On the Benefits of Adding Modes on Owners

— *a work in progress* —

Ownership, Uniqueness and Immutability

Johan Östlund

Tobias Wrigstad   Dave Clarke   Beatrice Åkerblom

Imagine a linked list with students at some university

Imagine a linked list with students at some university

We want the administrator to see who is registered

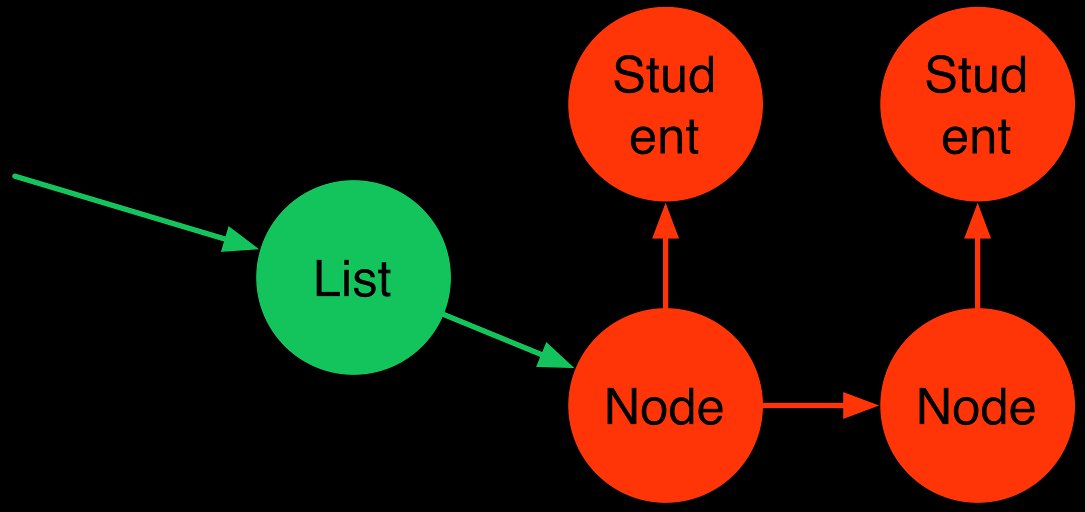
Imagine a linked list with students at some university

We want the administrator to see who is registered

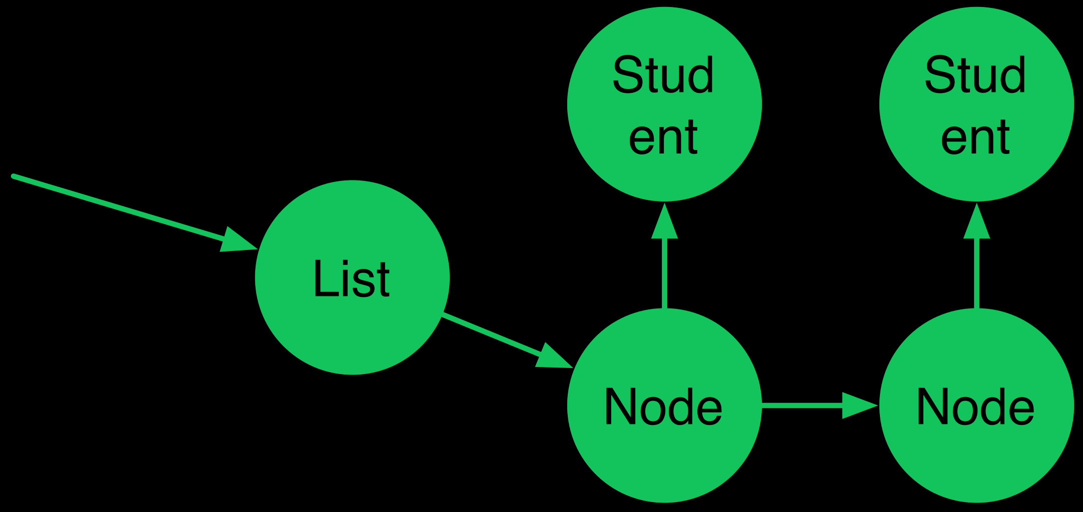
and we want the TAs to be able to mark the students

Read-Only Fails to do Both

Shallow (e.g., const)



Deep

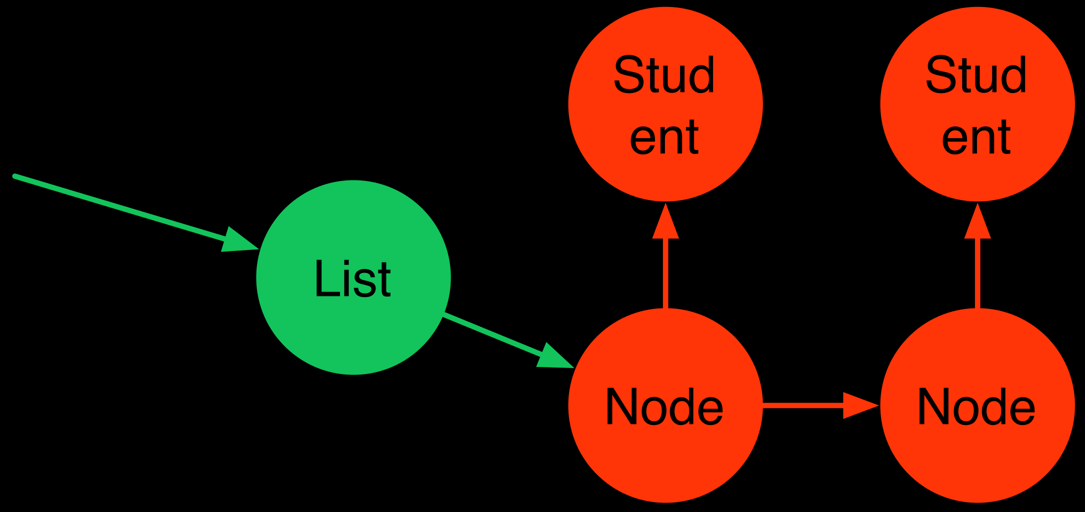


Mark students as passed on a course
But removing a student is also possible

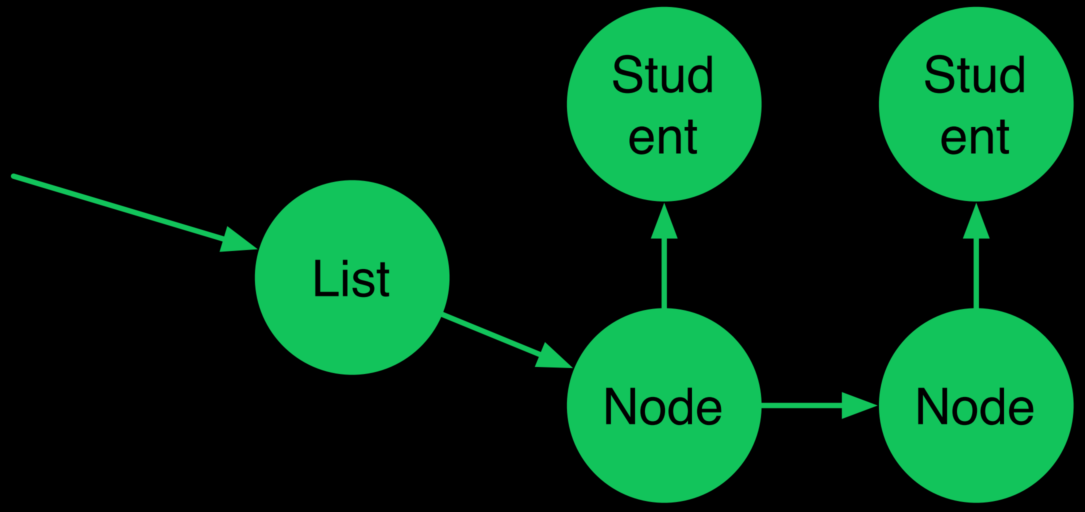
Check what students are registered
But cannot fix duplicate registrations

Read-Only Fails to do Both

Shallow (e.g., const)



