Eventrons: A Safe Programming Construct for High-Frequency Hard Real-Time Applications

Daniel Spoonhower

Carnegie Mellon University

Joint work with
Joshua Auerbach, David F. Bacon,
Perry Cheng, David Grove

IBM Research

PLDI – 13 June 2006

Motivating Application

End-to-end audio in Java

- *e.g.*, MIDI events, composition, audio driver
- tasks w/ wide range of real-time requirements

Today, focus on two tasks:

- tone generation – moderate real-time demands
- audio driver – hard real-time demands

Existing Solutions

Real-time garbage collection

- ✔ retain standard programming model
- ✘ best latencies around $\approx$ 0.5 ms [Bacon+ 2003]

Real-Time Specification for Java

(*i.e.*, manual memory management)

- ✔ better latencies than RTGC
- ✘ change in language semantics
- ✘ difficult to communicate w/ non-RT threads

Our Goals

Support for integrated real-time applications

- garbage collection for low-frequency tasks
- high-frequency tasks above 1 KHz
- *shared* data structures

Push latency as low as we can go

- 100 μs? 10 μs?

Introducing... Eventrons!

Arbitrarily preempt any other thread

- including the garbage collector

Can share heap with other threads

- real-time and non-real-time threads

No language extensions

- restricted to a strict subset of the language

"How" – In a Nutshell

Avoid synchronization with collector

- not by partitioning memory...
- ...but by restricting memory operations
- some GC support as well

Terminology: Eventron =

- code (call graph from "`void run()`")
- data structure (reachable objects)

Avoiding Synchronization (1/3)

No allocation in Eventron code

- no need to ask the GC for more space
- proscribe `new`, `newarray`, etc. bytecodes

Significant restriction, but not unreasonable

- for device drivers, sensor processing, etc.

Data structures allocated during *initialization* phase

- in contrast with *execution* phase

Avoiding Synchronization (2/3)

No pointer mutation in Eventron data structures

- isolate Eventron objects *from collector*
- Eventrons cannot change object reachability
- other threads cannot leak objects to Eventron

Reference fields in Eventron must be immutable

- use Java's `final` keyword for fields*
- restricts both Eventron and non-Eventron code

Avoiding Synchronization (3/3)

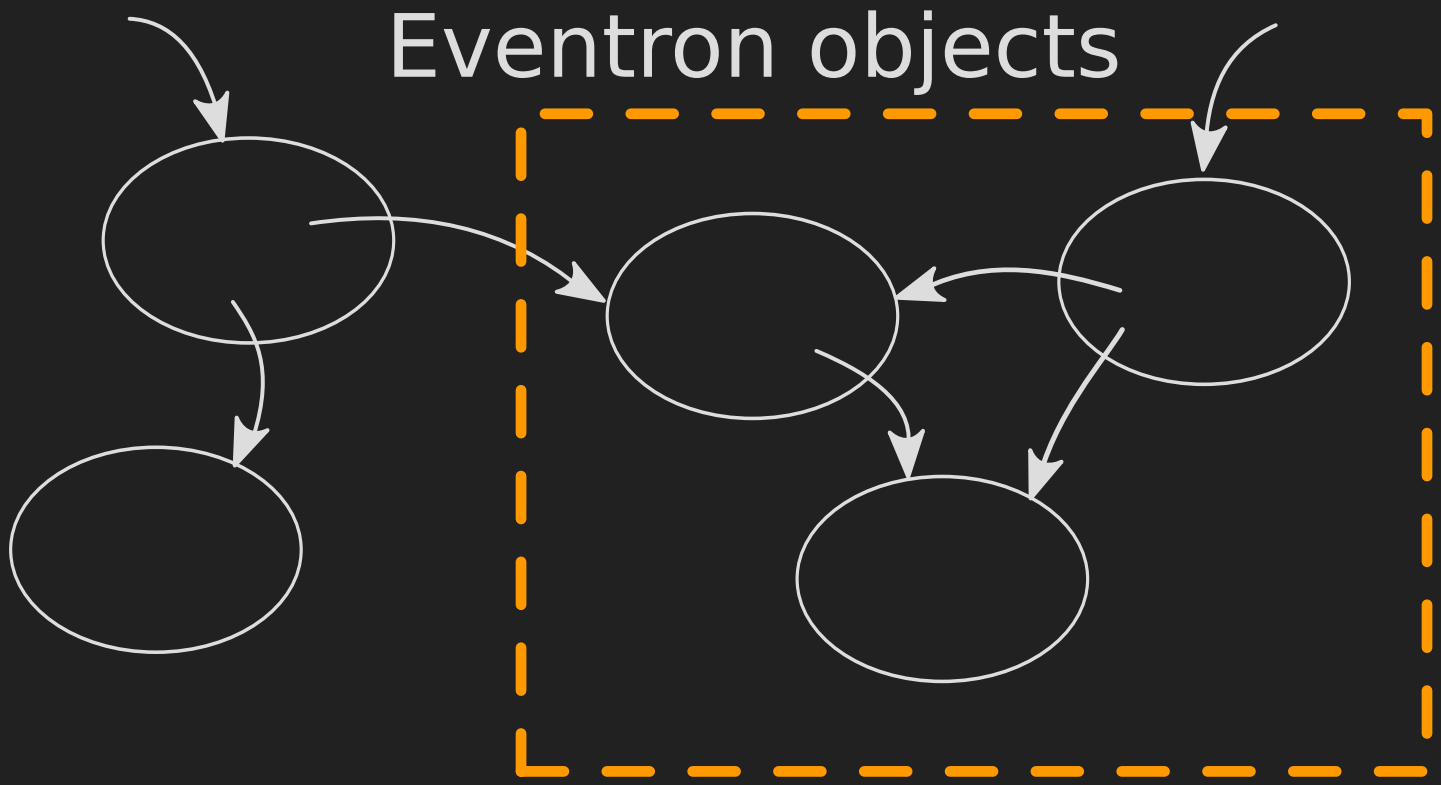
Use a garbage collector with support for pinning

- don't move objects used by Eventron
- $\Rightarrow$ it can't see objects in inconsistent state

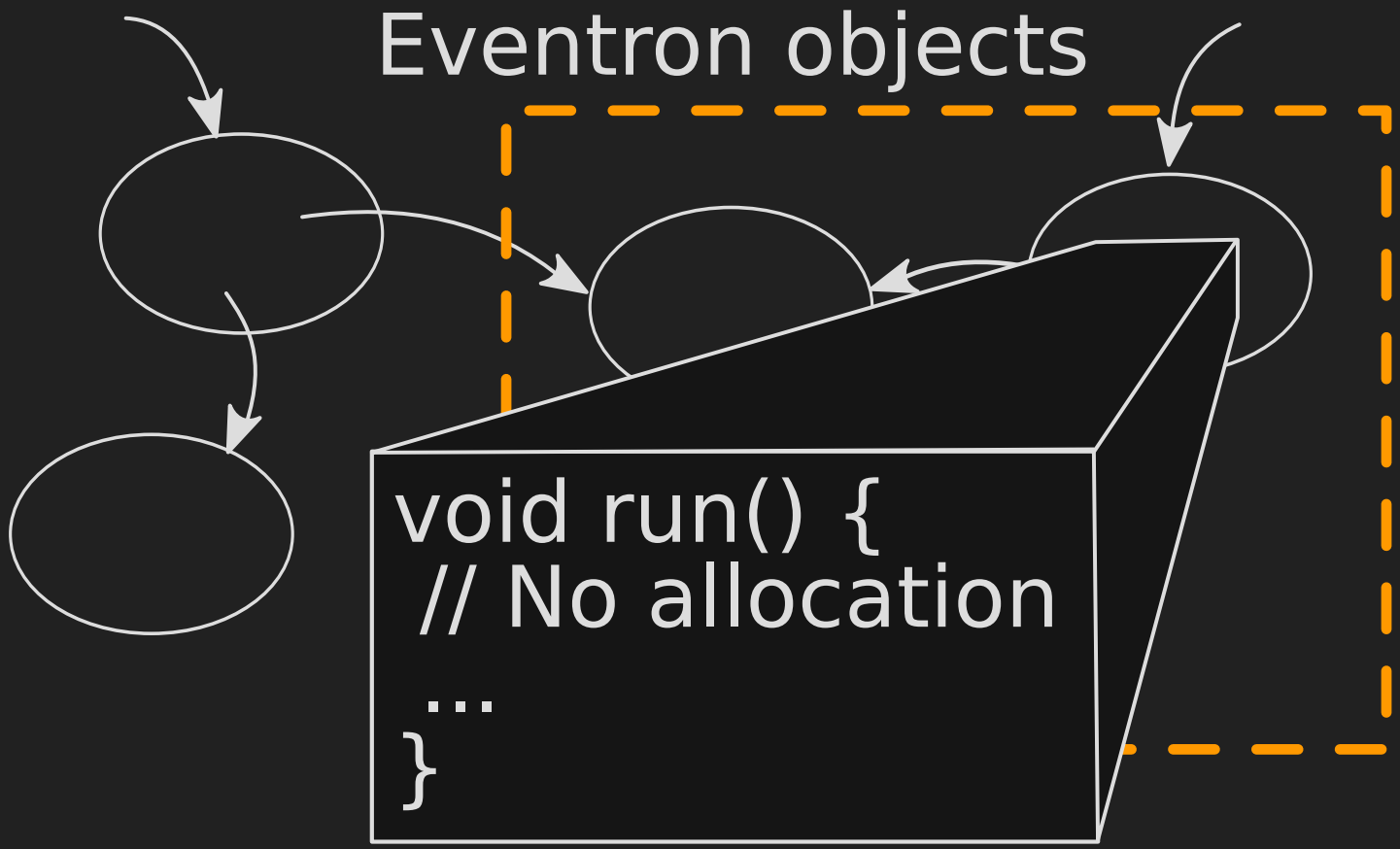
Contrast with RTSJ's `ImmortalMemory` class...

