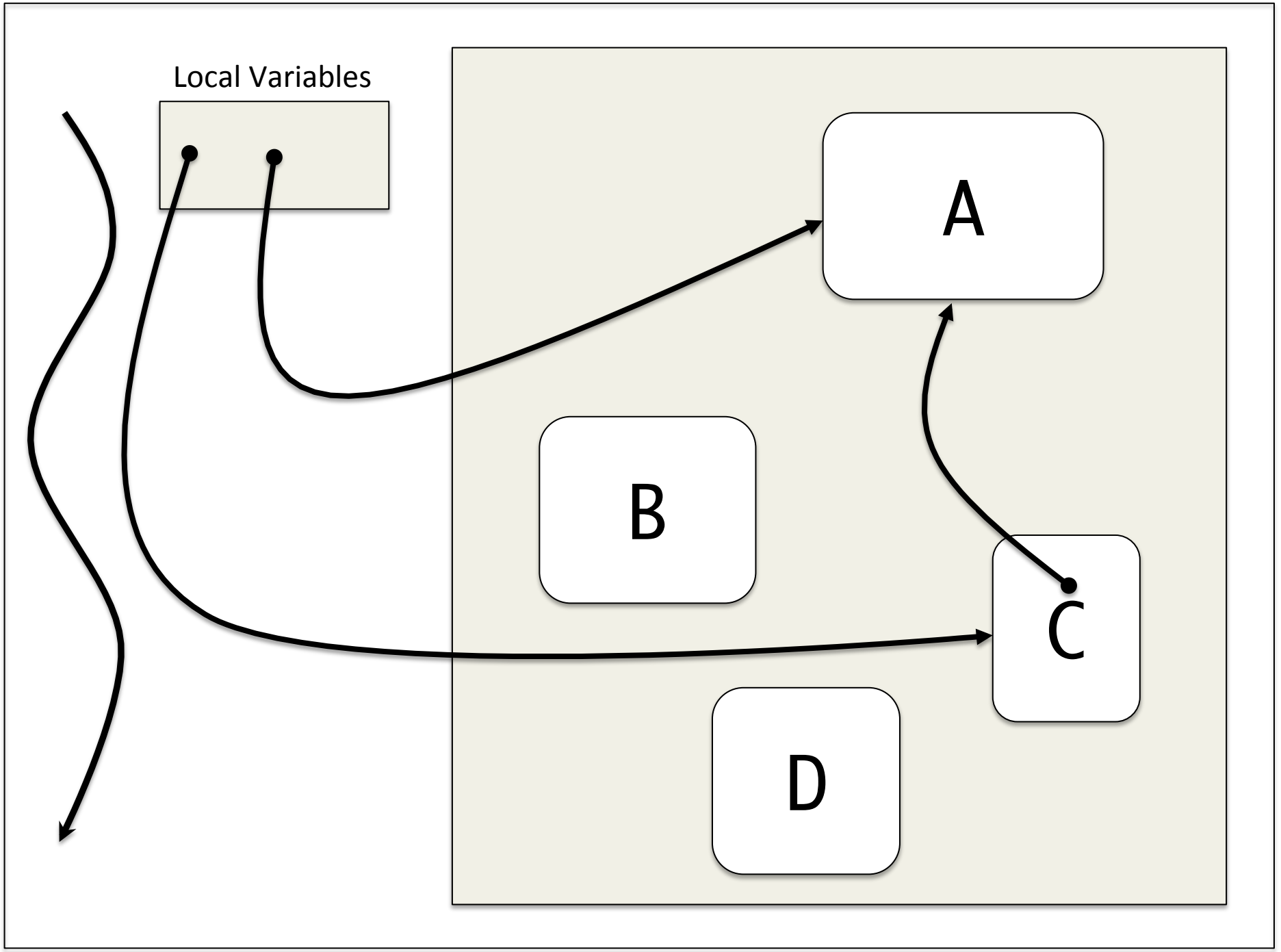
GC Reachability

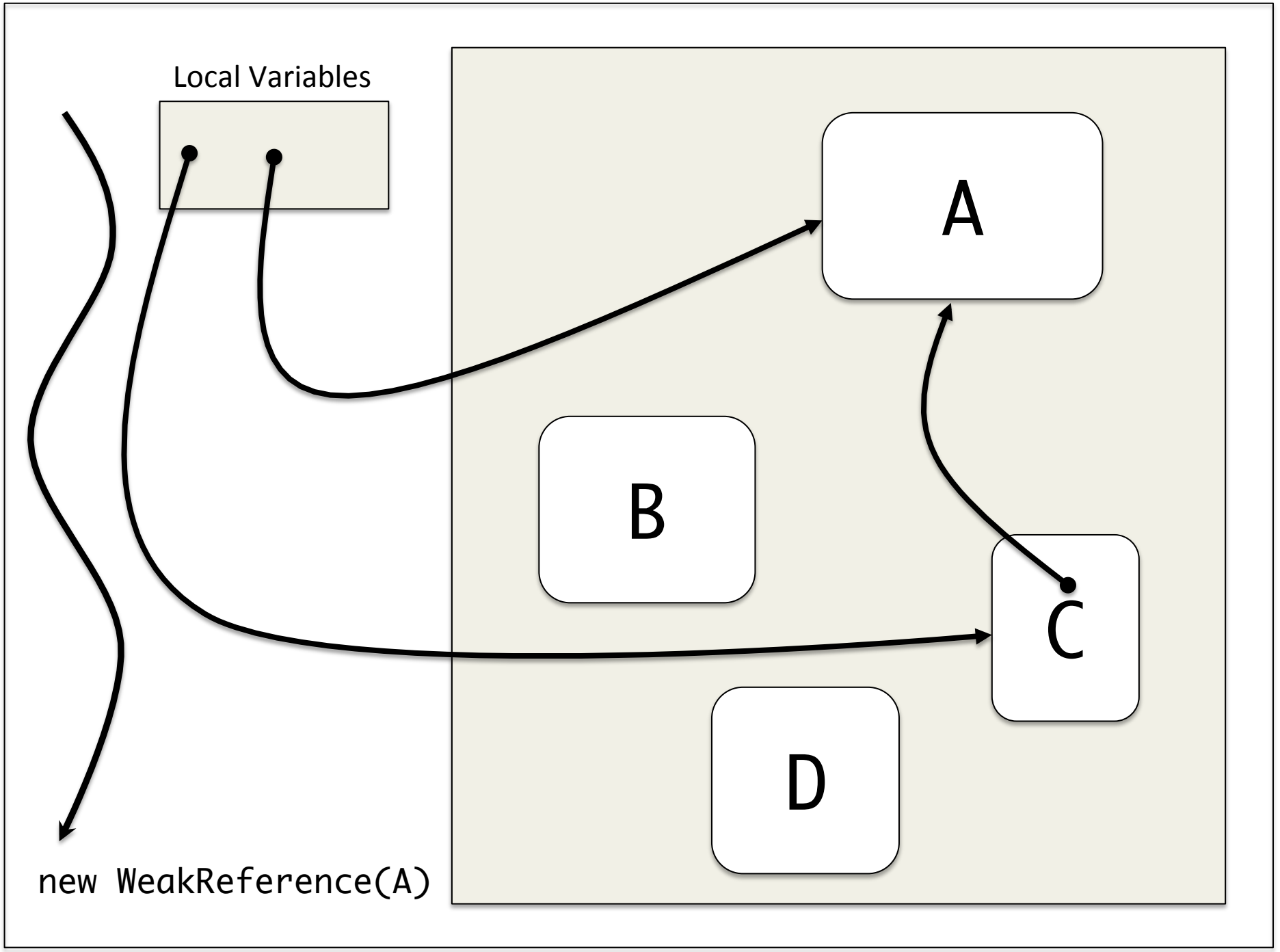
- Basic tracing model is simple
  - Roots define an entry point to the object graph
  - Live objects are reachable from the roots
- This model can sometimes be inconvenient
  - Not all references are equal
- Weak references allow reachability without liveness
  - Objects referred to only by weak references are garbage