Deep



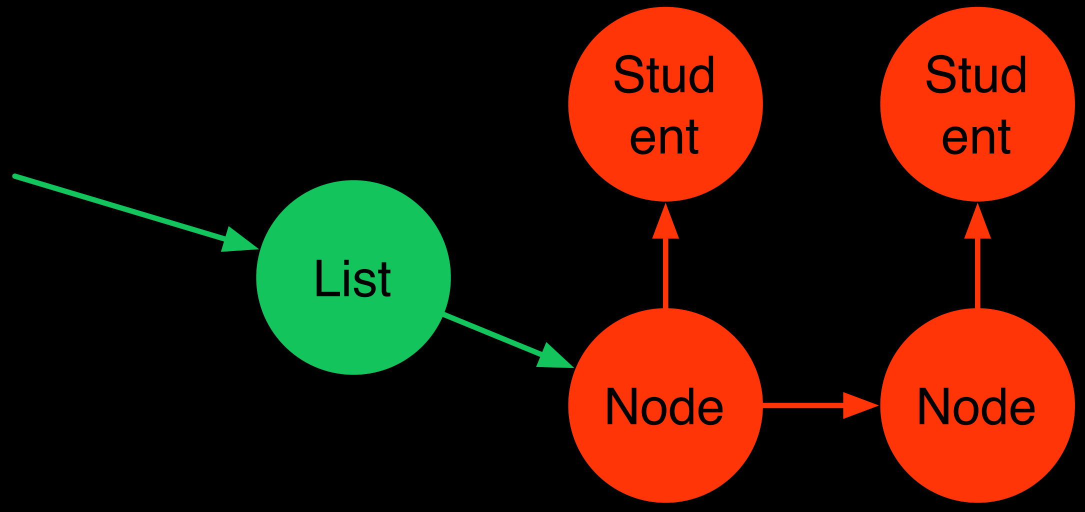
— too restrictive

Mark students as passed on a course
But removing a student is also possible

Check what students are registered
But cannot fix duplicate registrations

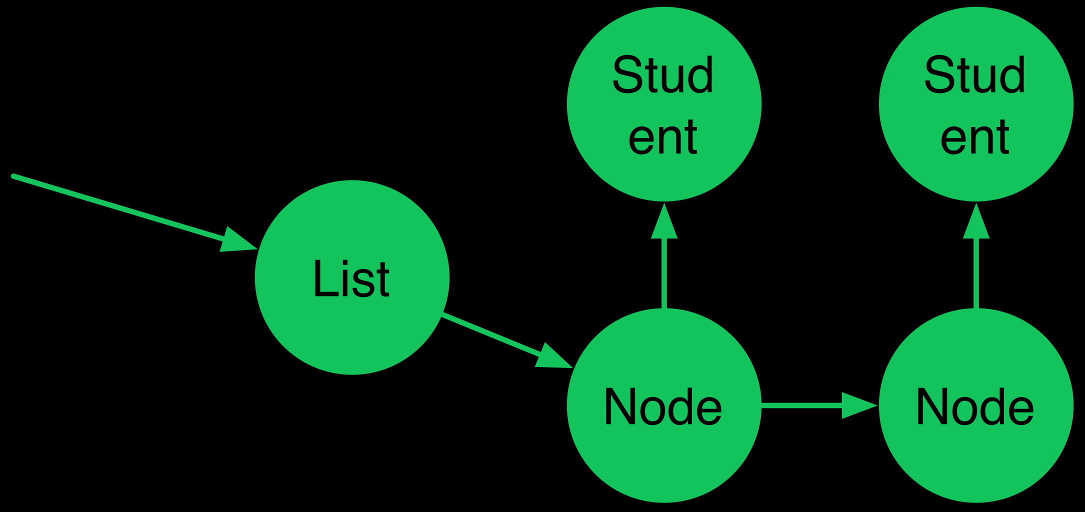
Read-Only Fails to do Both

Shallow (e.g., const)



— too permissive

Deep



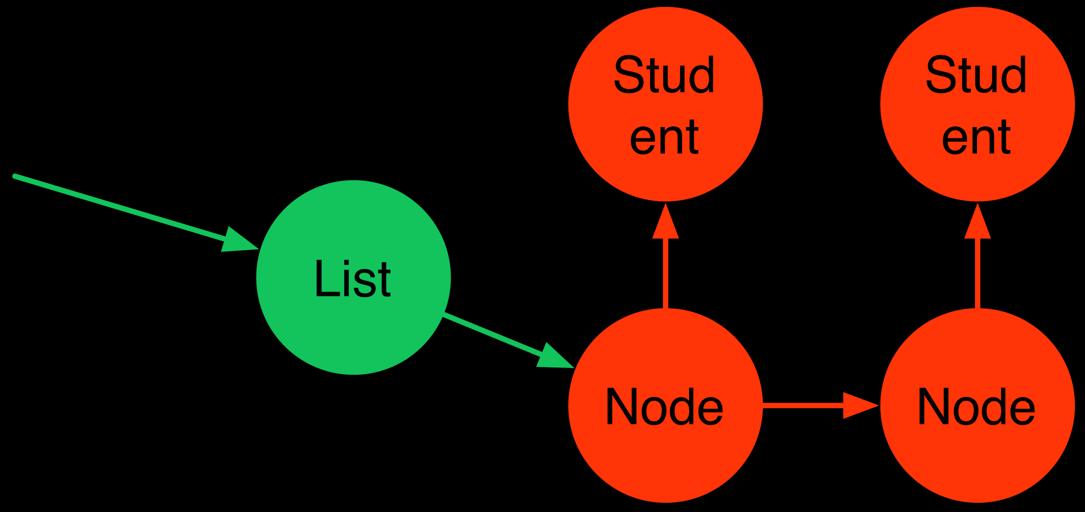
— too restrictive

Mark students as passed on a course
But removing a student is also possible

Check what students are registered
But cannot fix duplicate registrations

Read-Only Fails to do Both

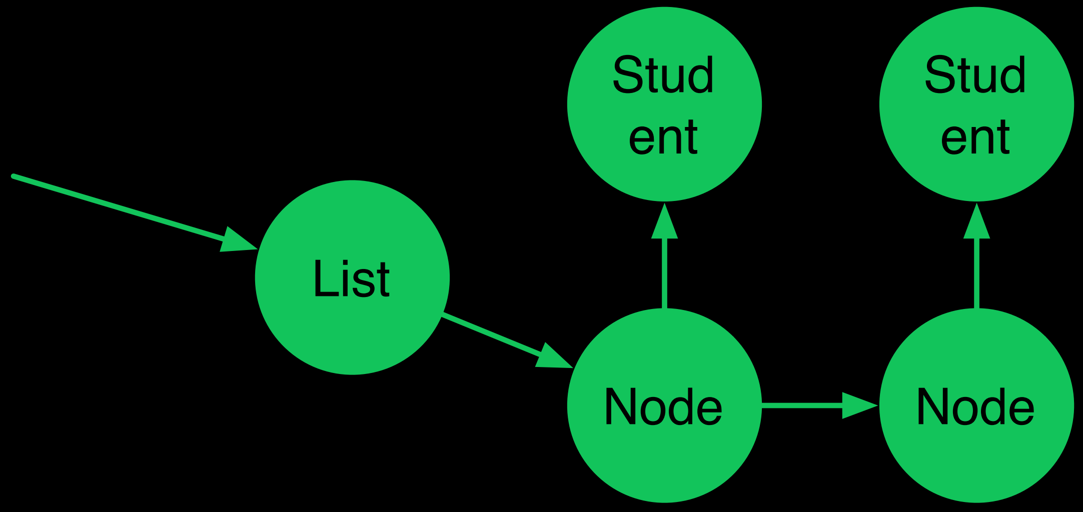
Shallow (e.g., const)



