Ownership in Object-Oriented Languages

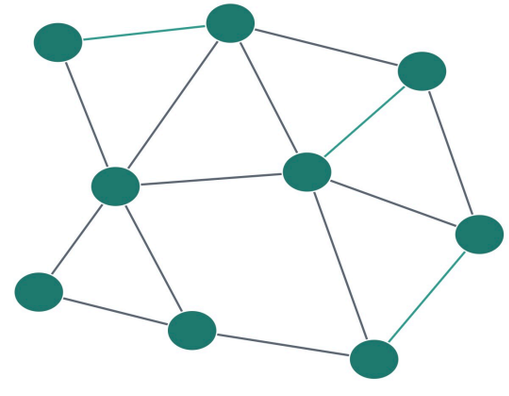
Marcus Denker, Inria Evref

The Object Graph is a Mess

Marcus Denker, Inria Evref

The Object graph

- Program at runtime: Objects and the reference between them
- Objects are nodes. References are directed edges
- The whole image is one large directed graph



Outgoing edges

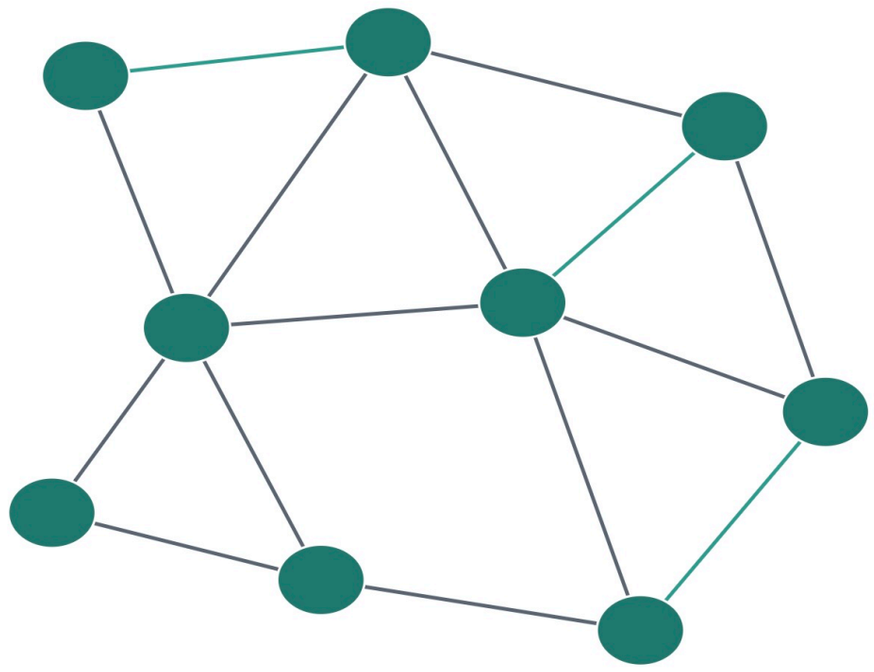
- **What the object points to**
  - Named — in the case of instance variable
  - Bounded — a fixed, countable set
  - Local — held inside the object
  - Reflective — instVarAt:, the slot protocol

Incoming Edges

- **What points to the object**
  - Anonymous — no record of who
  - Unbounded — any number, from anywhere
  - Global — added across the whole image
  - Invisible — no knowledge, no consent

It is a mess

- An object knows what it points to
- Has no idea what points to it



Missing: Reflective API

- No API to iterate over it
    - Implement it yourself (visitor pattern)
    - Use object level reflection (instVarAt:)
- No way to annotate nodes
- No way to annotate edges

The problem of why

- The “Outgoing edges” case looks nice at a first glance
- But: why have no idea *why* the edge exist
- Example: Some objects are private state of other objects

A trivial Program

```
Object subclass: #Order
    slots: { #items };
    package: 'Shop'.

Order >> initialize
    items := OrderedCollection new.

Order >> addItem: anItem
    items add: anItem

Order >> total
    ^ items inject: 0 into: [:sum :e | sum + e price]
```

items is private to Order

At runtime

- But at runtime, there is just a graph
- `items` is only protected by having no accessor
- We use patterns to avoid changing the collection
  - API to add/remove
  - No accessor
  - Or: accessor returning a copy

Known: Alias Problem

- As a problem to be solved this got attention with a Workshop 1992

- *“The Geneva Convention on the Treatment of Object Aliasing”* Hogg, Lea, Wills, de Champeaux, Holt

- Let’s look at some problems

Serialization: outgoing

- Serializers do not store single objects, but subgraphs
- We give a the Serializer a root object
- Follows all pointers from that object recursively
- To stop, we have to annotate which pointers *not* to follow
- Method on classSide (e.g. `#soilTransientInstVars`)