- *not* a memory leak
- objects in Eventron may be reclaimed after termination

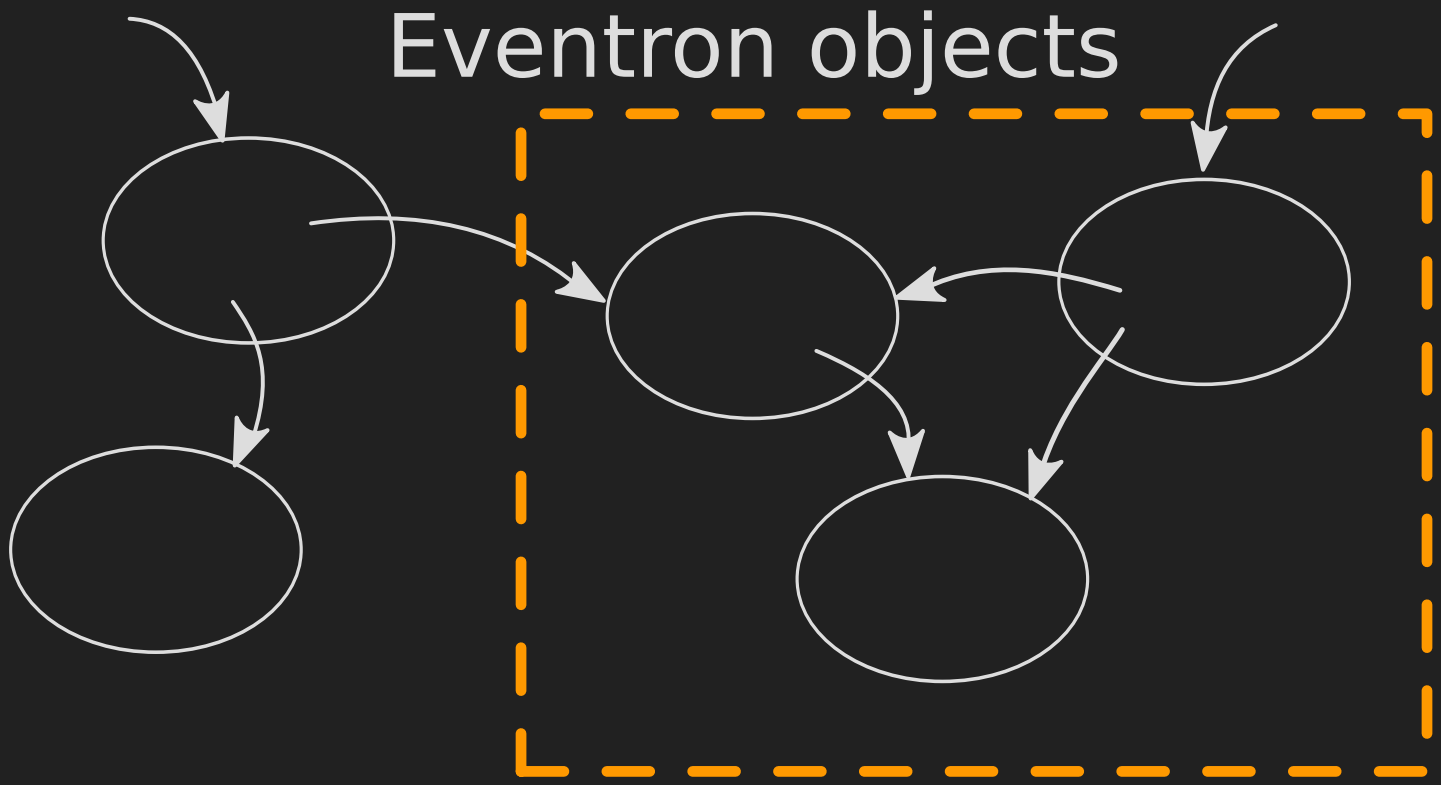
Eventrons Illustrated



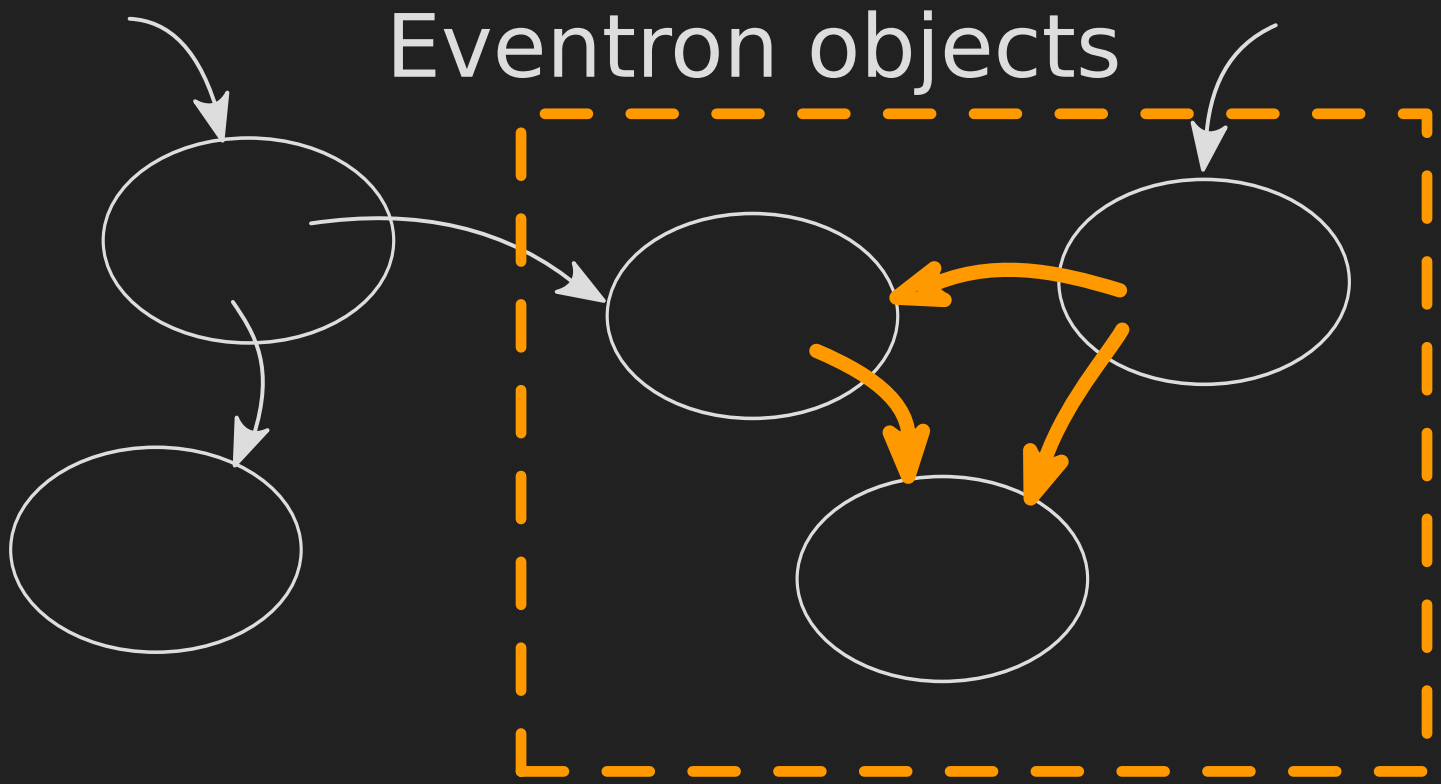
Eventrons Illustrated



Eventrons Illustrated

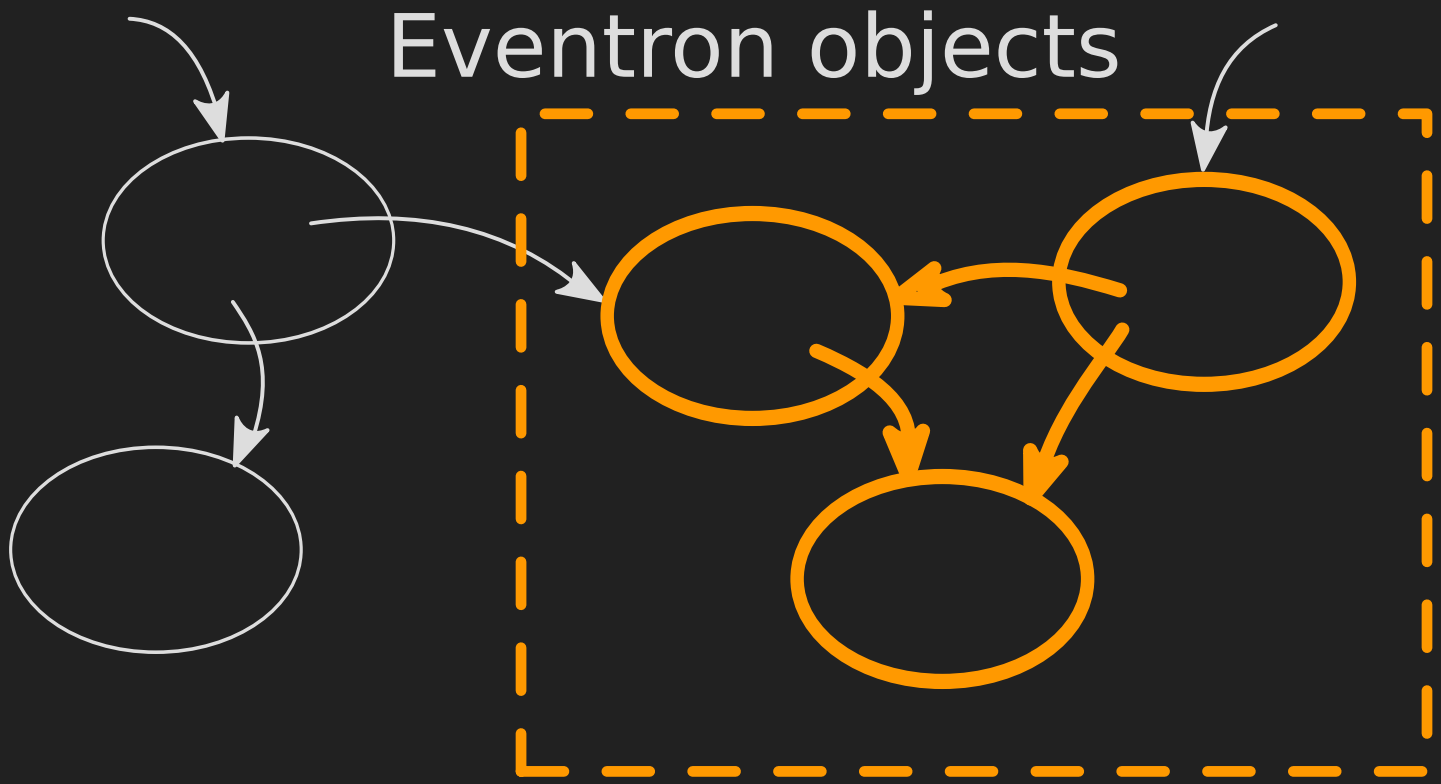


Eventrons Illustrated



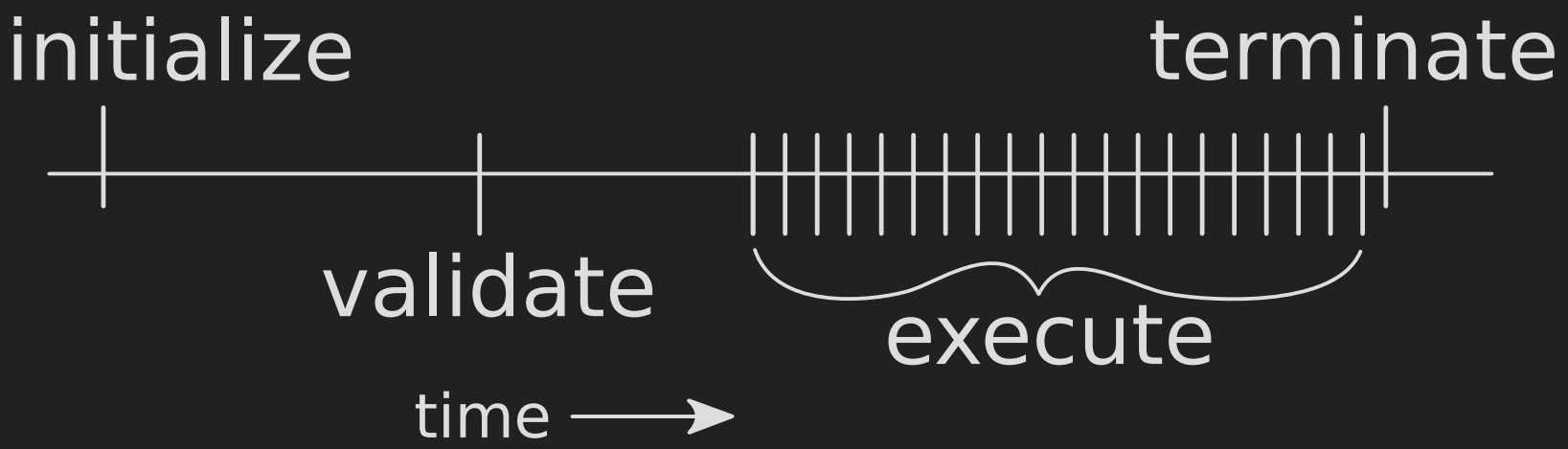
References must be **`final`**

Eventrons Illustrated

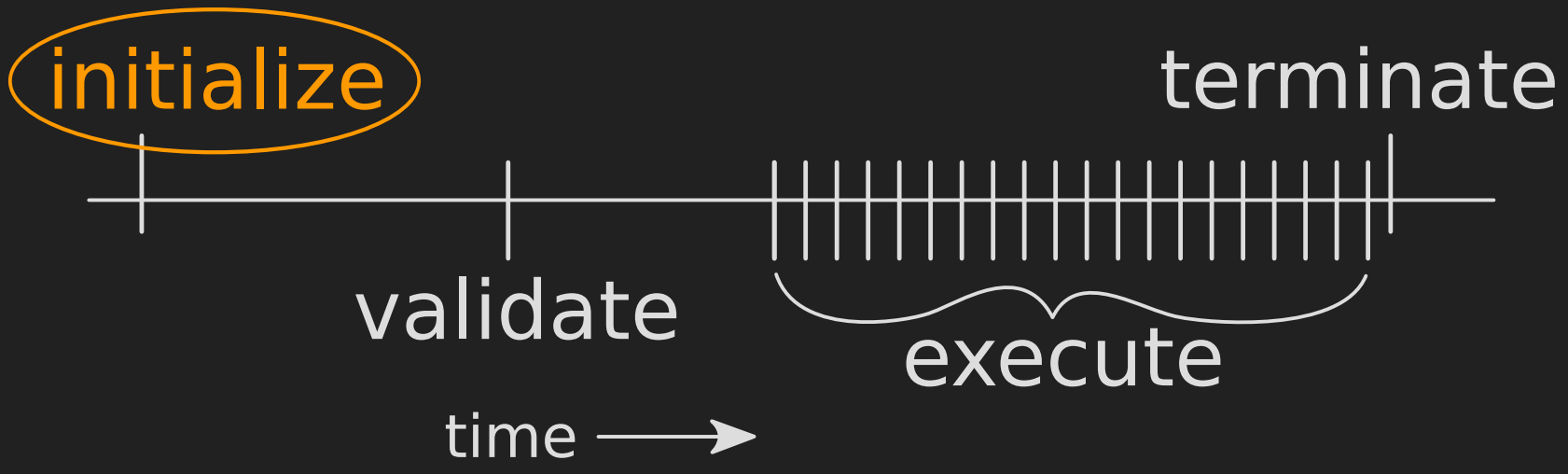


