Managing Arrays

with the OMR garbage collector

Prerequisites

THIS LAB IS "BRING YOUR OWN LAPTOP"

- You need:
  - Linux, osx, or windows laptop
  - a C++11 toolchain: msvc, clang, gcc
  - git
  - cmake
- Clone the skeleton project

```
git clone --recursive https://github.com/rwy0717/splash2018-omr-gc
```

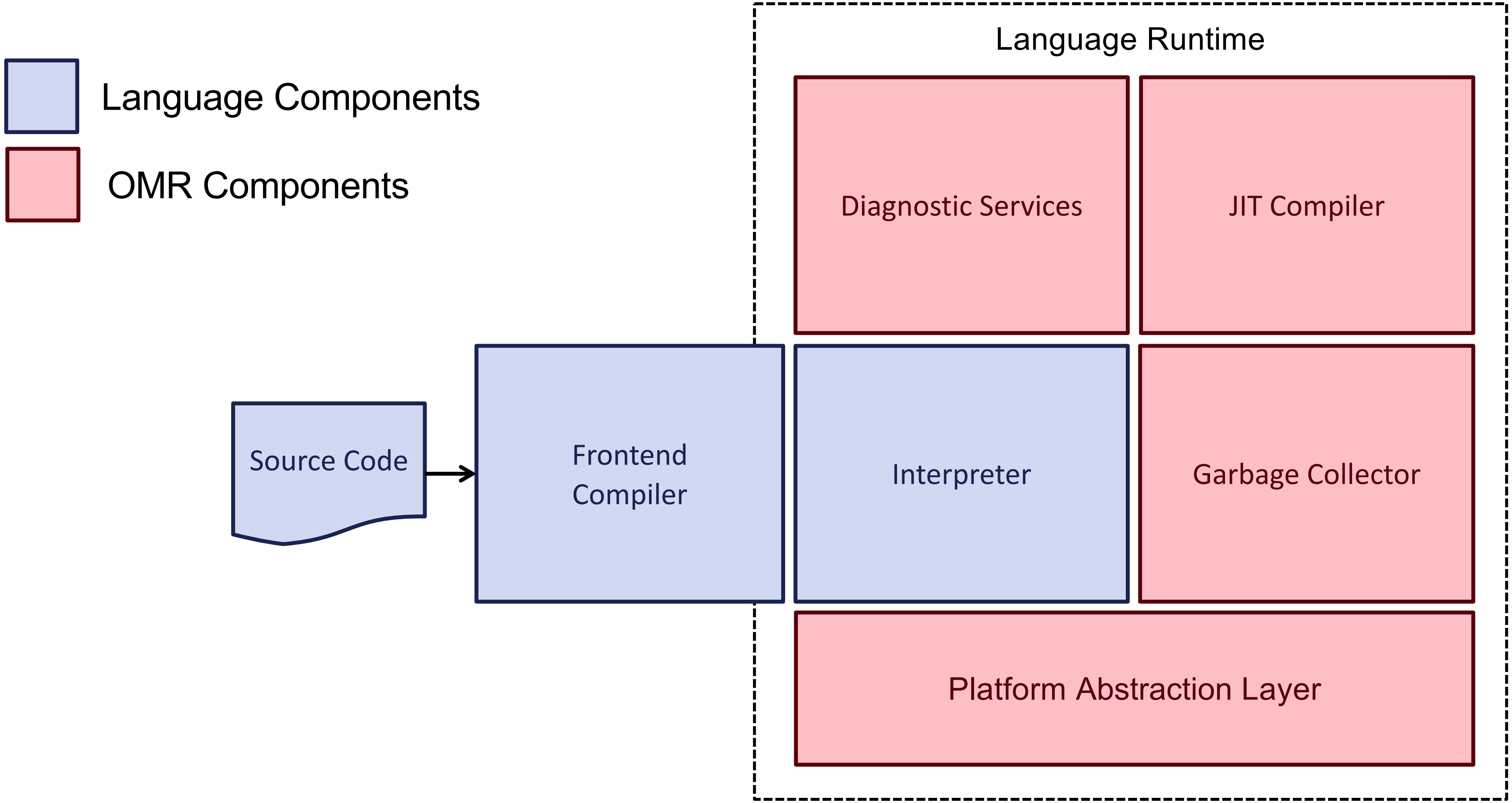
Don't forget your laptop charger!

Today, we’re going to implement a garbage collector for simple fixed-length arrays.

Eclipse OMR

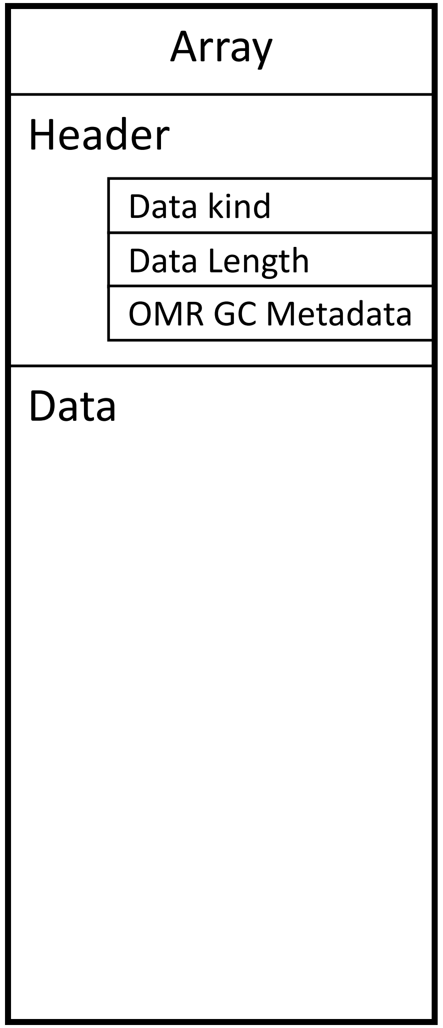
A toolkit for building language runtimes

https://github.com/eclipse/omr

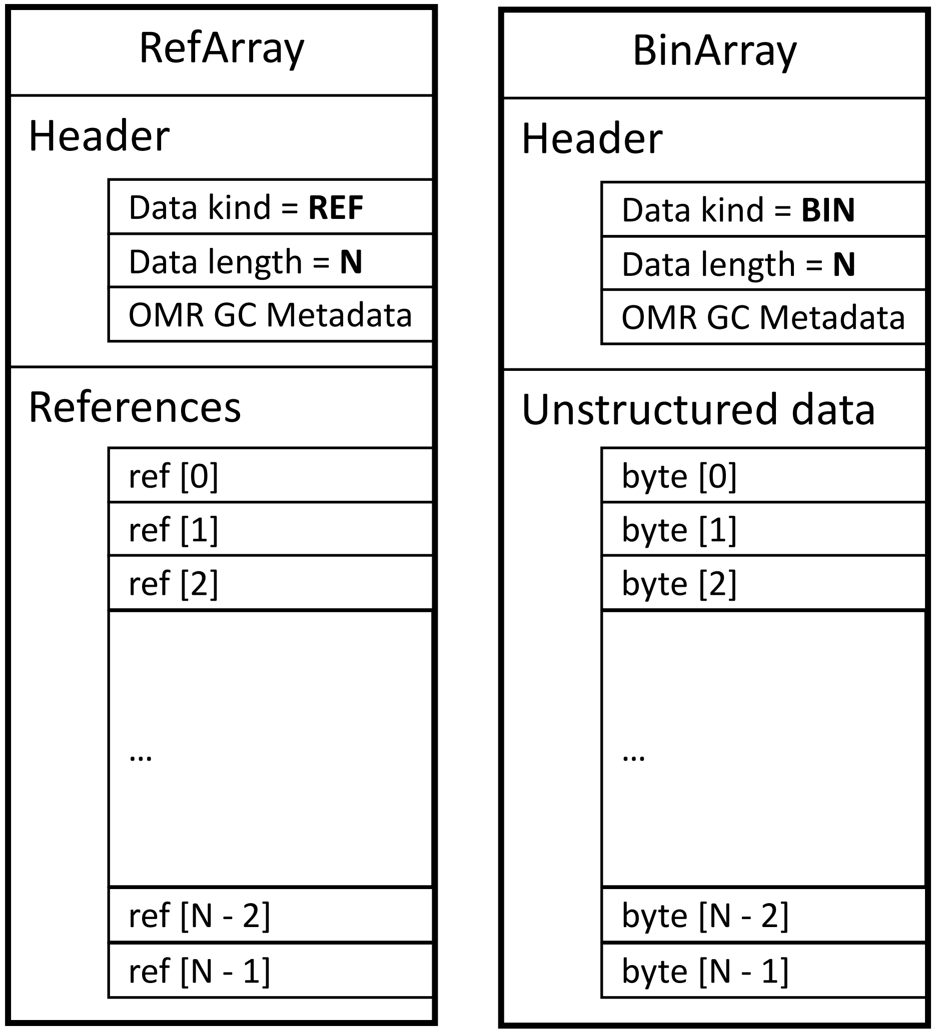

Language Components
OMR Components
Language Runtime
Diagnostic Services
JIT Compiler
Source Code
Frontend Compiler
Interpreter
Garbage Collector
Platform Abstraction Layer

Our array object model

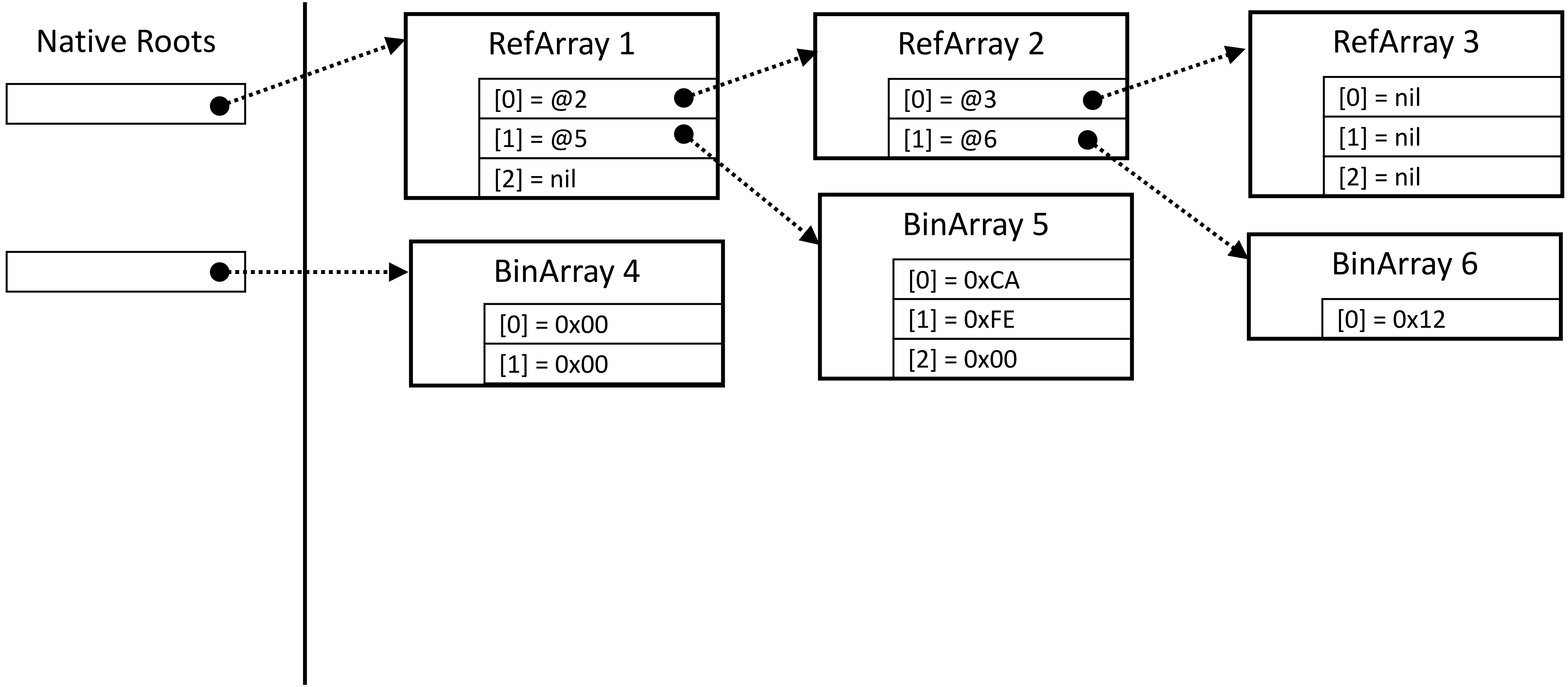
Our array object model:



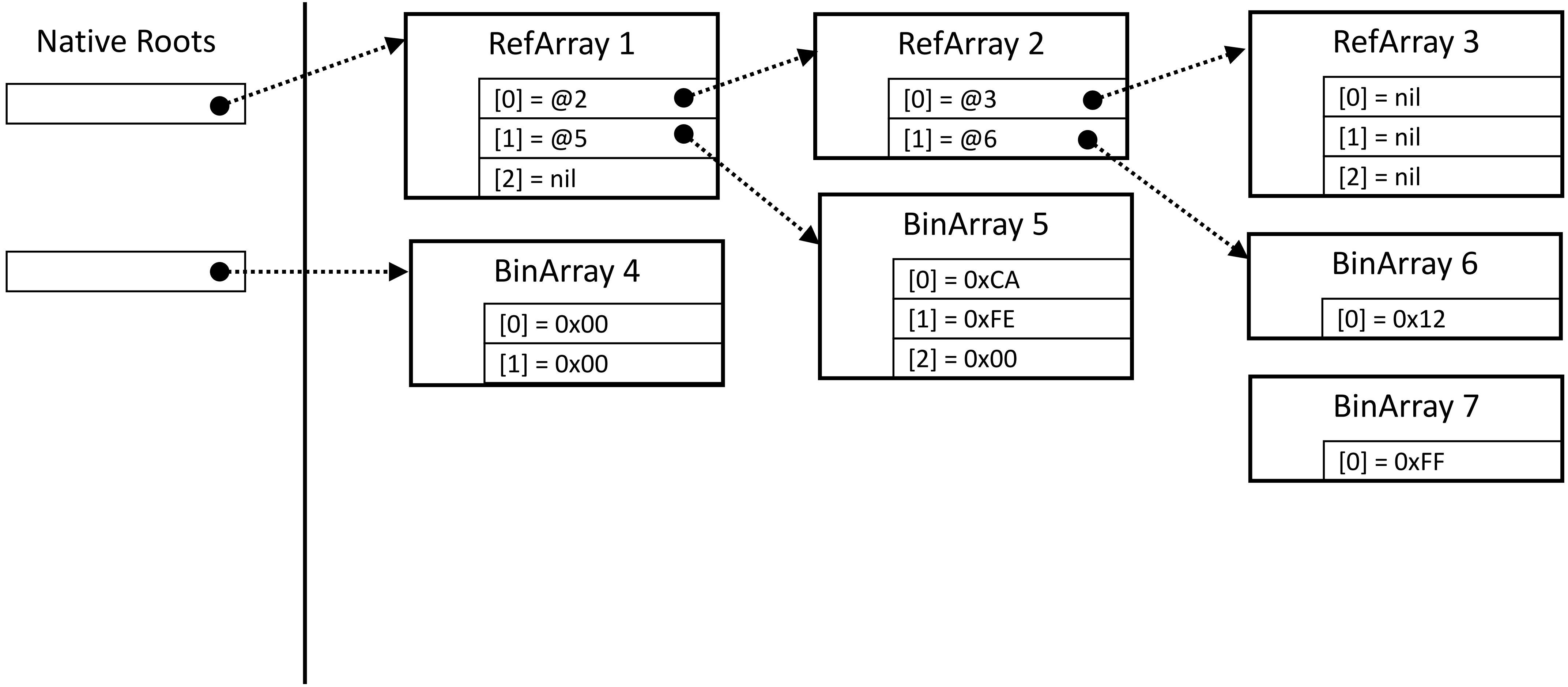
Two kinds of Data: References and bytes



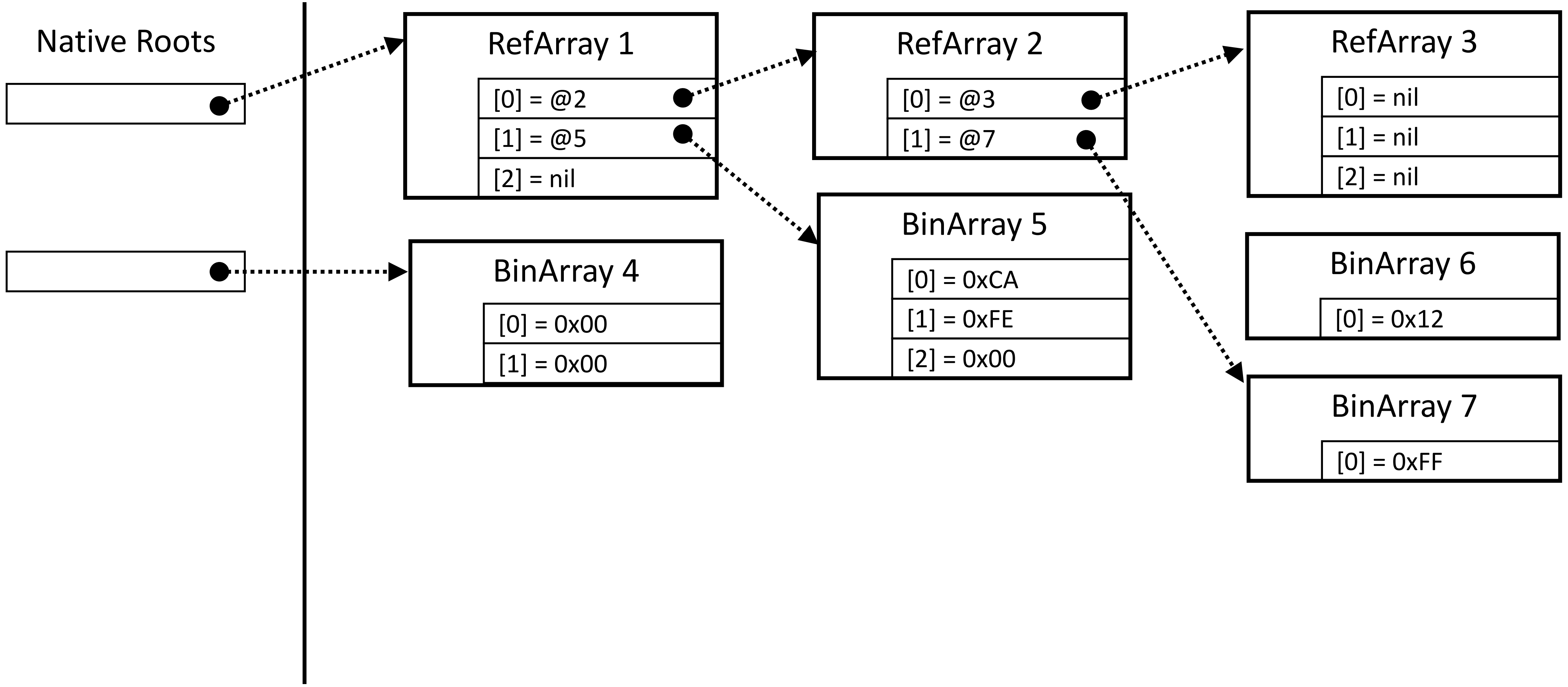
Memory: Graphs of objects



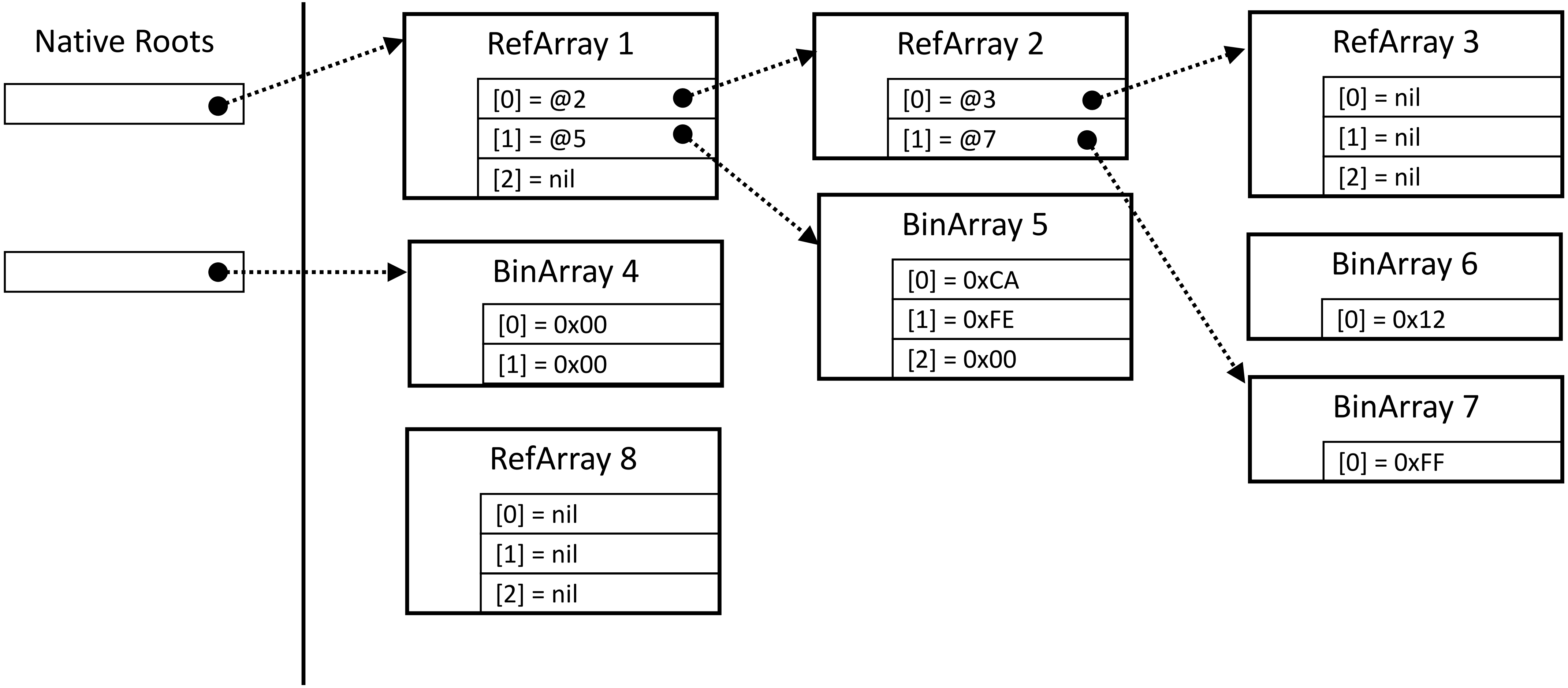
Memory: Graphs of objects



