Applications of Ownership Types

Ownership Application 1- restrict aliasing

```
College <p> c;  College <p> c';
```

We can deduce that

`c.ee` and `c.dramSoc`	cannot be aliases
`c.ee` and `c.visitors`	cannot be aliases
`c.dramSoc` and `c.visitors`	cannot be aliases
`c.doc` and `c.ee`	may be aliases

If `c` and `c'` are guaranteed not to be aliases, and not one inside the other, then

`c.ee` and `c'.ee`	cannot be aliases
`c.visitors` and `c'.visitors`	may be aliases

Ownership Application 2- restrict interference

We can express the footprint of some execution:

```
class Department<p1>{
      SList<this,this> ownedStList;

      void order() { … }
      footprint { ownedSTList }
      // order students according to SOLE results

      void markCW() { … }
      footprint { this }
      // update each student's cw records   }
```

If `c.doc` and `c.ee` are guaranteed not to be "inside" one another, then

`c.doc.order()` and `c.ee.markCW()`

have *disjoint* footprints, and can be executed in any order, or interleaved.

D. Clarke, S. Drossopoulou: Ownership, Encapsulation and Disjointness of Types and Effects, OOPSLA'02. Laos *Cameron & al*, MOJO, OOPSLA 2007.

Ownership Application 3 – cheaper reasoning

We can express the footprint of properties

```
class Department<p1>{
      … // as earlier
      boolean happyStudents() { … }
      footprint { this }
      // students' SOLE results were   ☺
}
```

If `c.doc` and `c.ee` are guaranteed not to be "inside" one another, then

`c.ee.markCW()` preserves `c.doc.happyStudents()`.

In other words, if `c.doc.happyStudents()` holds, then the property `c.doc.happyStudents()` still holds after execution of `c.ee.markCW(),`.

Matthew Smith: Ownership, Encapsulation and Disjointness of Types and Effects, OOPSLA'02

Ownership Application 4 – parallelism

Given that the footprints doe not overlap, if `c.doc` and `c.ee` are guaranteed not to be "inside" one another, then we can execute

```
c.ee.markCW()  ||  c.doc.happyStudents().
```

without need to lock anything!

R. L. Bocchino, V.S. Adve, D.Dig, S. V. Adve, S. Heumann, A Type and Effect System for Deterministic Parallel Java, OOPSLA 2009

NOTE: disjointness of the effects (andc.doc and c.ee not being aliases) can be checked at runtime.

Ownership Application 5 - atomicity/races

Consider an instruction `o.lock{ e }`, which locks object `o`, then executes `e`, and then releases `o`. Type system ensures that fields only accessed in contexts where owner of the object with the field has been `locked`.

```
class Slist<p1>{
    Slink<this,this> first
    void markCW( ){
          this.lock{ this.first.markCWS(); }
}   } }
class SLink<p1,p2>{
    Slink<p1,p2> next; Student< p2> s;
    void markCWS( ){
          this.s.mark = "A*";
         // NO NEED TO LOCK EACH STUDENT^
         … this.next.markCWS() …}        } }
```

Execution of `markCW()` will not to enter races, and is atomic.

Boyaparti, Rinard, A Parameterized Type System for Race-Free Java Programs, *OOPSLA 2001*

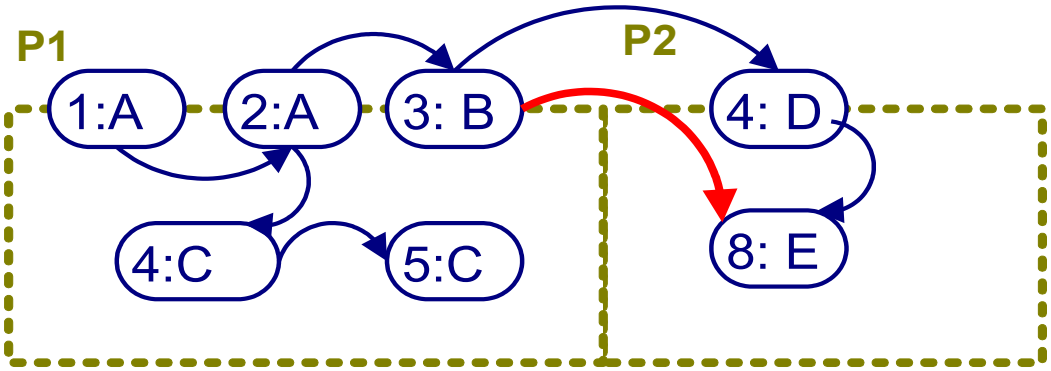
Ownership Application 6 – Confinement

A type can be declared to be *confined* within a package..

```
package P1 {
    class A{ ... }
    class B{ ... }
    confined class C{ ...
}
 }
```

```
package P2 {
    class D{ ... }
    confined class E{ ... }
 }
```

Objects of a confined class only accessible to objects of same package (red arrow = forbidden).