Objects are pinned

Life and Times of an Eventron



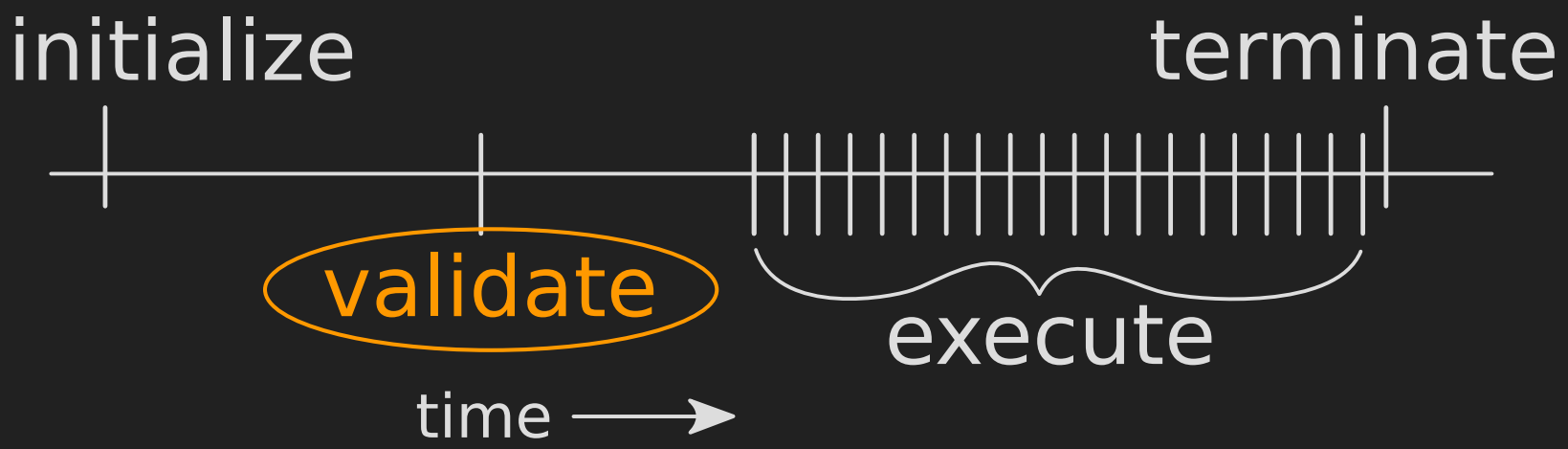
Life and Times of an Eventron



Programmer describes code, creates a data structure

- instance of `Runnable`
- ordinary `new` operations

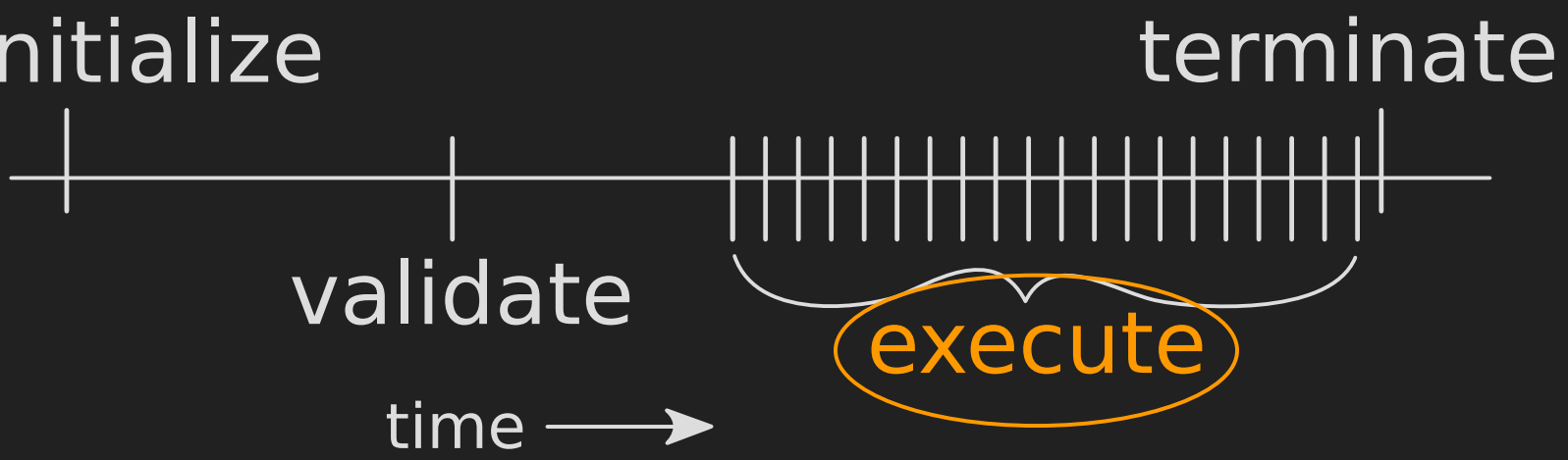
Life and Times of an Eventron



Given entry point and data structure, determine if Eventron is safe to run