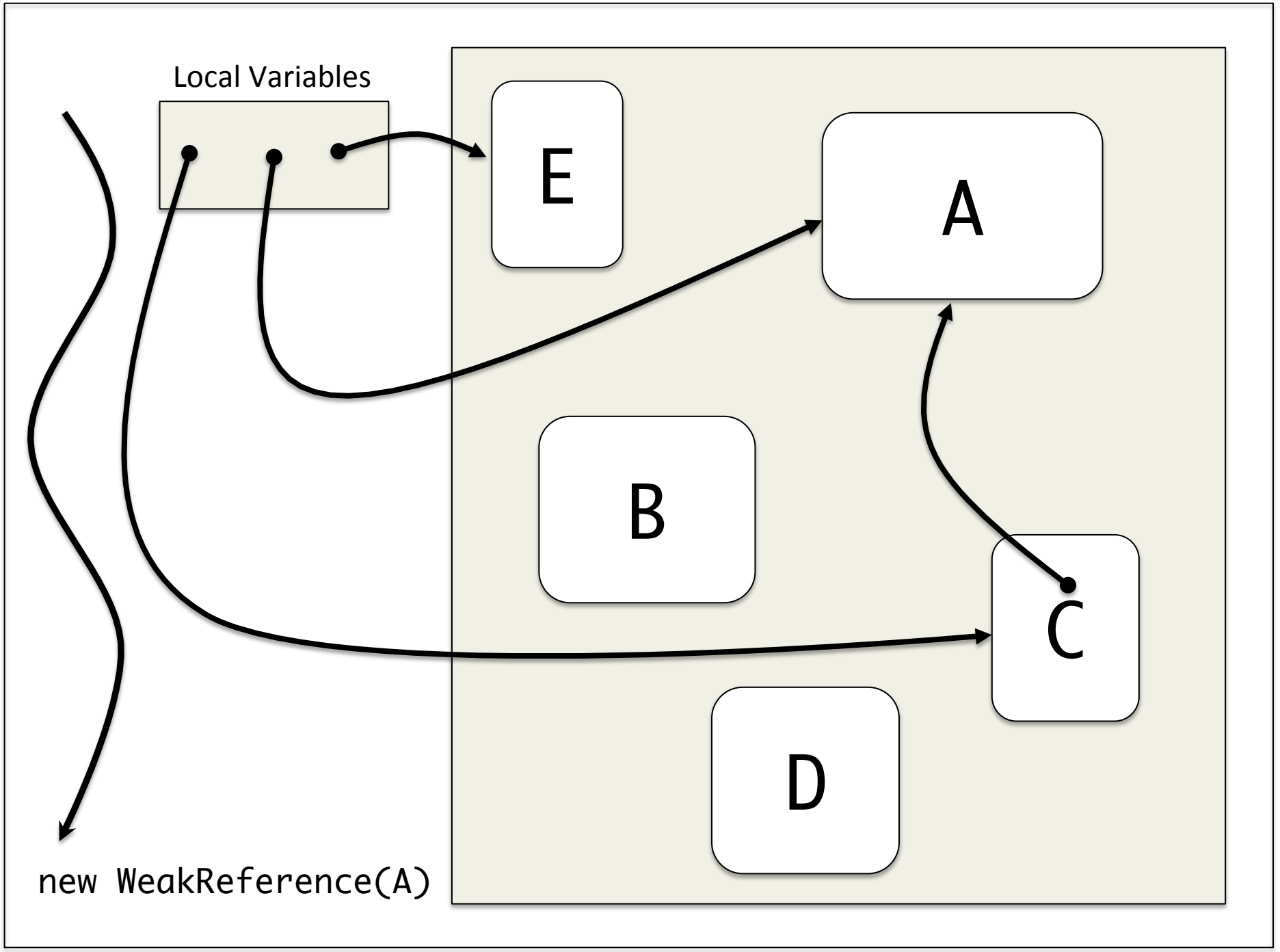


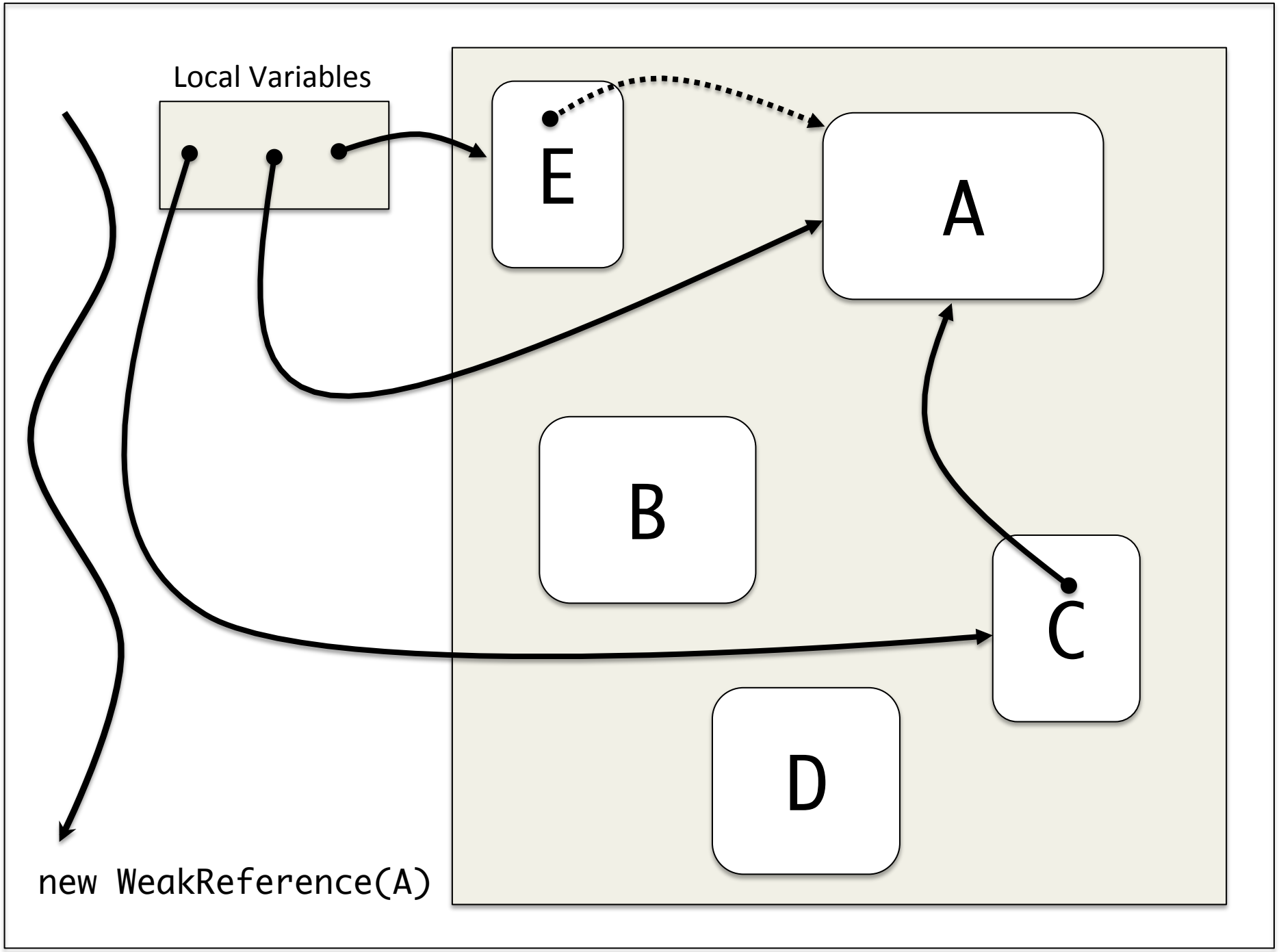

A
B
C
D

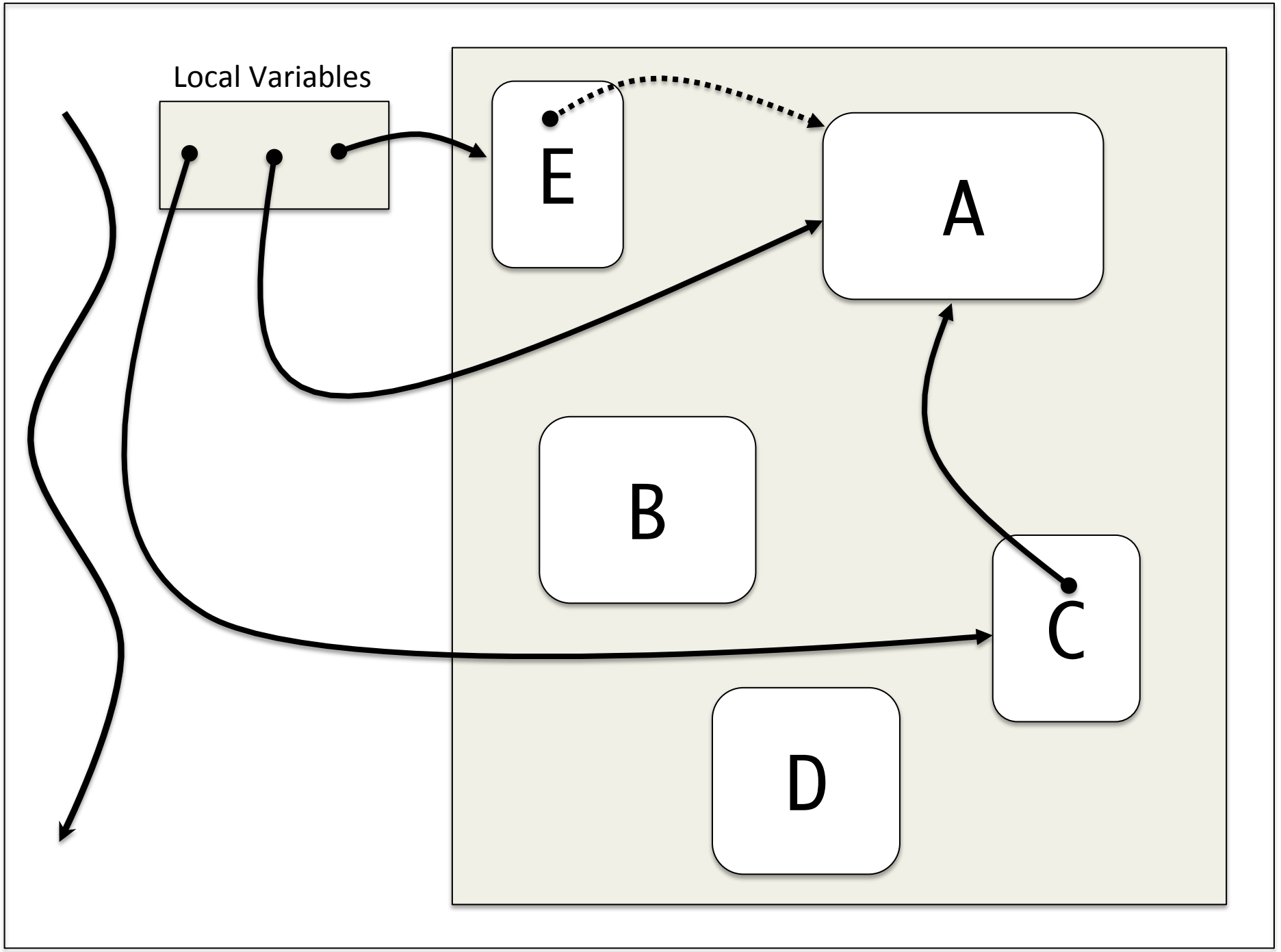

Local Variables
A
B
C
D

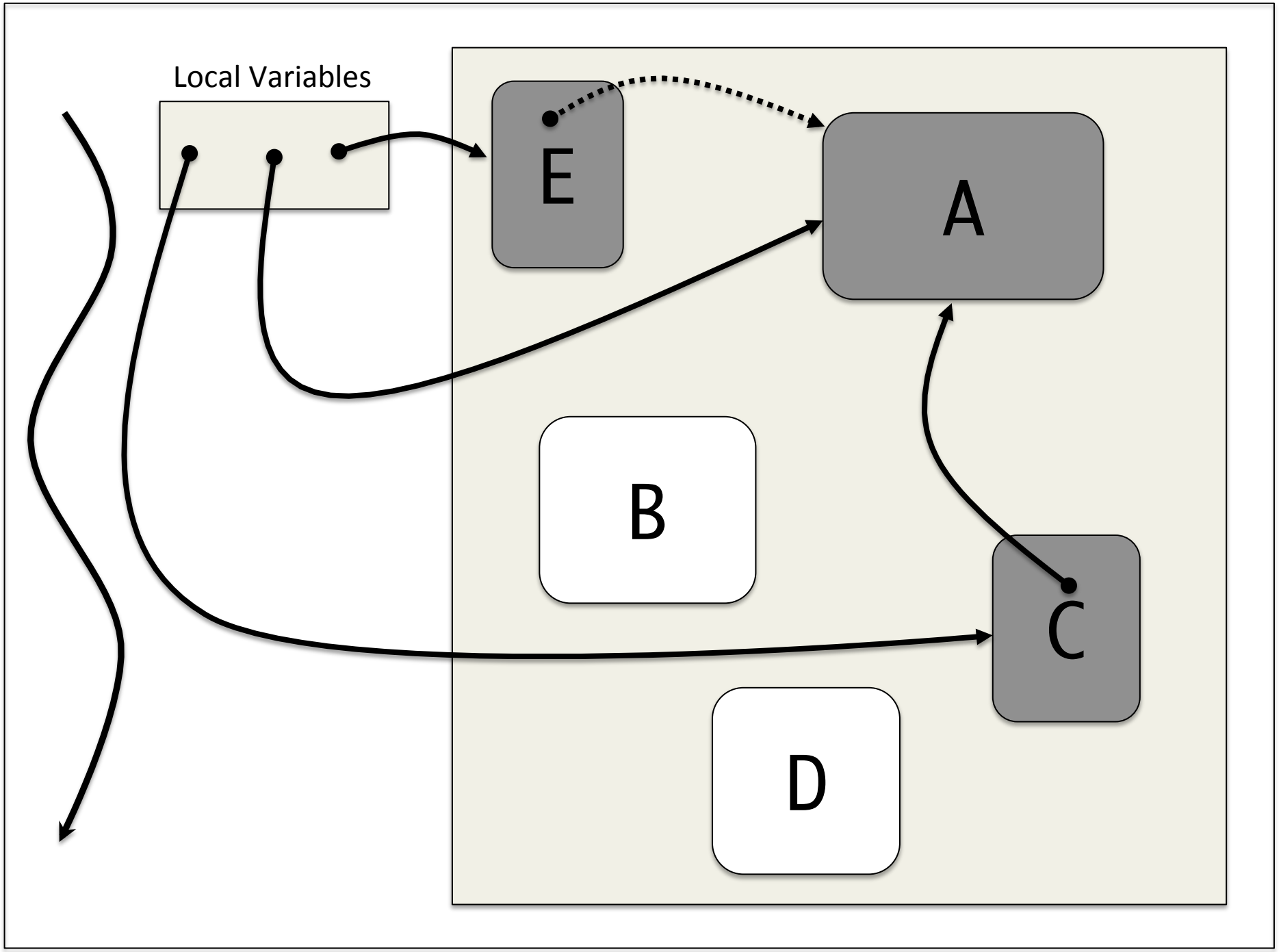

Local Variables
A
B
C
D

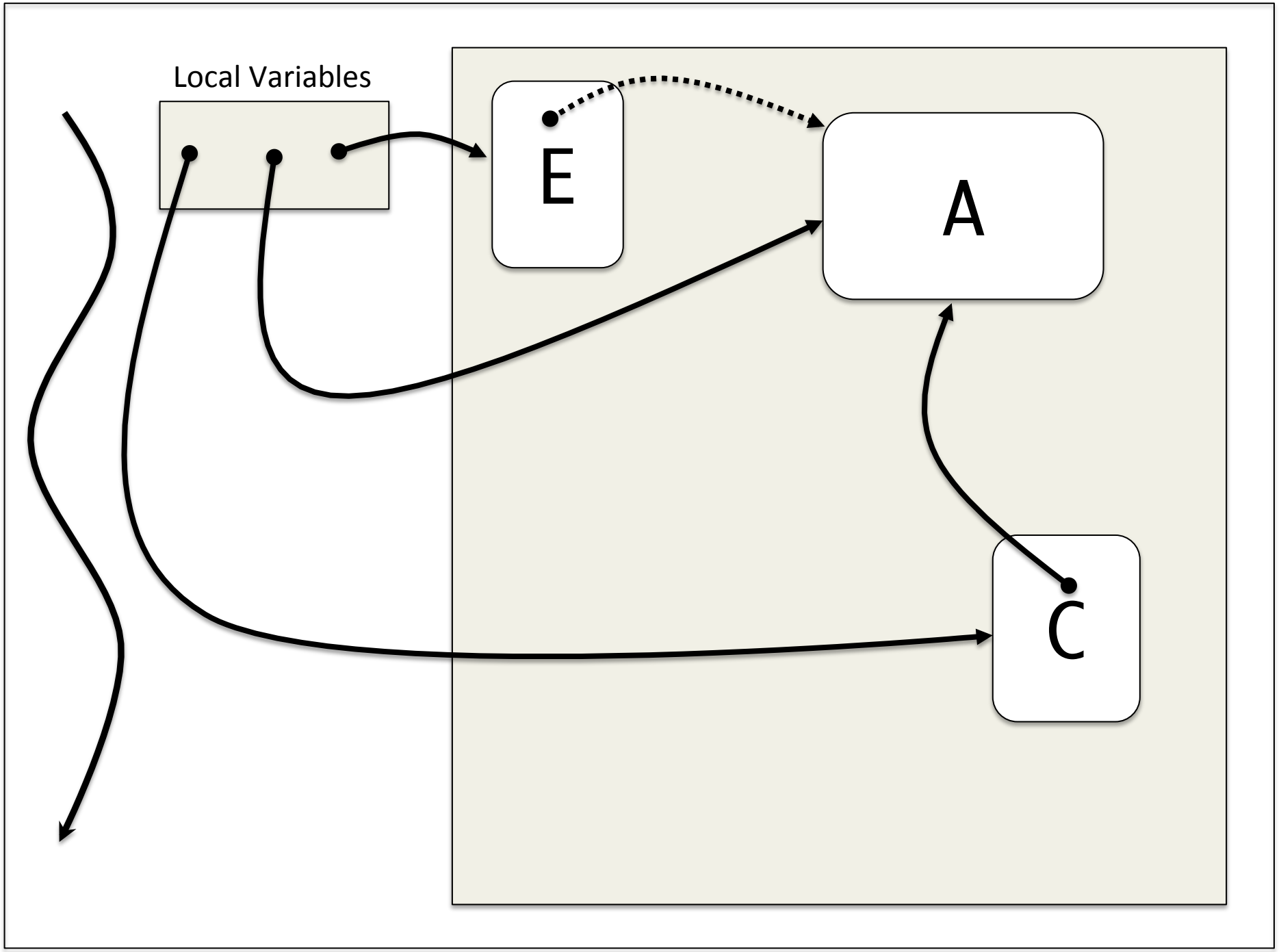

Local Variables
A
B
C
D
new WeakReference(A)

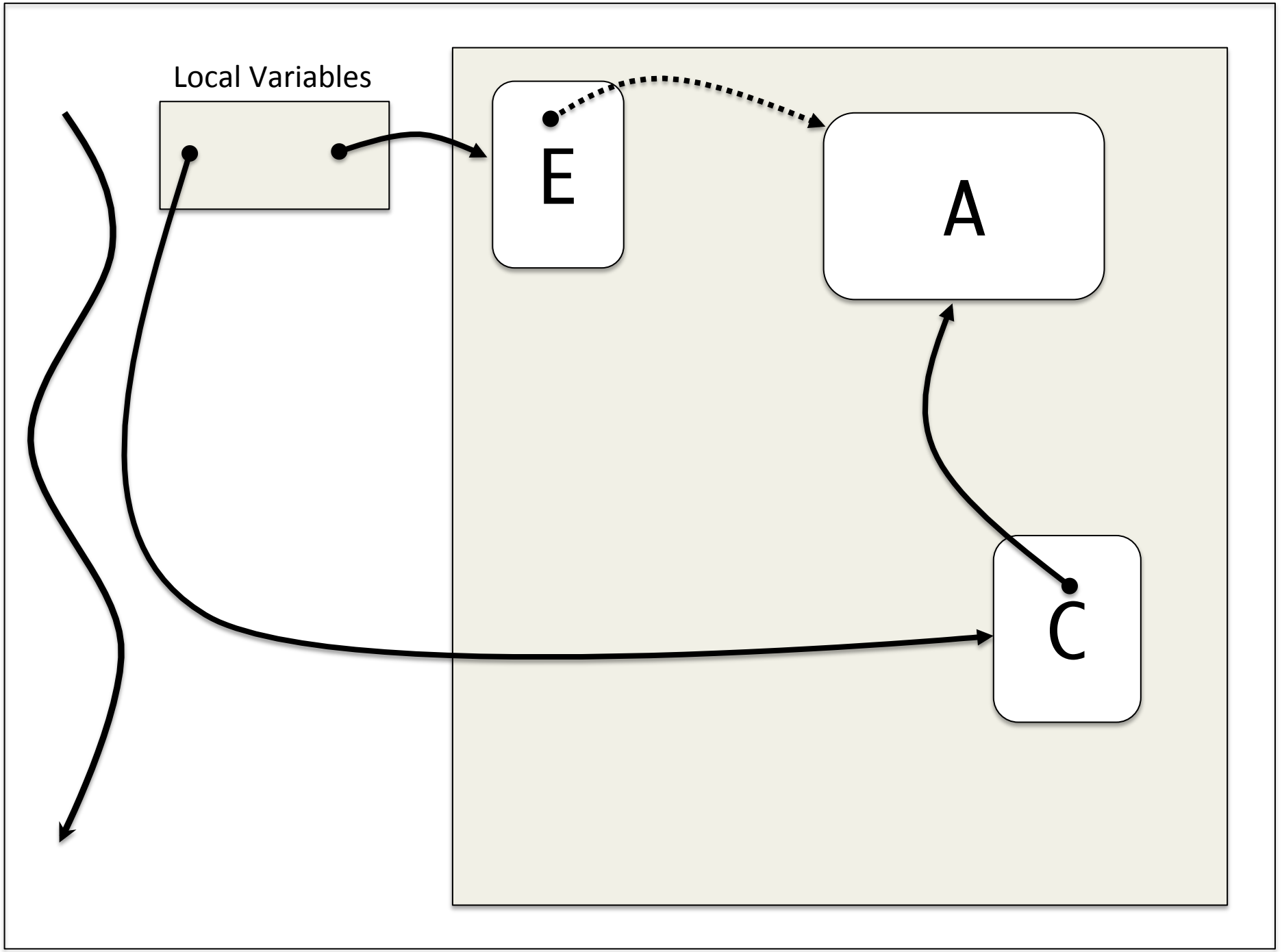

Local Variables
E
A
B
C
D
new WeakReference(A)

