Are many heaps better than one?

A new direction for persistence

Mario Wolczko

Architect, Oracle Labs

MoreVMs workshop, 23 March 2021

Safe harbor statement

Non-Volatile RAM is here!

- Intel has introduced NVRAM (Optane) for server-class machines.
  Others are likely to follow.
- NVRAM is organized as a filesystem. Individual files can be mapped into a process' address space, and then accessed and modified using memory ops (loads, stores, etc.)
- Caches are still volatile, and cached data must be written back to persist.

Optane memory vs DRAM

- Significantly **denser** (~8x)
- A little **slower** (~2–4x latency for reads, ~1x for writes, ~1/3x read bandwidth and ~1/6x for writes)
- **Cheaper** (per bit), depending on module capacity — and fluctuating DRAM prices!
- **Non-volatile**

For more detail:
https://medium.com/@mwolczko/non-volatile-memory-and-java-7ba80f1e730c

Automatic persistence: opportunities

- Simpler application code:
  - *The in-memory and persistent representations are one and the same*.
    No need to write code to serialize and deserialize…sometimes.
    Exceptions: portability (language-, machine-, OS-independence); simpler formats; standards
  - Not new: Lisp’s sys-out (1967), Smalltalk’s snapshots (1974),
    Atkinson, Bailey, Chisholm, Cockshott and Morrison, *PS-algol: a language for persistent programming* (1983)
- Performance, availability, scale
  - Reduced startup time — no long wait while data are read from block storage into RAM
  - Reduced update latency — updates can be durable in <1µs
  - Higher capacity and lower cost/bit

Automatic persistence: challenges

Consistency

- *The in-memory and persistent representations are one and the same*.
- Must have a way to ensure a batch of updates is atomic:
  - Logging and recovery, which impact performance

Position independence

- Not a good idea to rely on mapping to the same address
- Relocation — but what about failures, scaling?
- Better: *position-independent data*. E.g., pointers could be *self-relative* (a pointer at address *p* with value *v* points to *p+v*)
- This can be hidden in the runtime of a managed language

Software evolution

- How to update persistent data when data structure schema change?

One approach: Persistence by reachability

Some objects or values are designated as *persistent roots*.

Any object reachable from a persistent root is automatically persisted; everything else is volatile.

This approach was pursued in the 1980s and -90s; e.g., by PJama (Sun Labs & Glasgow Uni, JSR 20).

Requires objects to be moved from DRAM to NVRAM when they become persistent.

See the recent papers and thesis by Thomas Shull for a good exemplar.

Problems:

1. Some objects are problematic (e.g., threads with unmanaged frames, objects referencing external state [e.g. files, sockets])
2. Bugs which corrupt the heap

“Have you tried turning it off and on again?”

https://www.youtube.com/watch?v=nn2FB1P_Mn8

“Have you tried turning it off and on again?”

How often do you restart an application or system to resolve a problem? Daily? Weekly?

https://www.youtube.com/watch?v=nn2FB1P_Mn8

“Have you tried turning it off and on again?”

How often do you restart an application or system to resolve a problem? Daily? Weekly?

Why does that work?

https://www.youtube.com/watch?v=nn2FB1P_Mn8

"Have you tried turning it off and on again?"

How often do you restart an application or system to resolve a problem? Daily? Weekly?

Why does that work?

The in-memory (volatile) data must be in a "bad" state, while the persistent data are still fine.

https://www.youtube.com/watch?v=nn2FB1P_Mn8

“Have you tried turning it off and on again?”

How often do you restart an application or system to resolve a problem? Daily? Weekly?

Why does that work?

The in-memory (volatile) data must be in a “bad” state, while the persistent data are still fine.

Unsurprising: In-memory structure definitions change much more often, are much more complicated, and much more vulnerable to bugs and bit flips.

https://www.youtube.com/watch?v=nn2FB1P_Mn8

“Have you tried turning it off and on again?”

How often do you restart an application or system to resolve a problem? Daily? Weekly?

Why does that work?

The in-memory (volatile) data must be in a “bad” state, while the persistent data are still fine.

Unsurprising: In-memory structure definitions change much more often, are much more complicated, and much more vulnerable to bugs and bit flips.

Now, what does that mean for a persistent heap?

https://www.youtube.com/watch?v=nn2FB1P_Mn8

Introducing new terminology: *Base* and *derived* data

Data can be *base* or *derived*.

Base data can only be recovered from a copy in a different failure domain.

Derived data can be recovered by computation from less-derived (ultimately, base) data.

A fatal flaw?

Heaps are, and perhaps always will be, vulnerable to heap “corruption”.

But a restart will not resolve a problem in a persistent heap.

Using reachability to group data tends to mix base and derived data in the same heap. Discarding and recovering a heap containing base data may be expensive, or even impossible.

This is a serious weakness. Is there a mitigation?

A possible solution: multiple heaps

Partition the data, assigning each object to one of several heaps.

One of the heaps resides in volatile memory and is discarded at program termination.

In the simplest scenario, persistent data reside in a single persistent heap, but this suffers from the same corruption problem.

A more robust approach is to separate the persistent data so that base data live in one or more dedicated heaps, and derived data live in different heaps.

Then in the event of corruption, a restart will discard the volatile data; and heaps which do not contain base data can be discarded at will and rebuilt from base data.

Base data structures should be constructed as to maximize reliability and placed in a separate heap.

Base data heaps may be subject to more extensive protection and recovery.

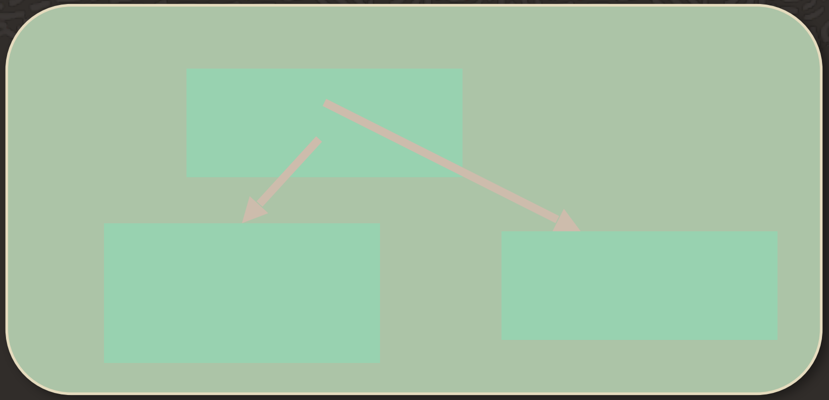
Example: Java

- Each heap is represented by an object (instance of subclass of `Heap`) that acts as a descriptor and interface for the heap.
- When the JVM starts, it starts with a single, volatile heap — just as before. This `MAIN` heap is a `VolatileHeap` (`Heap.MAIN`).
- Each `PersistentHeap` has a `root` field, whose type is specified by a parameter, e.g.

```
PersistentHeap<Map<String,Integer>> h;
…
h.root.add(“foo”,42);
```