- construct call graph, check for allocation
- inspect fields looking for mutable references

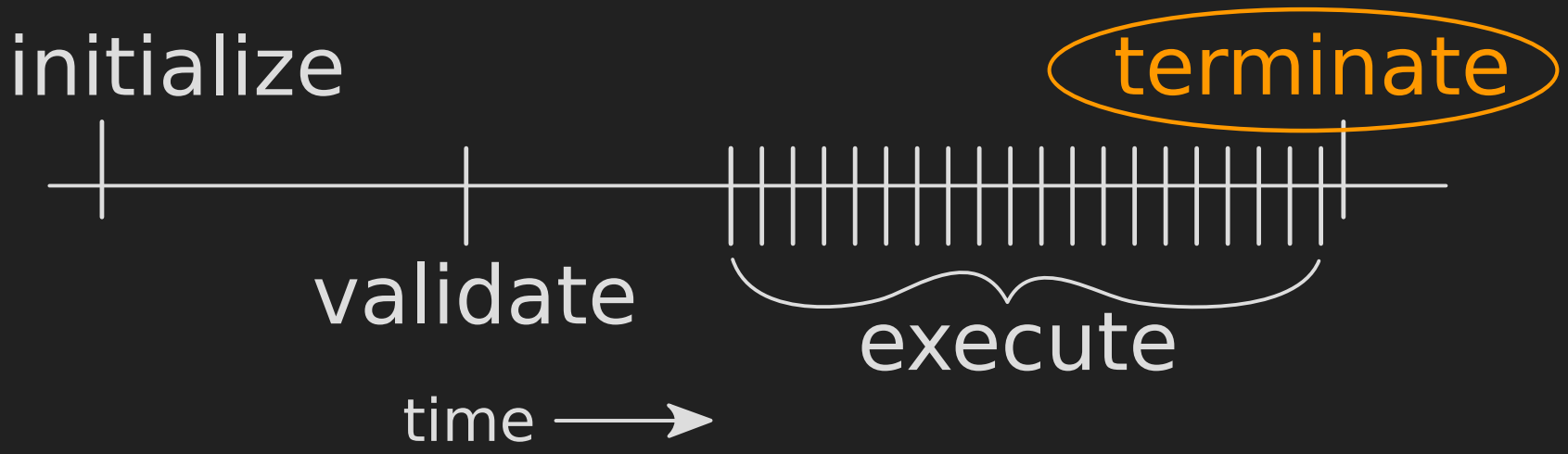
Life and Times of an Eventron



Once validated, Eventrons run over and over. . .

- on periodic schedule
- in response to interrupts

Life and Times of an Eventron



Eventrons may be terminated any time.

- during execution; by self or other threads
- safe b/c they hold no shared run-time resources