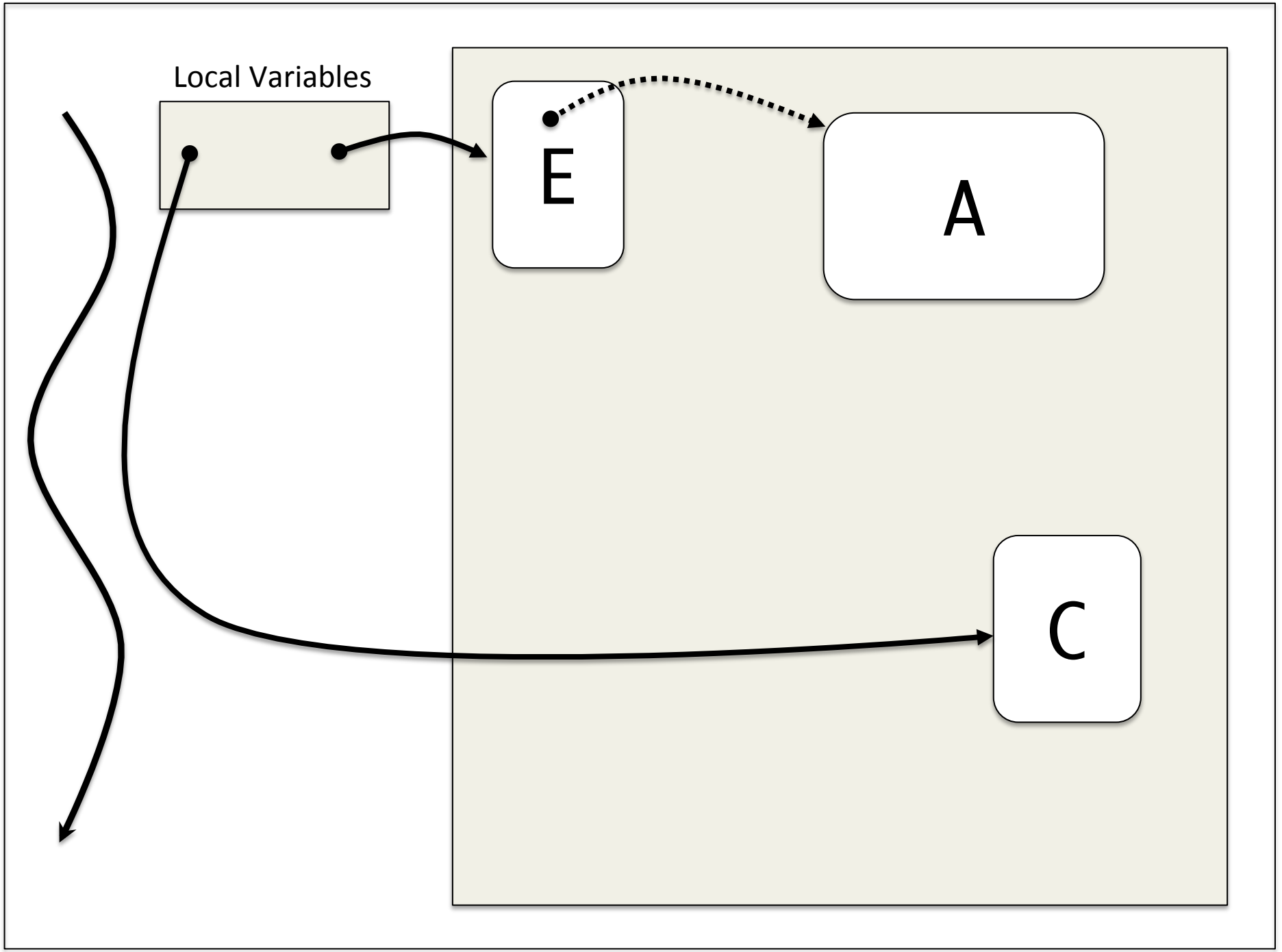

Local Variables
E
A
B
C
D
new WeakReference(A)

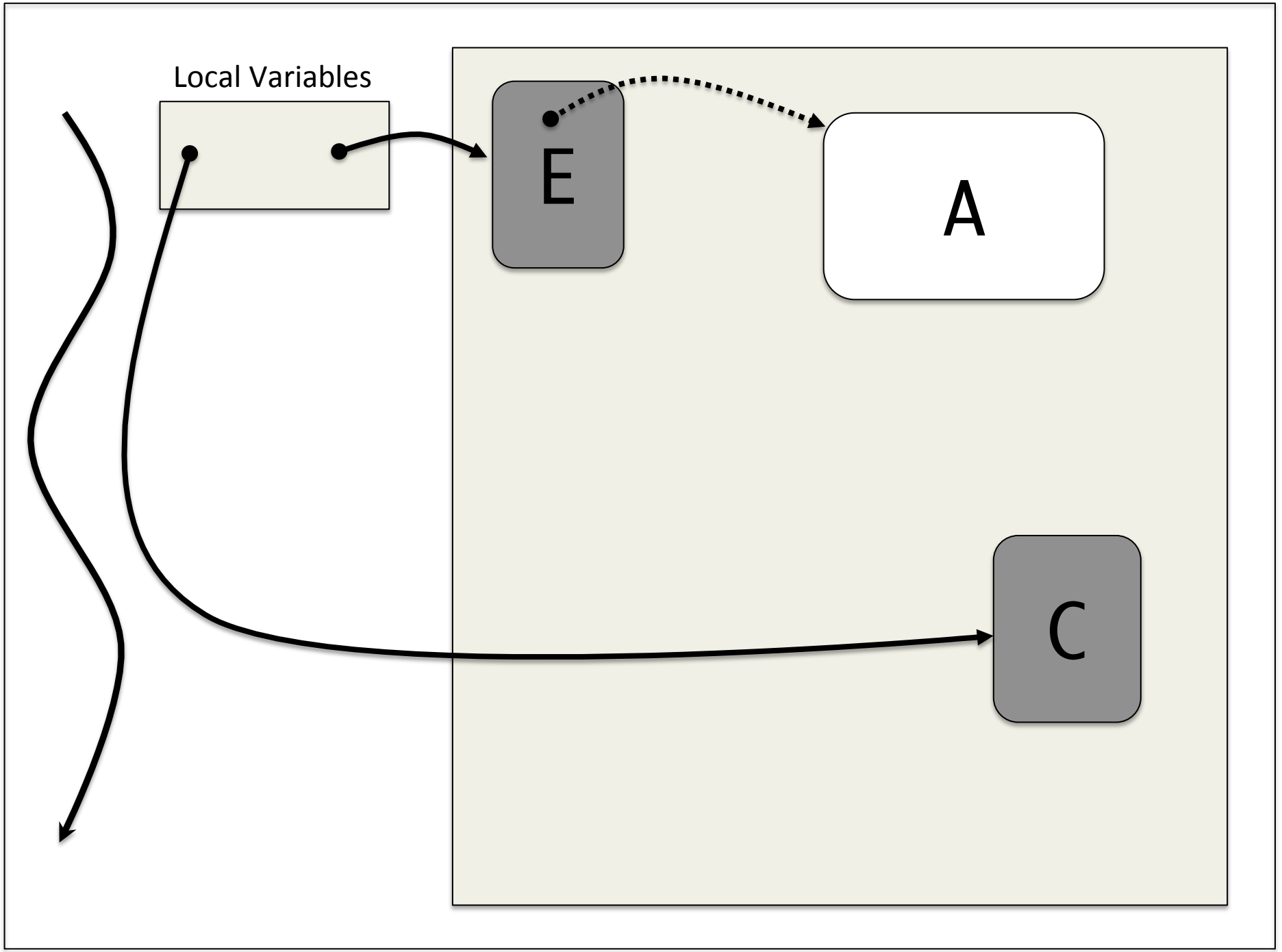

Local Variables
E
A
B
C
D

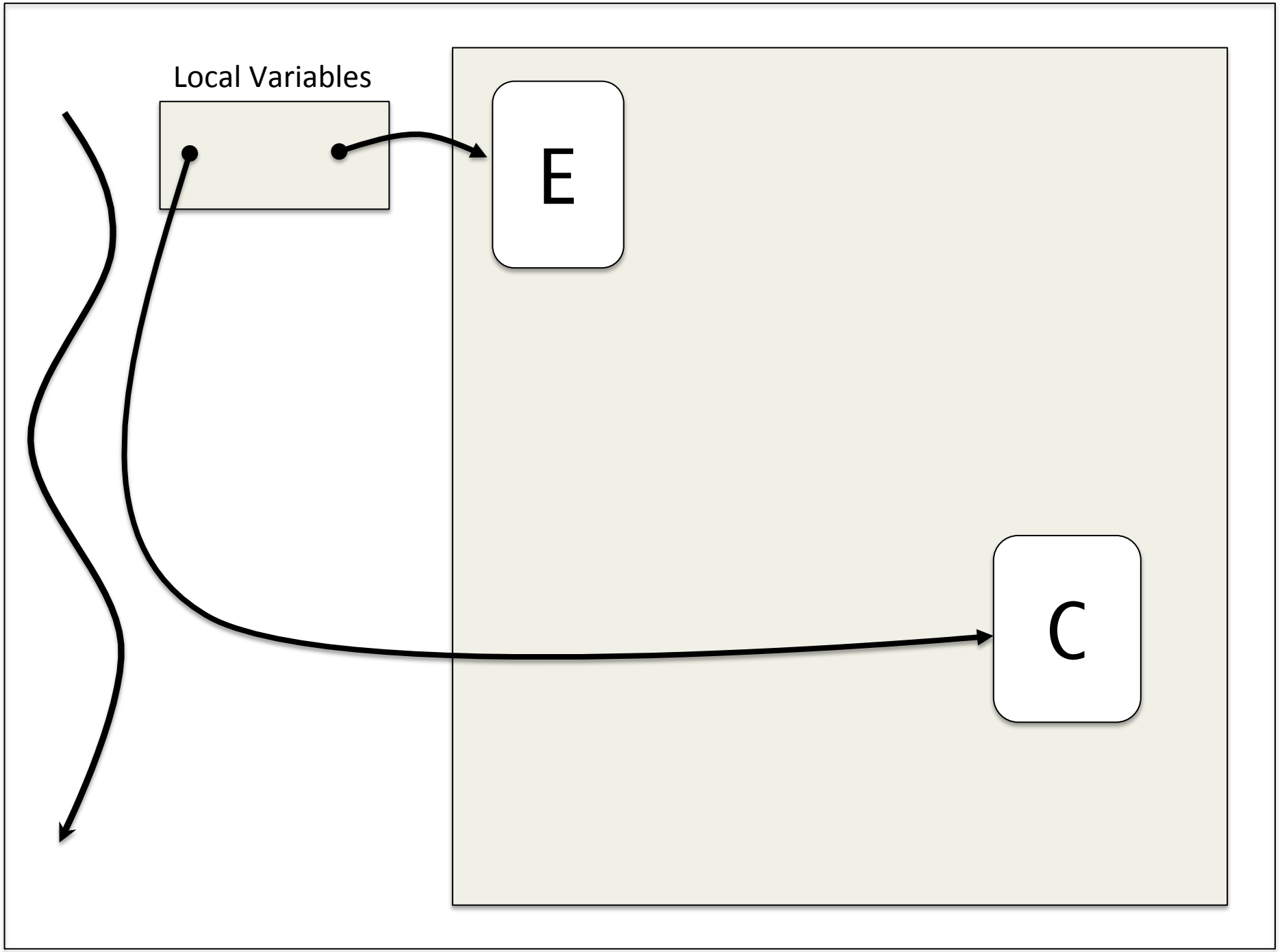

Local Variables
E
A
B
C
D


Local Variables
E
A
C


Local Variables
E
A
C


Local Variables
E
A
C


Local Variables
E
A
C


Local Variables
E
C

Weak Reference Uses

- Associative caches
  - Object is the key, value holds extra information
  - Key shouldn't keep an object alive
- Reference cycles
  - Long-standing issue with reference counting
  - Make one reference is a structure weak
- Observer pattern
  - Easily disassociate from event managers

Implementing Weak References

- WeakReference class has special semantics
  - Tracing algorithms treat it differently
  - Value may be changed by the garbage collector
- WeakReference is a primordial class
  - VM doesn't allow redefinition
  - Referent field is volatile
- GC must keep track of weak references
  - Need to nullify them if the object is collected

Reference Queues

- Gives visibility into object reachability
  - Weak reference is registered with a queue
- GC adds reference to the queue when nullifying
  - WeakReference object is placed on the queue
  - Referent is freed
- Developer can check for objects on queue
  - Knows which objects have been freed
  - Can take action on other related objects

Soft References

- Similar semantics to weak references
  - Soft reference does not keep object alive
  - Represented as a separate object in the Java heap
- Soft references are not cleared immediately
  - Softly-reachable object may survive multiple GCs
  - Free depends on memory pressure
- Guaranteed to be cleared before OutOfMemory

Using Soft References

- Large data structures that can be re-created
  - Result of complex computation
  - File, network or database resources
- Store structures in memory for fast access
  - Refer to cached structure using soft refence
  - Freed when the GC needs to reclaim the memory
- Program performance tied to GC algorithm