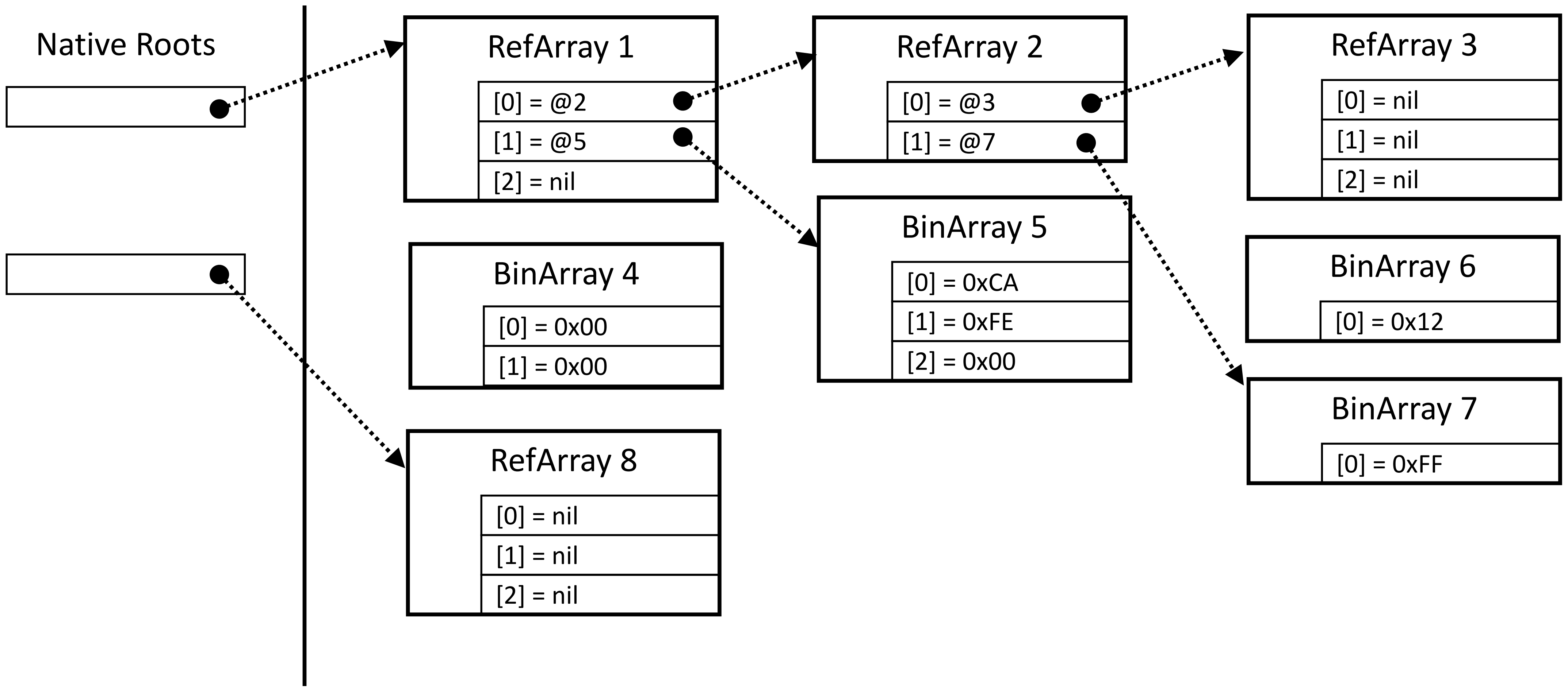
Memory: Graphs of objects



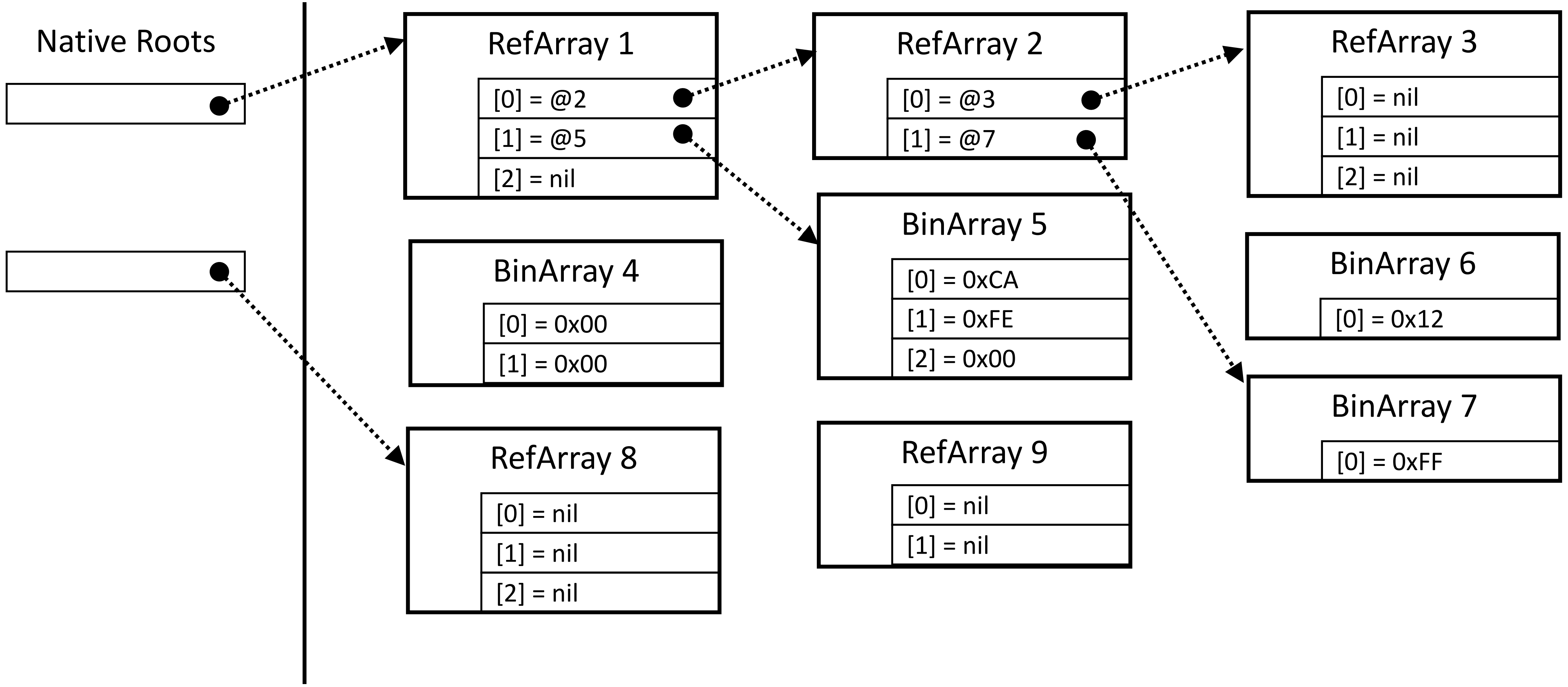
Memory: Graphs of objects



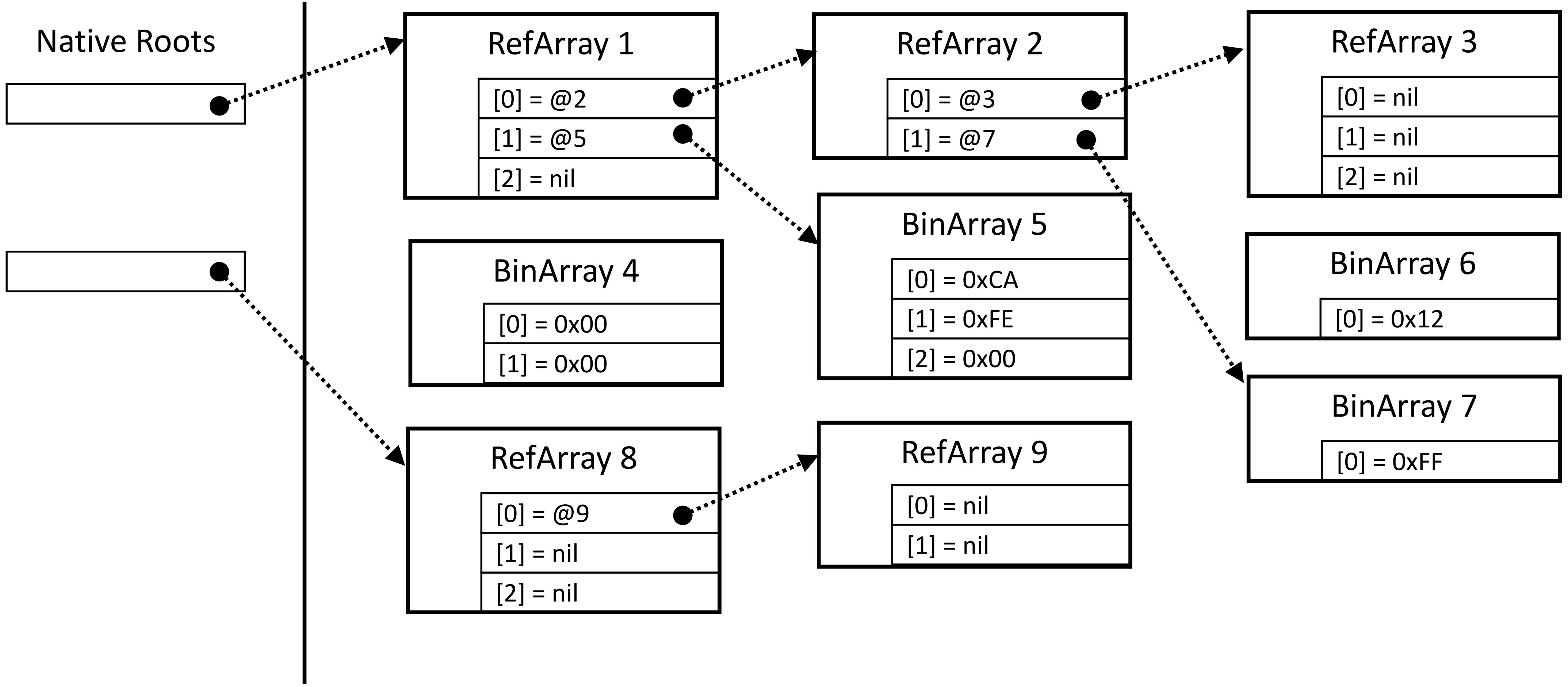
Memory: Graphs of objects