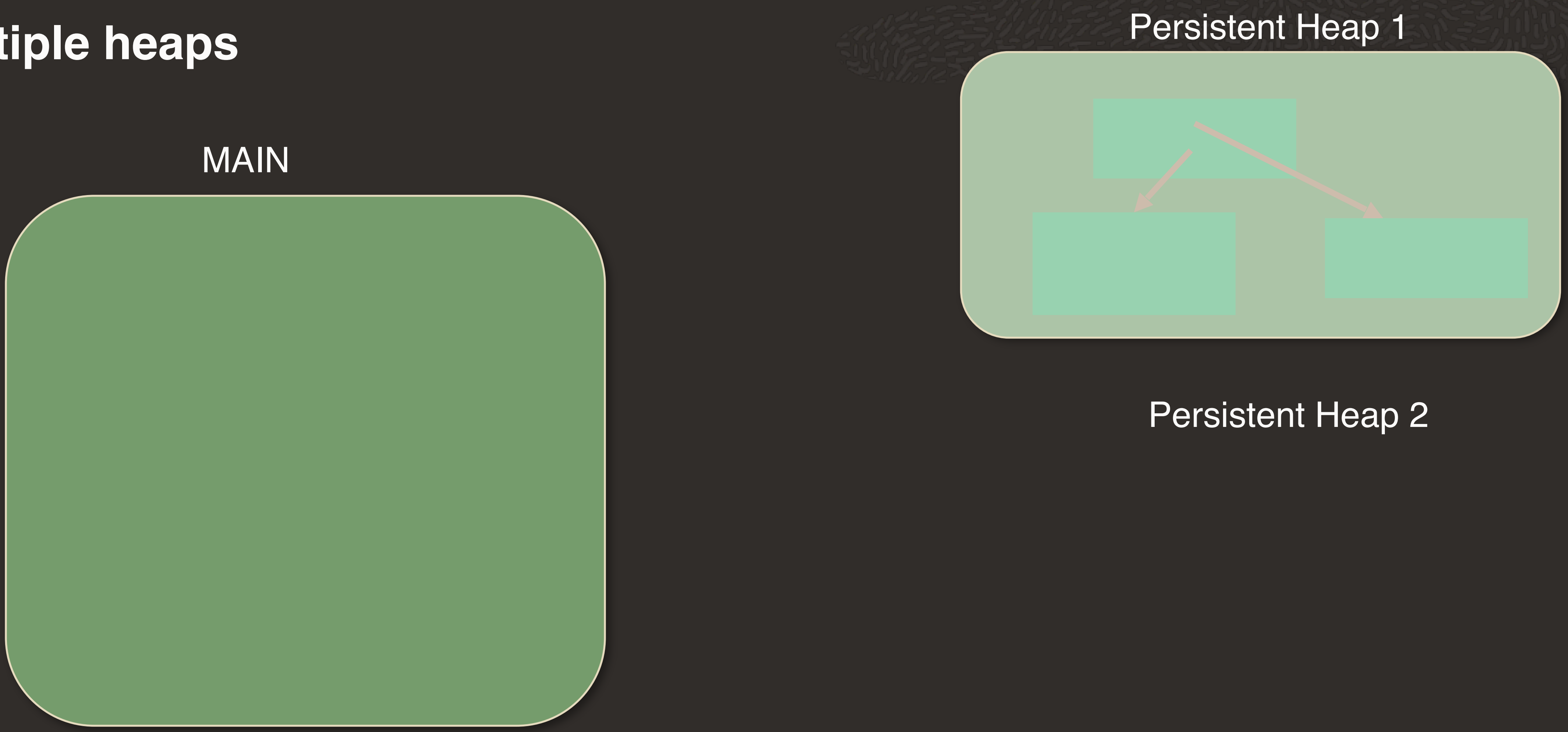
Multiple heaps

Persistent Heap 1

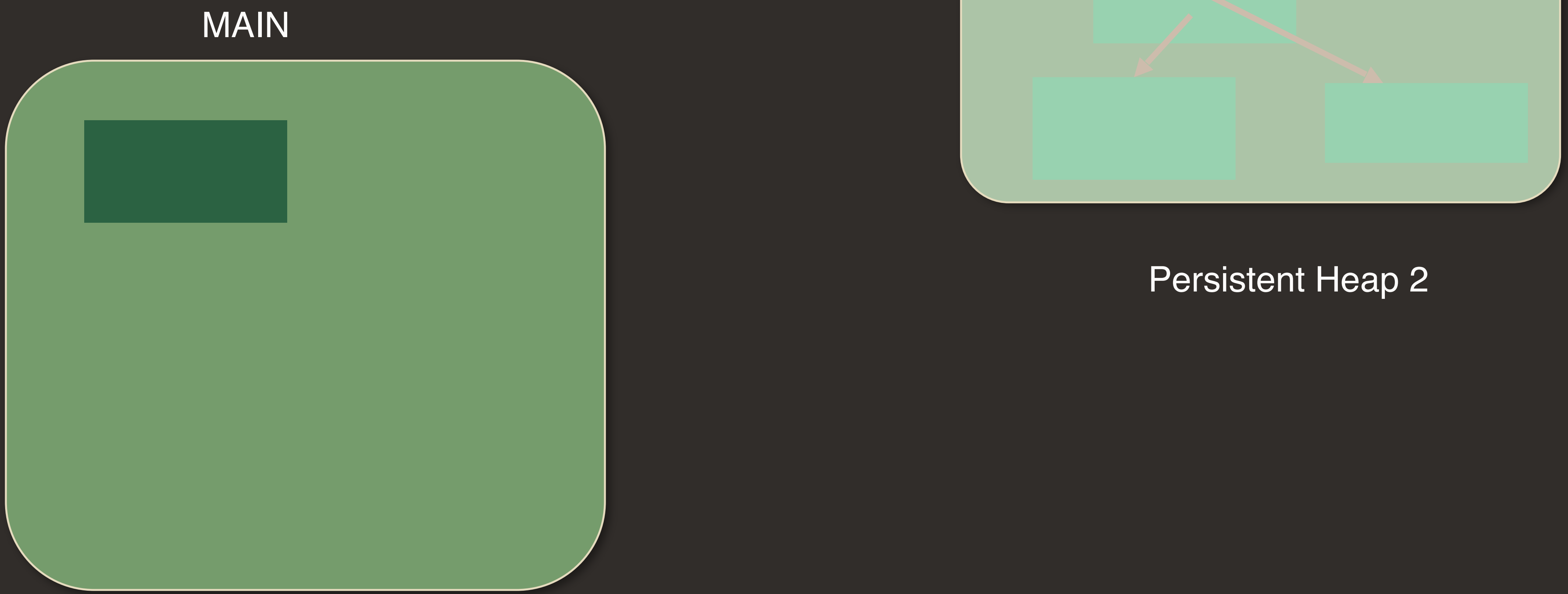


Persistent Heap 2

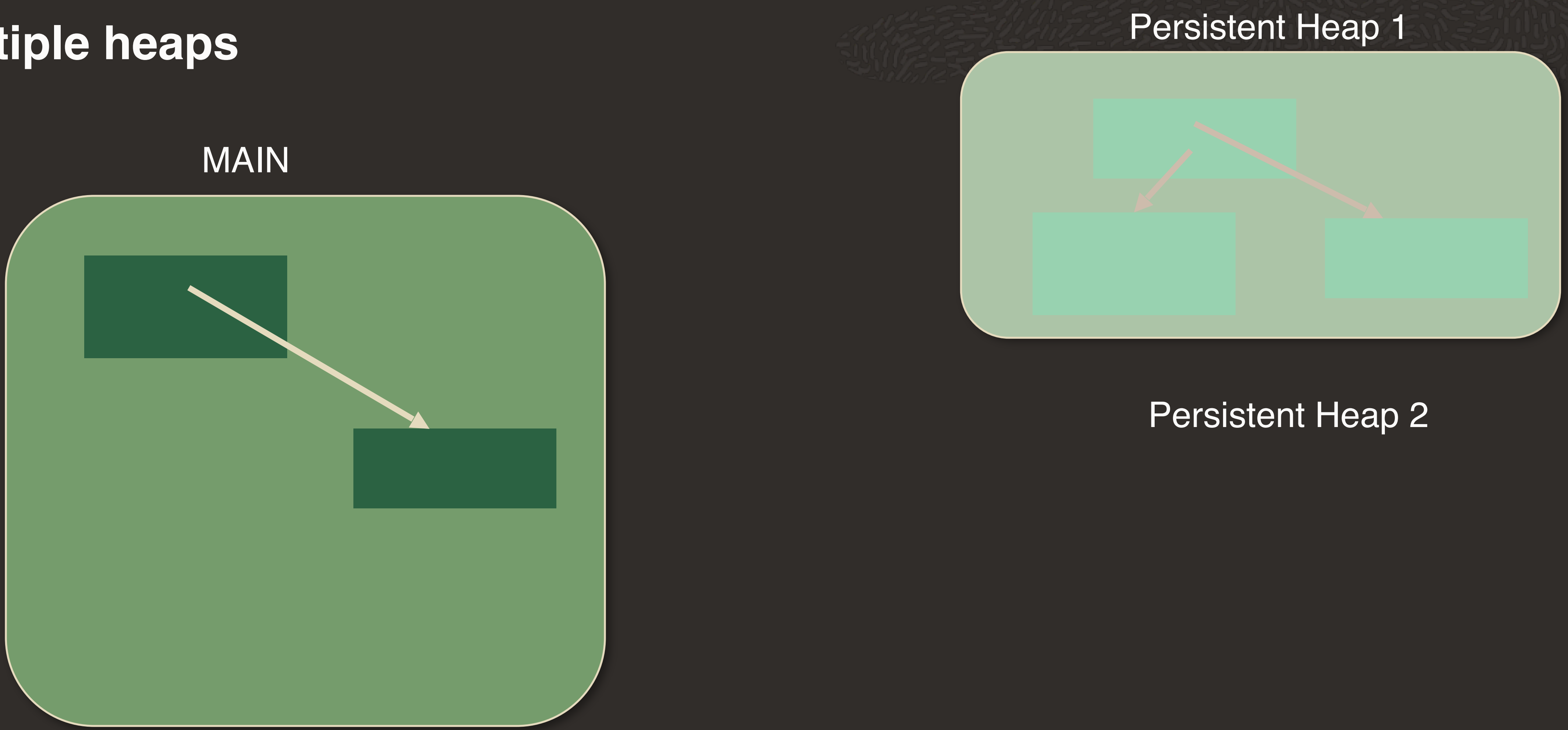
Multiple heaps



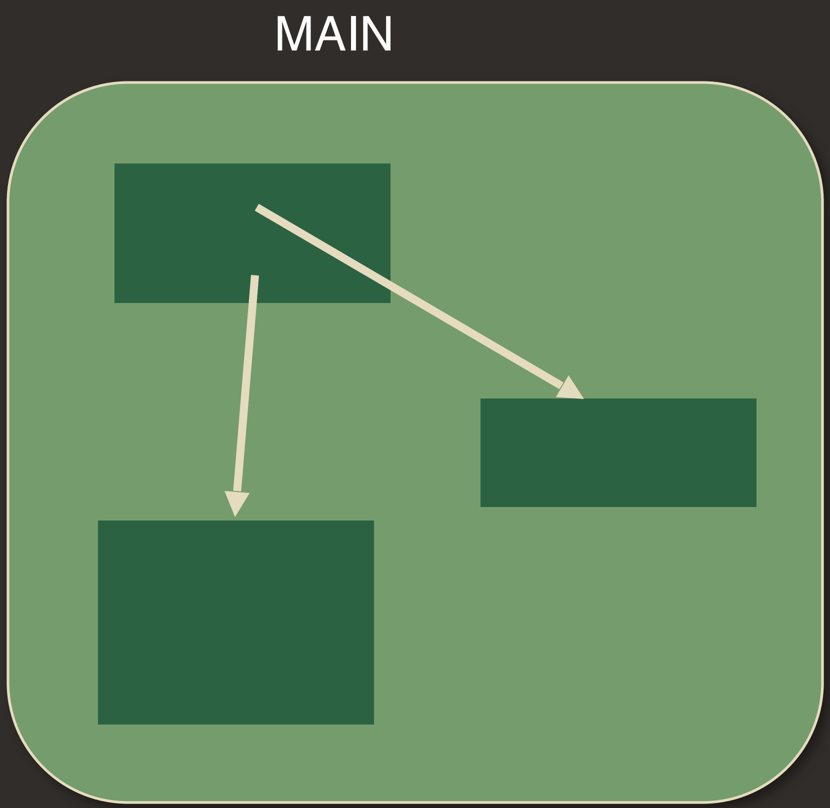
Multiple heaps