Freeing Soft References

- Free schedule is implementation specific
  - Can happen at any time from developer perspective
  - Spec guarantees that it is before memory runs out
- May want to release objects incrementally
  - GC tends to run slowly when memory is tight
  - Massive free operation leads to large pause
- VMs implement caching algorithms

Soft References in Hotspot

- SoftReference object has a timestamp
  - Set to the time of the last GC
  - Shows roughly when the reference was last accessed
- GC knows how much live data is in the heap
  - Figures out based on the mark phase
- Pre-set ratio of survival time to garbage ratio
  - Each GC has a dynamic limit on time since last use
  - Least recently used objects are freed first

Phantom References

- Third type of reference object in Java
  - Not commonly used
- Serves only as a notification of collection
  - Associate a PhantomReference with a ReferenceQueue
  - Test for the reference to be added to the queue
- Allows for complex clean-up behavior
  - Unpredictable

Reachability Hierarchy

- Strongly reachable
  - Can be reached by following normal references
- Softly reachable
  - Reachable only through soft references
- Weakly reachable
  - Reachable only through weak references
- Phantom reachable
  - Reachable only through phantom reference
- Unreachable
  - No references to the object

FINALIZATION

Destructors

- Destructors are a common concept
  - Code that runs when an object is freed
  - Useful for cleaning up resources
- Implementation in C++ is intuitive
  - Two types of storage – stack and heap
  - Stack objects freed when they leave scope
  - Heap objects have explicit free call
- Destructors run at well-understood times

Finalizers

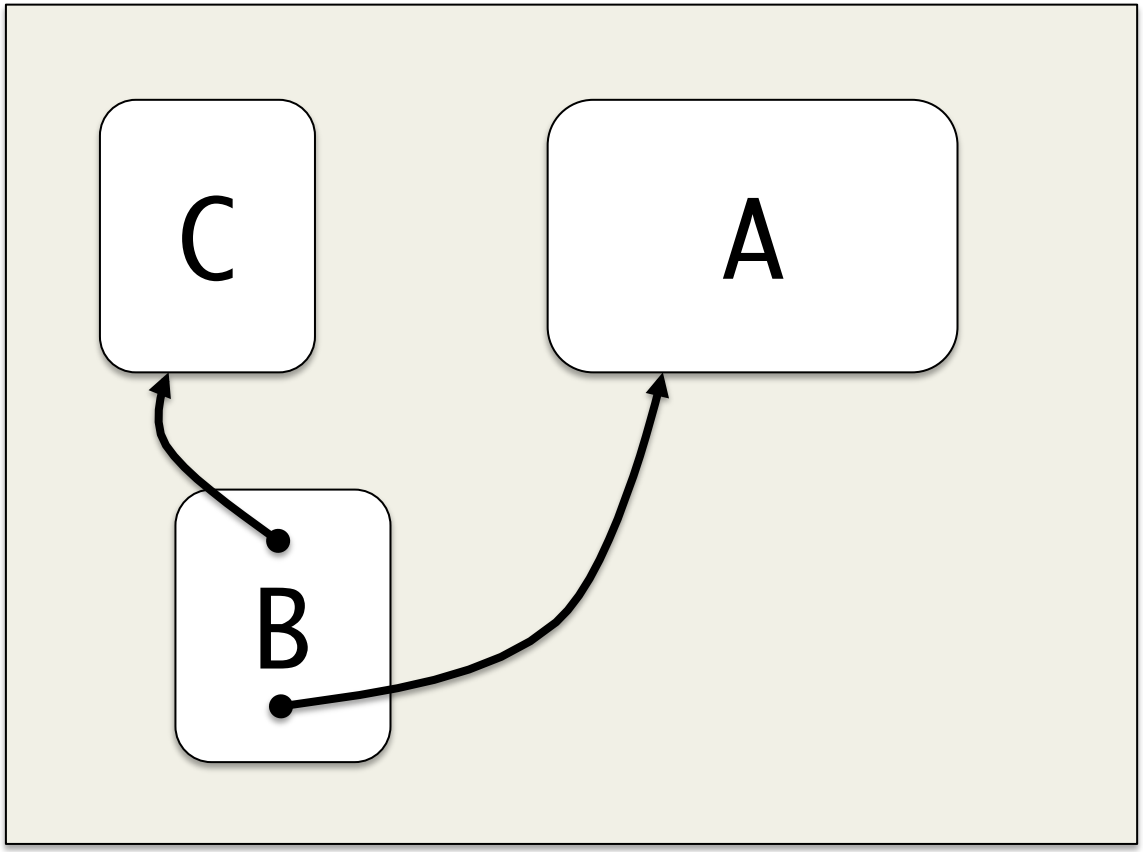
- Similar concept in managed languages
  - Any Java class can override Object.finalize() method
  - Called when an object is reclaimed by the GC
- Two key differences
  - Timing of finalizer is unpredictable
  - Finalization done in a separate thread or threads
- Finalize methods are harder to get right

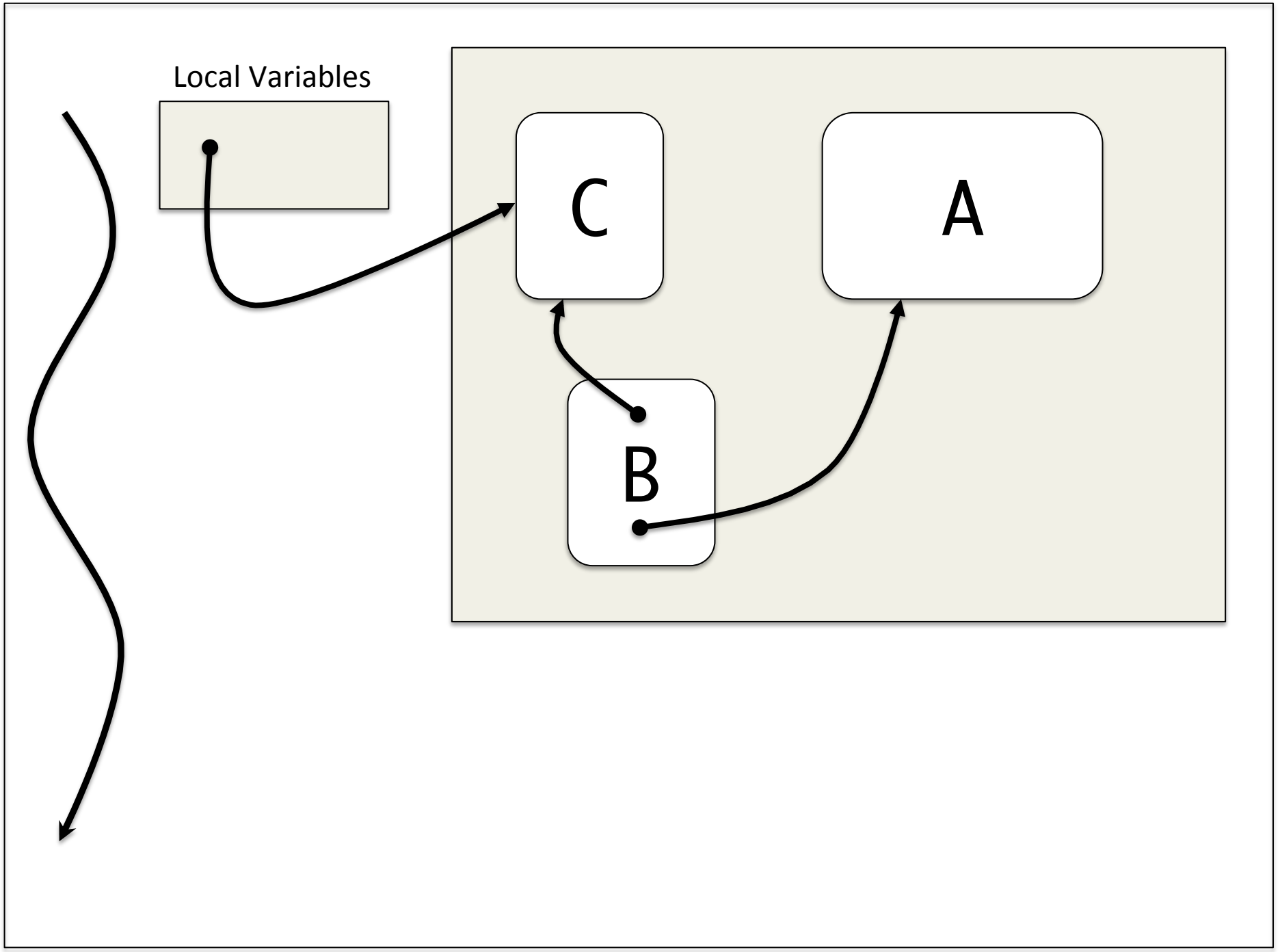
Finalization Semantics

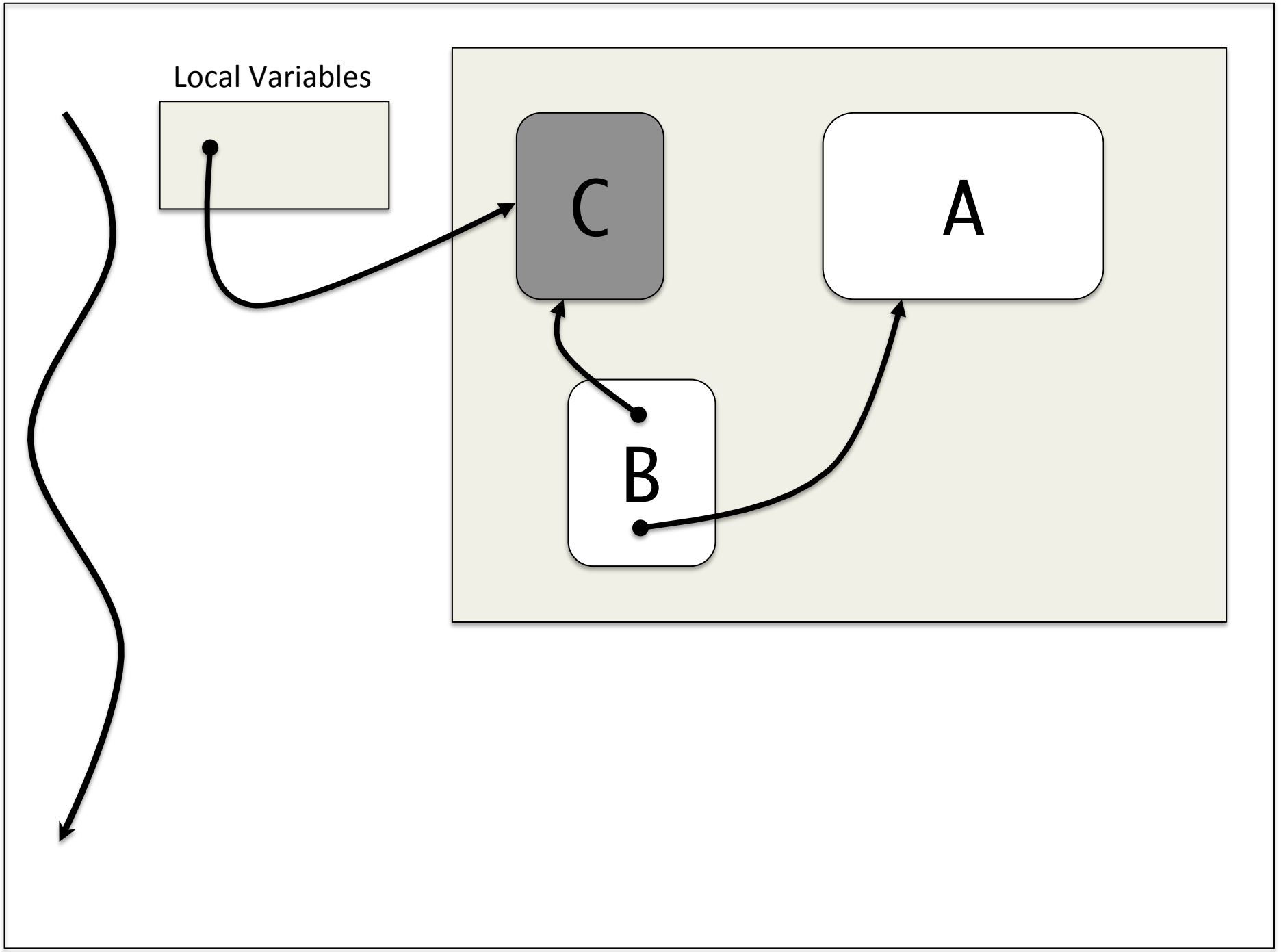
- Finalize method can contain any Java code
  - Recommended use is to clean up resources
- Finalization is handled by a daemon thread
  - Thread guaranteed not to currently hold any locks
- Exceptions thrown by a finalizer are ignored