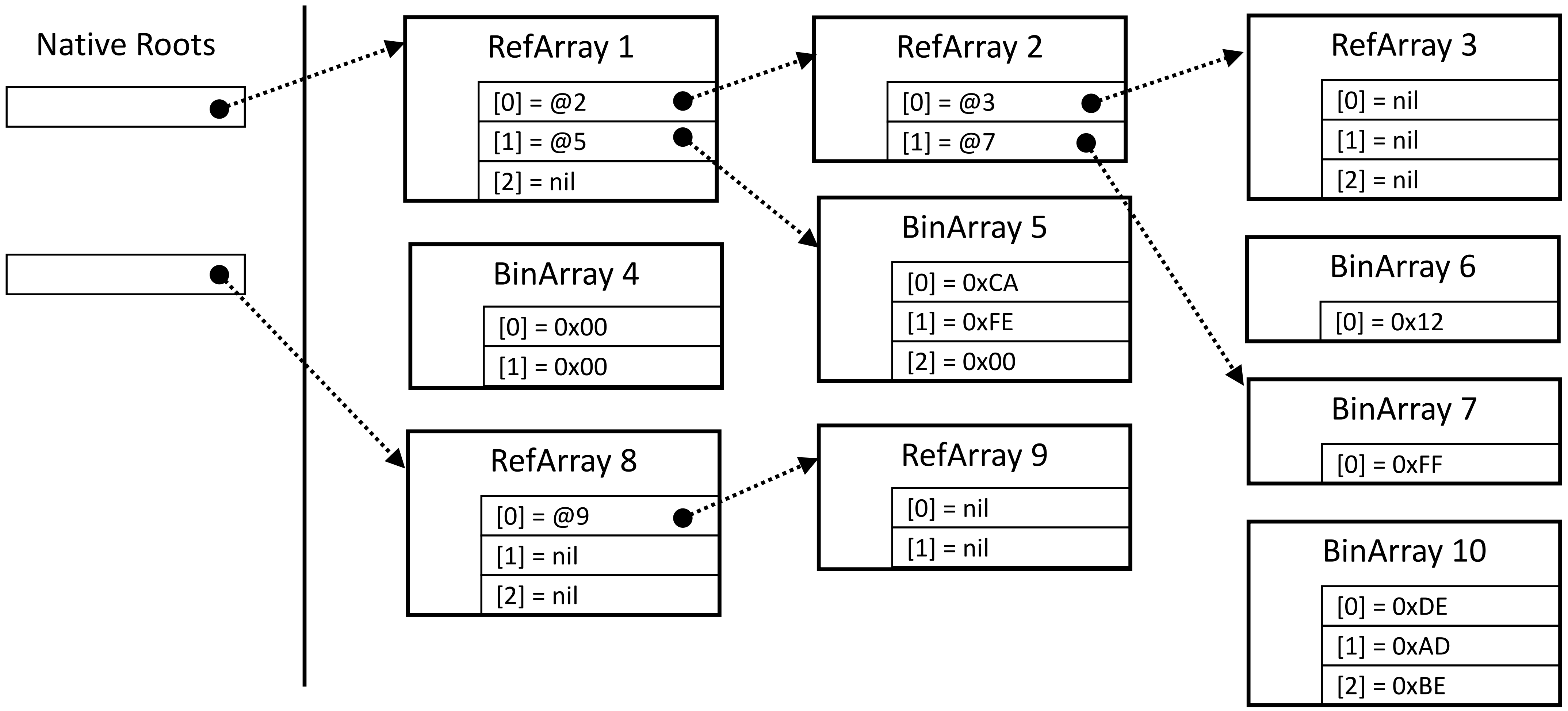
Memory: Graphs of objects



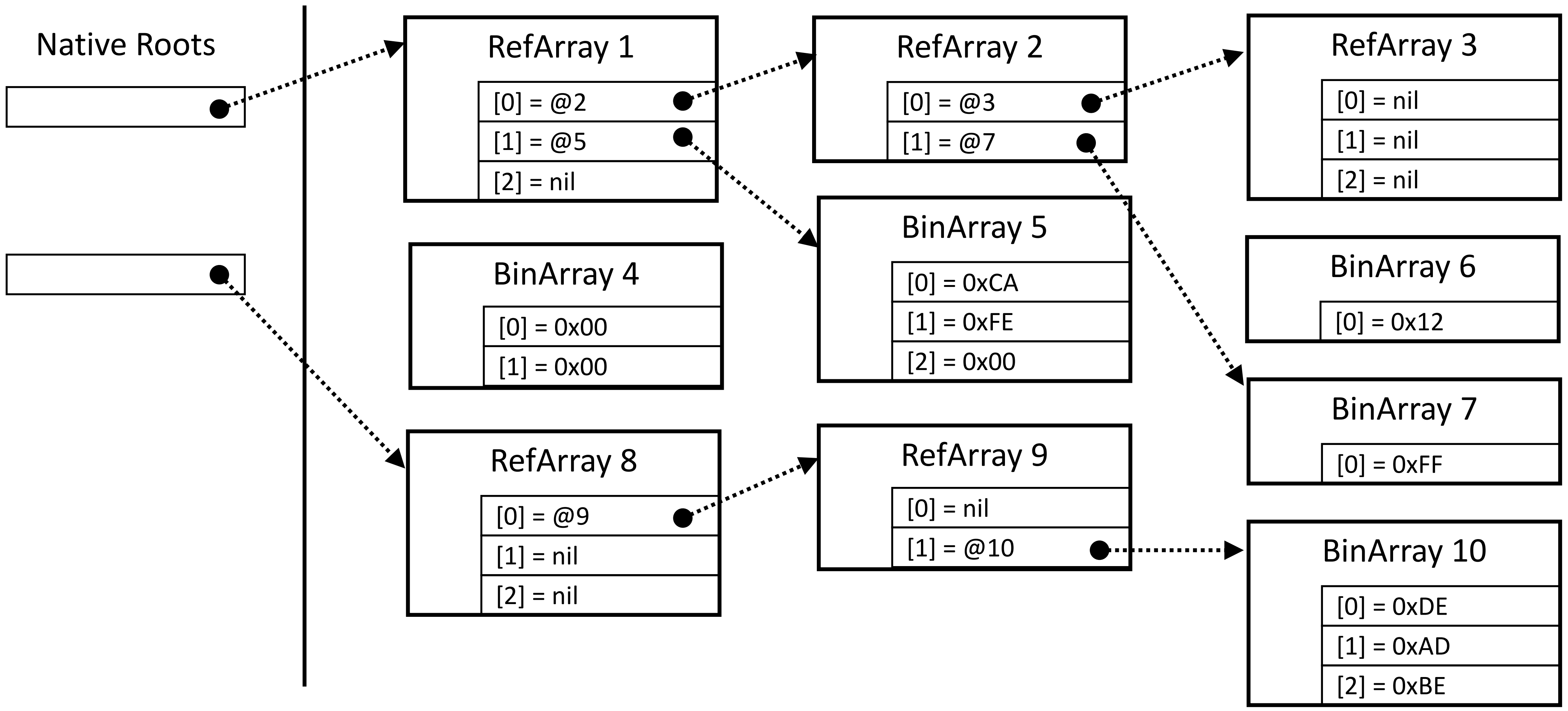
Memory: Graphs of objects



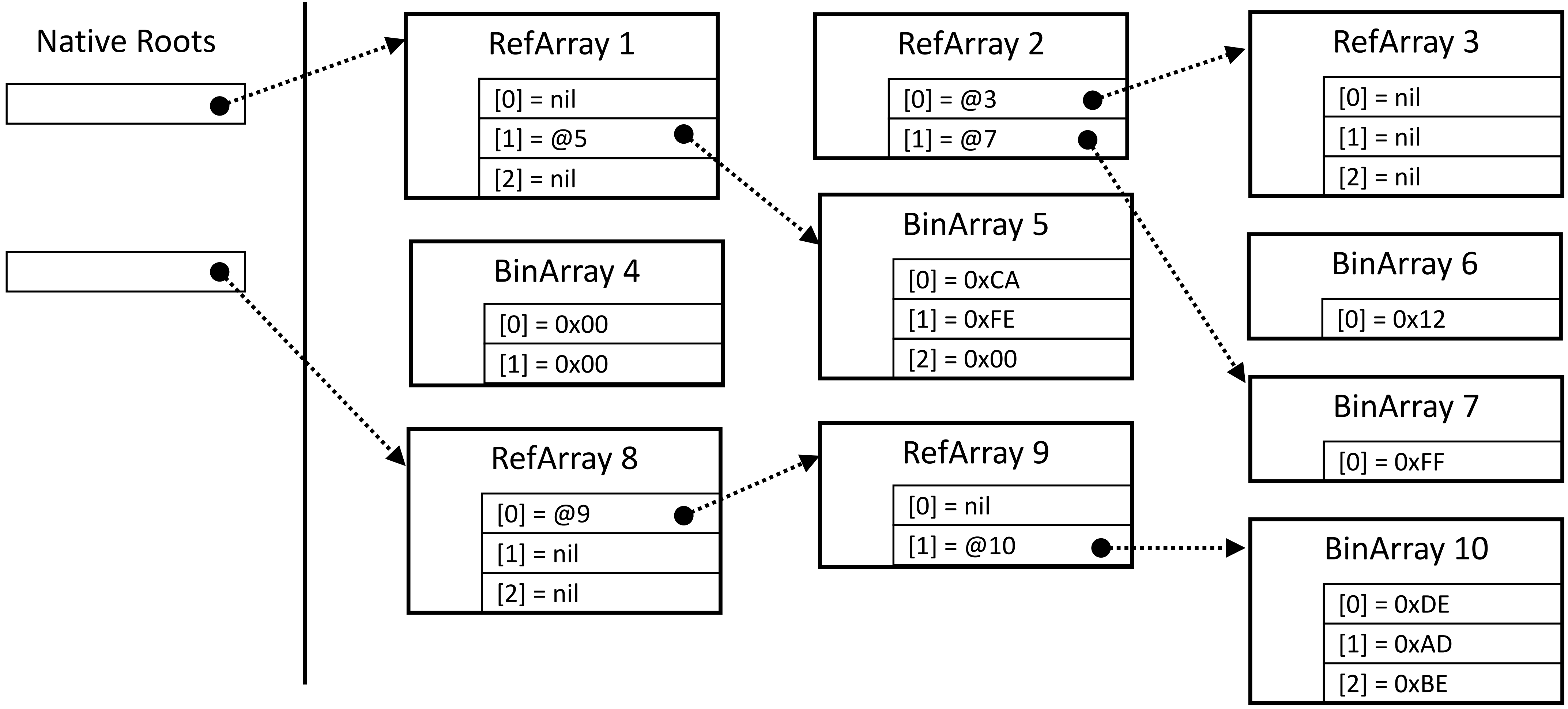
Memory: Graphs of objects