Software engineering applications.

Bokowski, Vitek. Confined types., OOPSLA 1991

Ownership Application 7 – Architecture

Jonathan Aldrich and Craig Chambers: Ownership Domains: Separating Aliasing Policy from Mechanism.. ECOOP '04

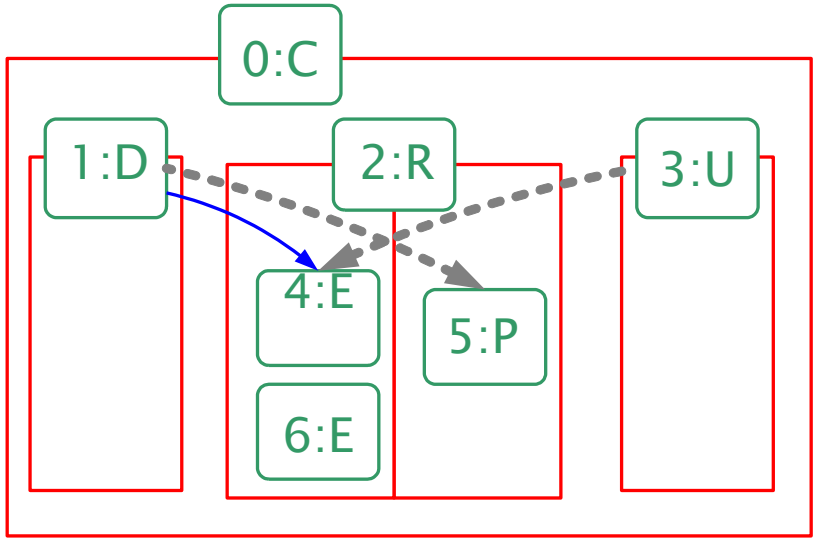
Characterize which boxes may have access to other boxes

```
class Registry <o1> {
    box PaymentData;
    box ExamsData
    …  }
class Departmt<o1> { .. }
class Union<o1> { .. }
```

```
class College{ {
    Registry<this> reg;
    Department<this> doc;
    Union<this>  studUnion;
    link reg.ExamsData with doc;
 }
```

The `link` assertion allows an object to access the contents of a box belonging to another object.

(Here the grey, dotted arrow forbidden)



Ownership Application 8 – Object Initialization

...

Ownership Application 9 – Object Cloning

...

Ownership Application 10 – Access Control

...

Ownership Application 11 – Energy

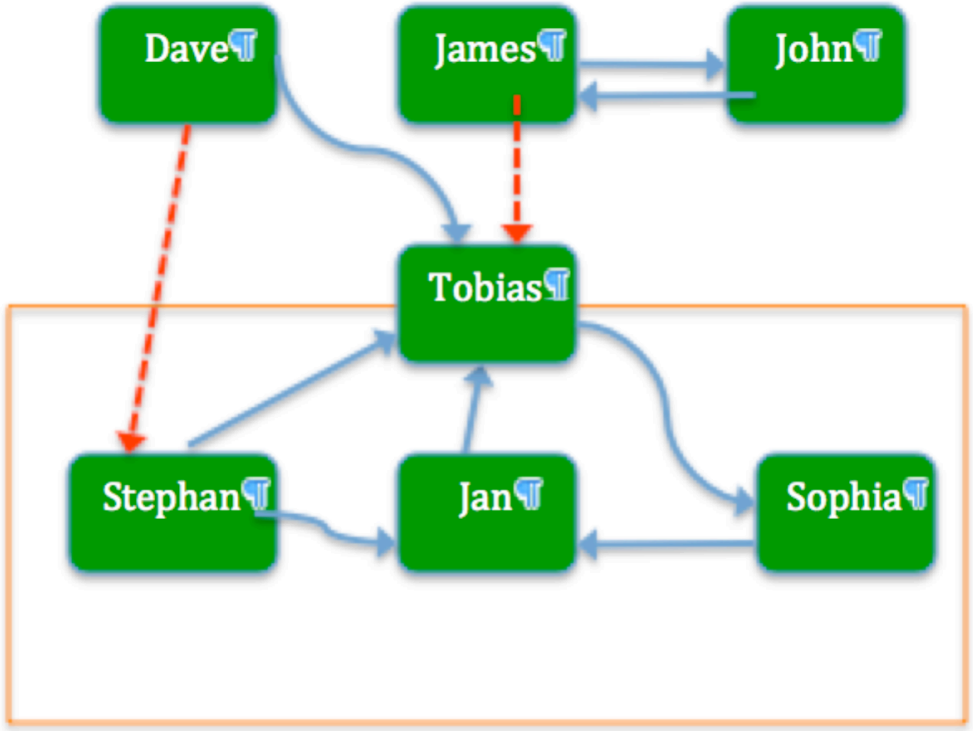
...