— too permissive

Mark students as passed on a course
But removing a student is also possible

Deep



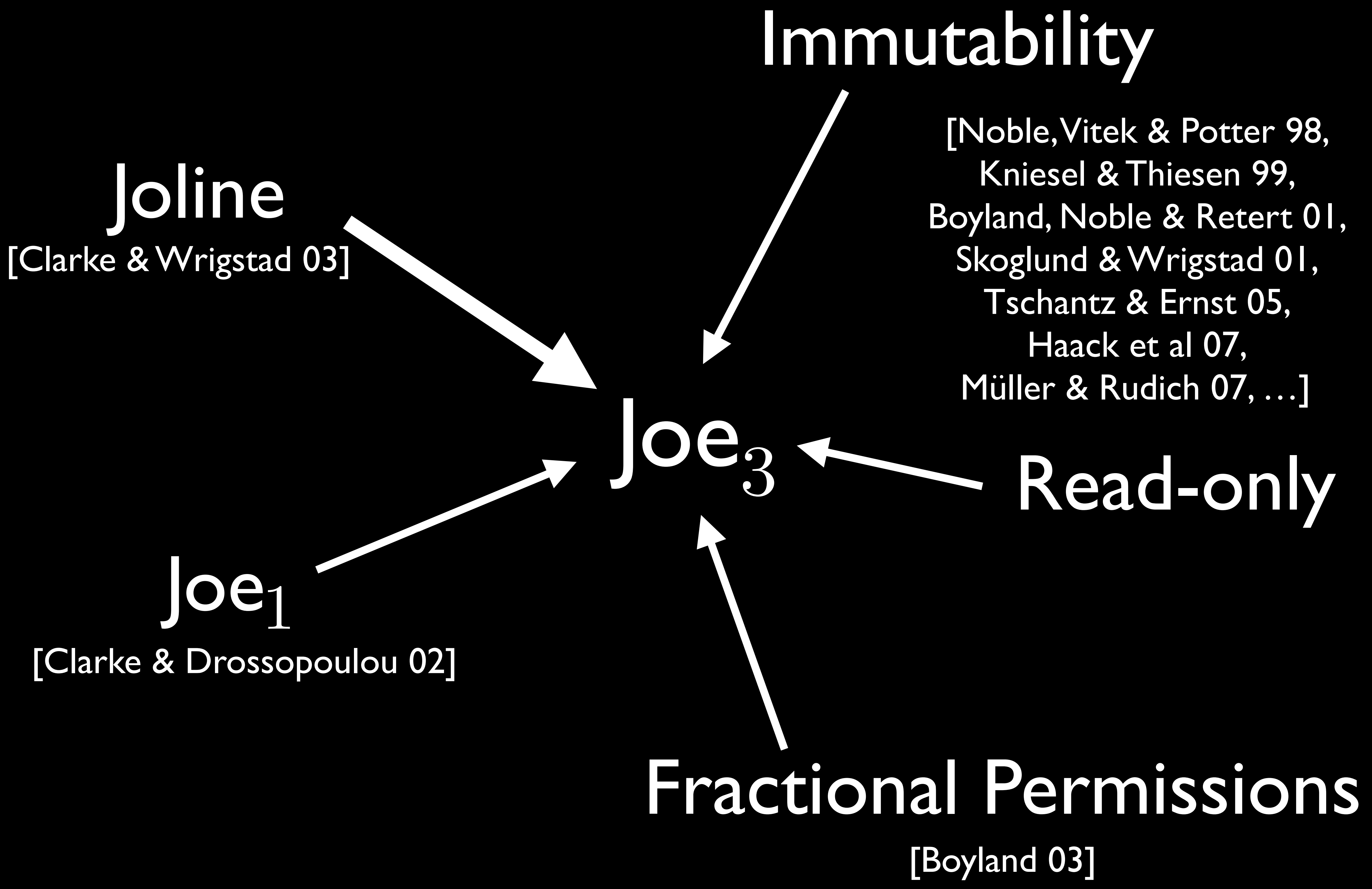
— too restrictive

Check what students are registered
But cannot fix duplicate registrations

Ad hoc — can be misused
[Tschantz & Ernst 05]

Design Goals

- Partial read-only in a non ad-hoc fashion
- Multiple simultaneous views of a single object in terms of modifiability
- One class for all views
- Not possible to circumvent read-only
- Co-existing read-only and immutability
- Fractional permissions-style immutables


Immutability
[Noble, Vitek & Potter 98,
Kniesel & Thiesen 99,
Boyland, Noble & Retert 01,
Skoglund & Wrigstad 01,
Tschantz & Ernst 05,
Haack et al 07,
Müller & Rudich 07, …]
Joline
[Clarke & Wrigstad 03]
Joe_3
Read-only
Joe_1
[Clarke & Drossopoulou 02]
Fractional Permissions
[Boyland 03]

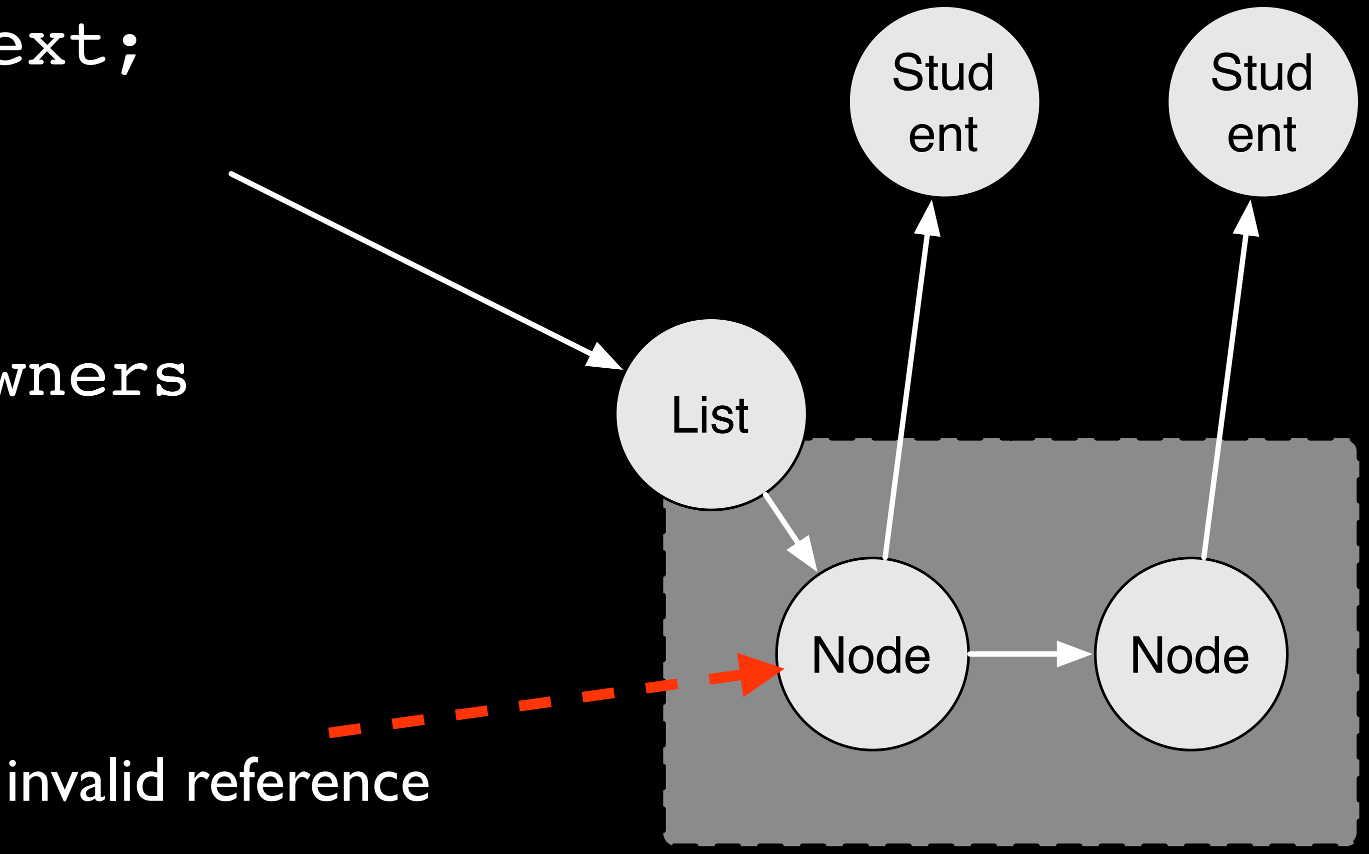
A Linked List in Joline

[Clarke & Wrigstad 03]

```
class List<data outside owner> {
  this:Node<data> first;
}

class Node<data outside owner> {
  data:Object stuff;
  owner:Node<data> next;
}


// a and world are owners
a:List<world> l;
```