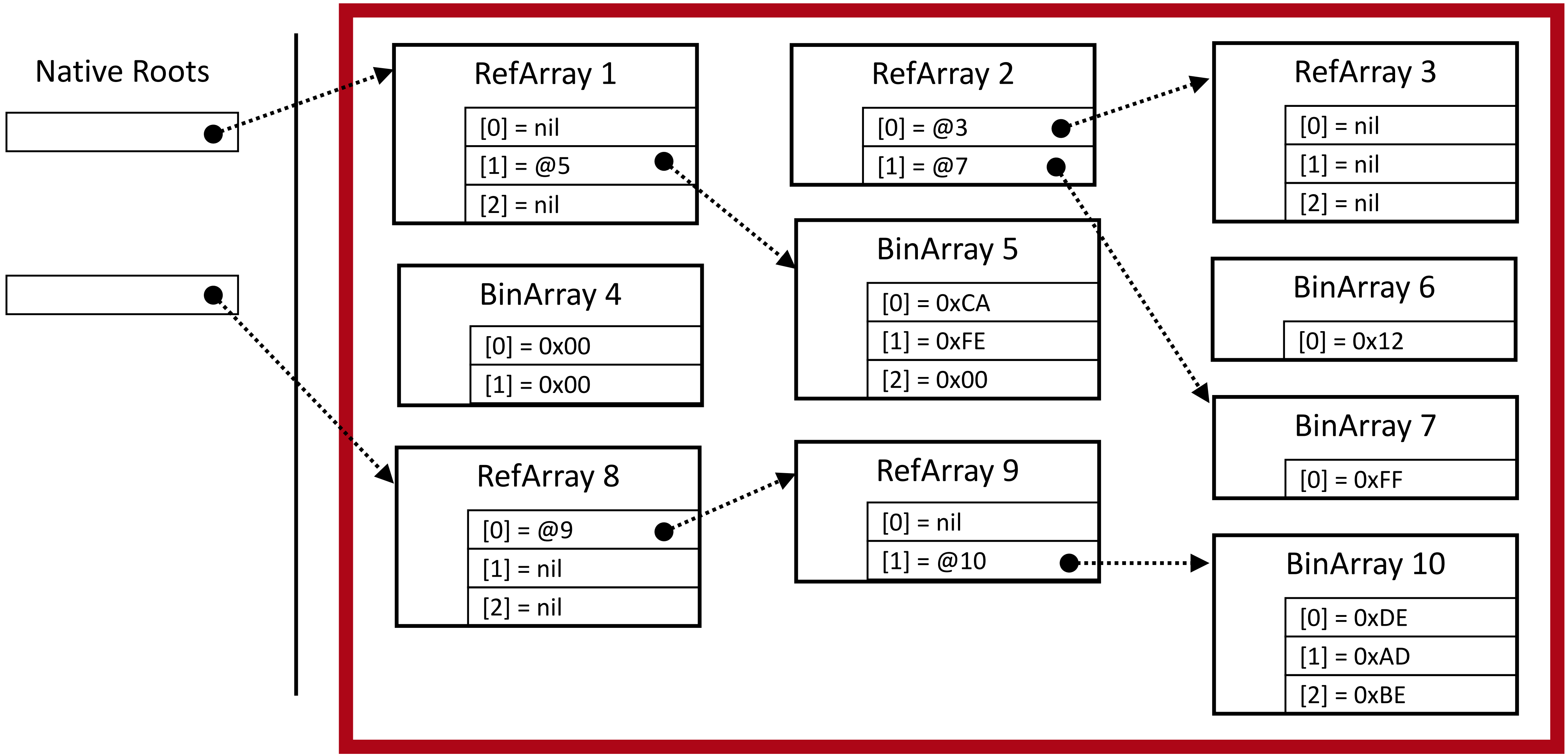
Memory: Graphs of objects



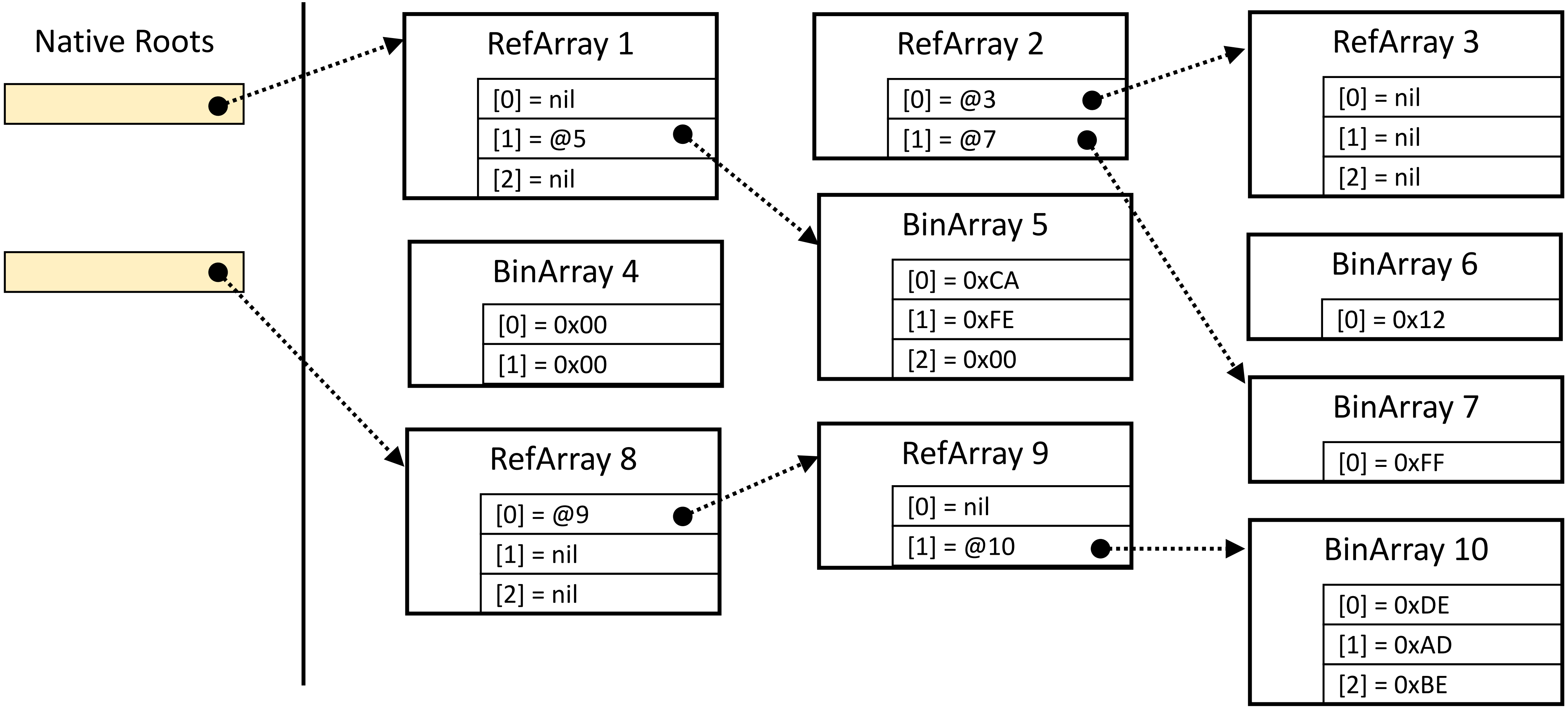
Memory: Graphs of objects



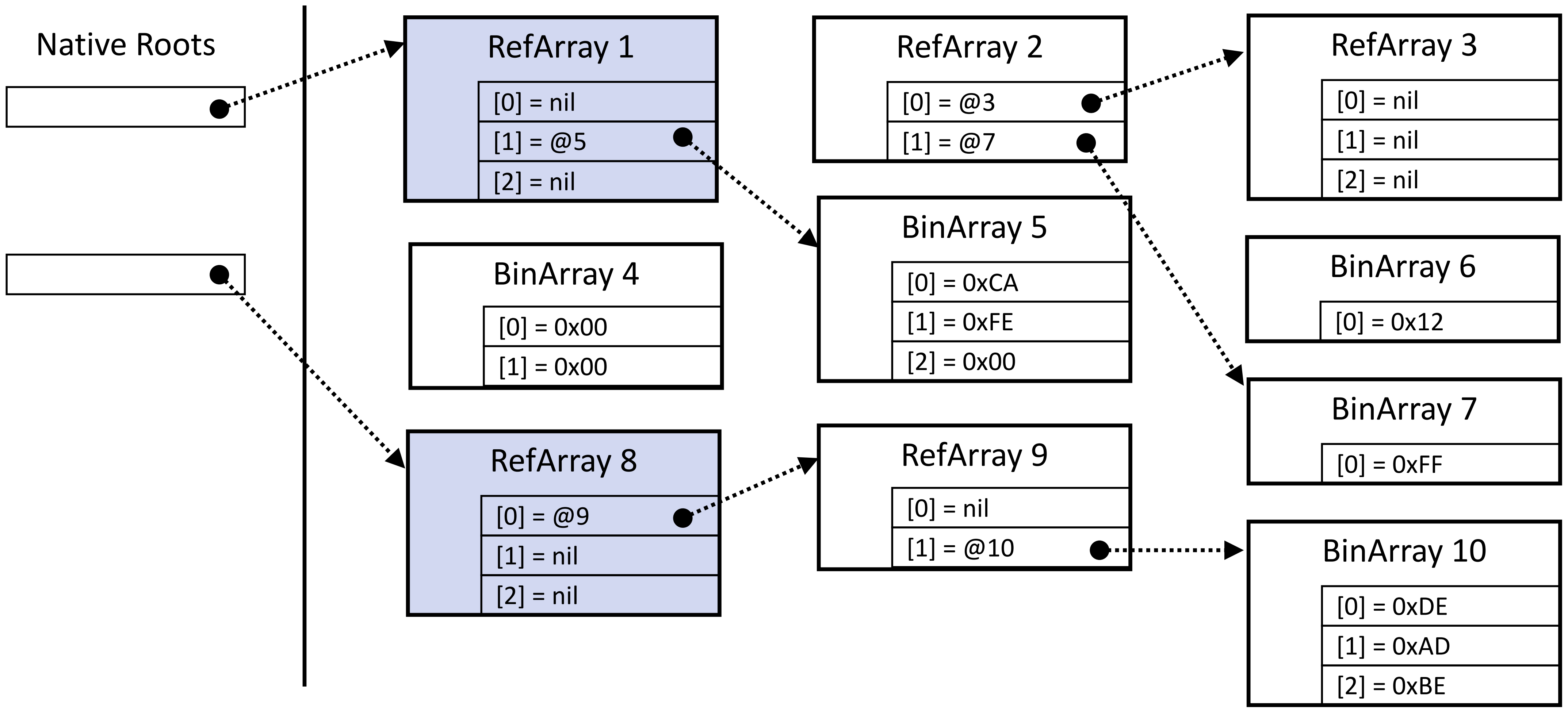
Out of memory!