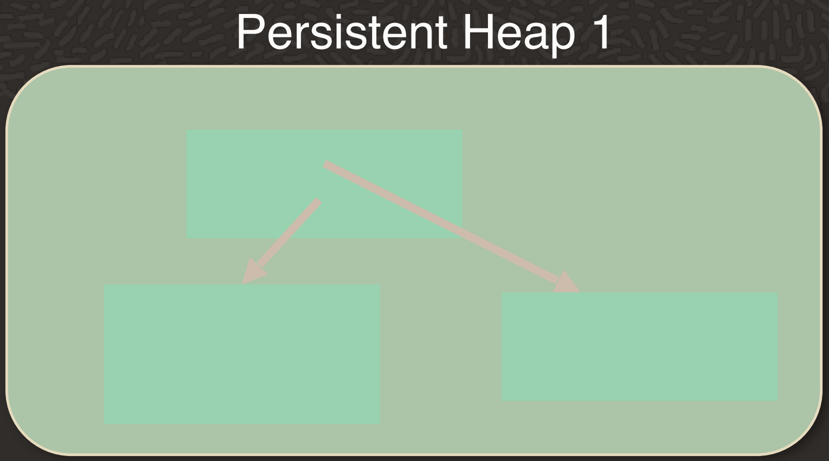


Multiple heaps



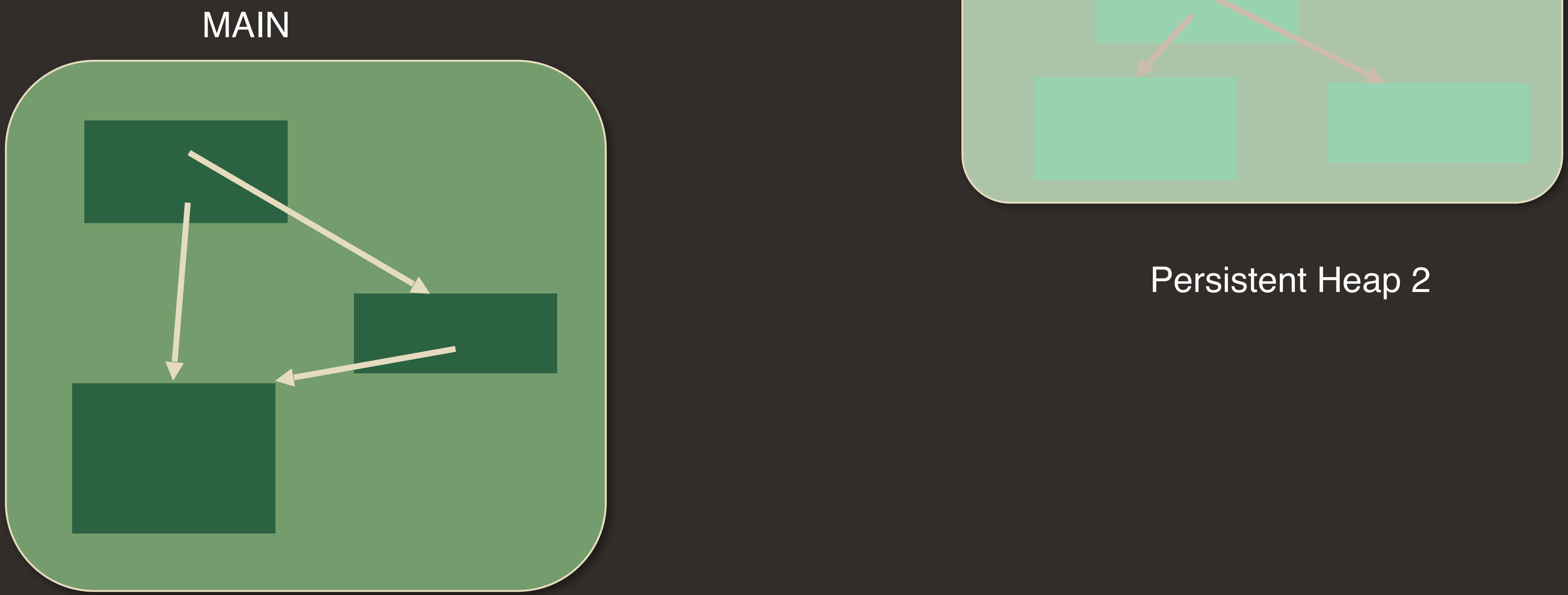
Multiple heaps



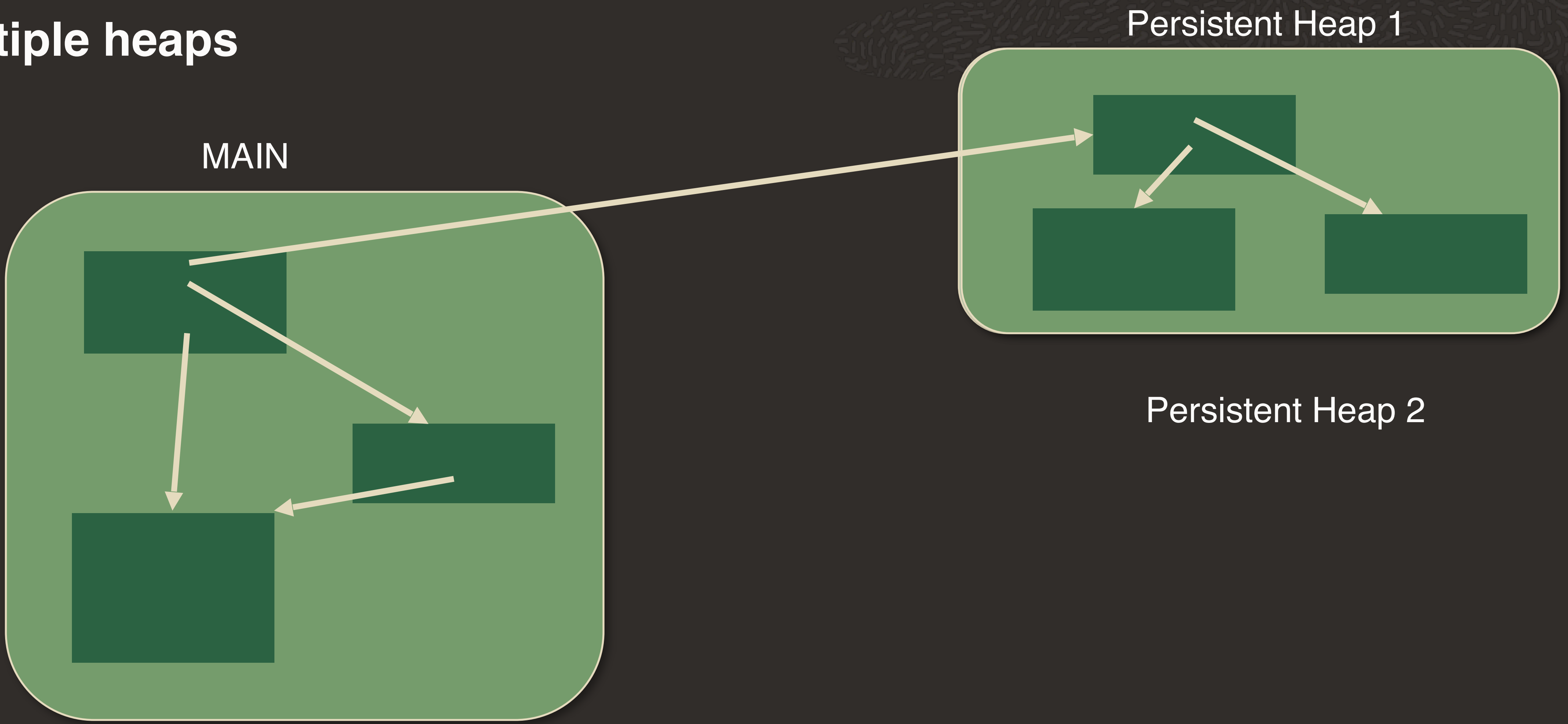


Persistent Heap 2

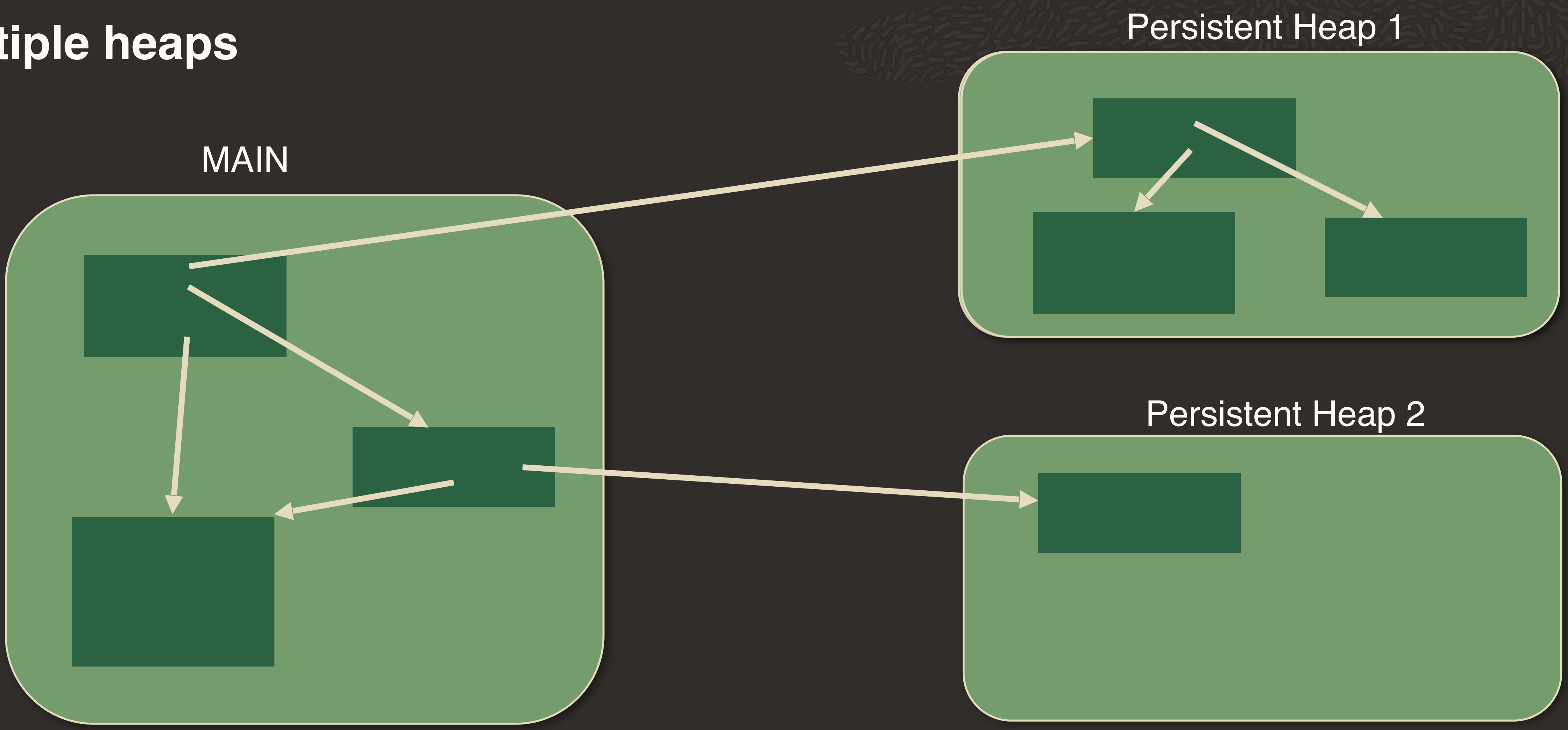
Multiple heaps