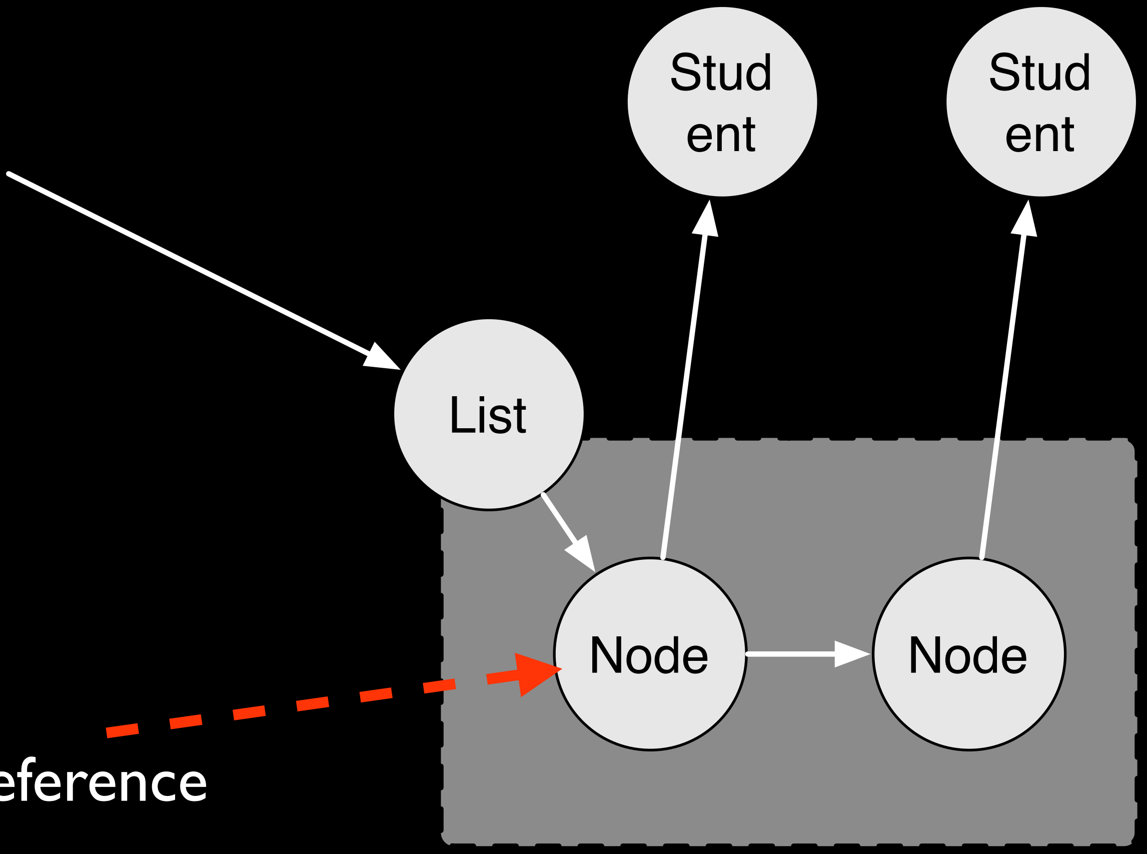
A Linked List in Joline

[Clarke & Wrigstad 03]

```
class List<data outside owner> {
  this:Node<data> first;
}

class Node<data outside owner> {
  data:Object stuff;
  owner:Node<data> next;
}


// a and world are owners
a:List<world> l;
```



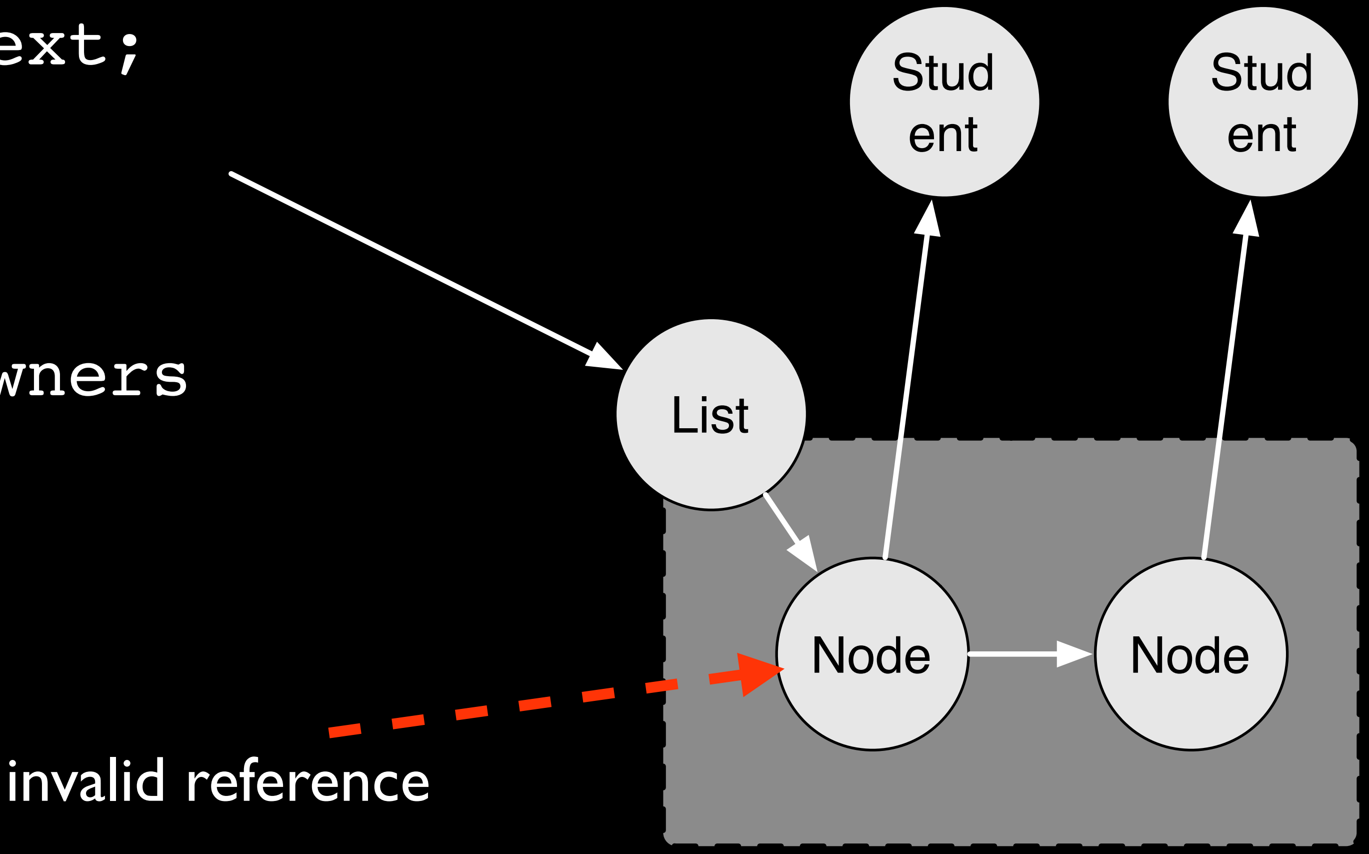
A Linked List in Joline

[Clarke & Wrigstad 03]

```
class List<data outside owner> {
  this:Node<data> first;
}

class Node<data outside owner> {
  data:Object stuff;
  owner:Node<data> next;
}


// a and world are owners
a:List<world> l;
```



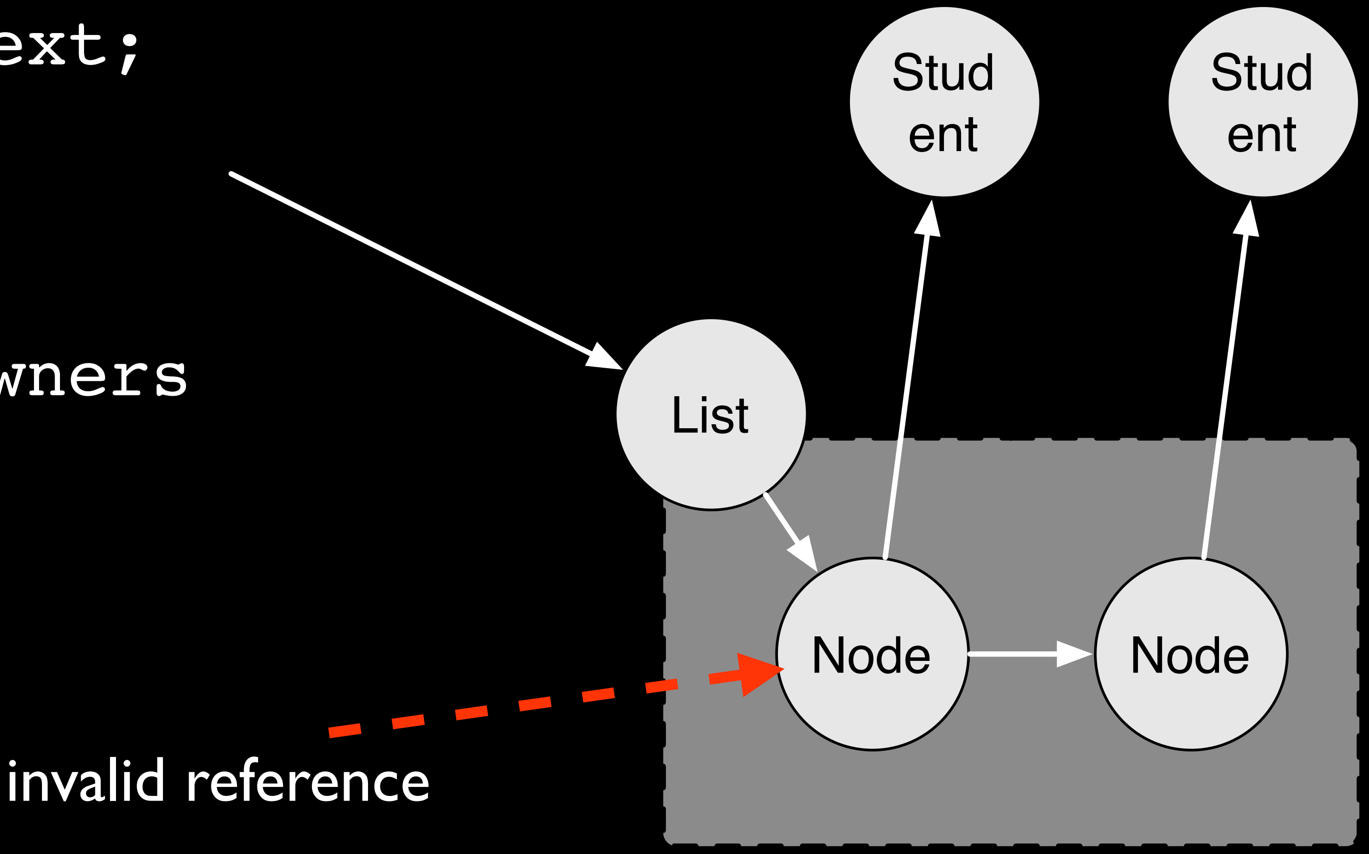
A Linked List in Joline

[Clarke & Wrigstad 03]

```
class List<data outside owner> {
  this:Node<data> first;
}

class Node<data outside owner> {
  data:Object stuff;
  owner:Node<data> next;
}


// a and world are owners
a:List<world> l;
```