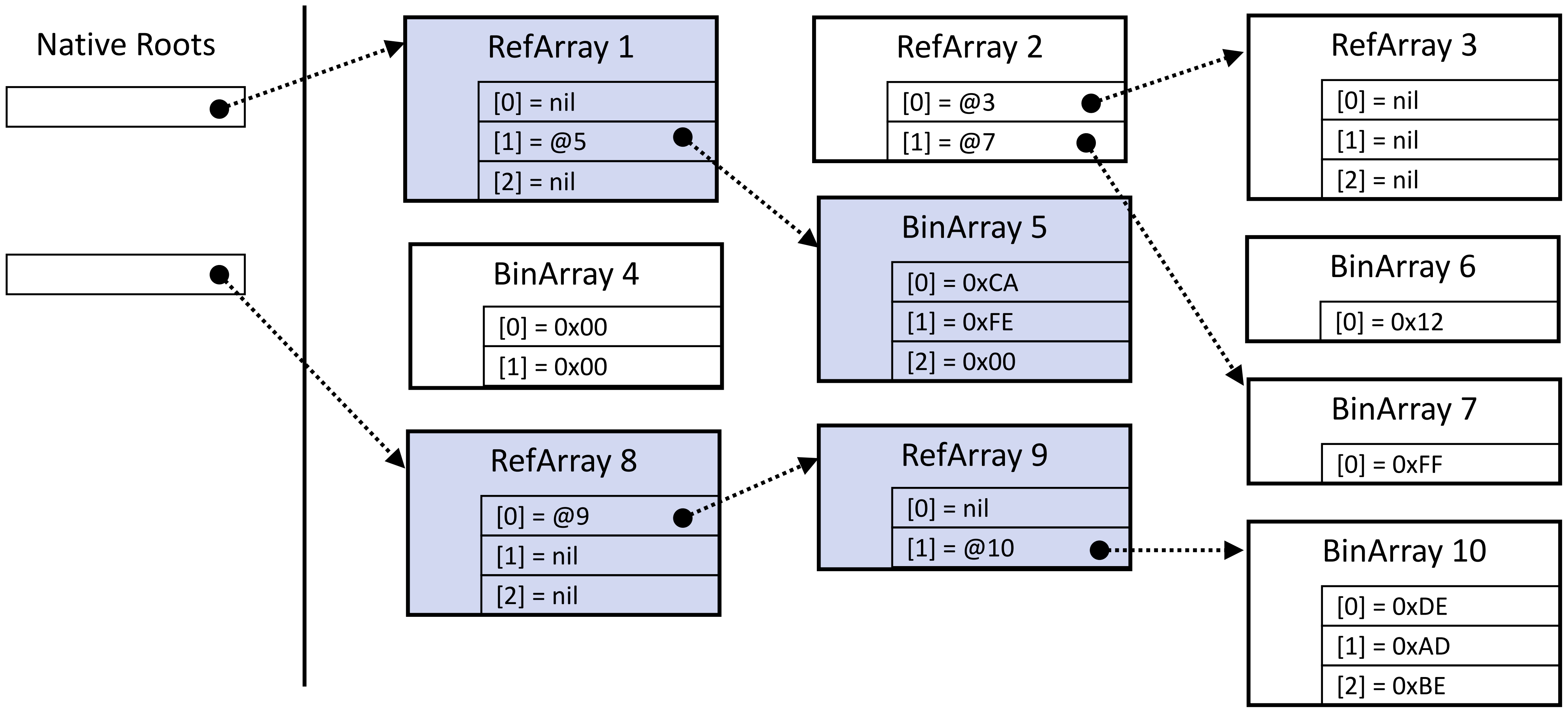
Memory: Graphs of objects



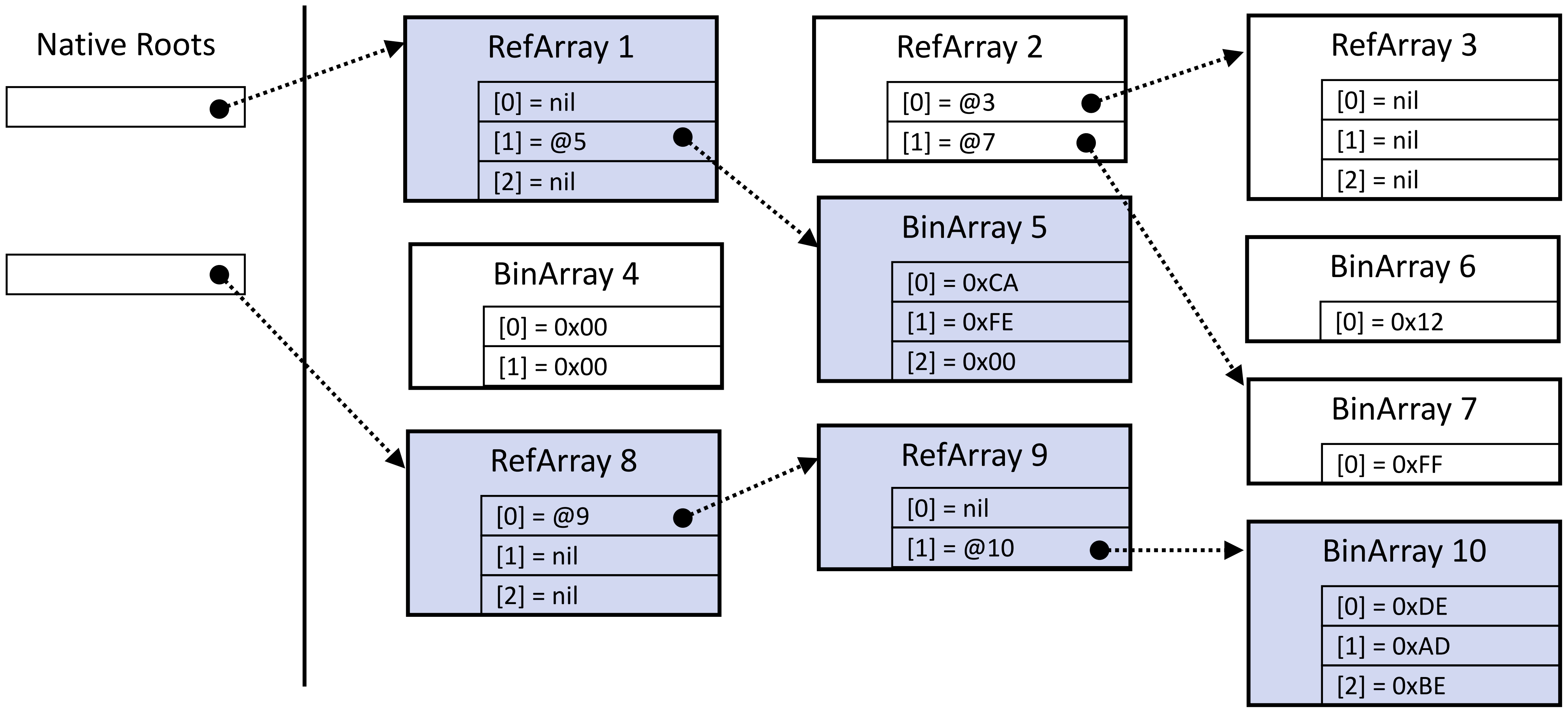
Memory: Graphs of objects



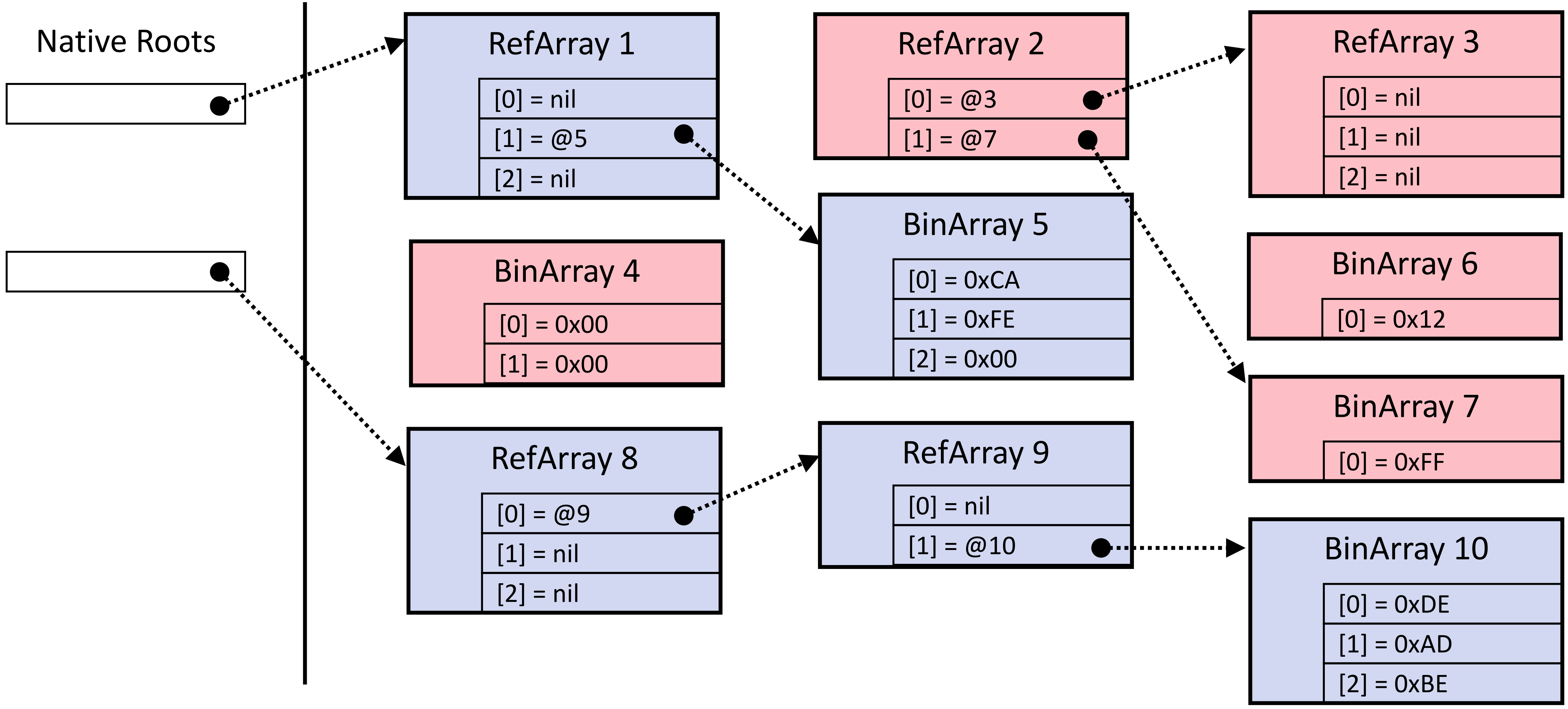
Memory: Graphs of objects



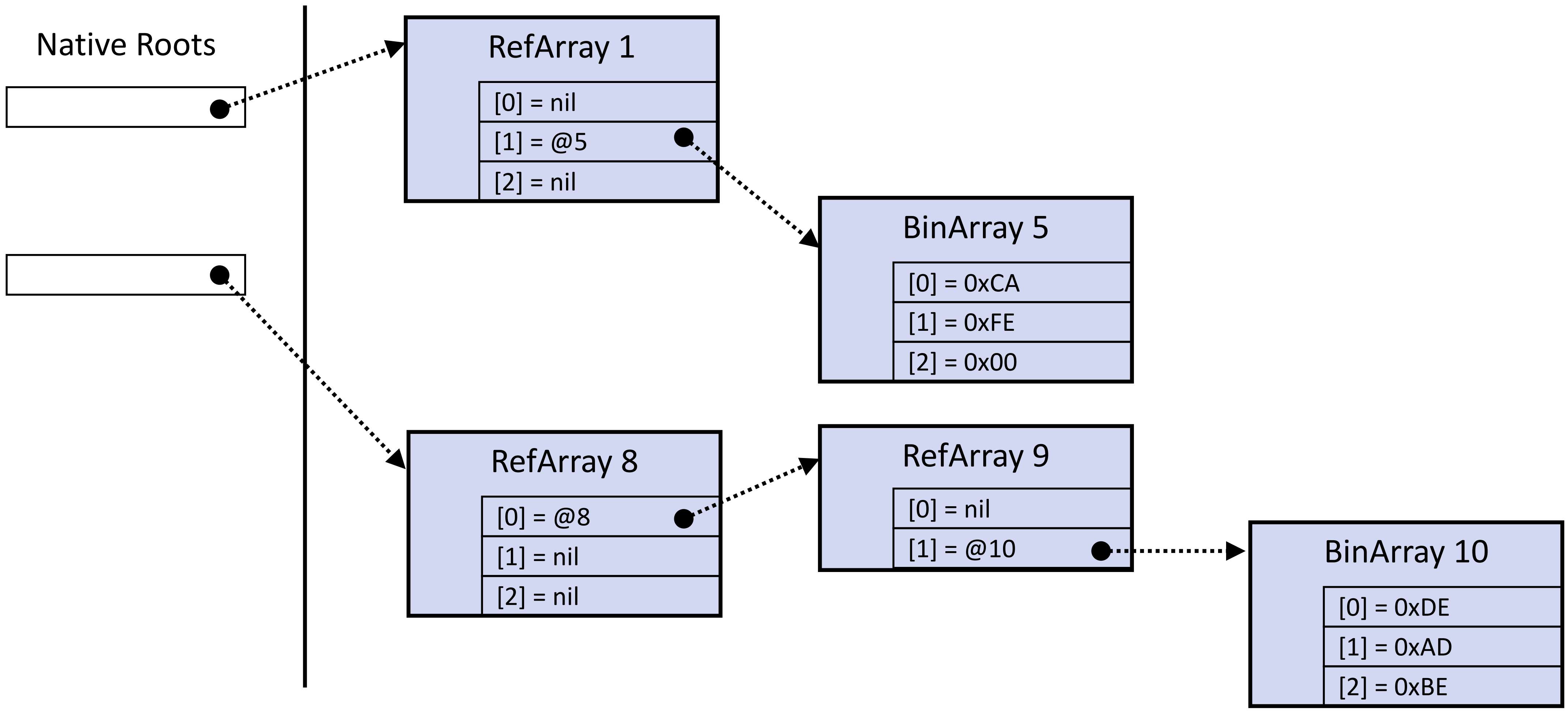
Memory: Graphs of objects



Memory: Graphs of objects



Memory: Graphs of objects



What’s going on?

1. The collector is using a classic “mark and sweep” algorithm
    1. Scan roots, mark reachable objects, put them on a work stack
    2. Scan the objects on the works stack, to find new live objects
    3. Build a map of the used and unused portions of the heap
    4. Add unused portions of the heap to the free list

- Memory is reclaimed in bulk, on demand
- Free memory is found in the space “between” live objects
- The GC has no “per-object” free operation (no destructors)

What does the GC need to know?

- Root scanning – what objects are we working with?
- Object size
- How to find references between objects:
  - Slot Location: object + offset
  - Slot Encoding: need to read and write references
- When is the graph changed?