Ownership Application 12 – AOP

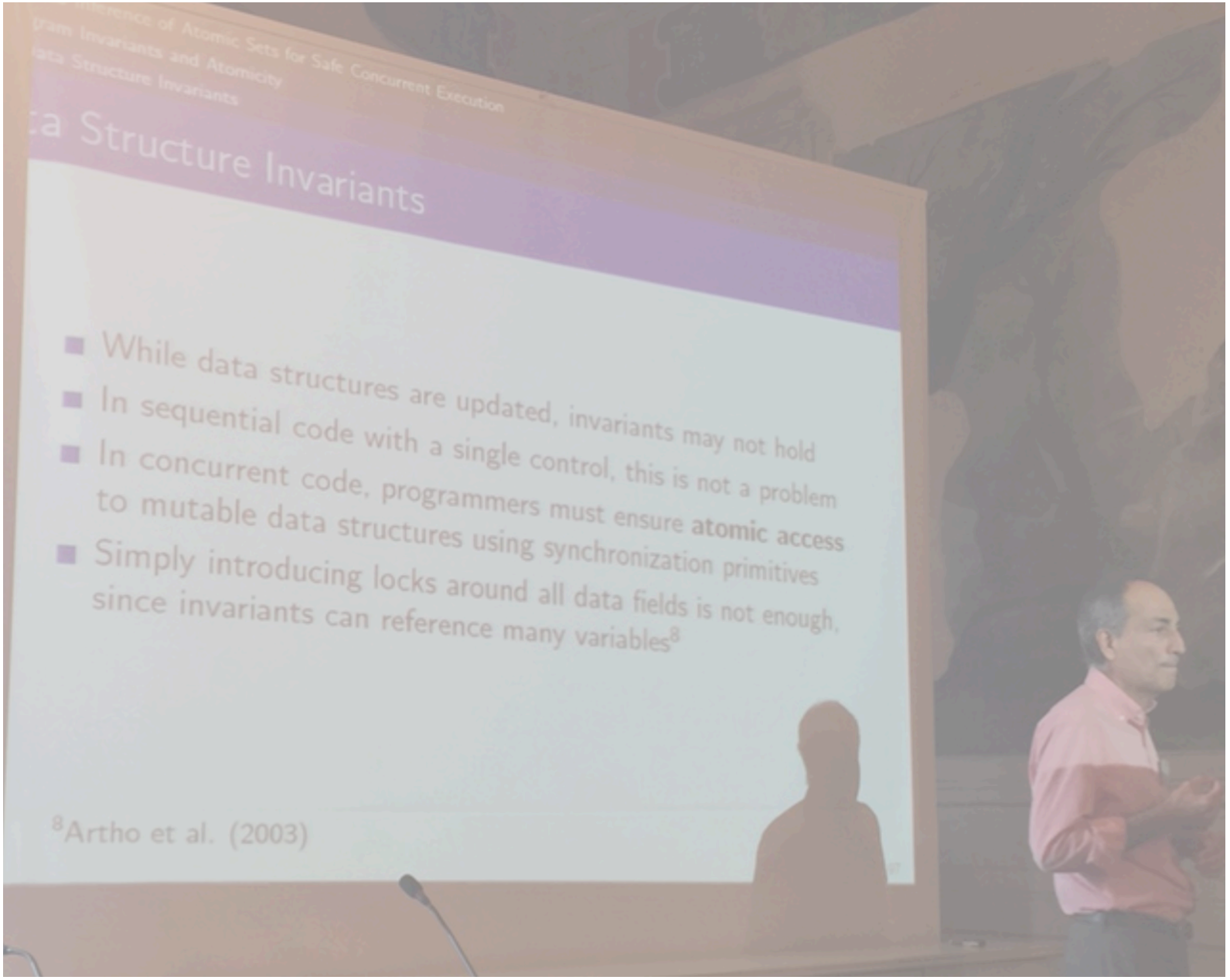
...