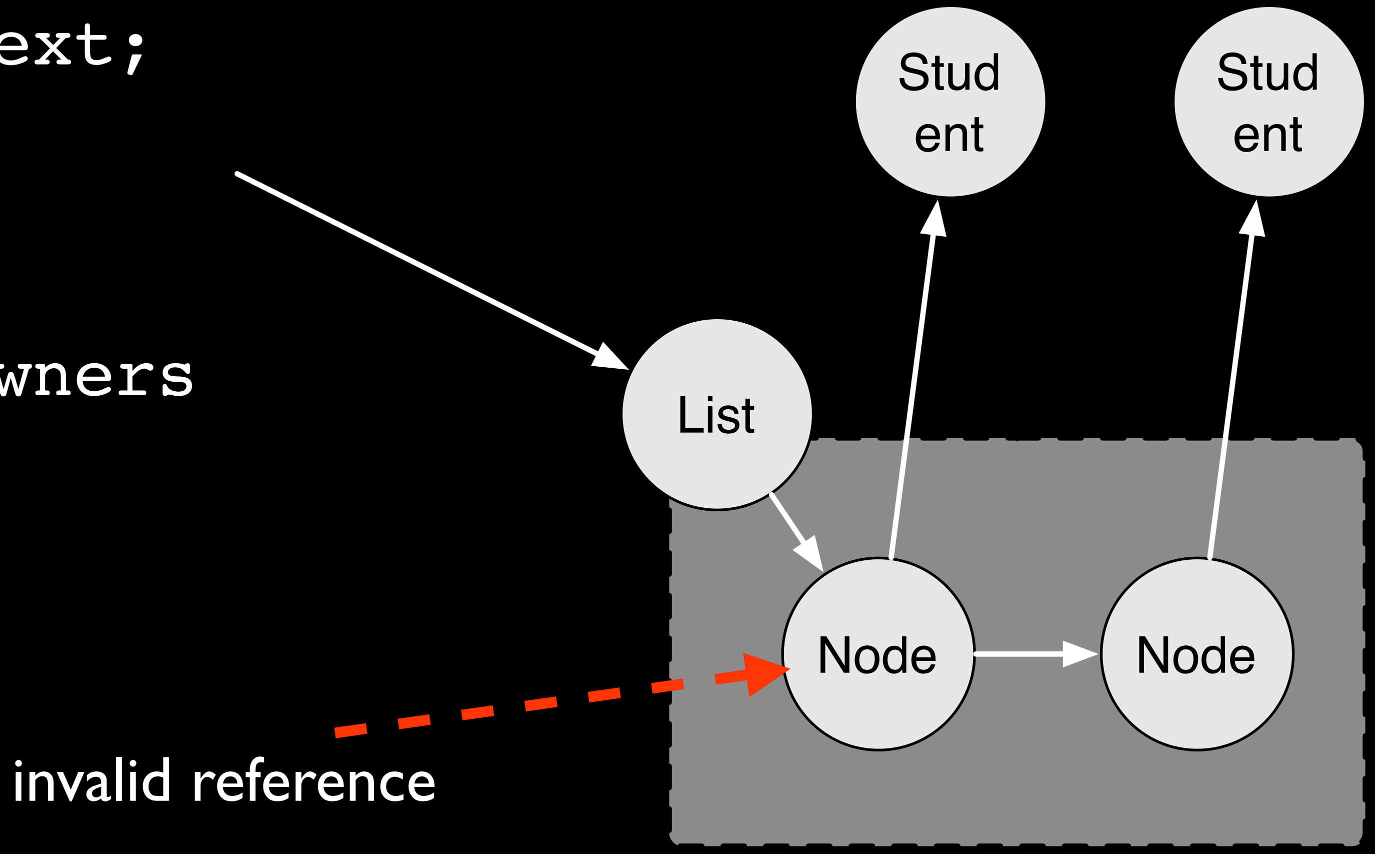
A Linked List in Joline

[Clarke & Wrigstad 03]

```
class List<data outside owner> {
  this:Node<data> first;
}

class Node<data outside owner> {
  data:Object stuff;
  owner:Node<data> next;
}


// a and world are owners
a:List<world> l;
```



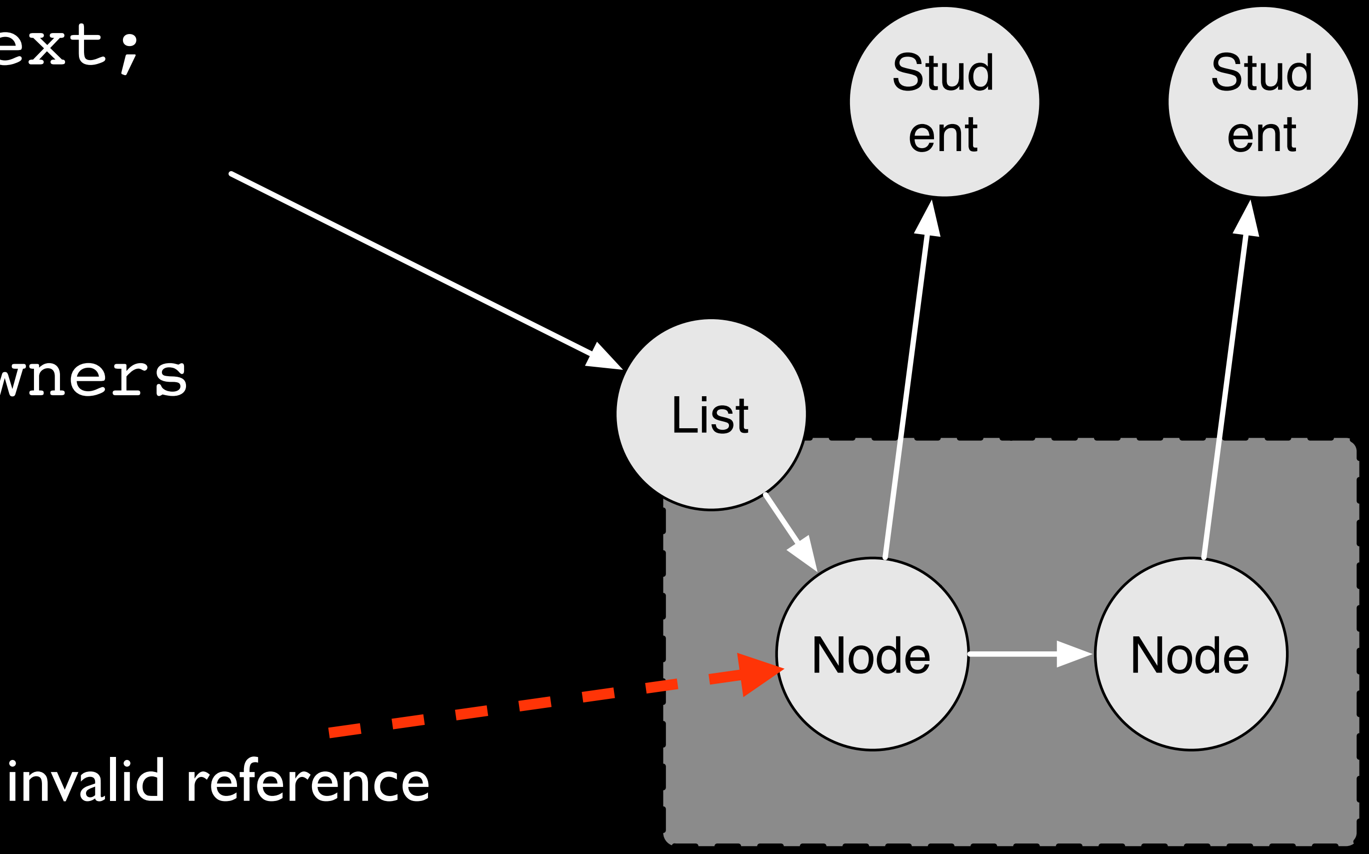
A Linked List in Joline

[Clarke & Wrigstad 03]

```
class List<data outside owner> {
  this:Node<data> first;
}

class Node<data outside owner> {
  data:Object stuff;
  owner:Node<data> next;
}


// a and world are owners
a:List<world> l;
```

