Ownership Types for Object Encapsulation

Chandrasekhar Boyapati
Barbara Liskov
Liuba Shrira

POPL 2003

- Motivation
- Ownership Type
- Subtyping and the problem
- Solution--Inner Class (major contribution)
- Effect
- Application & Summary

Motivation

- Goal is local reasoning about correctness
  - Prove a class meets its specification, using only specifications but not code of other classes
  - Requires no interference from code outside the class
  - Objects must be encapsulated

Motivation(cont'd)

- Three major relationships between the classes in ORD (Object Relation Diagram)
  - Inheritance
  - Association
  - Aggregation
- But modern OO programming(Java, C++, C#) languages don't support aggregation explicitly

Motivation(cont'd)

- UML
  - Aggregation: A special form of association that specifies a whole–part relationship between the aggregate (whole) and a component part.
  - Composition: A form of aggregation which requires that a part instance be included in at most one composite at a time, and that the composite object is responsible for the creation and destruction of the parts.
  - Both are transitive, and anti–symmetric,ir–reflexive
- Ownership corresponds to composition in UML

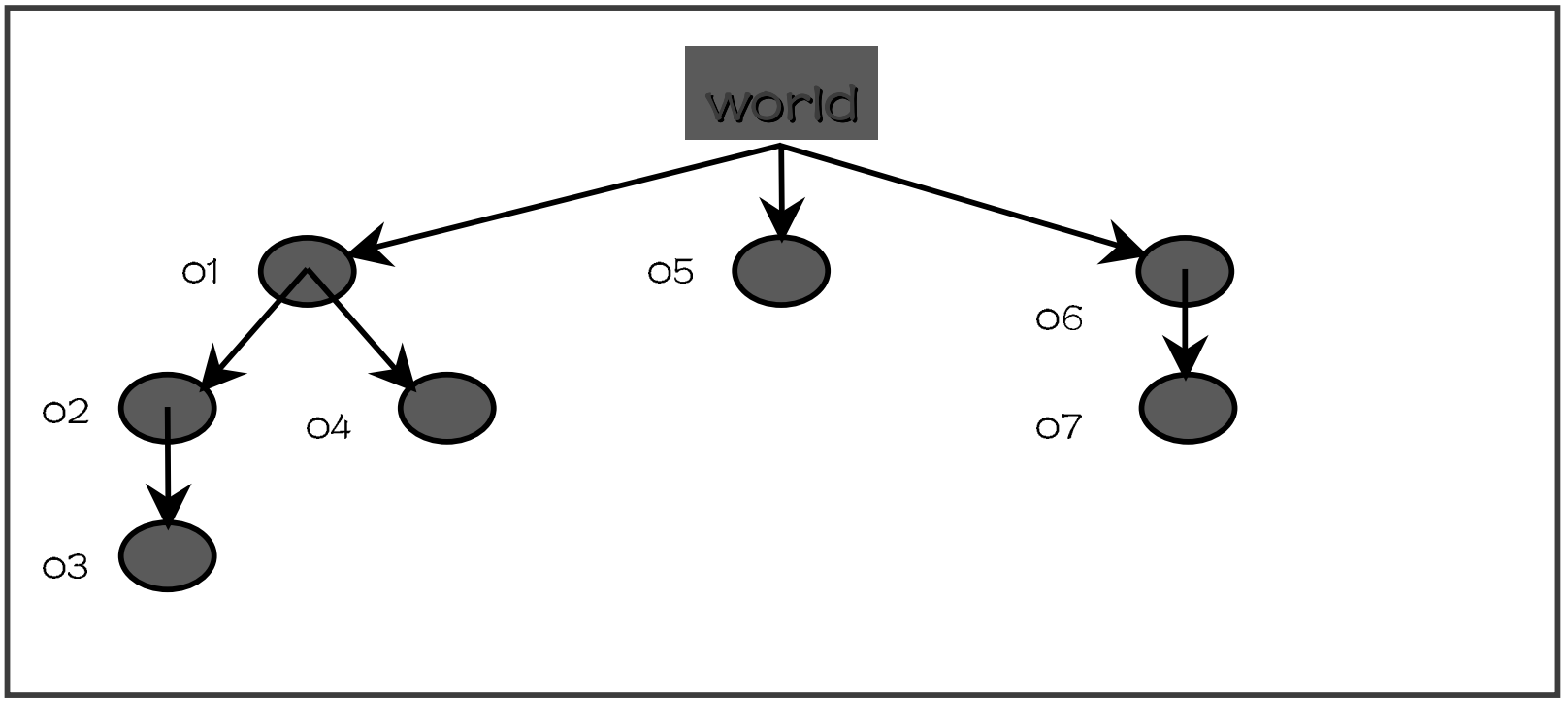
- Motivation
- Ownership Type
- Subtyping and the problem
- Solution--Inner Class (major contribution)
- Effect
- Application & Summary

Ownership Types

- Properties
  - P1:Every object has an (direct) owner
  - P2:Owner can be another object or world
  - P3:Ownership relation forms a tree
  - P4:Owner of an object cannot change