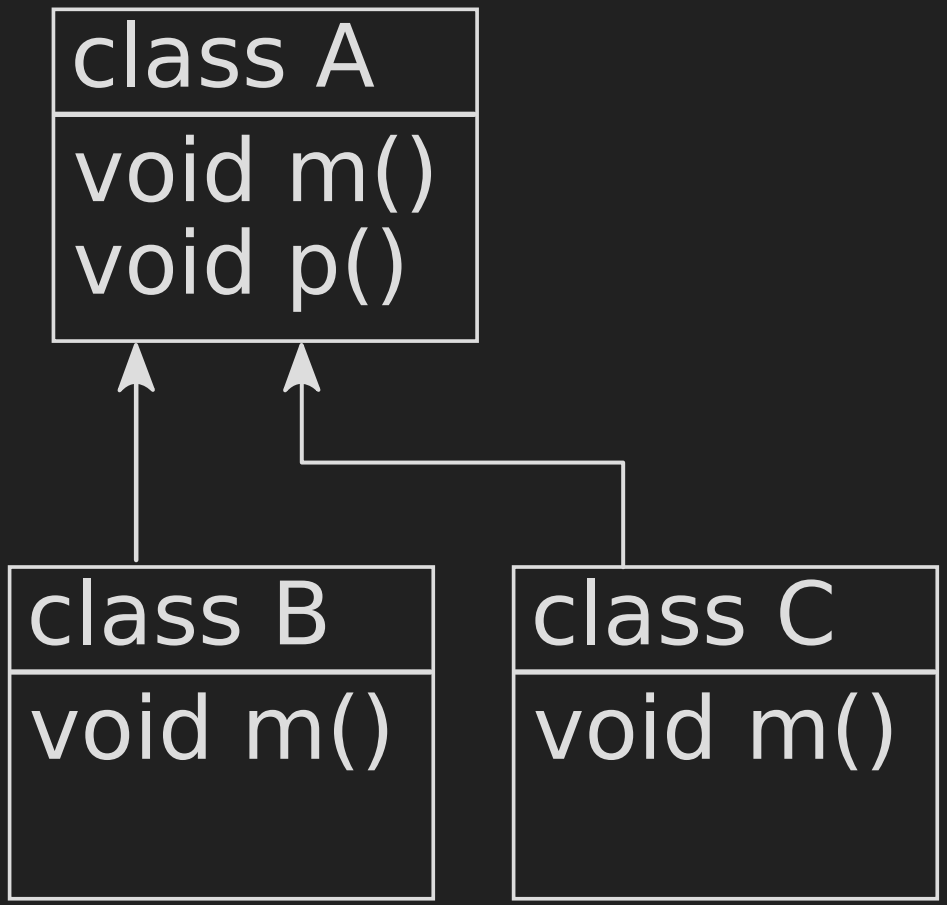
Closer Look at Validation

Ensure that Eventron-reachable heap remains fixed

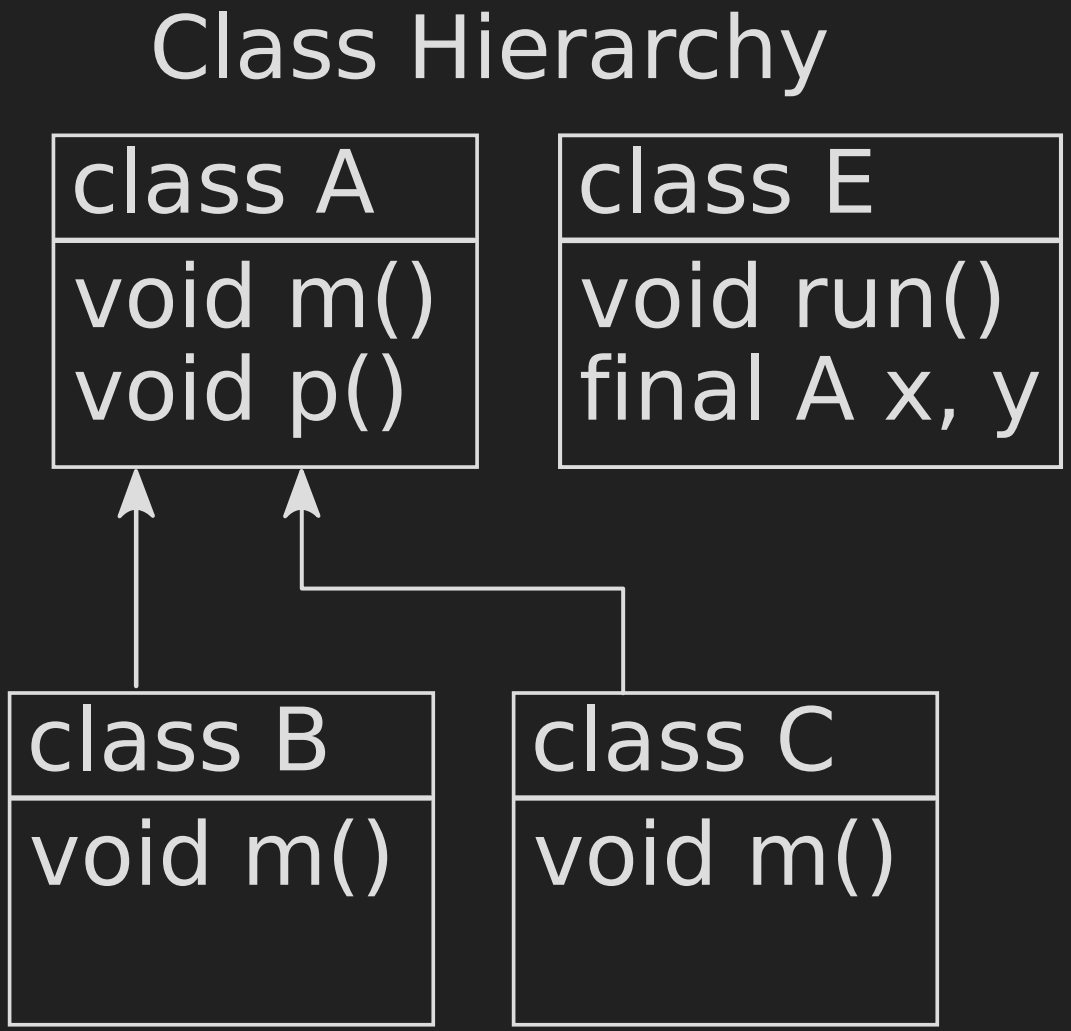
1. Use a data-sensitive call-graph analysis
   - exploits run-time knowledge of data structure
2. Inspect resulting code for offending operations