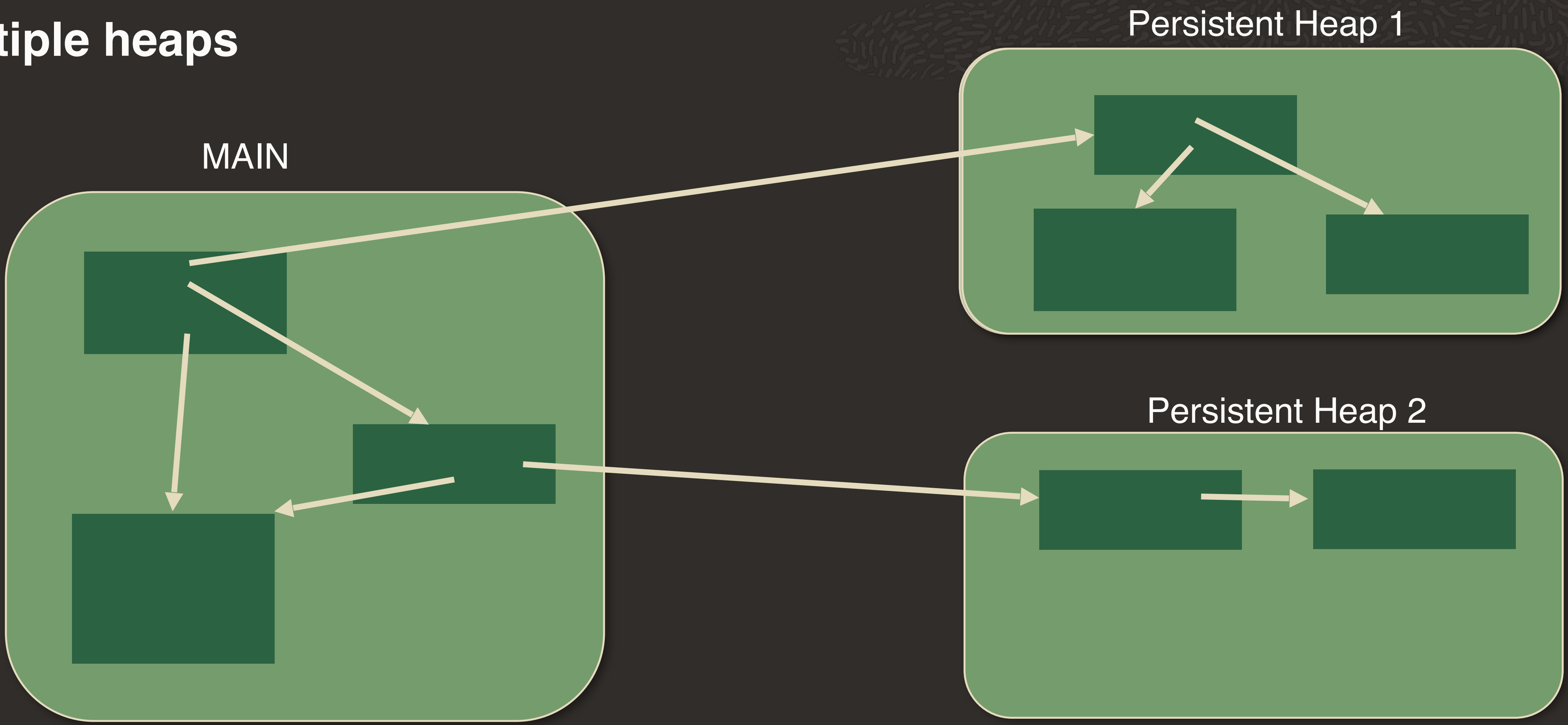
Multiple heaps



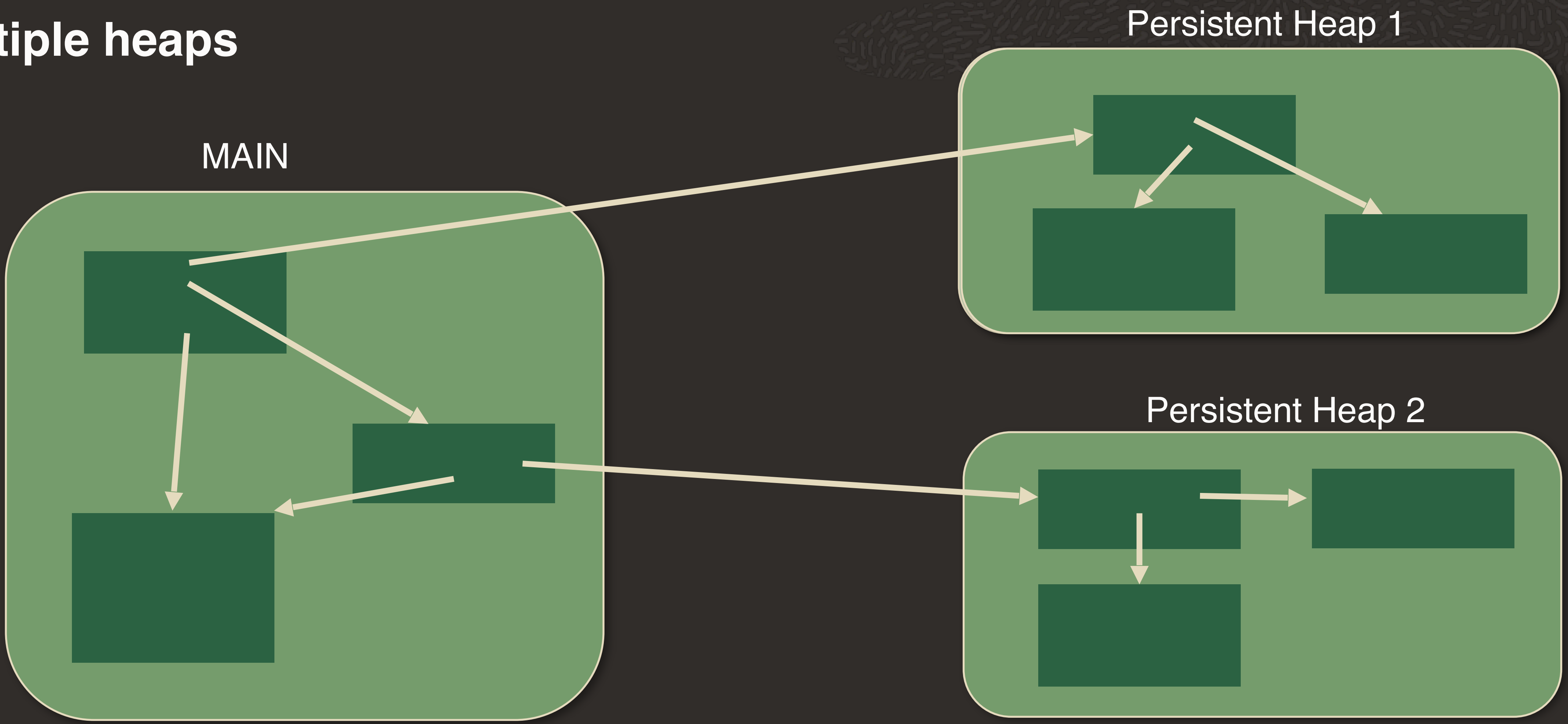
Multiple heaps



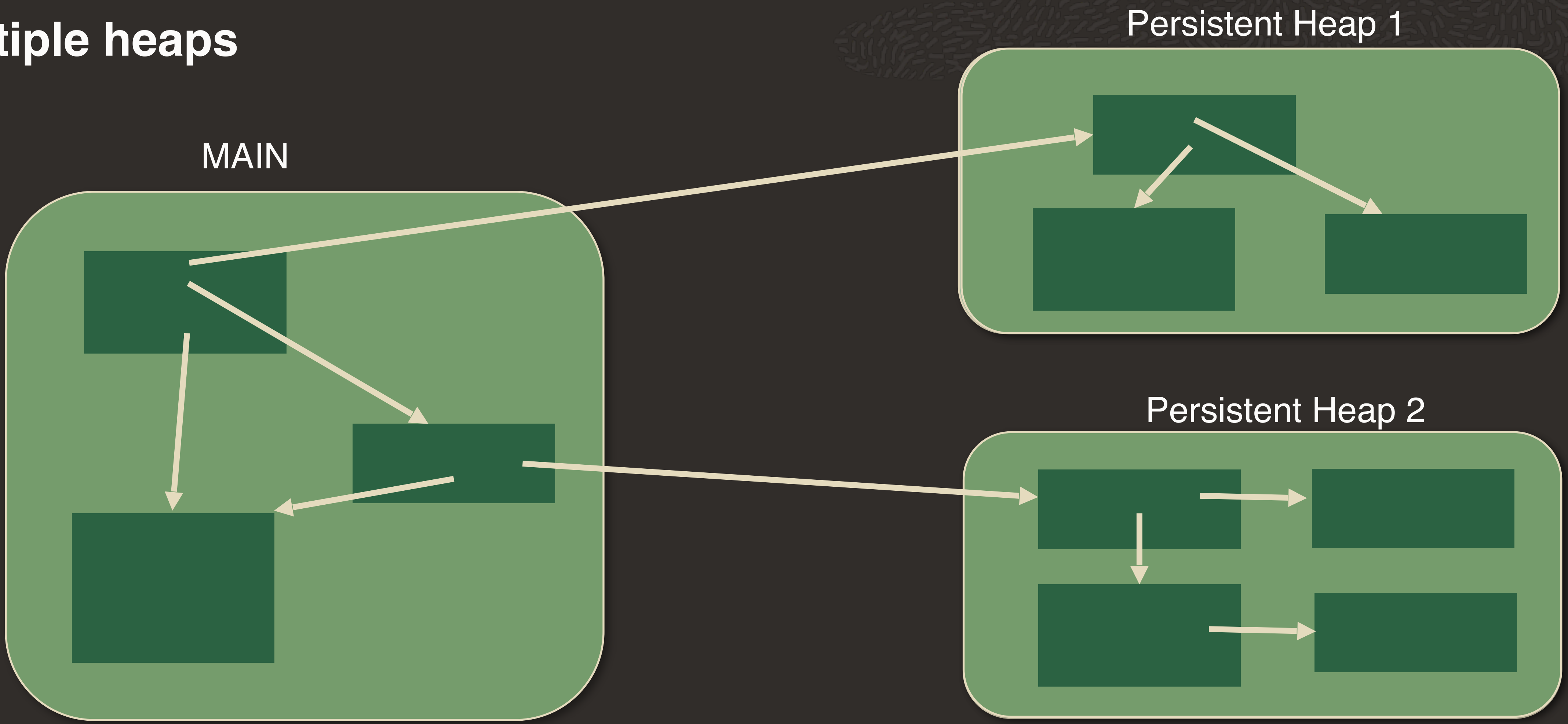
Multiple heaps



Multiple heaps



Multiple heaps



Pseudocode sketch of a minimal example