Ownership Application 13 – External Uniqueness

Any number of internal references allowed, but **only one external** reference to the aggregate



Wrigstad, Clarke: External Uniqueness is Unique Enough, ECOOP 2003


Atomic Sets for Safe Concurrent Execution
Invariants and Atomicity
Structure Invariants
Structure Invariants
While data structures are updated, invariants may not hold
In sequential code with a single control, this is not a problem
In concurrent code, programmers must ensure atomic access to mutable data structures using synchronization primitives
Simply introducing locks around all data fields is not enough, since invariants can reference many variables[8]
[8]Artho et al. (2003)

Ownership Application 14 – Reasoning

The concept of *monitor invariant* was proposed in the 70s by Tony Haore. This was adapted to the concept of object invariant in the 90s by Bertrand Meyer.

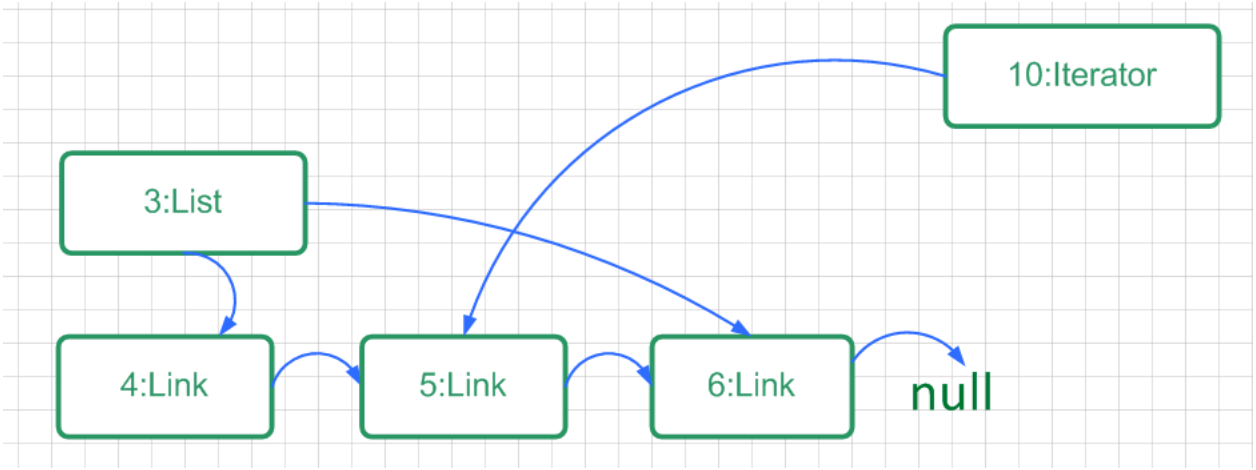
The *object invariant* is a property which is supposed to hold throughout the lifetime of the object (with small exceptions).

This means that upon method call we can expect the object invariant to hold (with small exceptions).

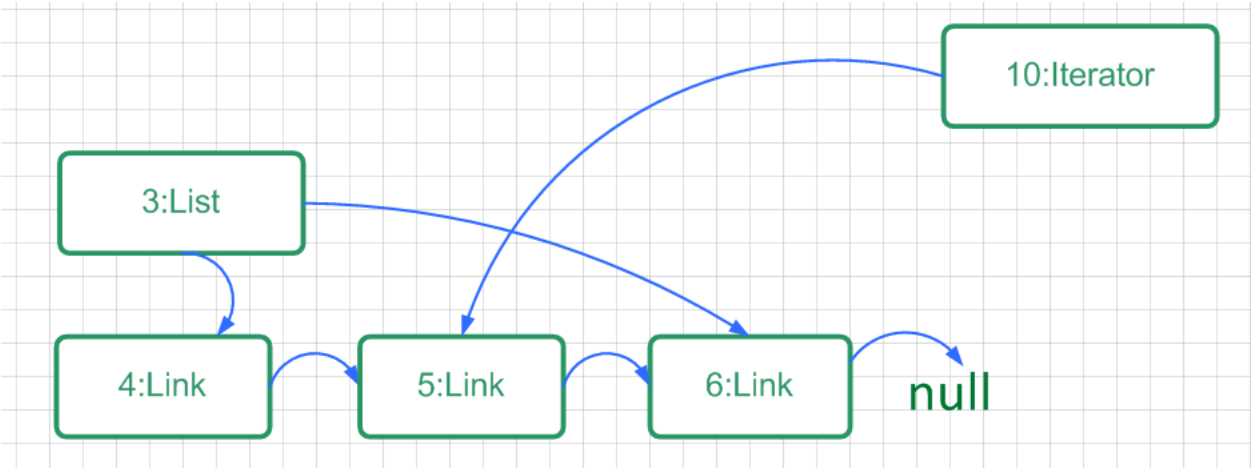
The difficulties arise when the invariant of an object depends on the state of other objects.