Data-Sensitive Analysis, Illustrated

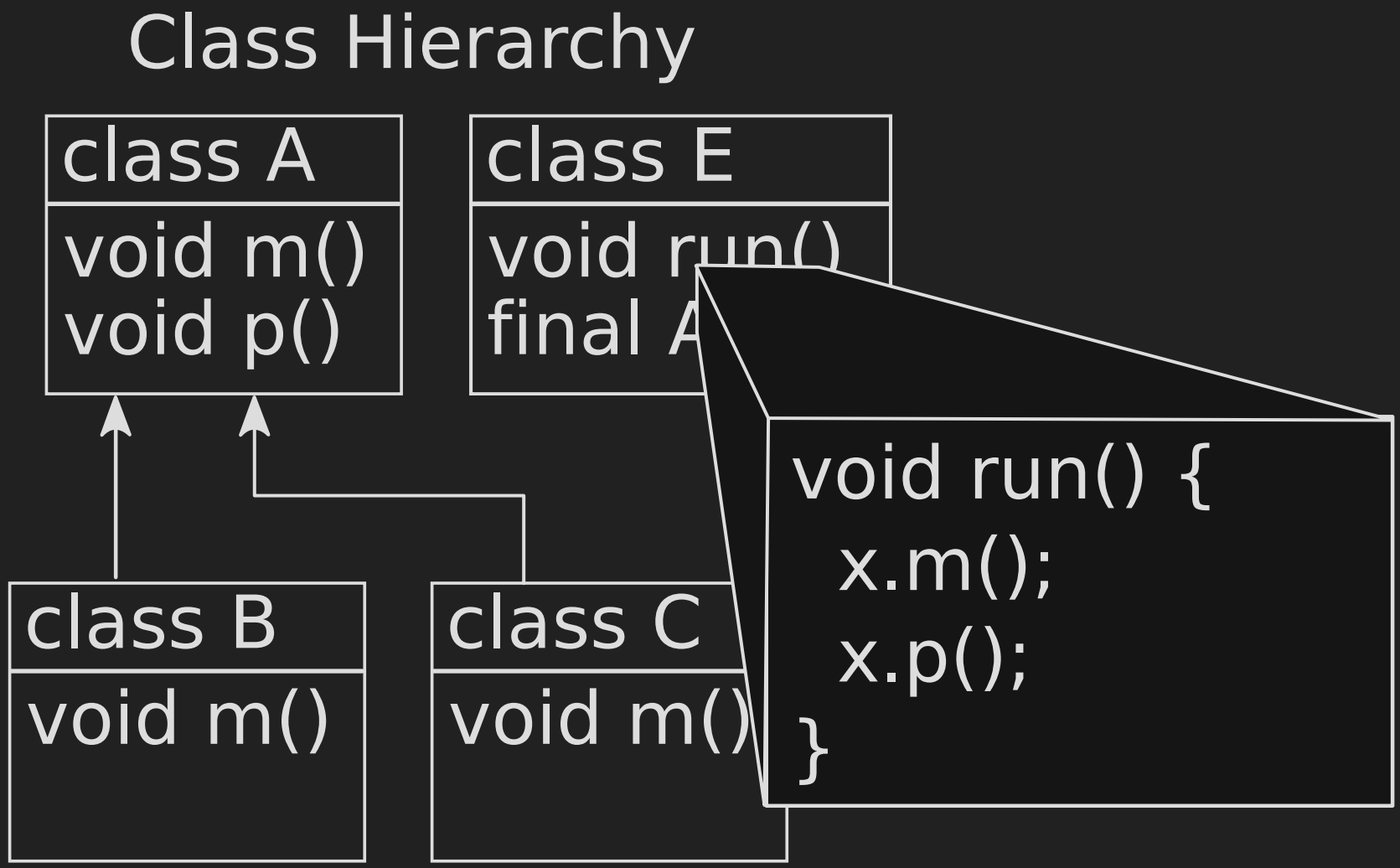
Class Hierarchy



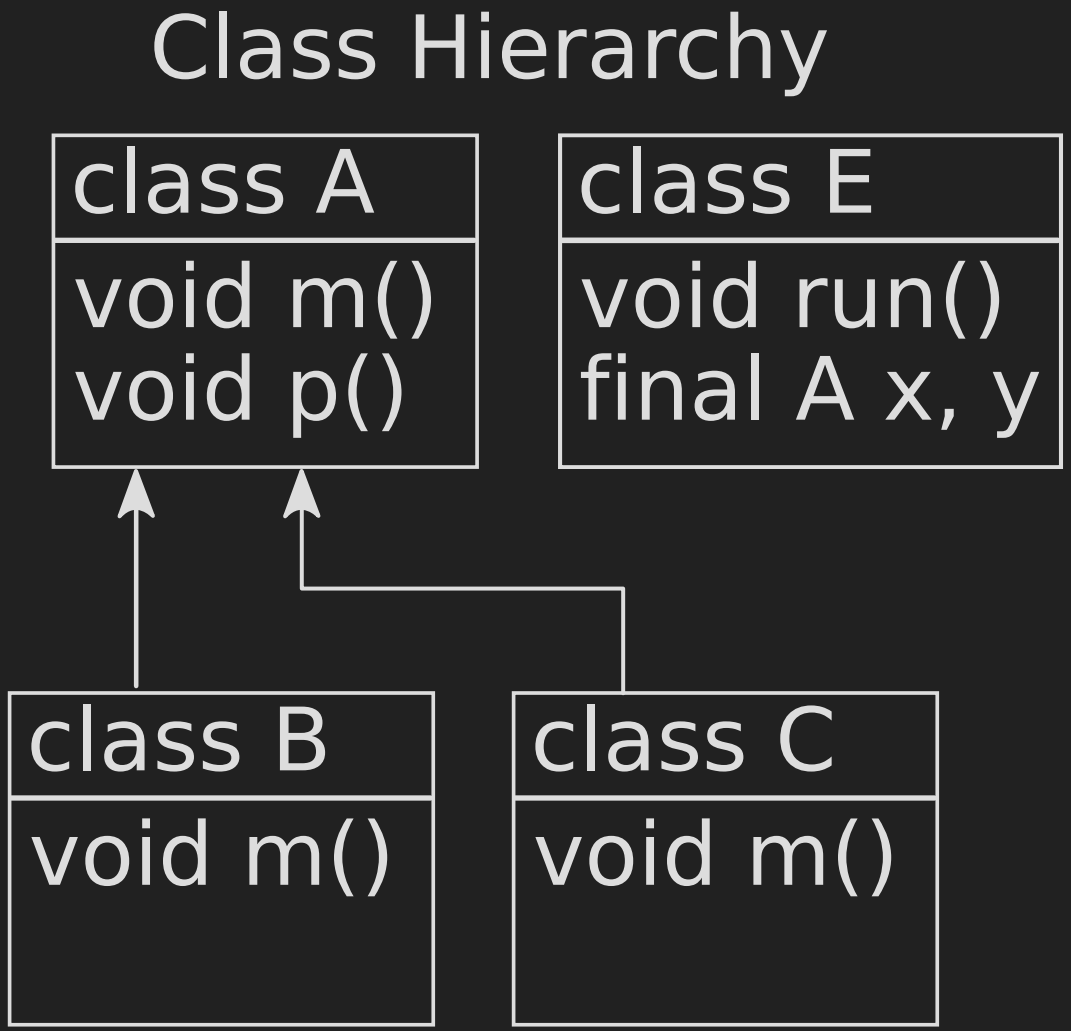
Data-Sensitive Analysis, Illustrated



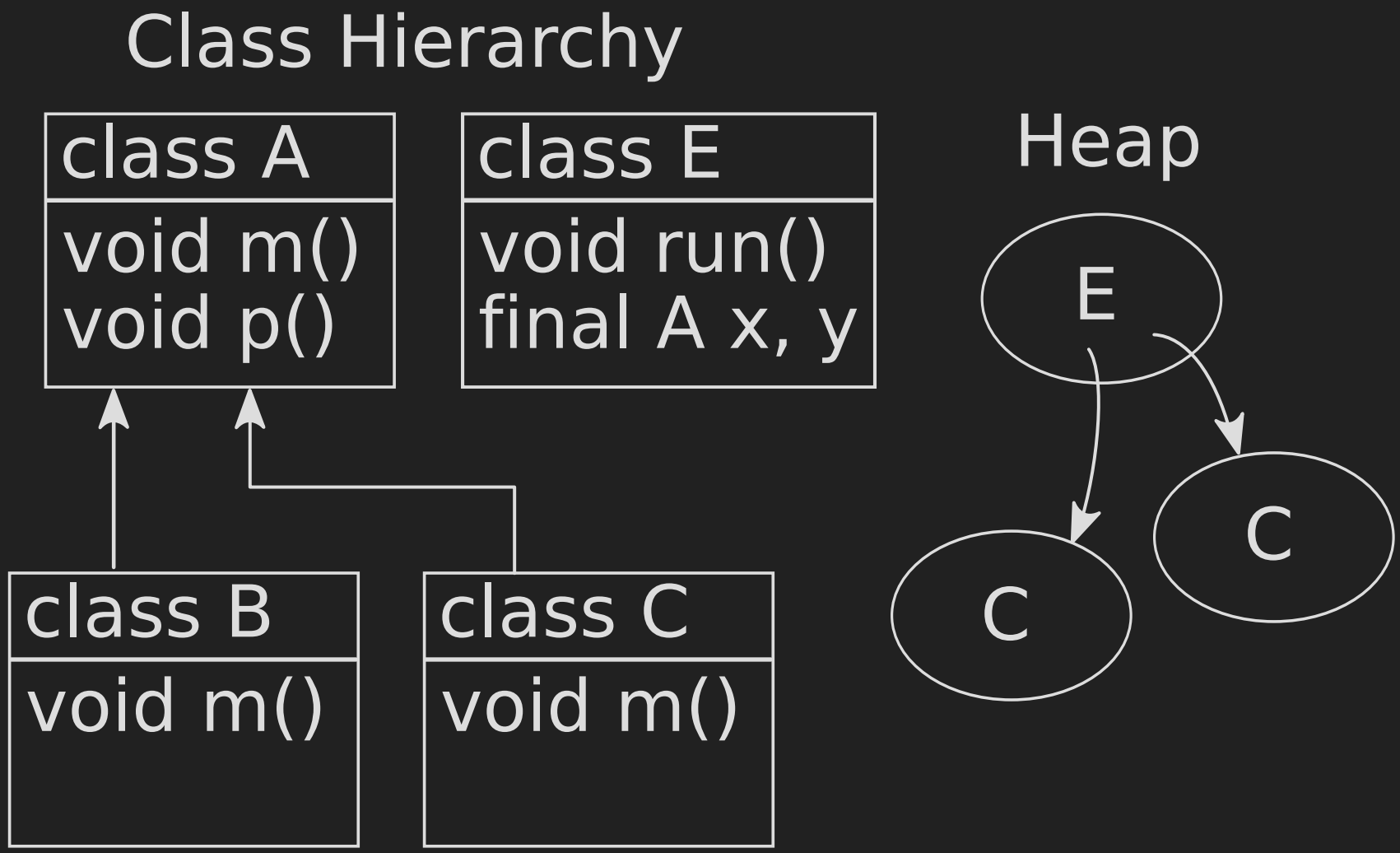
Data-Sensitive Analysis, Illustrated



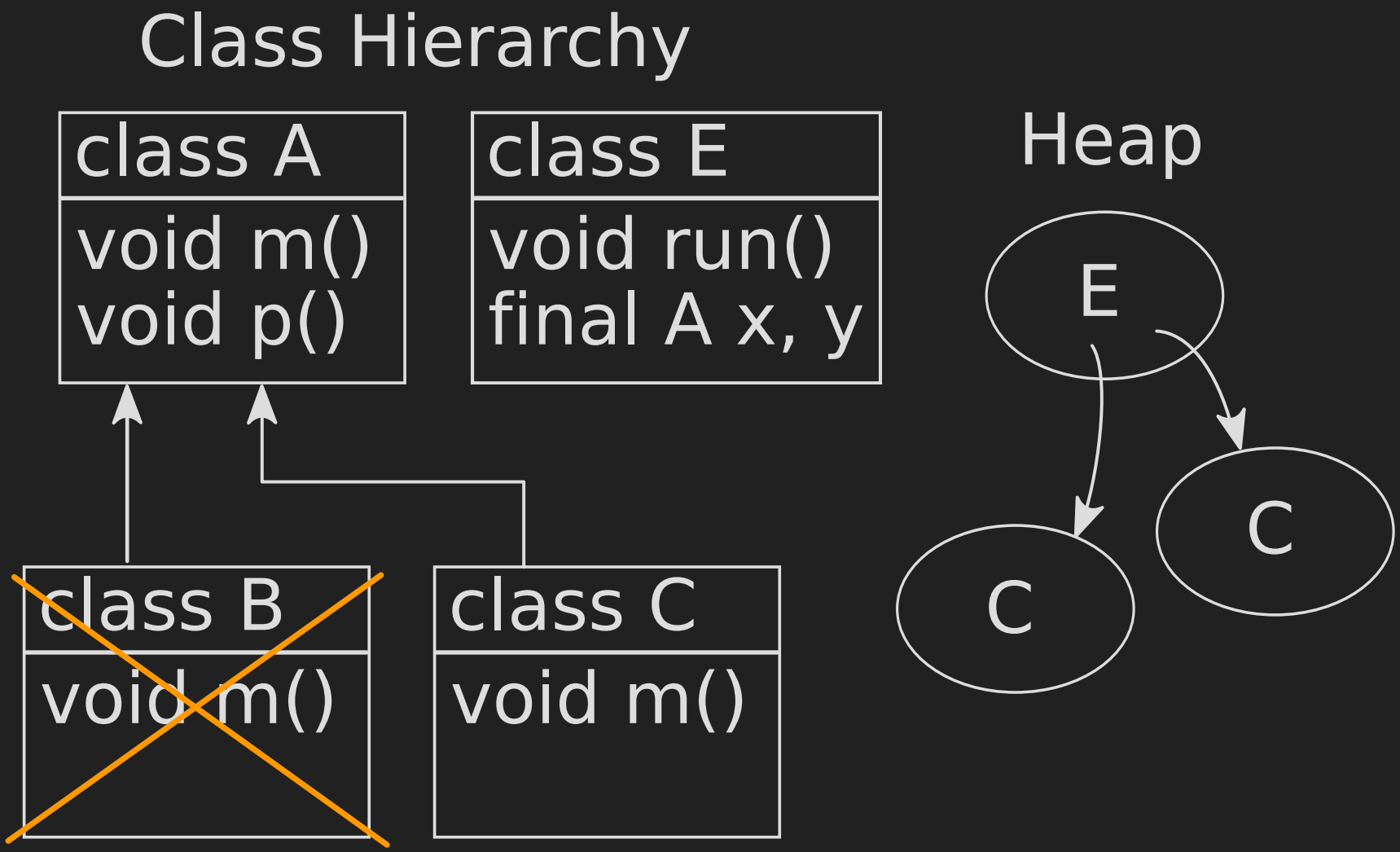
Data-Sensitive Analysis, Illustrated



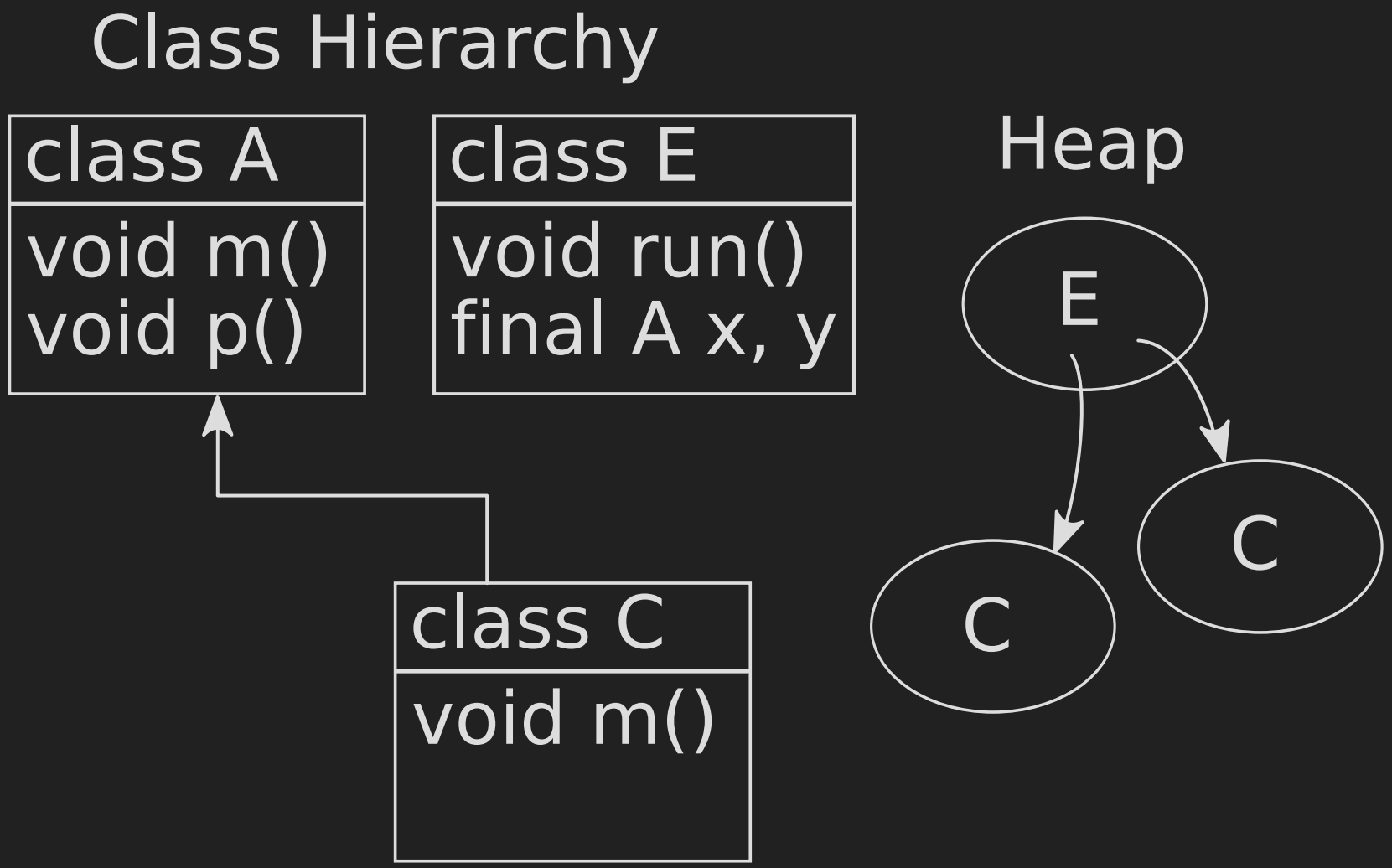
Data-Sensitive Analysis, Illustrated



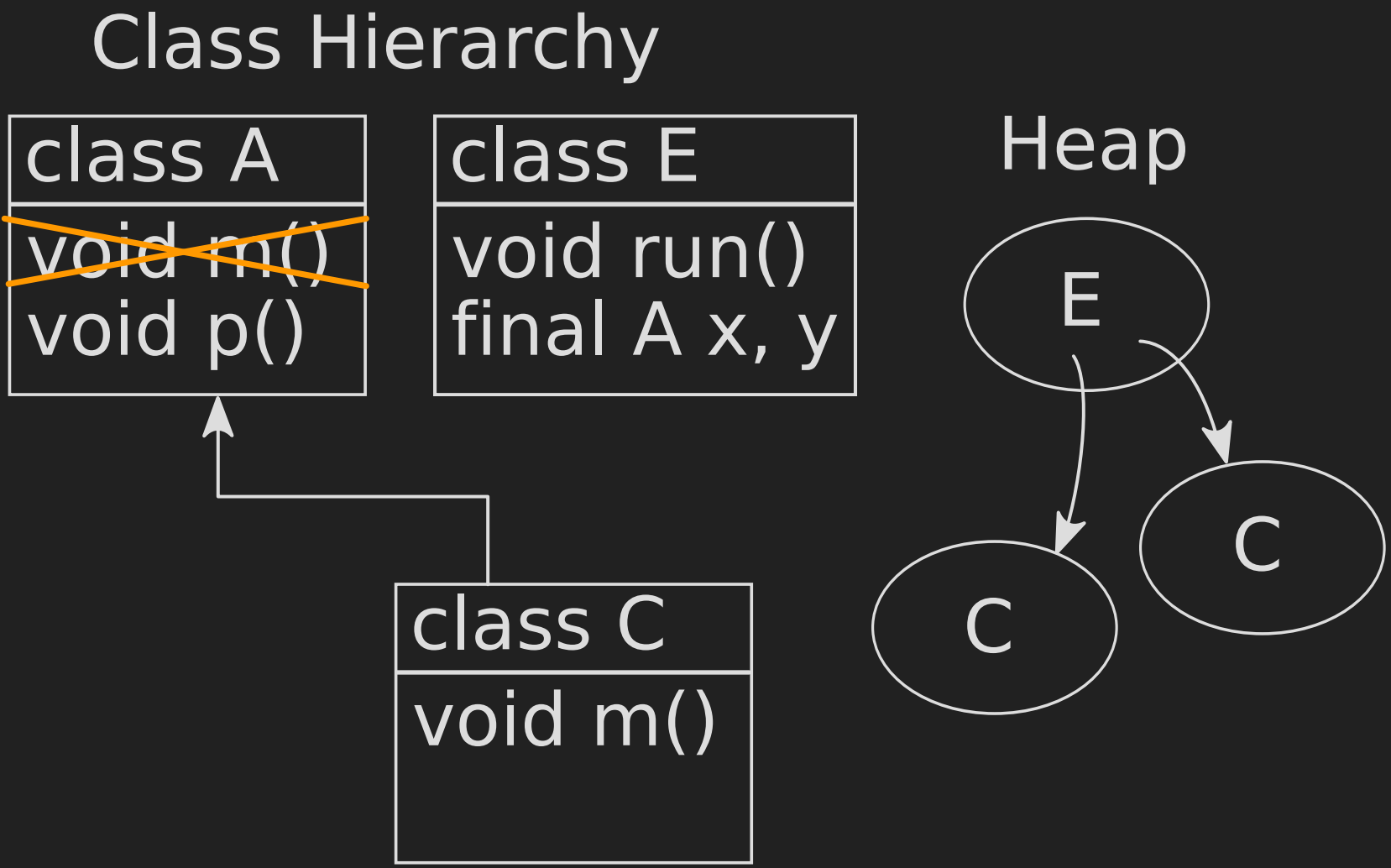
Data-Sensitive Analysis, Illustrated



Data-Sensitive Analysis, Illustrated



Data-Sensitive Analysis, Illustrated



Data-Sensitive Analysis

Doing run-time analysis yields additional precision

- alternative view: using method tables to enumerate methods in call graph

Requires that reference fields in Eventron objects are immutable

- ensures that referents present during analysis will persist

Empirical Results

Can Eventrons meet their real-time deadlines?