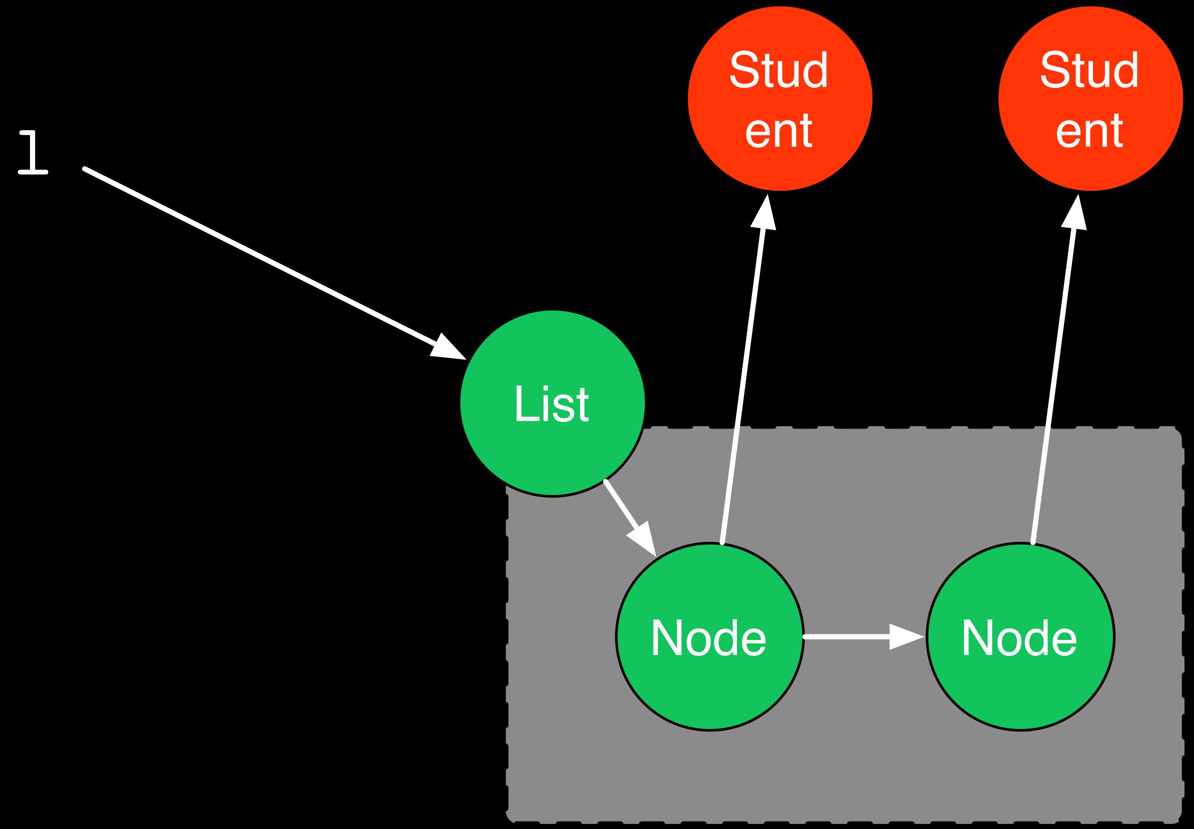


Modes in Joe$_3$

```
class List<data- outside owner> { // owner+ this+
  this:Node<data> first;
}

class Node<data- outside owner> { // owner+ this+
  data:Object stuff;
  owner:Node<data> next;
}

// Type controls usage
// a- and b+ are owners
a:List<b> l;
```

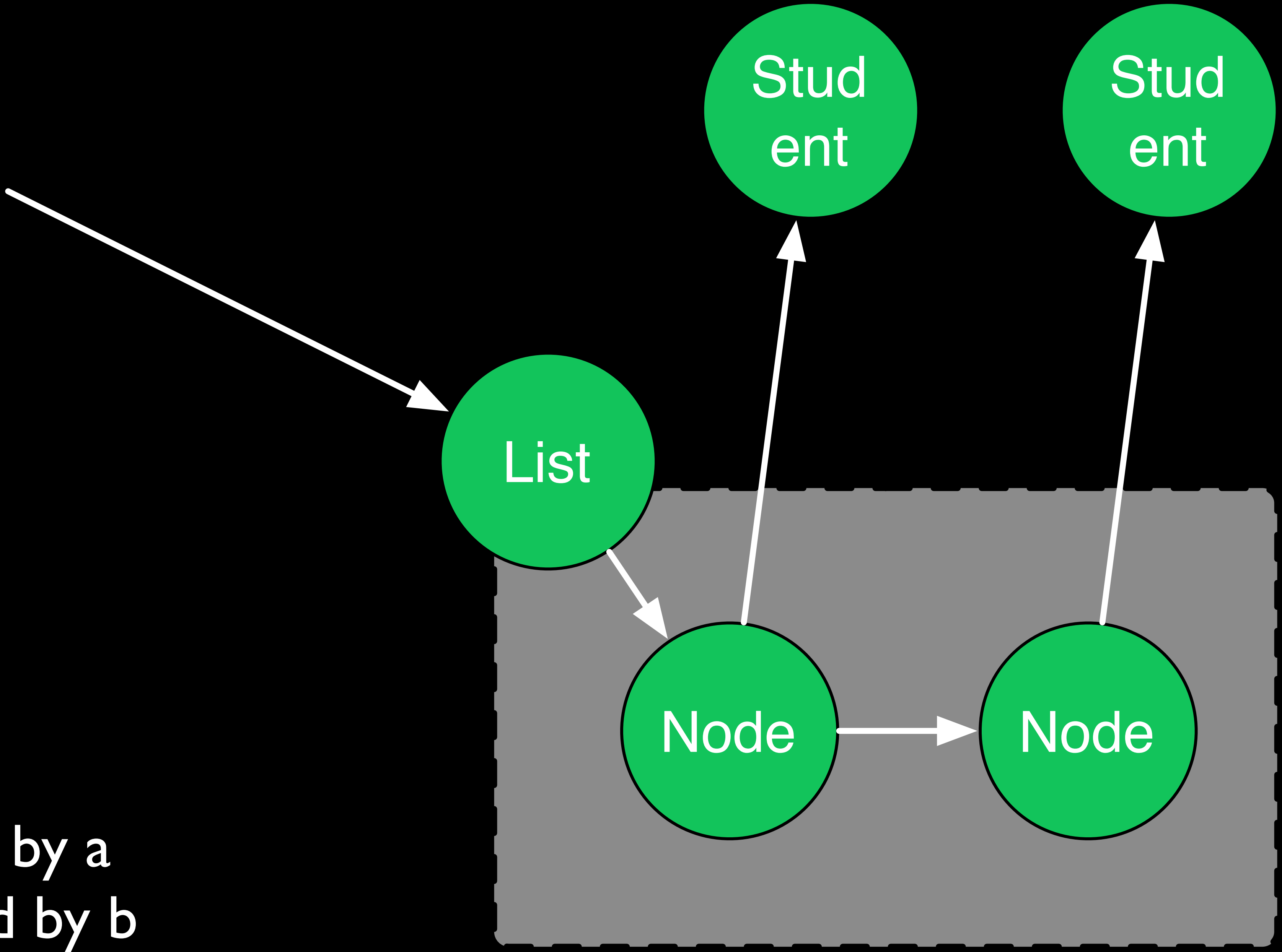


`a:List<b>` — check what students are registered (deep)

`a:List<b>` — Mark students as passed on a course (partial)

`a:List<b>` — Register and deregister students (partial)