```
… main(…)
{
        PersistentHeap<Integer> heap;
        File heapFile = …
        if (!heapFile.exists()) {
                heap = PersistentHeap.create(heapFile, …);
                heap.root = new Integer(0);
        } else {
                heap = PersistentHeap.attach(heapFile, …);
                heap.root++;
        }
        heap.checkpoint();
}
```

This is incorrect — correction to follow

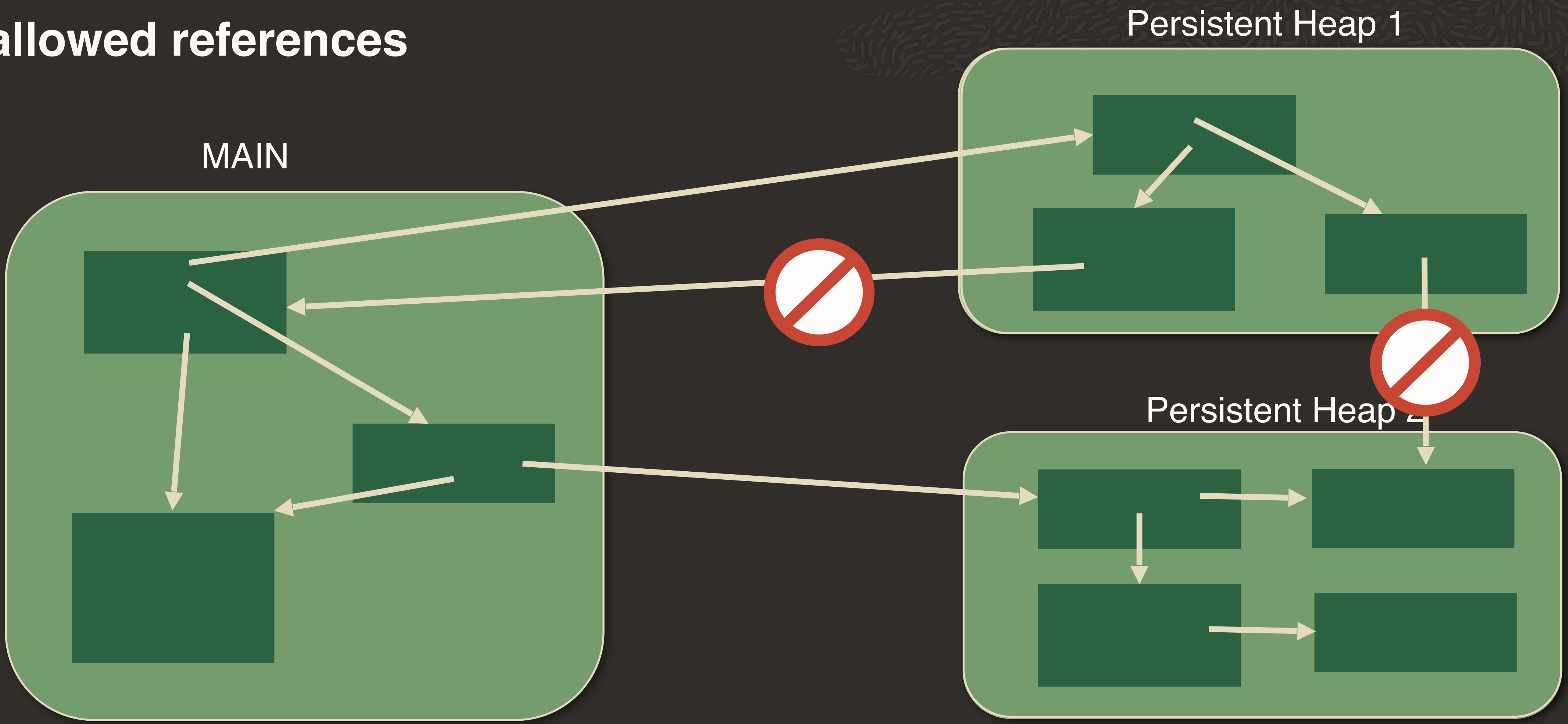
Inter-heap barrier

- A loadable heap must be *self-contained*.

  Add a write barrier to the JVM to prevent the creation of references to objects outside the heap;

  Throws an error when an invalid reference is being created
- Objects in the volatile heap can refer to any object.