Configuring the GC

- OMR is massively configurable at compile time
- You must teach the GC about your objects and runtime
- Users (that’s us!) implement “client” code
- A set of APIs defined by OMR, but implemented by consumers
- Client code is compiled and inlined into OMR
- Clients can incrementally develop their client code to enable new technology

The OMR GC API

An experimental set of APIs for the collector

Initializing the collector

- OMR::Runtime
  - Process wide singleton
  - Responsible for initializing the port & thread library
  - Required to bring up the GC subsystem
- OMR::GC::System
  - A complete garbage collected heap
  - Static configuration is optionally passed in to the constructor
  - You can bring up multiple heaps per process (hopefully, haha !)
- OMR::GC::Context
  - A per-thread GC context, required for most public APIs
  - Provides local heap caches, heap access locks, and rooting utilities

Collector initialization

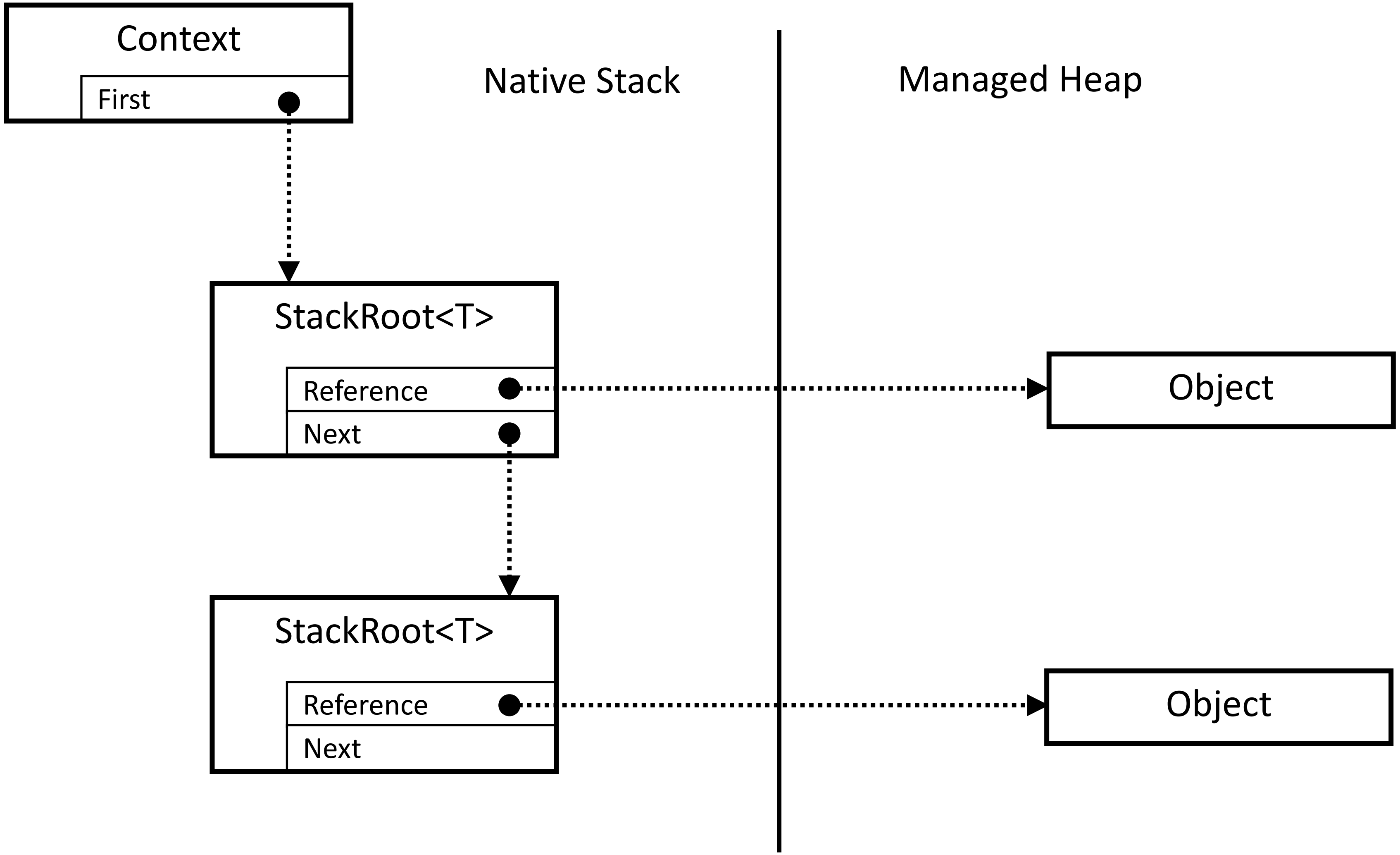
```
#include <OMR/GC/System.hpp>

// Process-wide singleton
OMR::Runtime runtime;

// Each system contains a unique heap. Per-VM.
OMR::GC::System system(runtime);

// Thread-local context to the GC::System.
OMR::GC::Context ctx(system);
```

The object-oriented allocator

```
template <typename T, typename Init>
T* OMR::GC::allocate<T>(cx, size, Init init);
```

- The return value is an unsafe heap reference.
- The initializer must put the new allocation into a scannable state
  - IE: set the objects size, and clear any reference slots.
  - The initializer cannot allocate.
- Collections can happen at allocation sites.
  - Do not hold raw heap reference across allocations sites!
  - Use the NoCollect API when it’s not safe to collect.

StackRoots: Automatically rooted pointers



StackRoot: Automatically Rooted References

Attached to a specific context, and null by default:

```
StackRoot<Object> root(cx);
```

Assignable, and comparable:

```
StackRoot<Object> r0(cx, allocateObject());
StackRoot<Object> r1 = r0;
r0 == r1; // true
```

Have a pointer-like API:

```
Root->field = 42;
(*root).field = 42;
```

Stack Roots have LIFO semantics and must be allocated on the stack.

Finding slots in Objects

- We need to show the GC how and where GC refs are stored
- We implement an object scanner that can notify GC visitors about edges between objects
- We give the visitor slot handles (pointers to slots).
- The GC uses these handles to read/write references from object slots.
- The OMR::GC::RefSlotHandle can be used for slots containing plain, untagged, full-width addresses
- Clients can provide their own slot handle types for defining custom read/write operations.

Let’s get started!

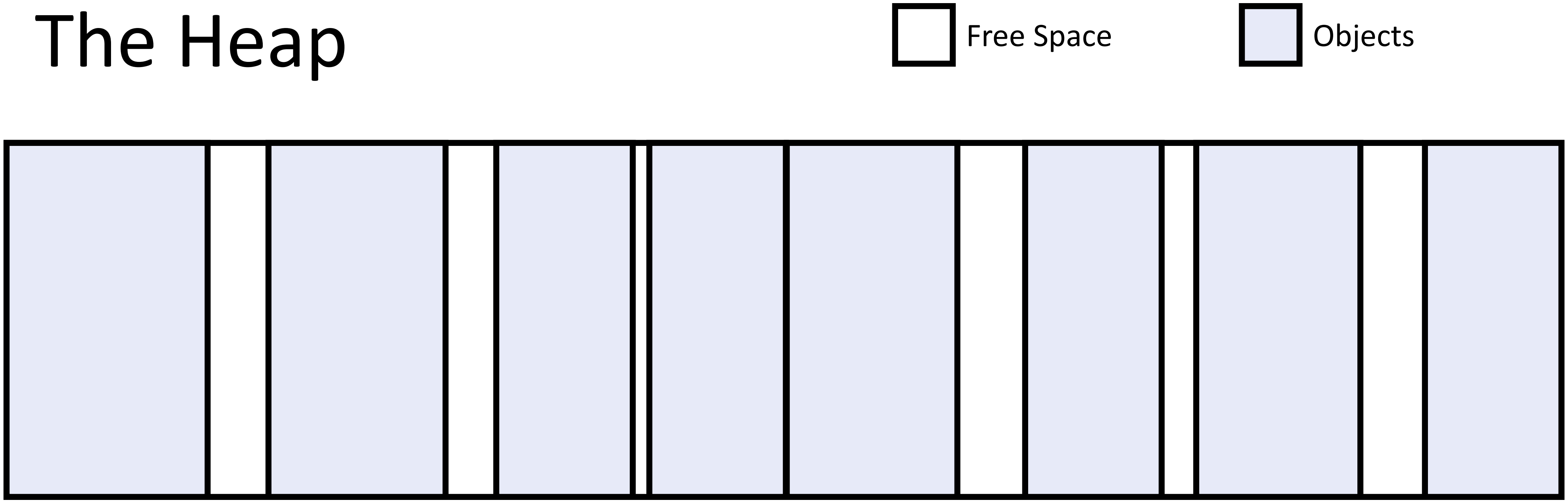
Coffee break!

Heap Compaction

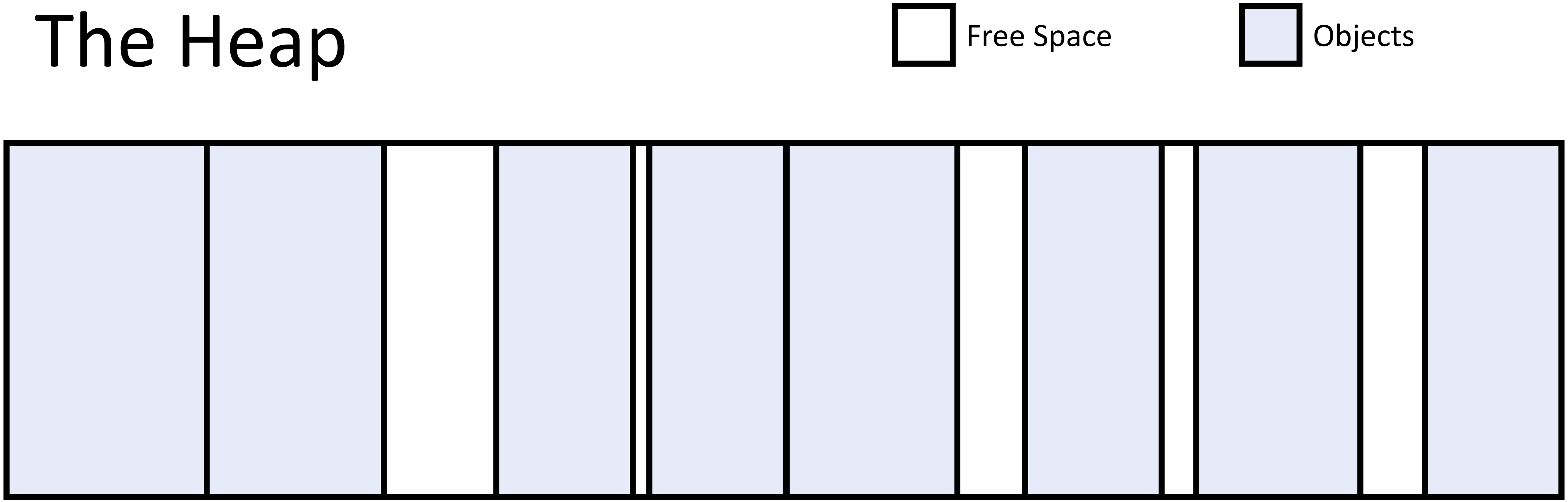
Heap Compaction

- Over time, heap memory becomes fragmented
- Heap fragmentation is bad for the application
  - Slow allocation
  - Bad data locality
  - Unusable heap memory
- The collector can slide all live objects together
- Groups heap into live, and free regions
- Extremely important for long lived applications