INVariants, an example

```
class List{
      Link first, last;
      INV last.next == null ∧ first.next*== last;
      …
}
class Link{
      Link next;
     …
}
class Iterator{  Link current; … }
```



INVariants, an example -- continued



Now, what if `lst1` points to `4`, and it executes

```
this.next.next:=this.next;
```

Or, of `iter1` points to `10`, and it executes

```
this.current.next := null;
```

Owners as Modifiers

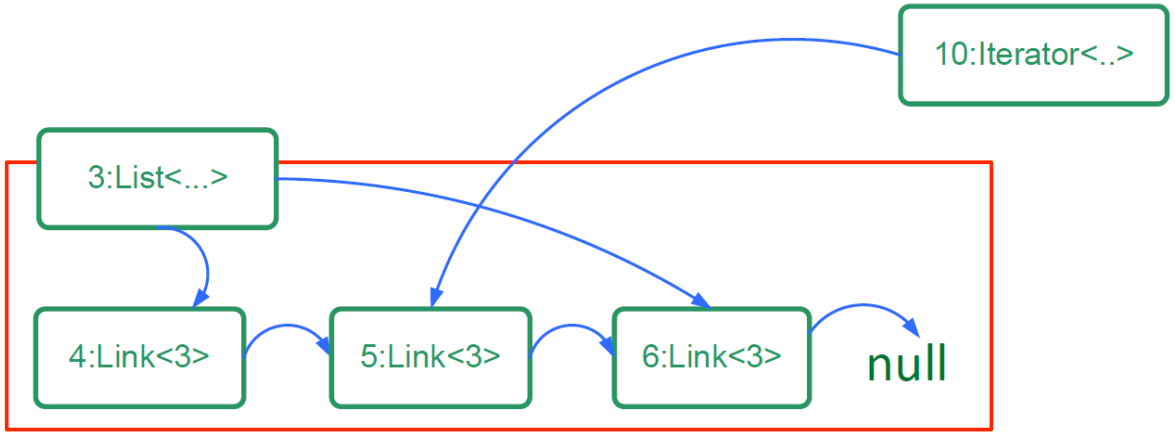
This concept was proposed in 2004 by Leino and Mueller. It had been adopted ion Spec#, and is used in MSR's VCC.

Owners as Modifiers mandate:

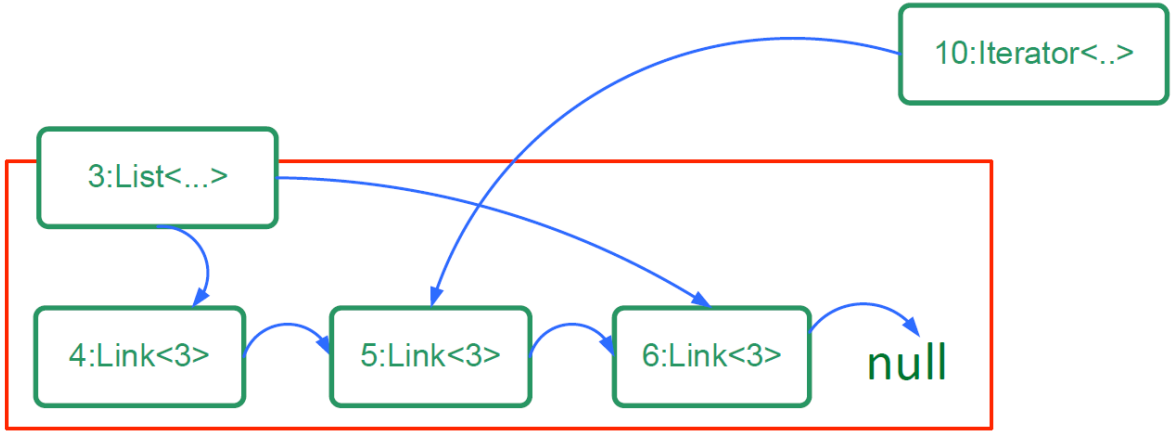
a) The invariants of an object $\circ$ may only depend on the values of the fields of the objects owner by $\circ$.
b) Only the owner may modify the fields of the objects which it owns.
c) Any method called on an object may assume upon entry that the receiver's invariant holds, and must establish at the end of the method body that it has restored the invariant.