Disallowed references



The current allocation heap

In the previous example, for

```
    heap.root = new Integer(0);
```

we must ensure the new `Integer` is allocated inside `heap`.

Add the notion of the current *thread-local allocation heap*, to be used when objects are created (with `new`). Then,

```
  heap.run(lambda)
```

runs the *`lambda`* with `heap` used for allocations.

Pseudocode sketch of a minimal example

```
… main(…)
{
        PersistentHeap<Integer> heap;
        File heapFile = …
        if (!heapFile.exists()) {
                heap = PersistentHeap.create(heapFile, …);
                heap.run(() -> heap.root = new Integer(0));
        } else {
                heap = PersistentHeap.attach(heapFile, Integer.class, …);
                heap.run(() -> heap.root++);
        }
        heap.checkpoint();
}
```

Exception handling omitted for brevity

Restrictions

- Threads must reside in `Heap.MAIN`
  - Java threads could be persisted with some implementation effort, but what about frames from foreign code?
  - `java.lang.Thread` is annotated as `@MAIN`, which forces all allocation to be in `Heap.MAIN`.
- Class objects must reside in `MAIN`
  - And by implication, static variables.
  - Objects in a persistent heap are “linked” to the corresponding class when the heap is attached.
  - Classes of objects in a persistent heap must be the same as those in the application.

Inter-heap references and reconnection

A variable annotated `@InterHeap` is interpreted in persistent heaps to:

1. Allow inter-heap references
2. Such references are nulled when a heap is attached.

A new interface, `Reconnectable`, enables objects to register with their enclosing heap so that when the heap is attached the object's `reconnect()` method is invoked. This can be used to re-initialize inter-heap references.

Class `HeapProxy<T>` implements `Reconnectable` and provides a single inter-heap variable of type T. Subclasses implement `reconnect()` to re-initialize the variable.

More elaborate data "linking" mechanisms can be built on top of this facility.

Statics

Some rewriting is necessary to deal with statics being in `Heap.MAIN`, e.g., referencing a global singleton which is needed in a persistent heap. Some approaches:

- Bypass.
  Example: the popular value cache in `java.lang.Integer`. I use this only for allocation in `MAIN`.
- Use reconnection.
  Example: Enums contain a static array of enum values. `EnumProxy<E>` wraps an enum E and holds on to the enum's int value to allow reconnection. (This transformation could be done mechanically.)
- Make a heap-local copy. The class `HeapLocal` (modeled after `ThreadLocal`) maintains a table of objects in each heap, indexed by the `HeapLocal` instance.
  Example: the `BASE_HEADER` sentinel in `java.util.ConcurrentSkipListMap`.

Making applications persistent

1. Introduce (create/attach) the persistent heap(s).
2. Identify the data structure(s) to be persisted, and which heap they belong in

- Ensure that they are self-contained, or where not, reconnectable – and make changes (inter-heap refs, reconnectables, or re-organize)
- Find statics which reference objects to be persisted and change (HeapLocal)
- Deal with enums
- Wrap code with `heap.run()` to ensure allocations happen in the correct heap

Find (or create) points of consistency where checkpoints can happen

All the code in the JDK will need this — no small task. I've been converting classes as I encounter problems; so far, nothing terribly challenging has emerged.

Persistent heaps are not just for NVRAM

An alternative implementation reads a saved heap into volatile RAM; a checkpoint writes the heap to a file. Checkpoints are slower than using NVRAM, but DRAM is ubiquitous.

(A more sophisticated approach is to write out just the modified parts during a checkpoint.)

This may be useful for fast startup in the cloud, to avoid keeping an app continuously provisioned.
GraalVM Native Image allows fast startup of code — this would allow fast startup of data.

The in-progress prototype

- Based on GraalVM Native Image
- The first version uses simpler implementation techniques:
  - The heap is read into DRAM; don't need NVRAM
  - Objects are relocated when the heap is attached, including "linking" to the class in the image
  - Relatively efficient inter-heap barrier (a few instructions)

Should be ready for experimental use later this year; will be seeking experimenters to evaluate usability and utility of the many-heap model.

Future versions will refine the model and API and also could:

- Improve performance and scaling
- Support NVRAM
- Allow for class evolution
- Make persistence available to other GraalVM languages.

DEMO

To conclude…

New model for persistence based on multiple heaps, with Java API

Prototype implementation to allow for experimentation and evaluation

For more info: Announcements, blog posts, etc., from my twitter account (@mwolczko)

Thank You

Questions? Comments?

Acknowledgements

The many-heaps model was inspired by Bill Bridge’s NVM-Direct.

Christian Wimmer, Peter Kessler, Eric Sedlar and Paul Wögerer for helpful discussions and/or SubstrateVM help.

Our mission is to help people
see data in new ways, discover insights,
unlock endless possibilities.

Backup slides

Cloud deployment of persistent heaps

Fast application startup opens up new economies for cloud apps

- Keeping an app continuously provisioned just for occasional short queries wastes resources
- Standard JVMs start up too slowly; add too much latency
  Graal Native Image loads code fast – but what about data?
  The image heap in NI cannot be updated without a rebuild — minutes of compute.

Need to be able to load pre-populated parts of the heap quickly and efficiently

- Deserialization is much too slow and inefficient
- Small updates also necessitate fast, incremental saves

Persistent programming: natively and seamlessly allow objects to continue existing after the program terminates.

An alternative: Nested transactions

- I discarded persistence-by-reachability because of heap corruption, but if the heap supports logging and recovery we could recover to an earlier, uncorrupted state.*
- To identify consistent earlier states, the programmer has to indicate points of consistency.
- This typically leads to a scheme of nested transactions, and a programming model that is a radical departure from existing models; legacy code typically must be rewritten.
- This *may* be the way of the long-term future, but adoption in the near term might be a challenge.
- For good examples, see the work of Hosking, Moss, et al., or NVM-Direct.

* But: memory leaks? Keep the log for all time?

Partitioned heaps in NVM-Direct

- NVM-Direct is an extension of C for persistence in NVRAM, by Bill Bridge (Oracle) https://github.com/oracle/nvm-direct
- **Explicit separation of volatile memory from persistent regions**
- Adds nested transactions, associated locks and compensation code
- A transaction is associated with a region
- Pointers to NVM, assignments to NVM: use new syntax
  - Compiler generates flushes and undo records
- Structs allocated in NVM must be declared persistent
  - Compiler and library use self-relative pointers

Java and NVM

- Millions of programmers
- Billions of lines of code
- My goal: large-scale near-term adoption
- Must limit disruption to existing code and practices

Other approaches:

- Intel Persistent Collections for Java:
- Off-heap persistence, Persistent types
- Espresso ():
    - persistent new

Can’t use existing Java code in a persistent context