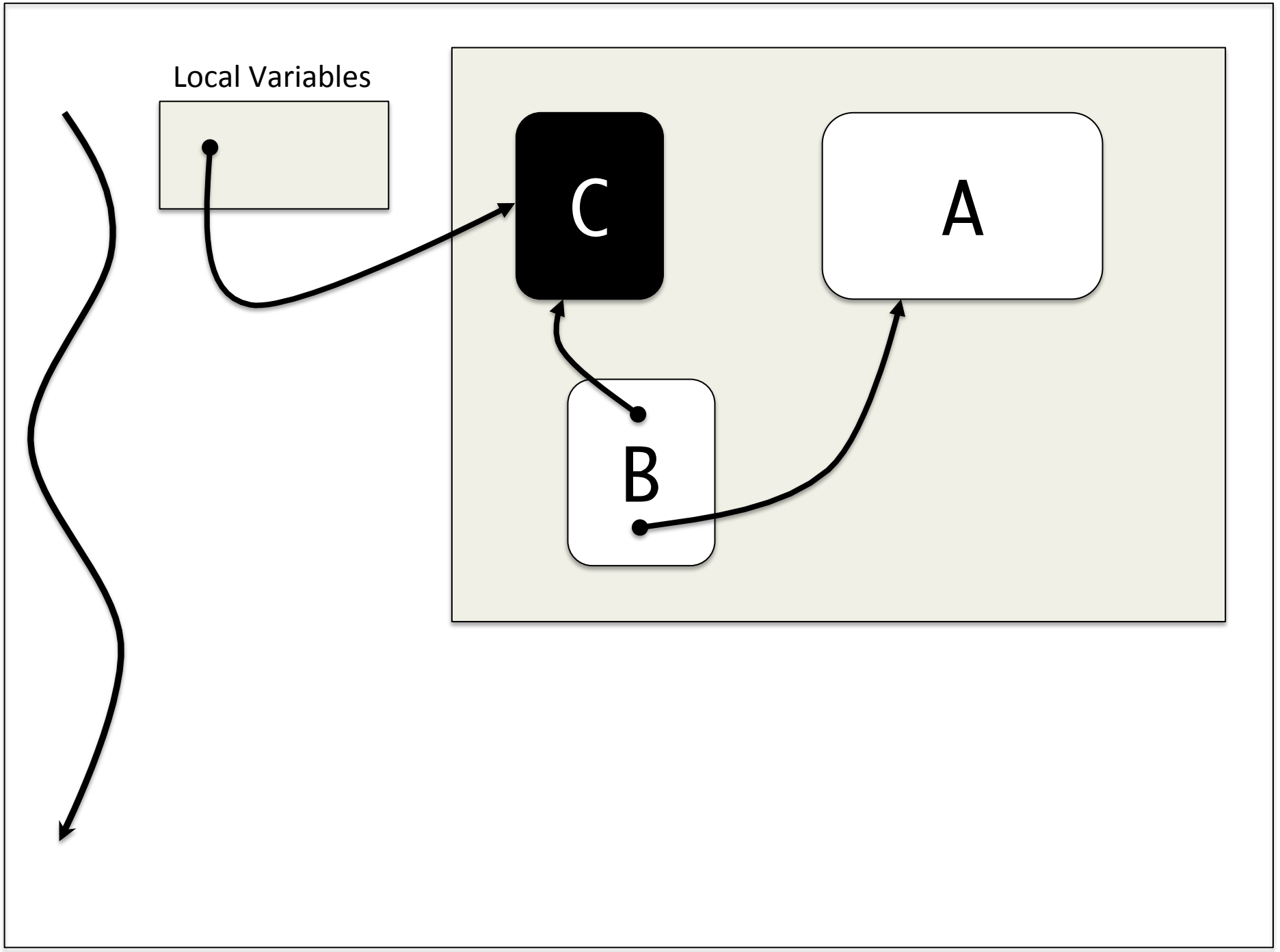
Implementing Finalization

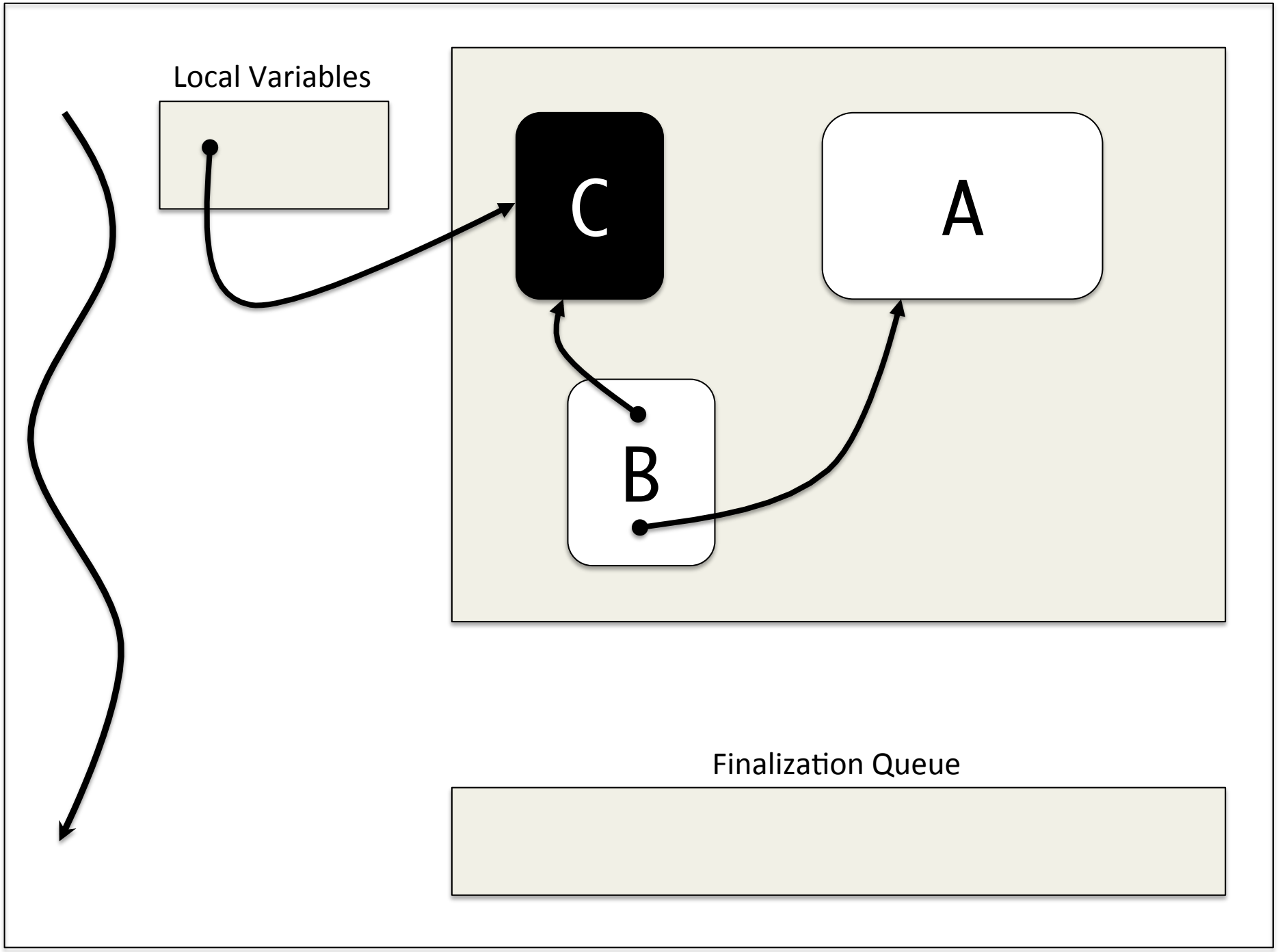
- Maintain a finalization queue
  - Check for a finalizer when freeing an object
  - May also flag finalizable objects at allocation time
- Add to the queue only before freeing the memory
  - Finalizers run before the memory is re-used
  - Some algorithms don't reuse all parts of the heap
- Run finalizers incrementally or concurrently
  - Finalizers can interact with other threads