Serialization: incoming

- What do we do with incoming pointers?
- Serializers do not know about them!
- Destroys the “we can easily store the subgraph on Disk” idea
- In Practice:
  - make sure that there are none

ImageSegments

- Use GC infrastructure to find incoming
  - If not defined as known: object not part of segment
  - Allows to replace incoming pointers with proxies
  - Re-connect incoming on load

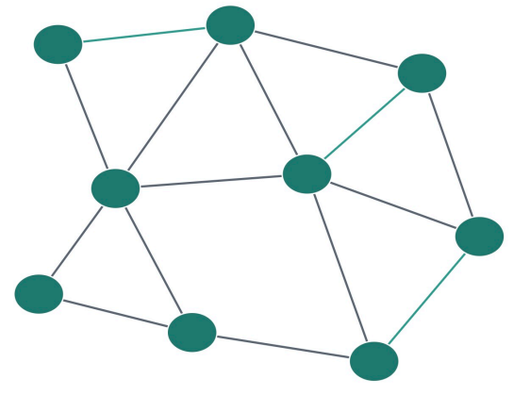
ImageSegments

- Adding a reference from the outside changes definition of what the segment contains

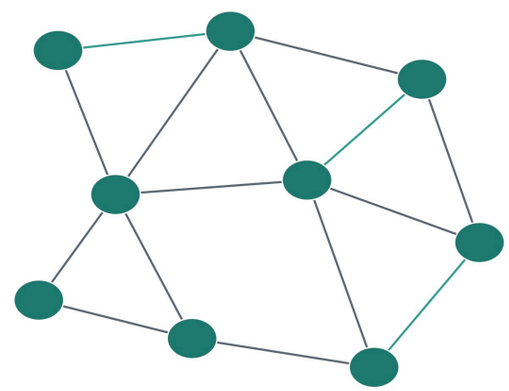
- Very brittle

- Full heap scan needed

The Object graph:
Unstructured mess



One idea: Ownership



Intuition

- Some objects are part of another object's representation
    - The items collection is part of the Order

A trivial Program

```
Object subclass: #Order
    slots: { #items };
    package: 'Shop'.

Order >> initialize
    items := OrderedCollection new.

Order >> addItem: anItem
    items add: anItem

Order >> total
    ^ items inject: 0 into: [:sum :e | sum + e price]
```

items is private to Order

Intuition

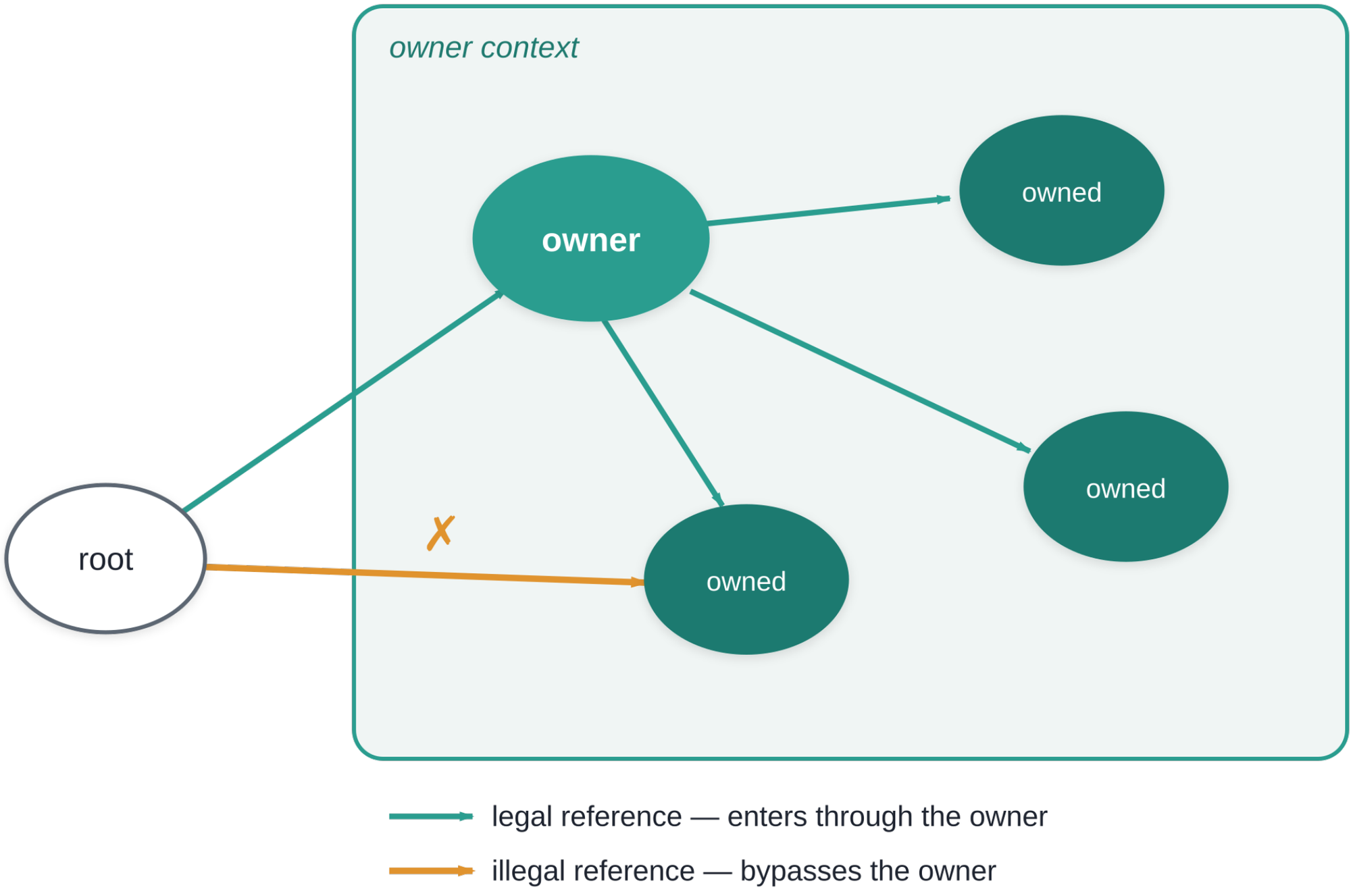
- Some objects are part of another object's representation
  - The items collection is part of the Order
  - The nodes are part of the linked list
  - The internal buffer is part of the stream