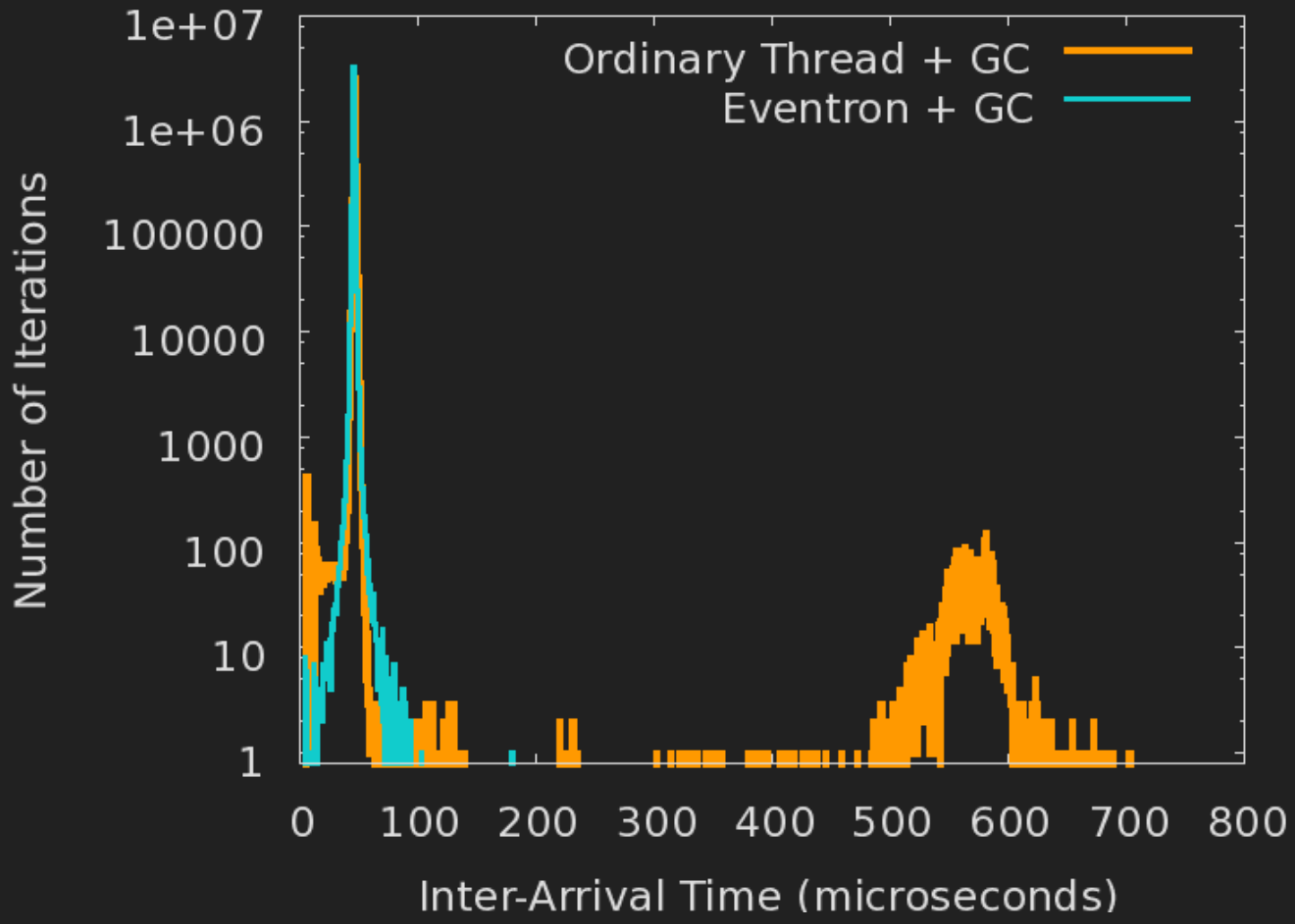
Measuring inter-arrival time for a periodic task

- frequency $=$ 22.05 KHz ($\approx$ 45 μs)
  (motivated by audio example)

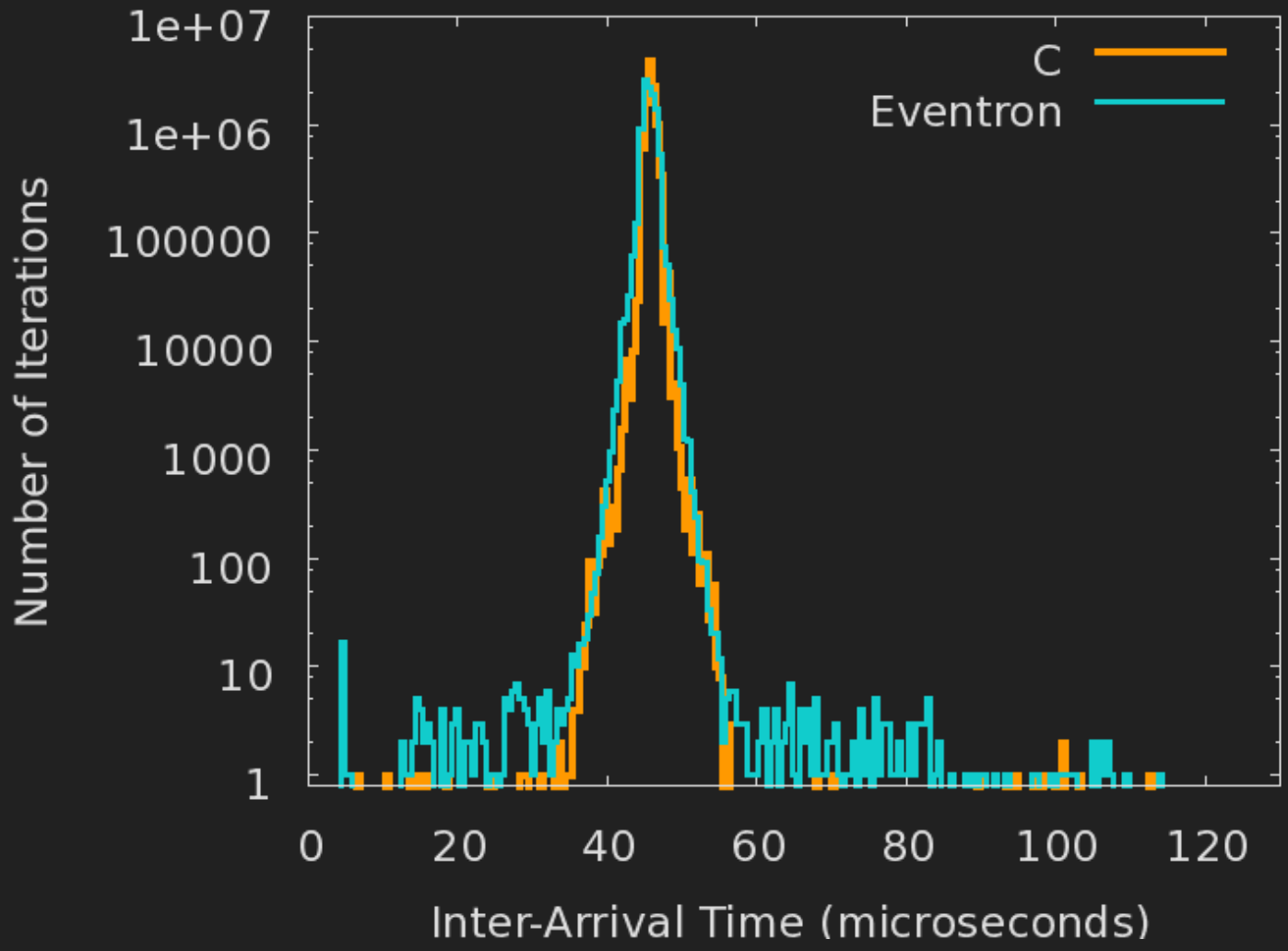
Experimental setup:

- 2 $\times$ Pentium 4 (2.4 GHz); 2 GB RAM
- Linux 2.6.14 w/ HRT & priority inheritance
- J9 Java virtual machine

Empirical Results, Comparing to RTGC



Empirical Results, Comparing to C



More Details in Paper

Ensuring the `final` references don't change

- despite concurrency, reflection, JNI

Run-time support

- object pinning in GC
- inspection of loaded bytecode

Programming with Eventrons

- code for music generation example
- *allocation-*, and *pointer-mutation-free* queues

Related Work

Real-time garbage collection

- Bacon, Cheng, Rajan [POPL 2003]

Programming with the RTSJ

- Bollella *et al.* [OOPSLA 2003]
- Pizlo *et al.* [OORTDC 2004]

Static analysis for RTSJ

- Boyapati *et al.* [OOPSLA 2003]
- Zhao, Noble, Vitek [RTS 2003]

Summary

Eventrons avoid synchronization with GC

- . . . therefore, can preempt it at *any* time

Restricted programming model, however. . .

- strict subset of current semantics
- can share objects with (non-)real-time threads

Data-sensitive analysis is very precise in practice

- use run-time data structures, not annotations
- immutable fields ensure soundness