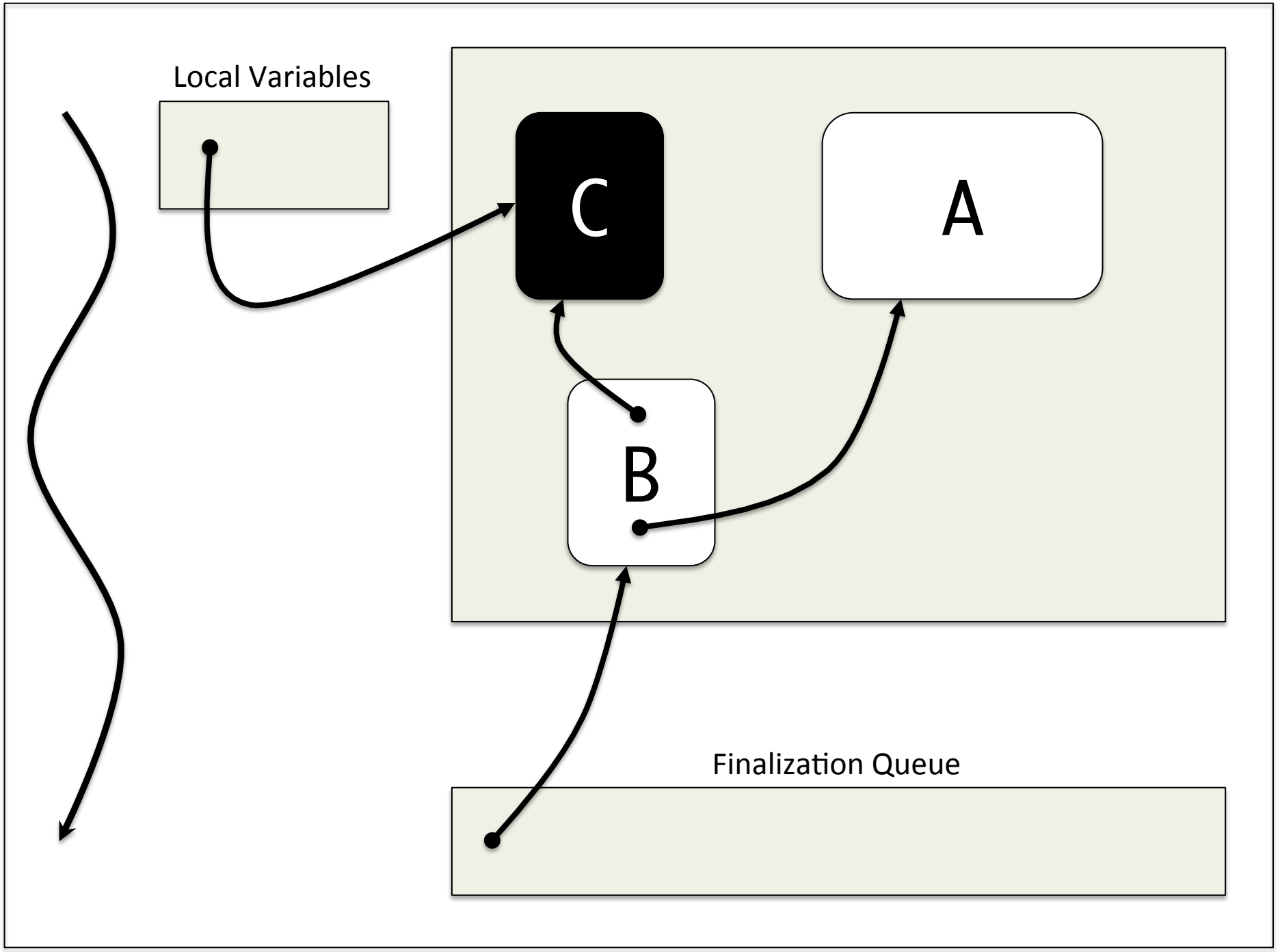

C
A
B

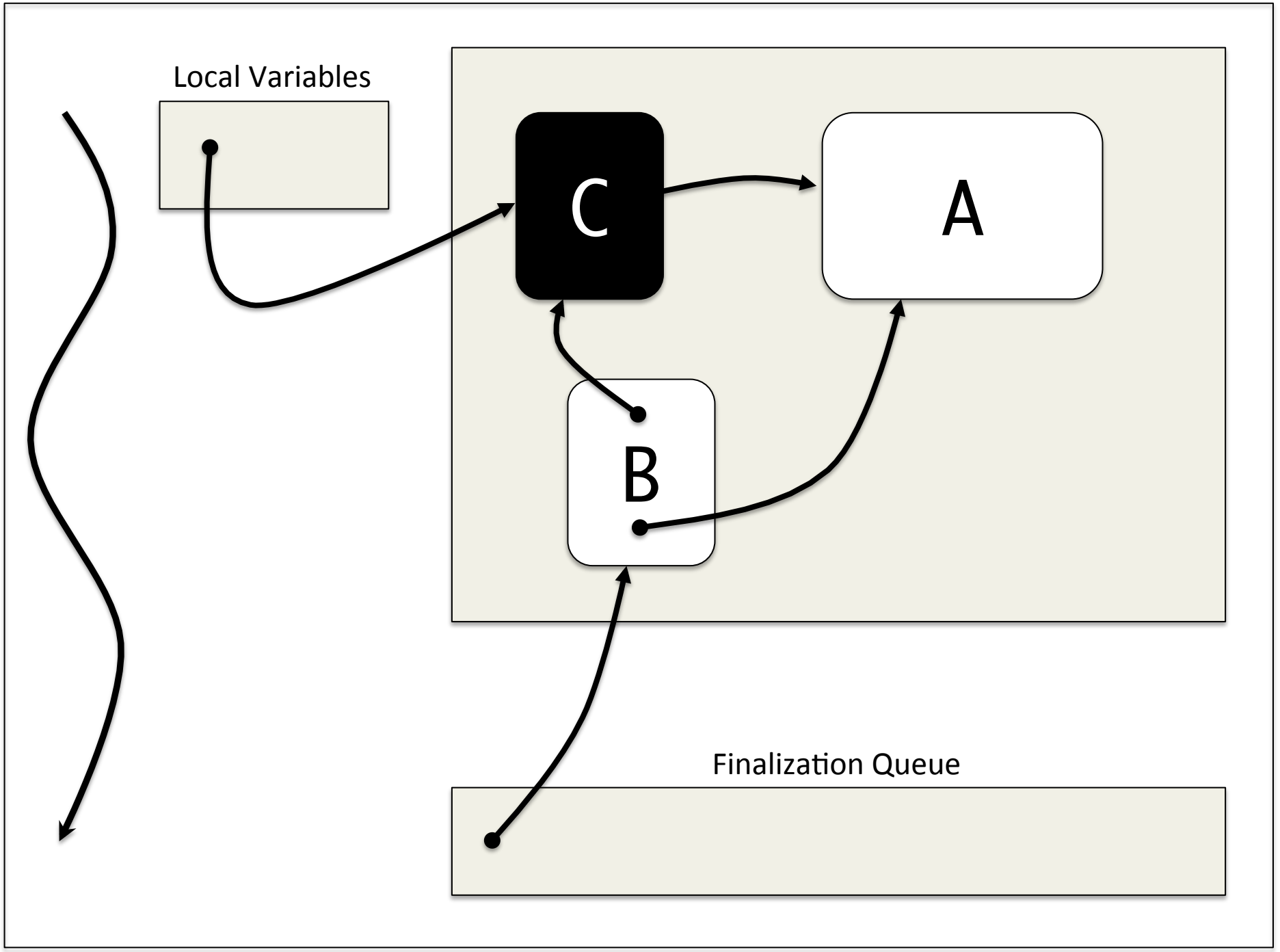

Local Variables
C
A
B

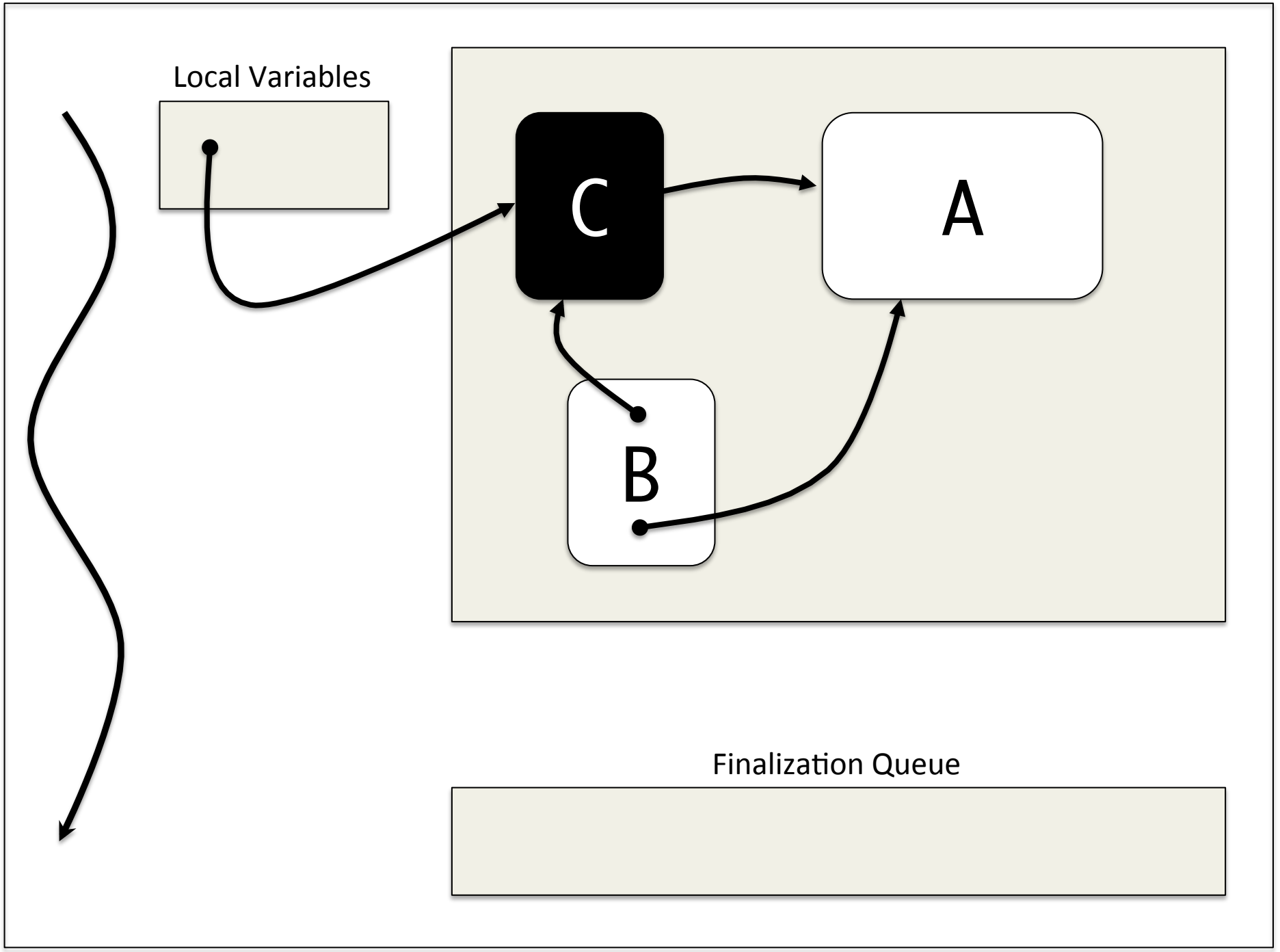

Local Variables
C
A
B

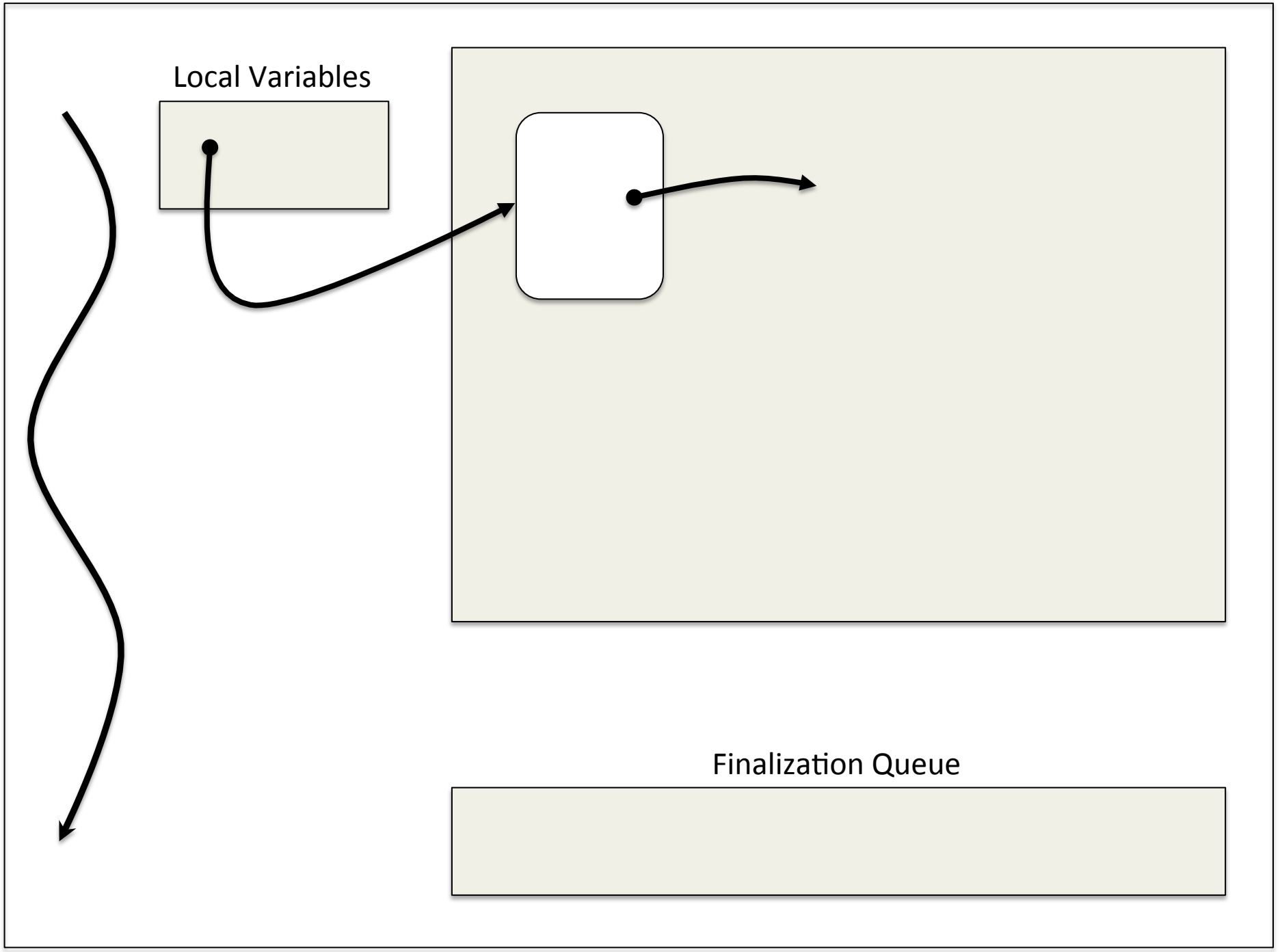

Local Variables
C
A
B


Local Variables
C
A
B
Finalization Queue


Local Variables
C
A
B
Finalization Queue


Local Variables
C
A
B
Finalization Queue


Local Variables
C
A
B
Finalization Queue


Local Variables
Finalization Queue

Object Resurrection

- A finalizer can make a garbage object reachable
  - May the the this pointer
  - May be another object entirely
- Resurrection greatly complicates GC
  - Can't delete any object that may be referred to
  - Need to wait until all finalizers run before deleting
- Can use a two-phase approach

Phased Finalization

- Trace the heap to find objects to be freed
  - Applies to all tracing algorithms
- Run all finalizers in the finalization queue
  - Remember that any object can only be finalized once
  - This phase may resurrect objects
- Trace the heap again
  - Identify object that are definitely garbage

Concurrent Finalization

- Some algorithms allow concurrent finalization
  - Extension of the phased finalization algorithm
- Works if garbage can be retained for a GC cycle
  - Ensure that all finalizers run before the next GC
  - Tracing phase of next GC finds unreachable objects
- Track phantom objects separately
  - Those found unreachable and not resurrected

Double Finalization

- A given object's finalizer is called at most once
- Resurrected objects may have different semantics
  - An object may be resurrected after finalization
  - Objects may resurrect themselves
- The object will not be finalized again

Multi-Threaded Finalization

- Objects are finalized by a finalizer thread
  - May be multiple parallel threads
  - Could be a specialized daemon
- The VM can borrow an application thread
  - Must ensure that no locks are held
  - Incremental finalization on constrained systems
- Finalizers can cause deadlock

Why Use Finalizers

- Free up native memory
  - Natural extension to GC semantics
- Last-resort resource clean up
  - Log errors in explicit resource tracking
- Can be useful if you understand the semantics

AHEAD OF TIME COMPILATION

Ahead of Time Compilation

- AOT compilation is a familiar model
  - Write your code
  - Run the compiler
  - Ship the resulting executable
- Java AOT step is generally very minimal
  - Convert the Java code to bytecode
- AOT compilers can build executable from Java

AOT Advantages

- Shift compilation cost from the runtime
  - No interpretation and profiling phase
  - No JIT compilation
- Produce a static binary
  - Single image contains all required code
  - Can do whole-program analysis
- Well adapted to constrained environments
  - Low power prohibits JIT compilation
  - Uncooperative environments don't allow dynamic code

AOT Limitations

- Up-front compilation cost
  - Resulting executable is platform specific
- Can't use dynamic classloading
  - All classes must be declared ahead of time
  - Harder to adapt the application to the environment
- Limited profiling and debugging
  - JVMTI instruments bytecode
  - Requires interpreter or JIT compilation

Ahead of Time Compilers

- Various compiler implementations
  - GCJ
  - Excelsior
  - LLVM
- Compiler requires a class library implementation
  - Licensing issues with the standard JDK library
  - GNU Classpath offers partial compatibility

LLVM

- Open source compilation framework
- Various language-specific front-ends
  - Common intermediate representation
- Basis of various compilers in everyday use
  - Clang used as a replacement for GCC
- VMKit was a JVM leveraging LLVM
  - LLVM + MMTk + pthreads + glue

GC in LLVM

- Limited support for garbage collection
  - Doesn't provide an actual GC implementation
  - Provides hooks for runtime developers
- Binary interface in LLVM support intrinsics
  - Creating GC safepoints
  - Computing stack maps
  - Read and write barriers
- Intermediate representation supports other aspects
  - Type maps
  - Root locations

LLVM as a JIT

- LLVM was thought to be static only
  - Optimizations required large analysis
  - Doesn't take advantage of profiling data
  - Support for mixed mode operation limited
  - Compilation is much slower than required
- Apple released LLVM-based JIT in Webkit
  - Dynamically compiling Javascript in Safari
  - Significant engineering work