`a:List<b>` — Your regular reference



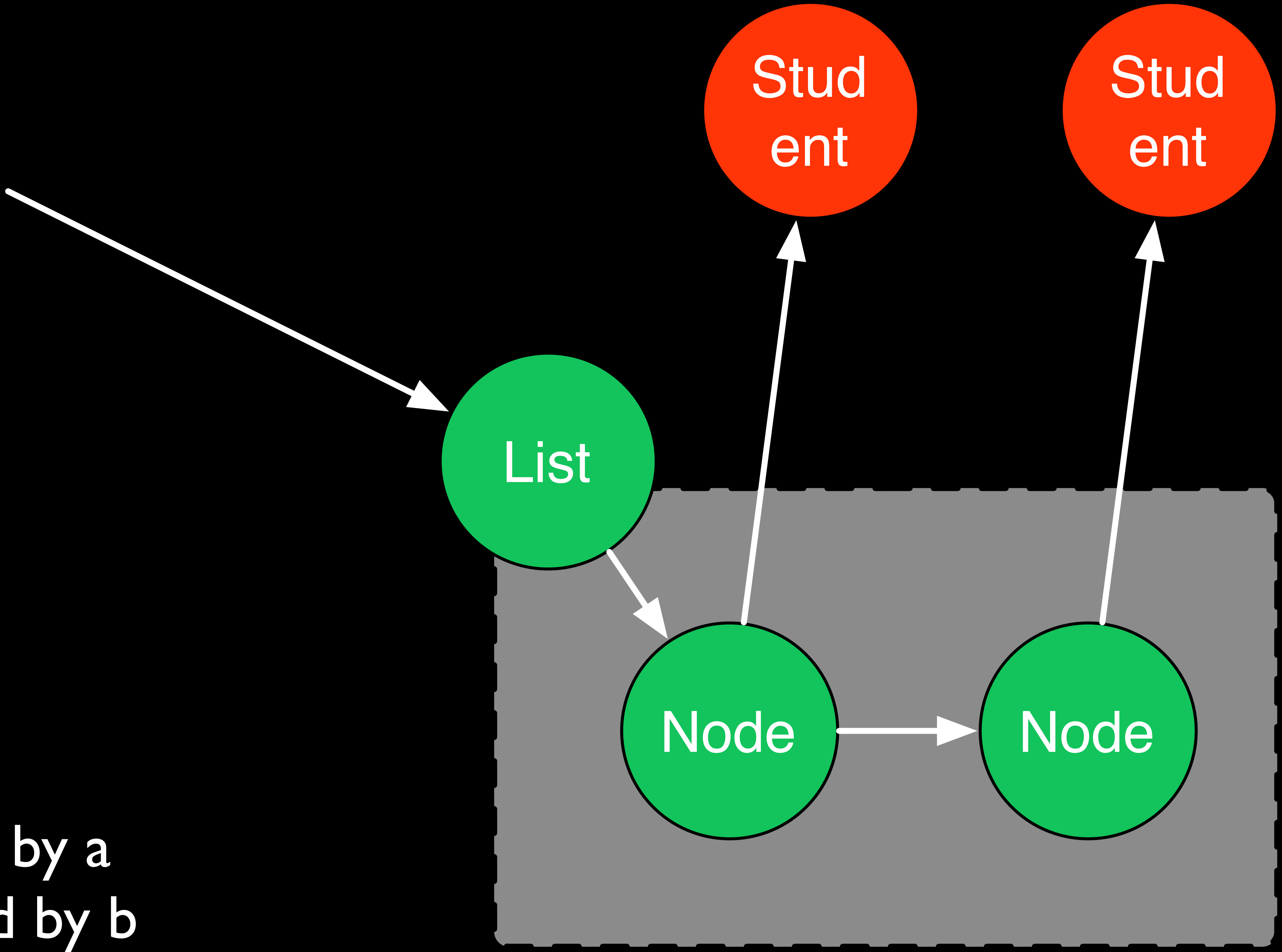
List is owned by a
Stuff is owned by b

`a:List<b>` — check what students are registered (deep)

`a:List<b>` — Mark students as passed on a course (partial)

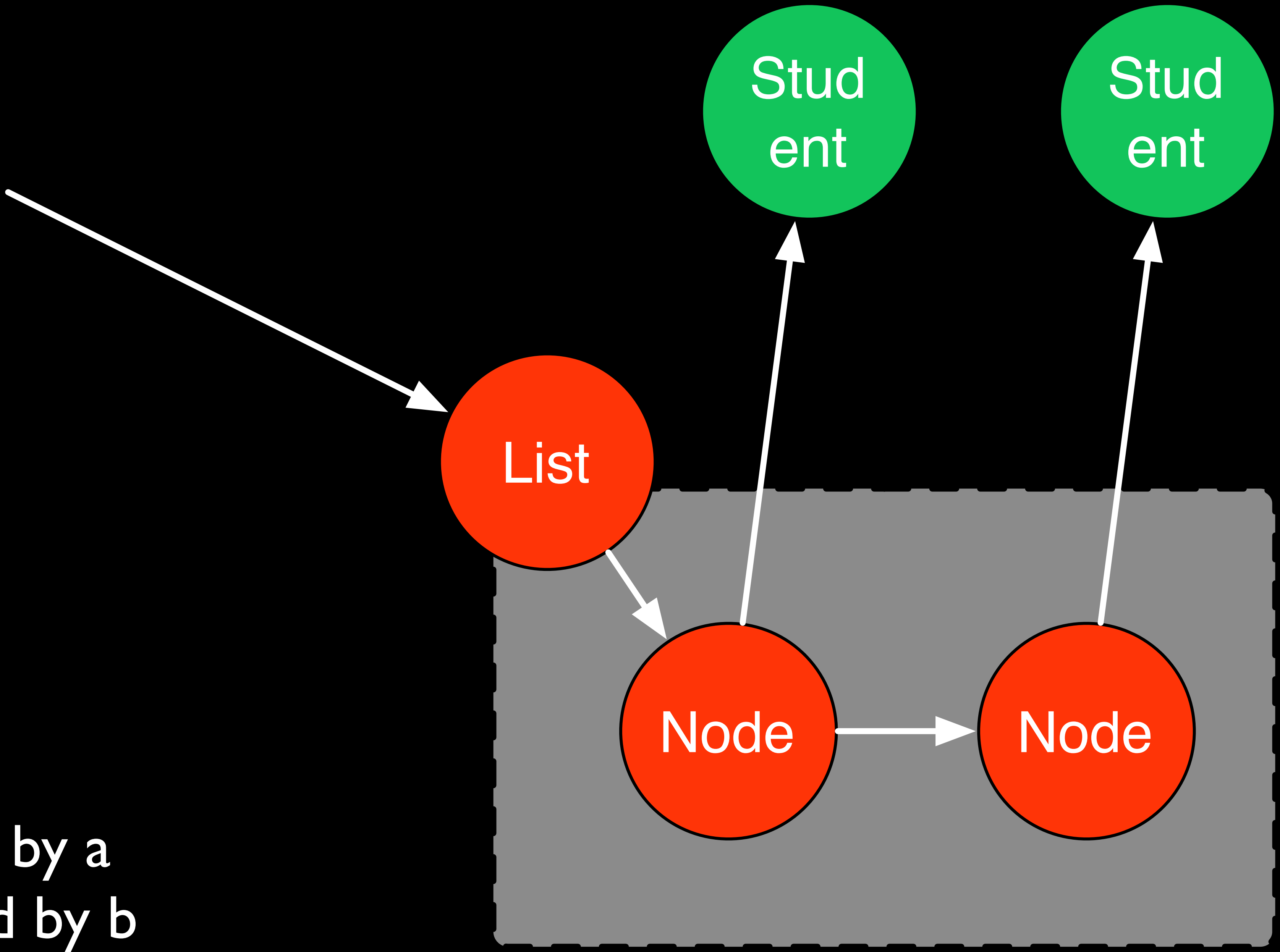
`a:List<b>` — Register and deregister students (partial)