References to representation objects should not leak out.

Owners-as-dominators

- Every path from the root of the system to an object passes through it's owner
- The owner dominates its representation (graph-theoretic dominator)
- Tree of nested owner contexts — boxes inside boxes
- **Crossing a boundary without going through the owner: forbidden**

Owners-as-dominators



Every path from the root to an owned object must pass through its owner.

Ownership types

- Classes are parameterized by ownership contexts
- The type system statically rejects any expression that would leak an owned reference
- Our simple example would not compile

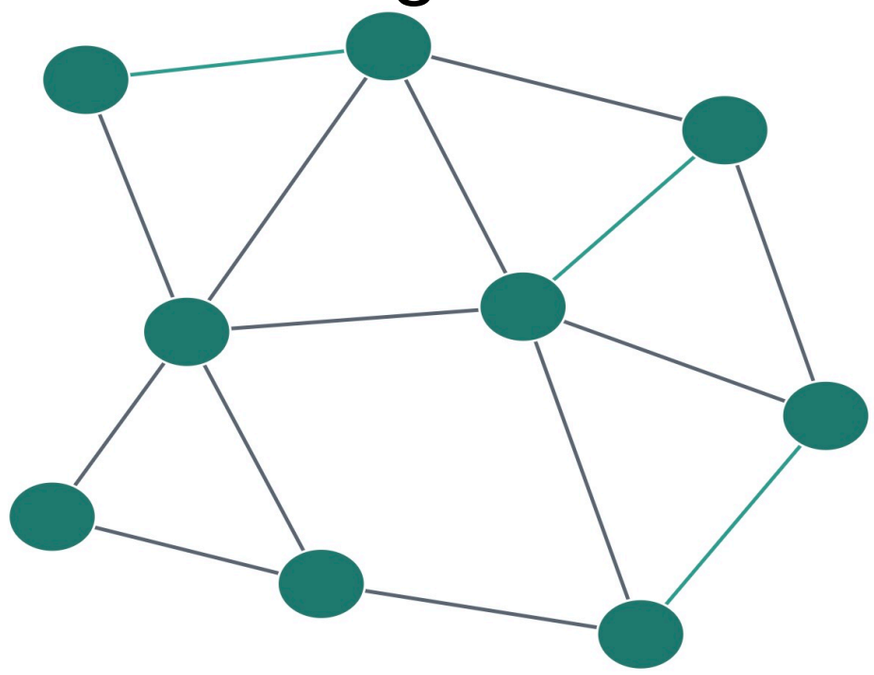
Clarke, Potter, Noble — Ownership Types for Flexible Alias Protection · OOPSLA 1998

Industrial Systems

- Rust: every value has an owner
  - assignment moves, borrowing
  - Quite Complex
- Swift, Mojo

But: Typesystems

- These are Typesystems: They restrict the structure of the build graph and programs manipulating it
- The Object Graph at runtime knowns nothing



Smalltalk: Not a good fit

- No types
- Reflection
- Debugger / Inspector
- Liveness: Code changes with existing objects

Dynamic ownership

- Model ownership as a feature of the object graph
- Check ownership on write
- What exactly? No Idea !
    - Challenge: Memory and Performance

Questions?

- Object Graph is an unstructured mess at runtime

- We need to explore how to do better !