INVariants, and Owners as Modifierfs

```
class List<o>{
      Link<this> first, last;
      INV last.next == null ∧ first.next*== last;
      …
}
class Link<o>{
      Link<o> next;
       …  }
class Iterator{  Link<_> current; … }
```



INVariants, and Owners as Modifierfs - 2



If `lnk1` points to `4`, then

`link1` does not own `this.next`

`link1` may NOT execute

```
this.next.next:=this.next;
```

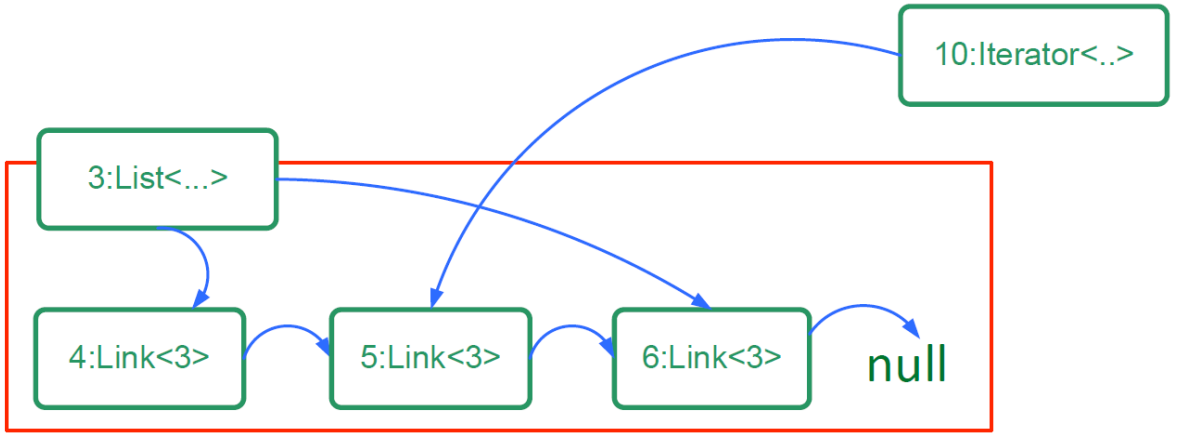
If `iter1` points to `10`, then

`iter1` does not own `this.current`

`iter1` may NOT execute

```
this.current.next := null;
```

INVariants, and Owners as Modifierfs - 3



If `lst2` points to `3`, then it MAY execute

`iter2` owns `this.first.next`

it MAY execut

```
this.first.next.next := null;
```

but then, the Owners as Modifiers discipline also forces it to establish the INVariant, eg by executing:

```
this.last := this.first.next;
```

Enforcing Owners as Modifiers through the type system

In traditional oo types we have

$$\frac{e: C \qquad e' : D \qquad \textbf{class}\ C\{\ \ldots\ D\ \ f\ \ldots.\ \}}{e.f = e' : D}$$

Enforcing Owners as Modifiers through the type system

In ownership type systems we have

```
e: C<q1, ….qn>
class C<p1,…pn> {  … D<r1, …. rm> f …. }
e’ : D<r1,…rm[q1,….qn/p1….pn]>
r1 = this
-------------------------------------
e.f = e’: D<r1,…rm[q1,….qn/p1….pn]>
```

Blocking

- The cause of many high-variance slowdowns
  - More cores → more slowdowns and more variance
    - Blocking Garbage Collection accentuates impact
- Reducing blocking
  - Help perform prerequisite action rather than waiting for it
  - Use finer-grained sync to decrease likelihood of blocking
  - Use finer-grained actions, transforming ...
    From: Block existing actions until they can continue
    To: Trigger new actions when they are enabled
- Seen at instruction, data structure, task, IO levels
  - Lead to new JVM, language, library challenges
    - Memory models, non-blocking algorithms, IO APIs

http://gee.cs.oswego.edu