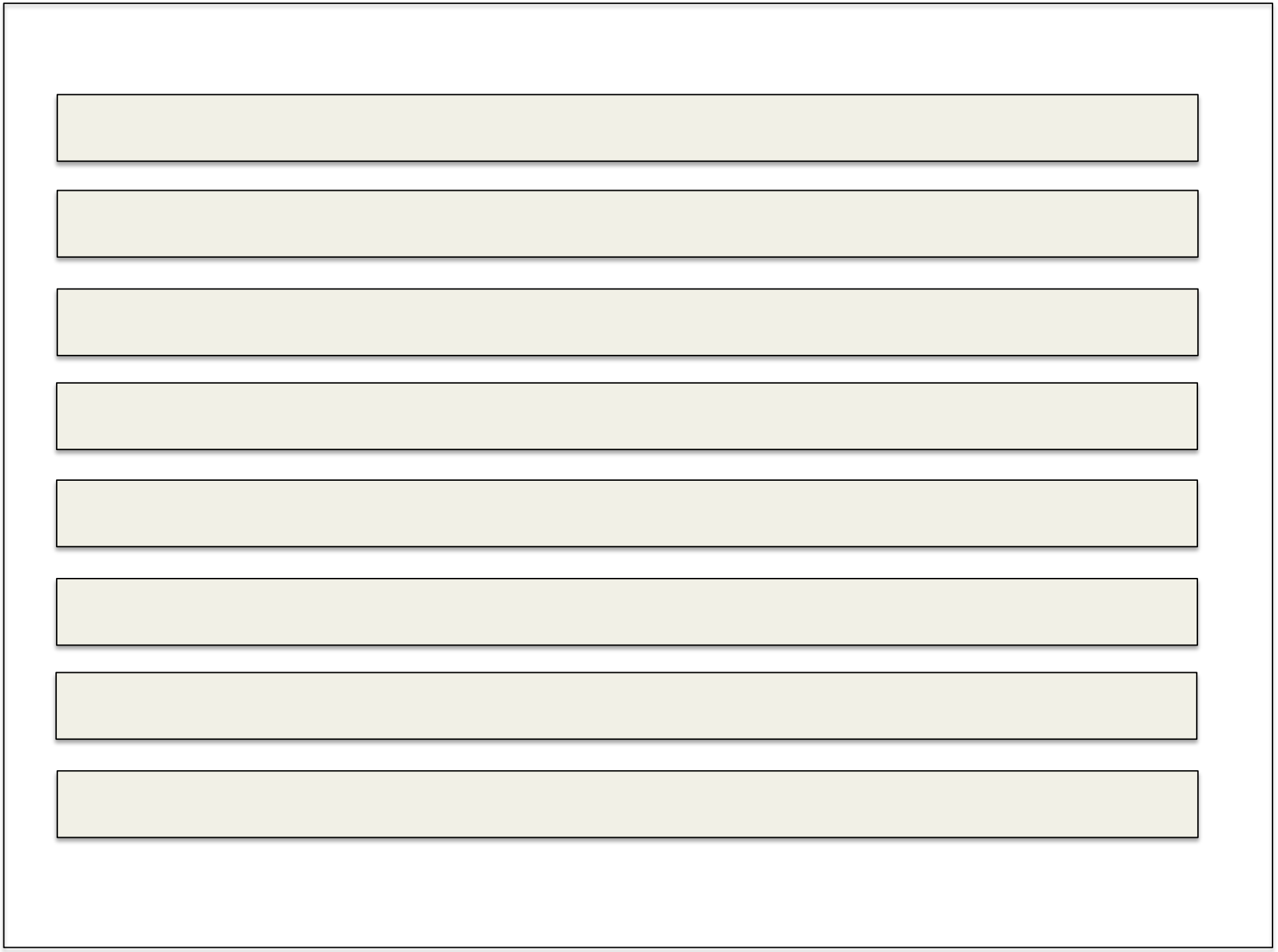
GARBAGE FIRST ALGORITHM

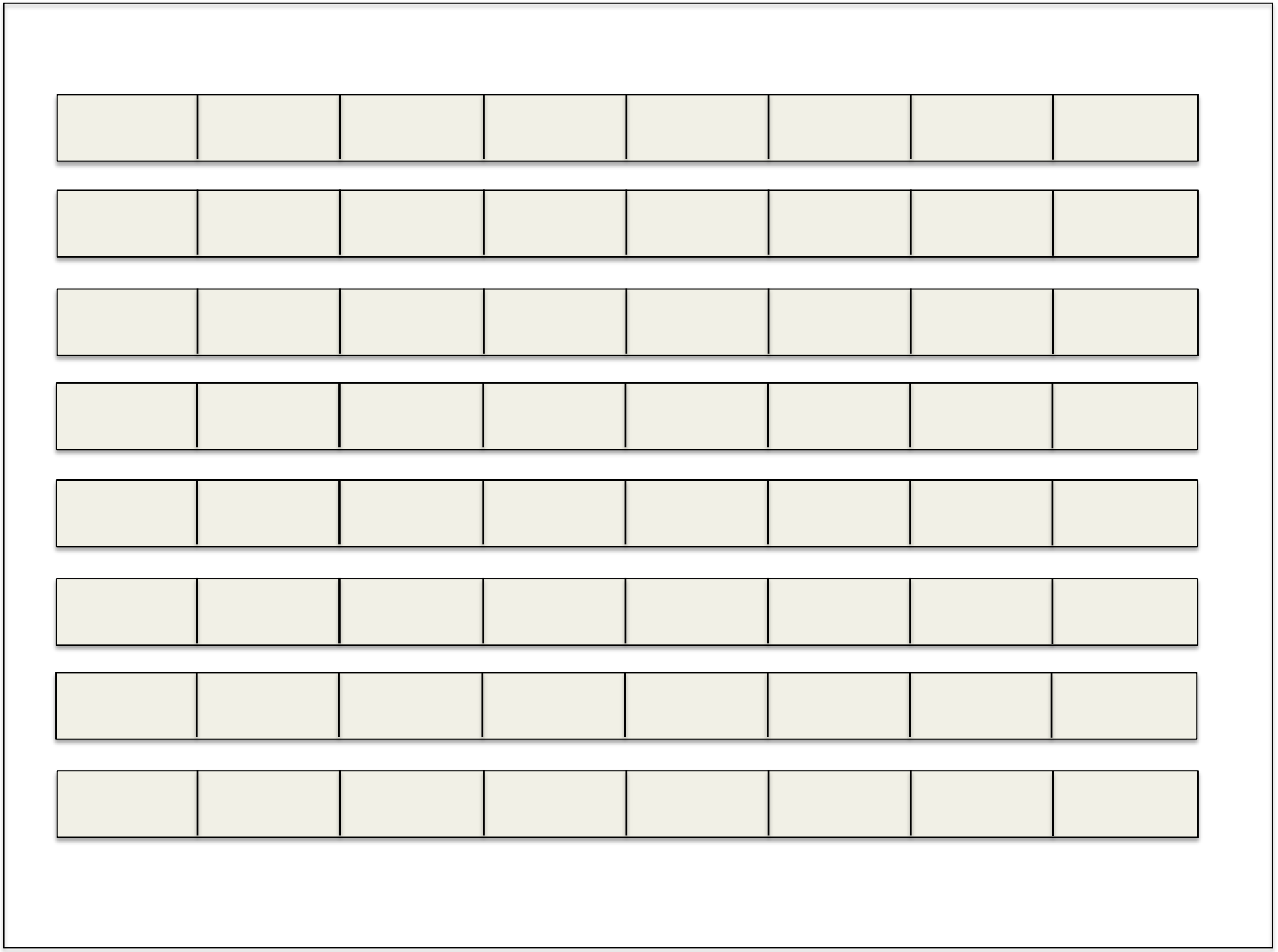
Garbage First Algorithm

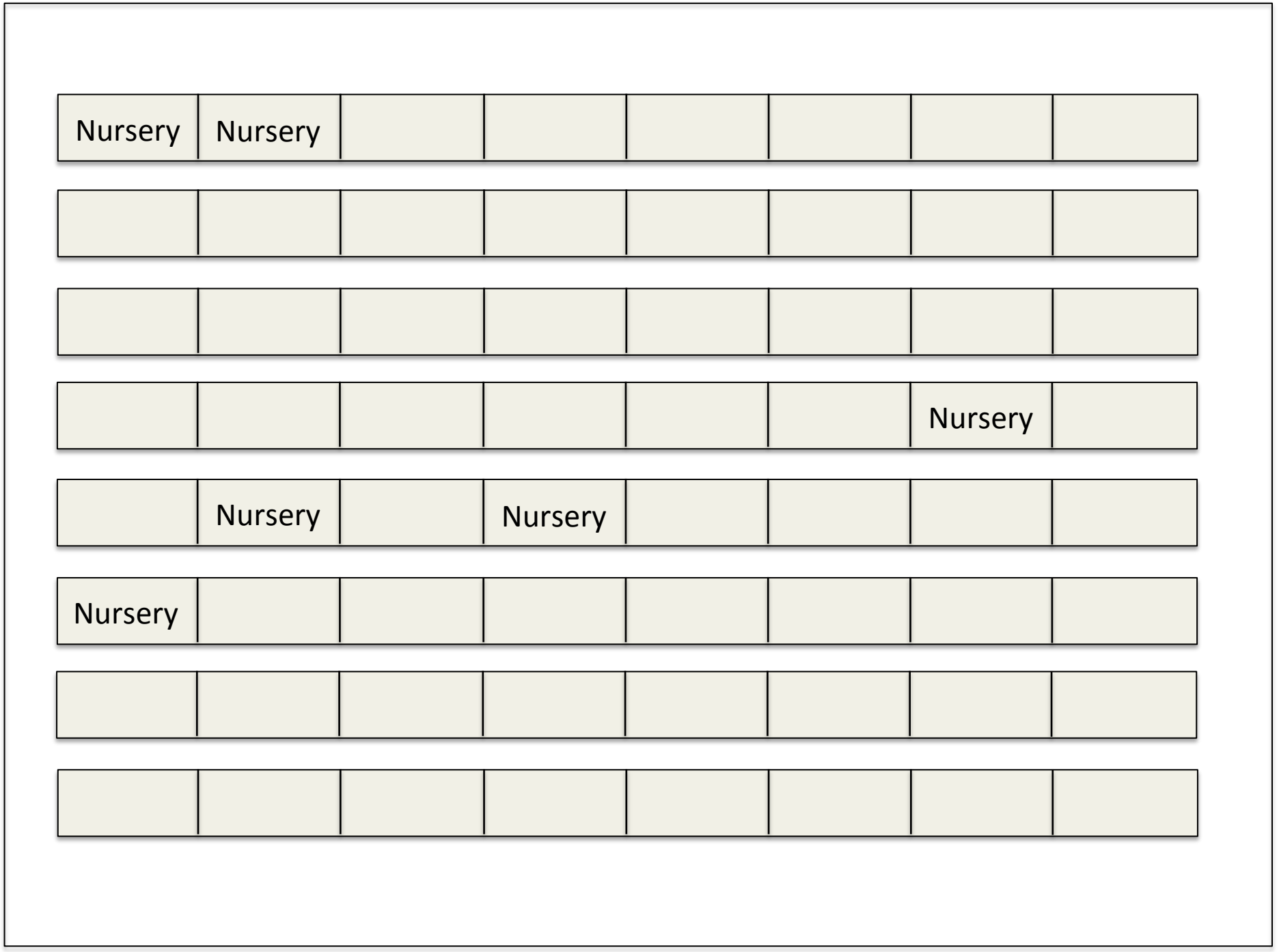
- Latest iteration of the Hotspot collector
  - Not enabled by default
  - Targets pause times over throughput
  - Many server applications want the inverse
- GC is parallel and concurrent
  - Copies objects to compact the heap
  - Pauses are predictable and tunable
  - Offers soft real-time without guarantees

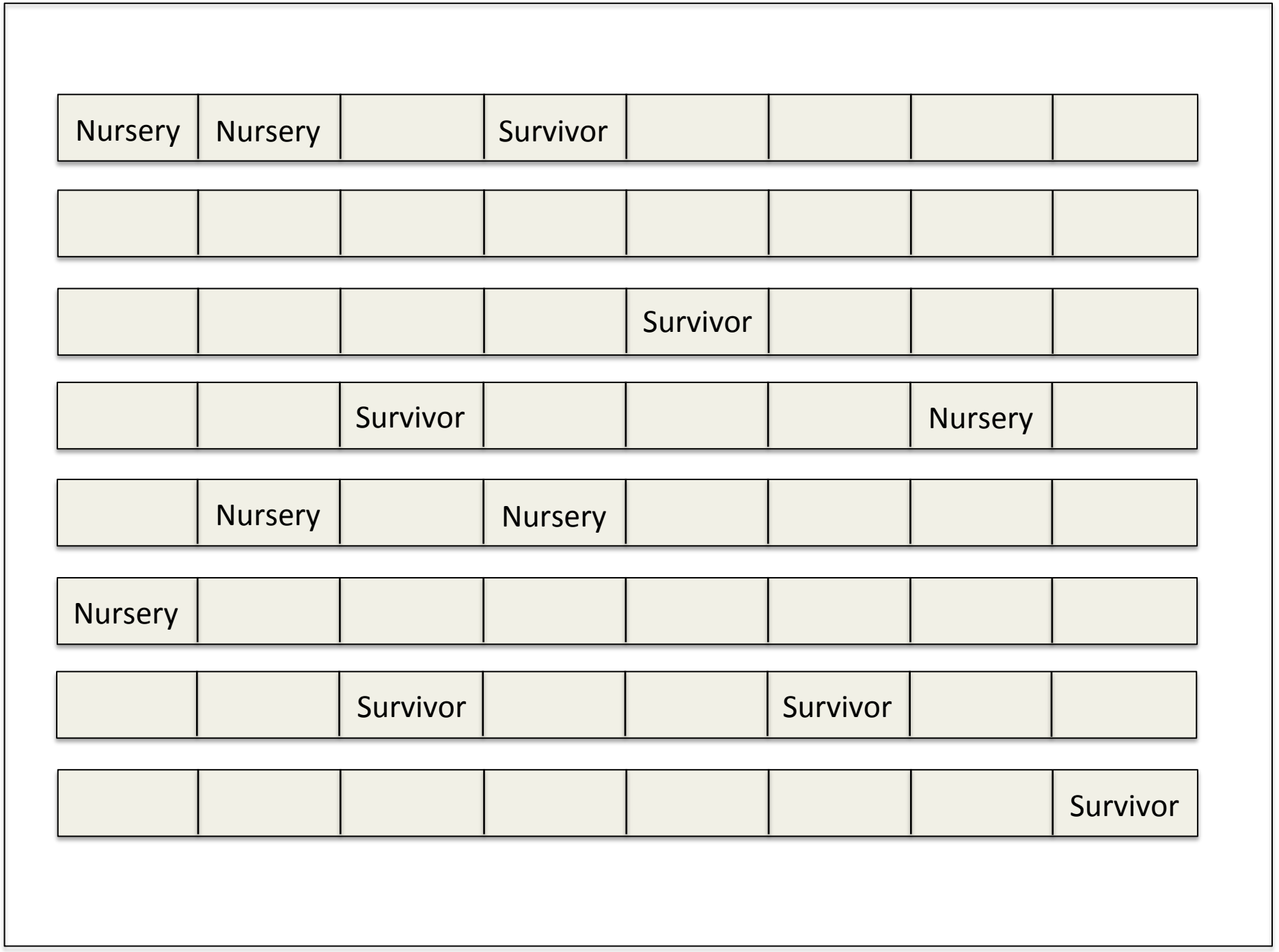
G1 Regions

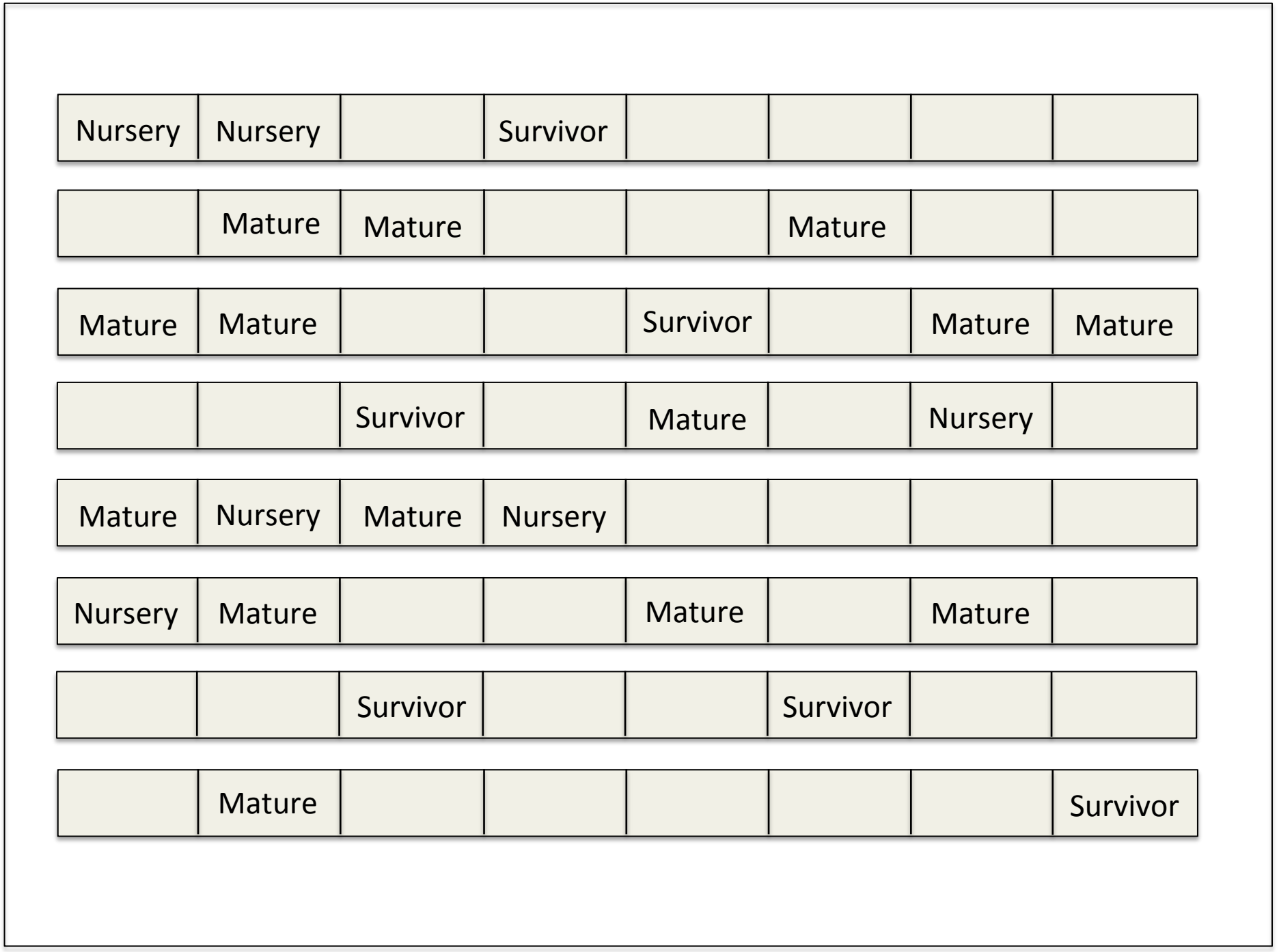
- Heap is split into equally-sized regions
  - Region sizes range from 1-32 Mb
  - Aim to have no more than 2048 regions
  - More tuning to enable more than 64 Gb heap
- Young generation made up of two block types
  - Nursery (or Eden)
  - Survivor set
- Very large objects given a region to themselves
  - Multiple regions can coalesce if necessary