`a:List<b>` — Your regular reference



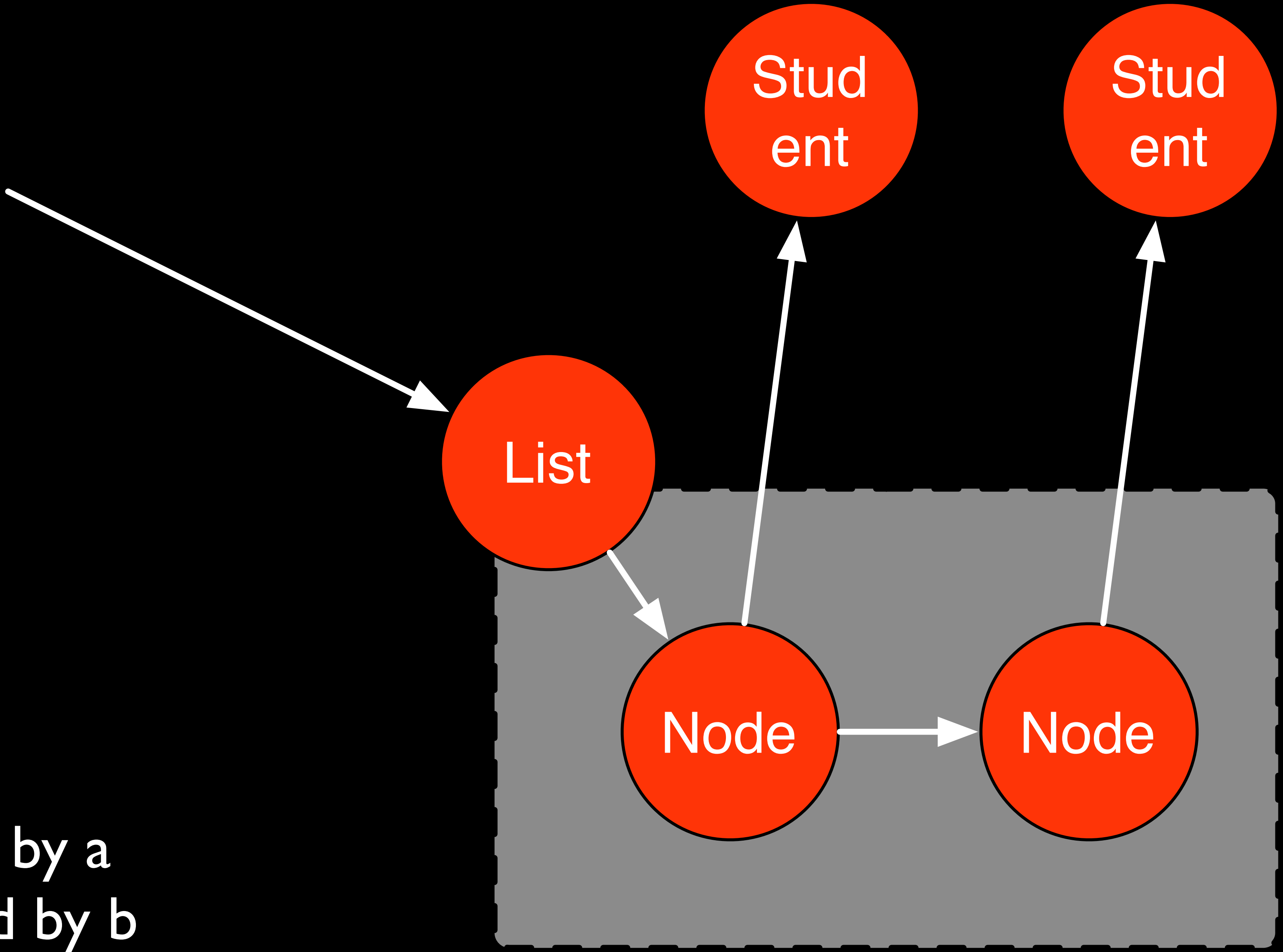
List is owned by a
Stuff is owned by b

`a:List<b>` — check what students are registered (deep)
`a:List<b>` — Mark students as passed on a course (partial)
`a:List<b>` — Register and deregister students (partial)
`a:List<b>` — Your regular reference



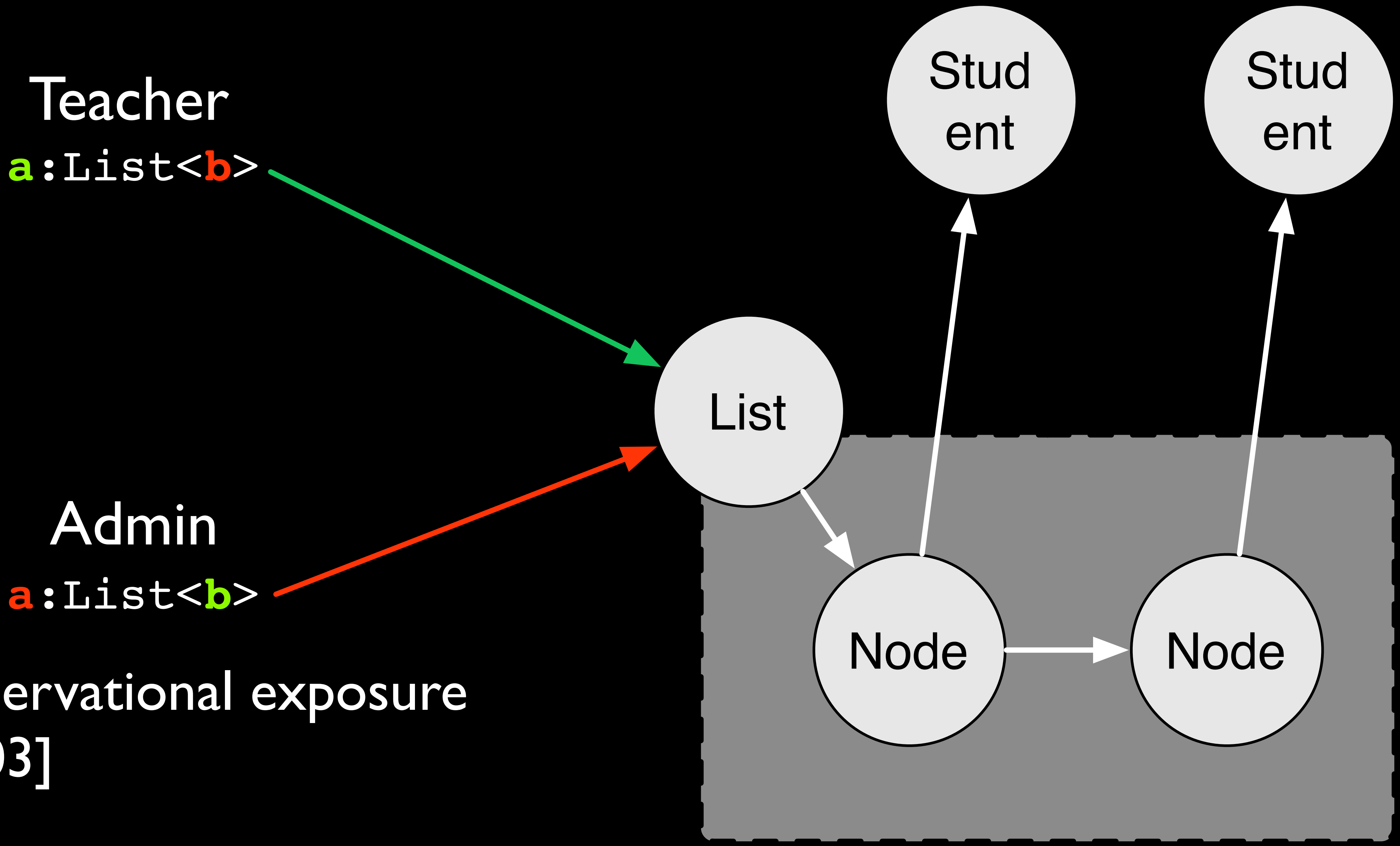
List is owned by a
Stuff is owned by b

`a:List<b>` — check what students are registered (deep)
`a:List<b>` — Mark students as passed on a course (partial)
`a:List<b>` — Register and deregister students (partial)
`a:List<b>` — Your regular reference



List is owned by a
Stuff is owned by b

Multiple Views



Bad: Observational exposure [Boyland 03]

Good: Context-based read-only

Immutability & the * Mode

```
<a* inside world, b* outside a>
         int averageMark(a:List<b> students) {
    ...
}
```

- Immutable can trivially be achieved by read-only plus unique — but the information is lost
- The *-mode captures immutability in Joe3
- Only unique pointers can achieve *-dom
- Nice staged initialisation

~Fractional Permissions

```
unique:List<d> l;
borrow l as x*:temp in {
  // temp : x:List<d> for duration of block
  ...
}
```

- Borrowing allows unique variables to be treated as immutable for the duration of a scope
    - Temporarily nullifies the source variable
    - Automagic confinement through temporary owner
- Essentially Boyland's [03] Fractional Permissions

Joe$_3$'s Static Semantics

- Trivial extension to Joline's static semantics
- Modes added to owners in type environment
- Trivial changes to four rules to check that the modes on a receiver is respected by method calls, field updates and borrowing
- Revoke clause added to enable finer granularity

```
void method() revoke this {...}
void method(x:Object) revoke x {...}
```

Modes & Inheritance

(not in the paper)

- Subclassing must preserve immutable modes
- Subclassing to narrow permissions is straight-forward
- Subclassing to widen permissions is possible
  - Overriding methods must always obey the most restrictive modes of any super class
  - Modification only possible in new, non-overriding methods

Future Work

- Prove soundness by extending Joline’s proofs
- Properly formulate the guarantees of our constructs
- Explore Universes-style owner-as-modifier
- Modes on types, not just owner declarations

```
class Ex {
  owner+:Object rep;

  owner-:Object getRep() { return rep; }
  void setRep(owner+:Object o) { rep = o; }
}
```

Thank You! Questions?