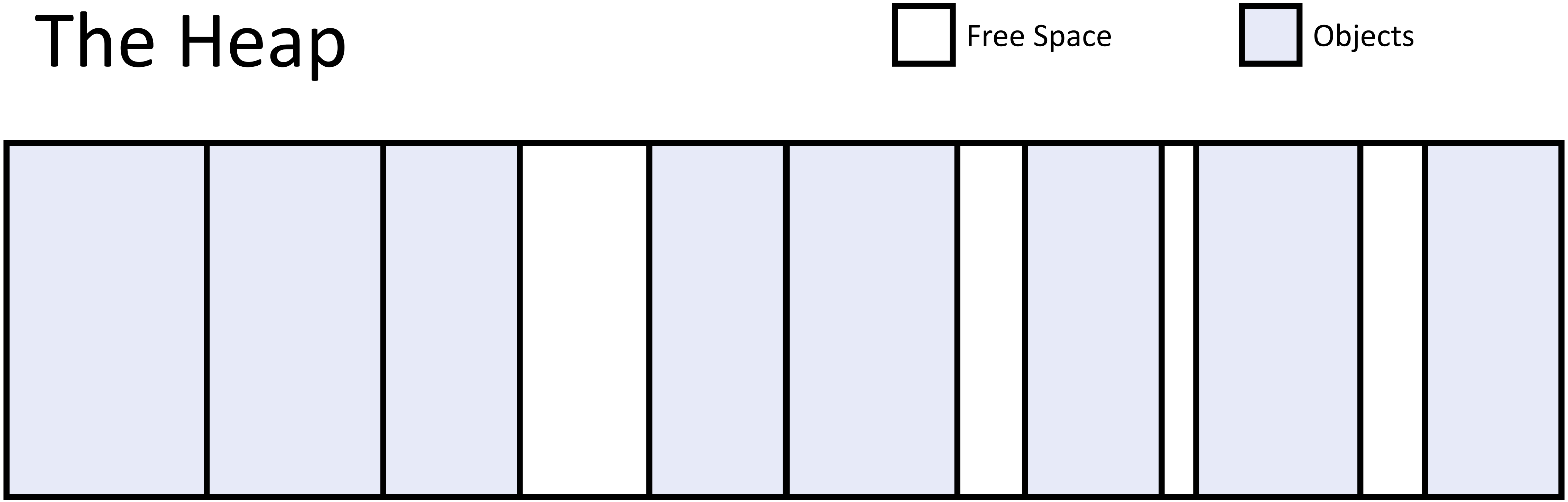
The Heap



The Heap

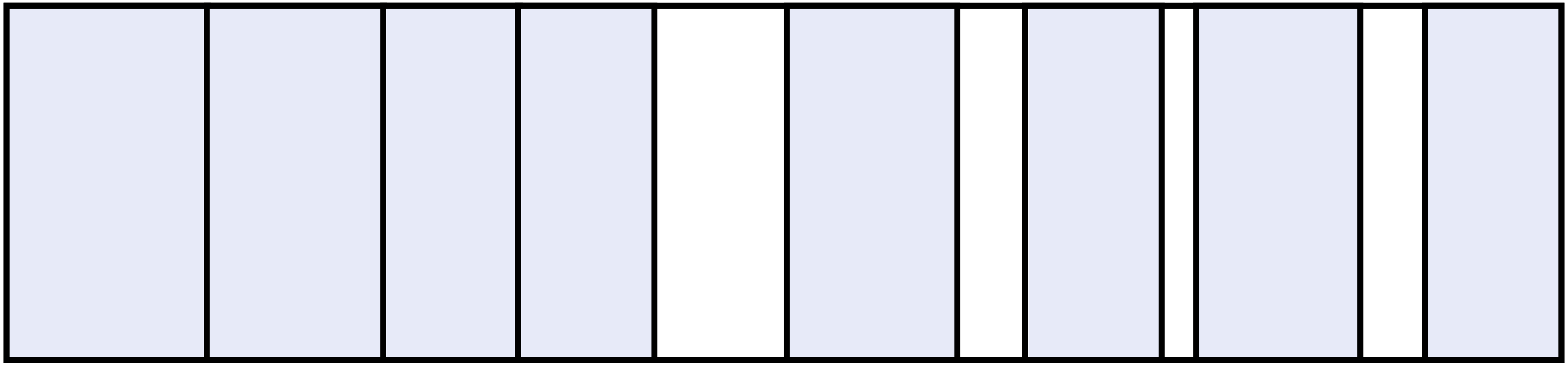


The Heap

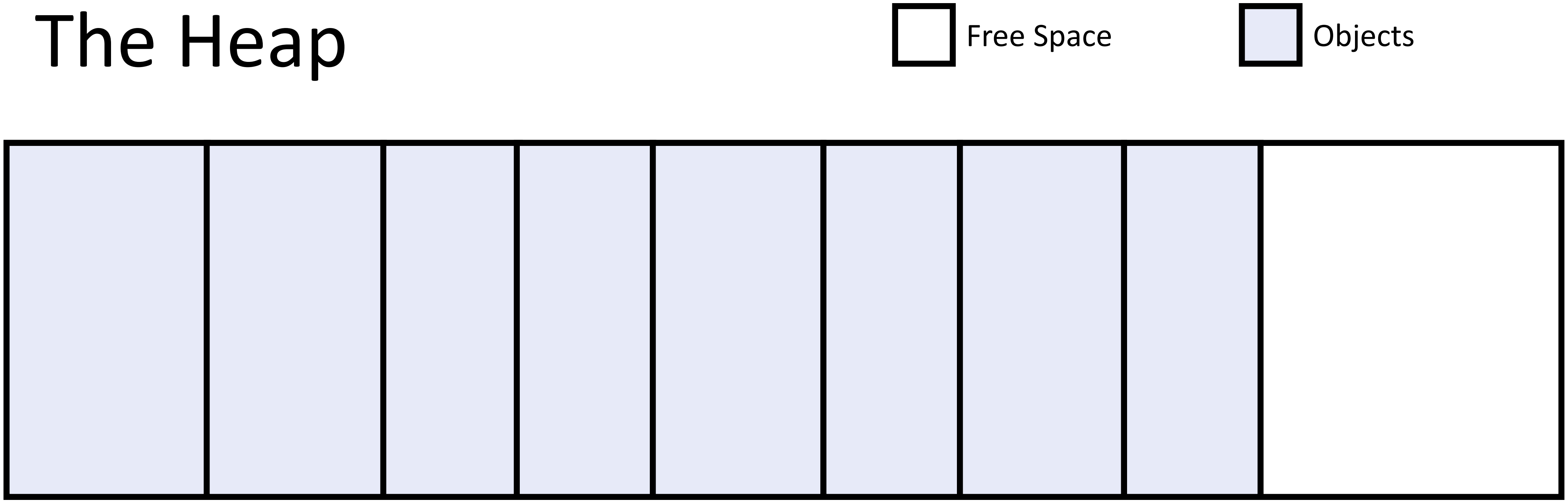


The Heap

Free Space Objects



The Heap



Generational collection (Scavenging objects)

What is generational GC?

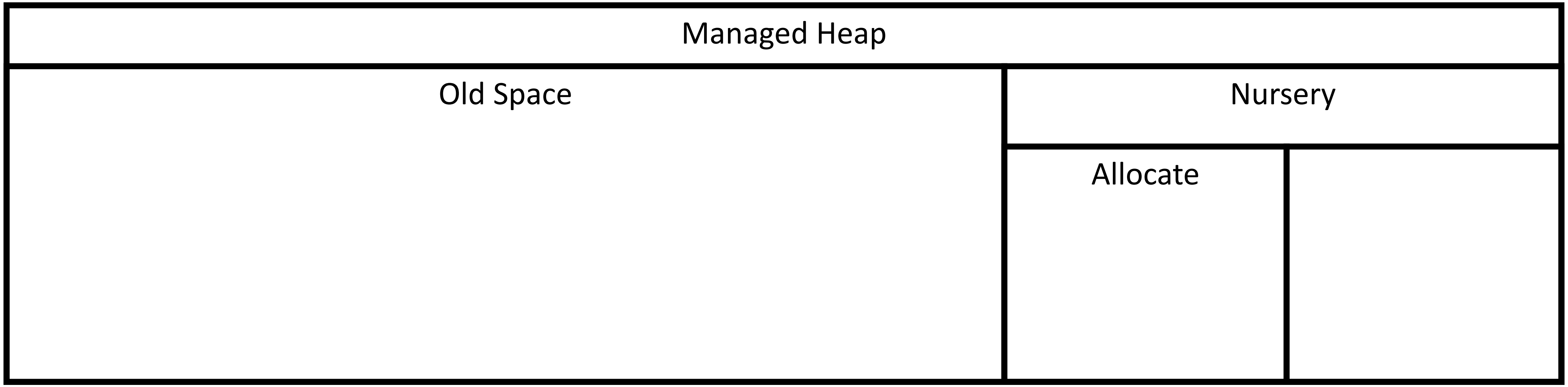
- Weak generational hypothesis:
  - Young objects are more likely to die, or
  - The longer an object lives, the more likely it is to survive.
- The plan: scan only newly allocated objects
- Old objects will survive
- When objects survive long enough, tenure to old-generation
- Also known as “local collection”

Remembering old objects

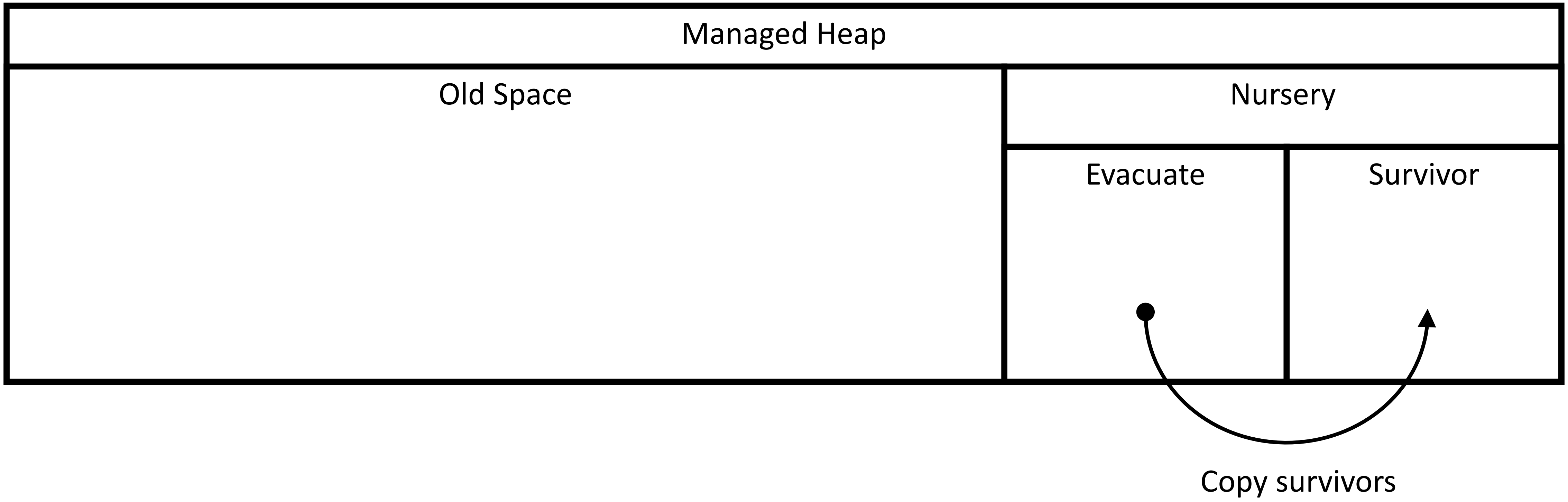
- When an old object references a new object, we “remember” it
- During a local collection old objects will survive
- In a local collection, treat remembered objects as roots
- How do we find old -> new references?
- Use a “write barrier” to track all object graph updates
- Every time a reference is stored:
  - Is the referrer in old space?
  - Is the referent in new space?

  => Remember the referrer

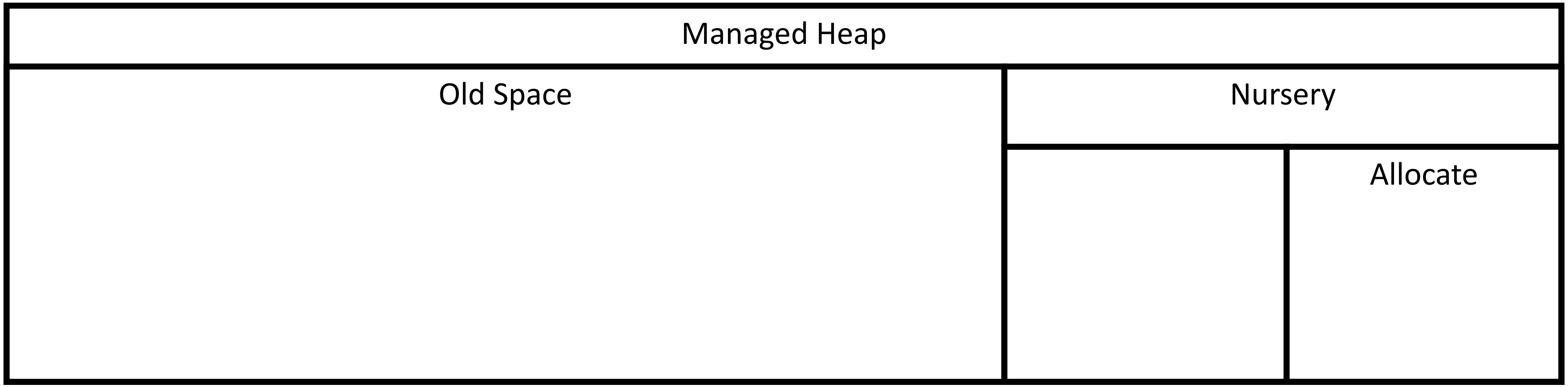
The Generational Heap



The Generational Heap: Scavenge



The Generational Heap



OK, Back to work

Thank You!!