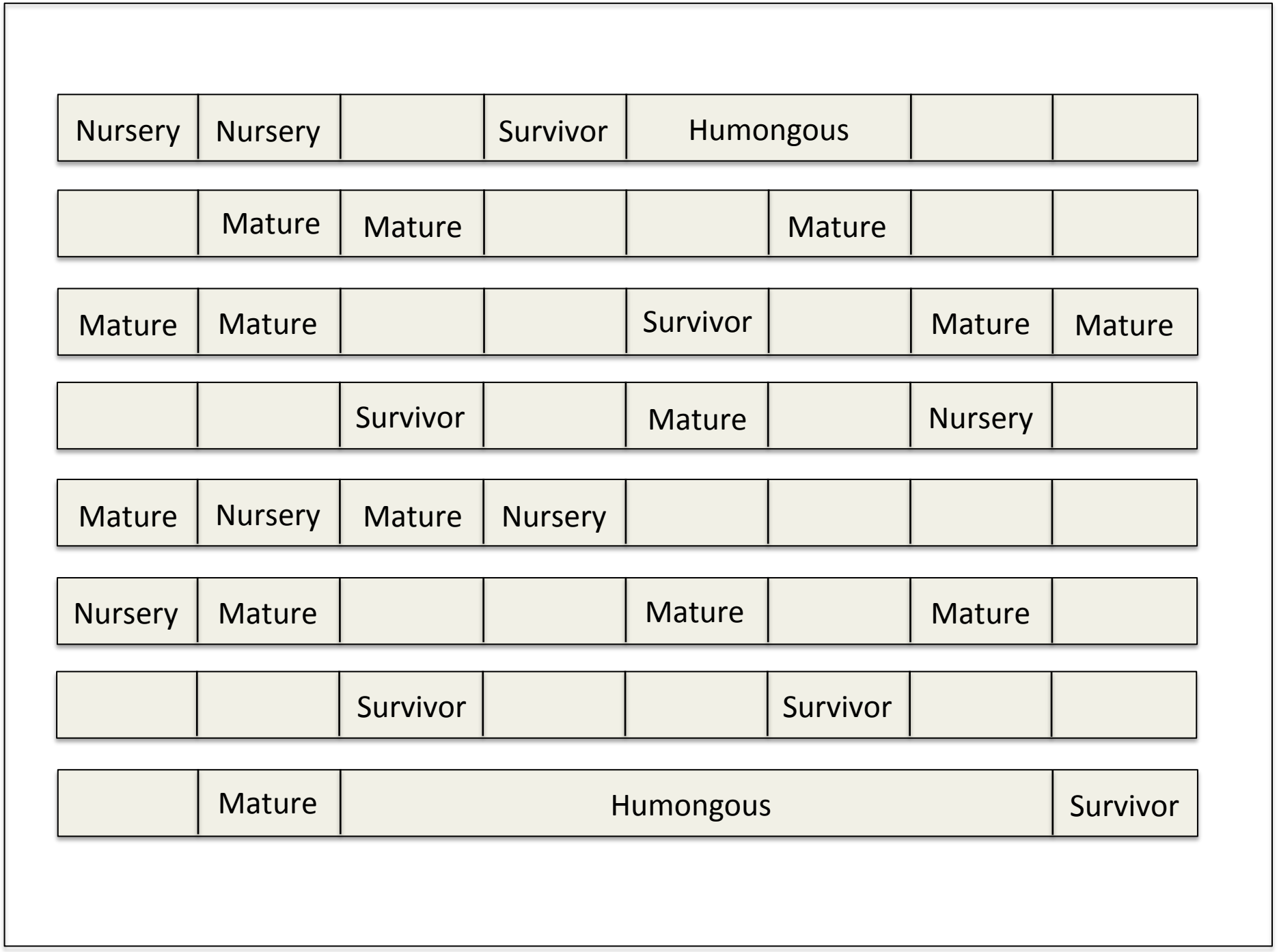





Nursery
Nursery
Nursery
Nursery
Nursery
Nursery
Nursery


Nursery
Nursery
Survivor
Survivor
Survivor
Nursery
Nursery
Nursery
Survivor
Survivor
Survivor


Nursery Nursery Survivor
Mature Mature Mature
Mature Mature Survivor Mature Mature
Survivor Mature Nursery
Mature Nursery Mature Nursery
Nursery Mature Mature Mature
Survivor Survivor
Mature Survivor


Nursery Nursery Survivor Humongous
Mature Mature Mature
Mature Mature Survivor Mature Mature
Survivor Mature Nursery
Mature Nursery Mature Nursery
Nursery Mature Mature Mature
Survivor Survivor
Mature Humongous Survivor

Remembered Sets

- Maintain remembered sets for each region
  - Track references to that region from outside
  - Don't track internal references
- Maintain set using a write barrier
  - Use block alignment to generate region ID
  - Simple shift operation
- Remembered sets give estimate of live data

Minor GC

- Parallel stop-the-world copying collection
- Promote all live young objects
  - Move objects from nursery to the survivor regions
  - Move objects from the survivor to the mature
- Don't evacuate all young objects at once
  - Determine nursery regions by pause time
  - Determine survivor regions by garbage ratio estimates

Concurrent Marking

- Initial marking to scan the root set
  - Requires that threads be stopped
  - Piggy-backs on a minor GC pause
- Concurrent marking across the heap
  - Uses incremental update
- Remark phase to complete incremental update
  - Stops the world

Cleanup Phase

- After mark phase we know all live objects
  - Remembered sets aggregate the information
- We can explicitly target regions with more garbage
  - Minimize copy overhead
  - Reclaim more garbage in a bounded time
- In many cases a region is entirely garbage
  - Remember the train algorithm

Evacuation

- Final part of major collection
- Iterate over regions selected for copying
  - Copy all live objects to an empty region
  - Highly parallel
  - Uses a form of TLAB to minimize contention
- Number of regions copied can be tuned
  - Trade-off throughput for pause time

BENCHMARKING

Benchmarking

- Benchmarks are big business
  - Drive adoption of a given platform
  - Encourage investment in technology
- They provide great ammunition for flame wars
  - What would Slashdot be without benchmarks?
- Benchmarks are almost all worthless

Apples to Apples Comparison

- An ideal benchmark compares like for like
  - Single application running on different JVMs
  - Identical hardware and software platforms
  - Running in single-user mode
  - Networking and other interrupts turned off
- A good experiment uses many runs
  - Average over many (n > 30) iterations
  - Take standard deviation, confidence intervals, etc

Micro Benchmarks

- Small kernels of code that test a given operation
- Generally implemented as a tight loop
  - Allocations per second
  - Monitor acquisitions and releases
  - Floating point operations per second
- Very hard to make a micro benchmark interesting

```
public class SquareRoot {

    public static void main(final String[] args) {
        int iterations = Integer.parseInt(args[0]);
        long start = System.currentTimeMillis();

        for (int i = 0; i < iterations; i++) {
            Math.sqrt((double) i);
        }

        long end = System.currentTimeMillis();

        System.out.println("Time: " + (end - start));
    }

}
```

JIT Compilation

- The JIT is designed to defeat microbenchmarks
  - Dead loop elimination
  - Constant folding
  - Escape analysis
- Many microbenchmarks measure startup costs
  - Code runs in interpreted mode for a time
  - JIT compiler is triggered
  - Benchmark behavior changes completely

Phase Behavior

- Native code tends to start at full speed
  - Compilation happens ahead of time
- Managed runtimes take time to get started
  - Startup time required to initialize the VM
  - Initially run in interpreted mode
  - JIT compilation kicks in after a while
  - Runs at full speed after compilation period

Macro Benchmarks

- Aim to emulate real-world applications
- Various SPEC benchmark suites
  - SPECjbb2000, 2008, 2013
  - SPECjvm98, 2008
- DaCapo benchmark suite
  - Targets interesting memory management behavior
- Java Linpack
  - Java version of popular floating point benchmark

Macro Benchmark Benefits

- More likely to reflect real applications
  - Run for a reasonably long time
  - Workloads can be analyzed and considered
  - More often designed to account for phase behavior
- More varied operations
  - Don't obsess over a single part of the VM
  - Use a mix of custom and library code

Macro Benchmark Issues

- May not test what you think
  - Applications sometimes spend time in strange places
- Repetitive workloads
  - Increase run time by doing the same thing again
- Macro benchmark suites become obsolete
  - SpecJVM98 now runs in seconds, not minutes
- Limited number of benchmark suites
  - VM vendors can target and optimize for them

GC Benchmarking

- GC is notoriously hard to benchmark
  - Synthetic applications behave differently from real
  - Benchmark times too short for interesting behavior
- Micro benchmarks focus on the nursery
  - Generate lots of small, similar objects
  - Create occasional large, long-lived objects
- Many macro benchmarks never trigger full GC
  - Well-tuned system can run for surprisingly long

Java Benchmarking Advice

- Include a sufficient warm-up phase
  - Get past the startup overhead
  - Run the workload for a while to let the JIT work
- Measure steady-state behavior
  - Assume that Java applications run for a long time
- Avoids benchmarks targeting Java's weaknesses
  - By focusing on Java's strengths?

General Benchmarking Thoughts

- The only important benchmark is your application
  - Make sure that your own code runs well
  - Target the right platform for the job
- Benchmarking shows that any application can be tuned
  - Use that to your advantage
  - Profile your realistic workloads
  - Optimize the important parts
- Try out your application with different VM settings
  - Or even running on different JVMs

MULTITENANT VM

VM Overhead

- The Java VM is a heavyweight system
  - VM internal data structures and mechanisms
  - Full compiler implementation
- Many applications use the same libraries
  - Collections, Spring, XML processing, etc
- Every Java application incurs the same overhead

Multitenancy

- Ideally, multiple programs could share the VM
  - Single VM startup pause
  - Compile common classes once
- Classloading provides limited sharing
  - Doesn't give the illusion of a private VM
  - Class loading semantics decrease potential sharing
  - Difficulty in terminating the program
  - Semantics for JVMTI

Multi-Tasking Virtual Machine

- Project Barcelona in Sun labs (pre-Oracle)
- Introduced isolates to Java
  - Defined an isolate as a sub-process within a VM
  - Isolates share no state with other isolates in the system
  - Each isolate appears to be its own VM
- Allowed for aggressive optimization across isolates
  - Transparent to the developer

MVMs Today

- Specification is available to implementers
  - JSR 121: Application Isolation API
- Two reference implementations from Sun
  - MVM, MVM2
- Interest died down after the initial work
  - Machine-level virtualization made it less interesting
- Interest appears to be picking back up now
  - Vendors interested in PaaS cloud offerings

VM SHUTDOWN

Shutdown Modes

- Ideally, the JVM always shuts down gracefully
  - May happen if the VM encounters an internal bug
- VM can abruptly halt
  - Receives a SIGKILL or TerminateProcess interrupt
  - The host machine crashes
- Graceful shutdown happens in several cases
  - Final non-daemon thread exits
  - VM receives an external interrupt signal
- Shutdown involves several phases

Shutdown Hooks

- Program designates work during termination
  - Flush buffers
  - Close connections
- Hook is specified as an un-started Thread
  - Recommendation for defensive programming
  - Expectation that work should not take long
- Hooks run in any order at shutdown time

Forcing Finalization

- VM option forces all finalizers to run on exit
  - Gets around the limitation on whether finalizers run
- Finalizers can contain arbitrary code
  - Developers may not have considered shutdown
  - Code could deadlock or slow shutdown
- May be many objects to finalize
  - Large heap sizes imply more objects to finalize

JVM Exit Preparation Step

- Any JVM shutdown hooks run after Java code
  - Delete-on-exit files
- Clean up most VM structures and threads
  - Profiler, signal thread, GC threads
- Notify JVMTI agents
  - VM death events sent to all registered agents
- Native code may still be running
  - Set a flag in the JNI thread environment

Final Shutdown

- Terminate any remaining Java threads
  - Daemon or shutdown hooks
- Bookkeeping tasks for any Java thread
  - Release JNI reference handles
  - Mark thread as terminated
- Terminate the VMThread
  - Bring any threads to a VM safepoint
  - Stop JIT compiler threads
- Return control to the OS

METACIRCULARITY

Self-Hosting

- A compiler is just a regular computer program
  - Takes source files as inputs
  - Produces executable code as output
- A self-hosting is one that can compile itself
  - Compiler written in the target language
- Chicken-and-egg problem
  - Interrupt the recursion with a bootstrap compiler

Metacircularity

- Special case of a self-hosting runtime
  - The language's own features optimize the runtime
- Both VM and application are optimized together
  - VM data structures managed by the GC
  - VM code passes through the JIT
- VM can to be written in a higher-level language
  - Use Java synchronization rather than pthreads

JIT Compilation

- Much of the VM implementation relies on the JIT
  - Compiles both the VM and the application
- The JIT is an metacircular JVM is written in Java
  - Which must be JIT compiled
- Same problem as self-hosted compilers
  - Solve in the same way
  - Create a small bootstrap VM
  - Bootstrap VM is used only to get the system running

Intrinsics

- Some required language features are unavailable
  - Low-level pointer access is the most obvious
- Remember JIT compiler intrinsics
  - JIT has a better implementation than the class file
  - Class file is ignored
- We can use intrinsics to implement new features
  - Intercept all calls to a given class or interface
  - JikesRVM uses the org.vmmagic package

Magic

- Magic package split into two parts
- Pragma annotations
  - Tell the compiler to change the method
  - Remove safepoints, control compilation, etc
- Intrinsic annotation
  - Tell the compiler to replace the entire method

Magic Intrinsics

- `org.jikesrvm.runtime.Magic` has method stubs
  - Directly access registers
  - Manipulate stack frames
  - Cast arbitrary bits to Java types
  - Pointer arithmetic
  - Atomic operations such as CAS and memory fences
- Method stubs just throw runtime exceptions
  - Really there for the javac compiler
  - JIT compiler recognizes calls and rewrites