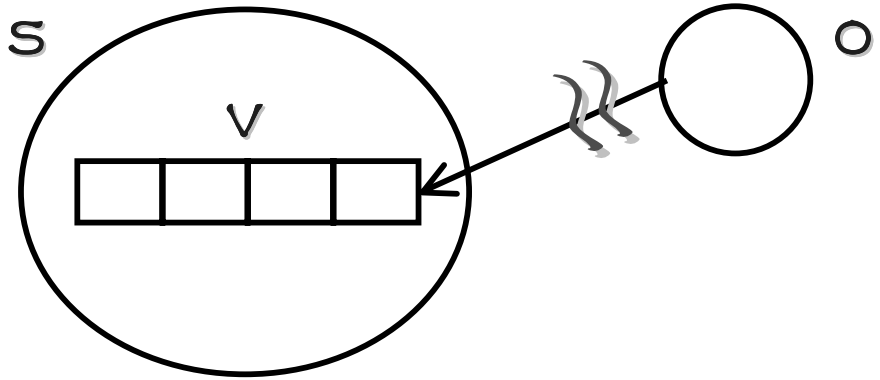
- An Object is only allowed to access
  - Itself and objects they (directly) own
  - Its (transitive) ancestors and objects it (directly) owns
  - Globally accessible objects

Ownership Types(example)



Object Encapsulation(Example)

- Consider a Set object s implemented using a Vector object v
- The ownership type system enforces encapsulation
  - If v is inside s and o is outside
  - Then o cannot access v

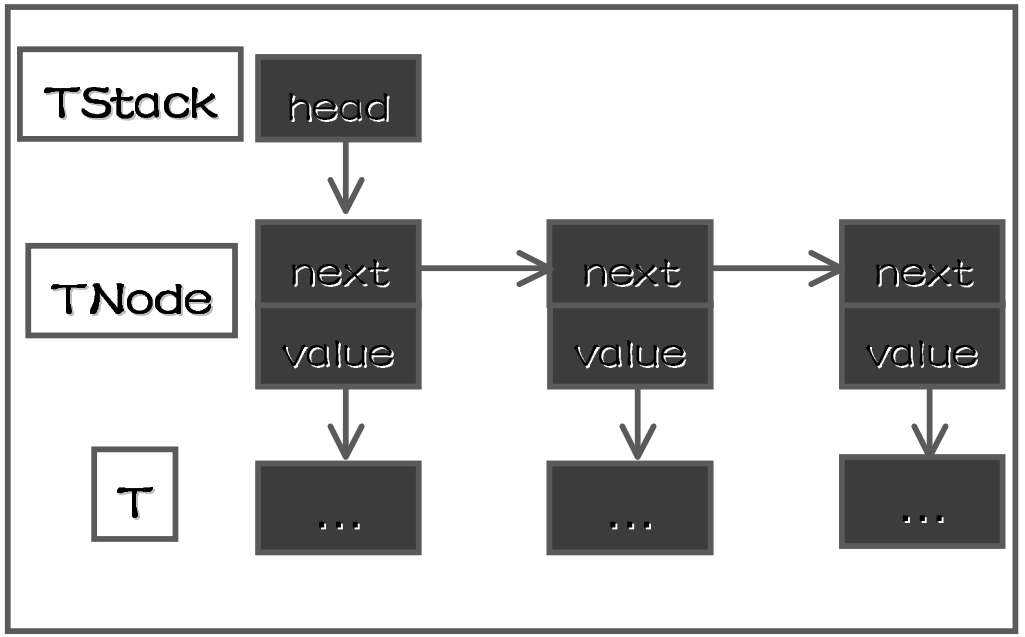


Ownership Types for Encapsulation

- Ownership allows a program to statically declare encapsulation boundaries that capture dependencies
- An object should own all the objects it depends on
  - Directly, Transitively
  - Overstatement……

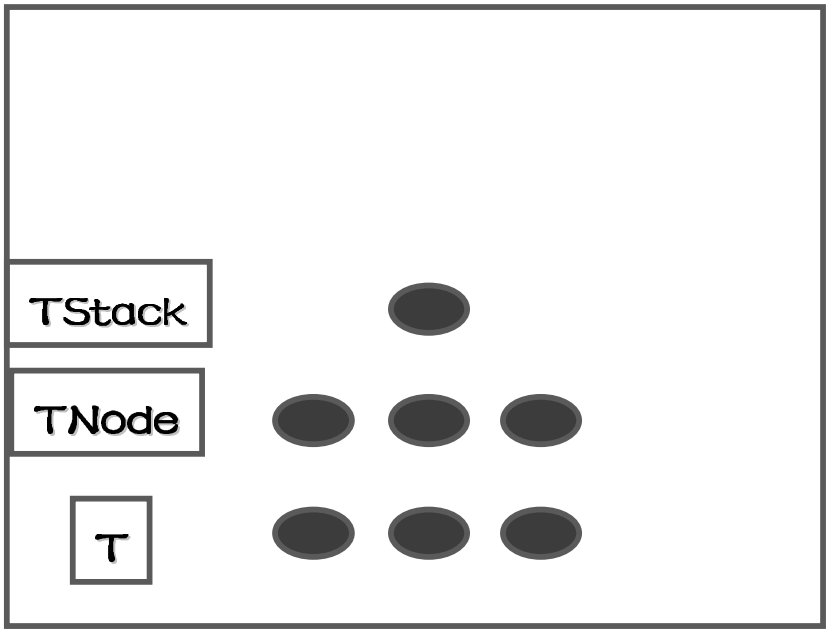
TStack Example (No Owners)

```
class TStack {
    TNode head;

    void push(T value) {…}
     T pop() {…}
}

class TNode {
    TNode next;
    T value;
    …
}

class T {…}
```



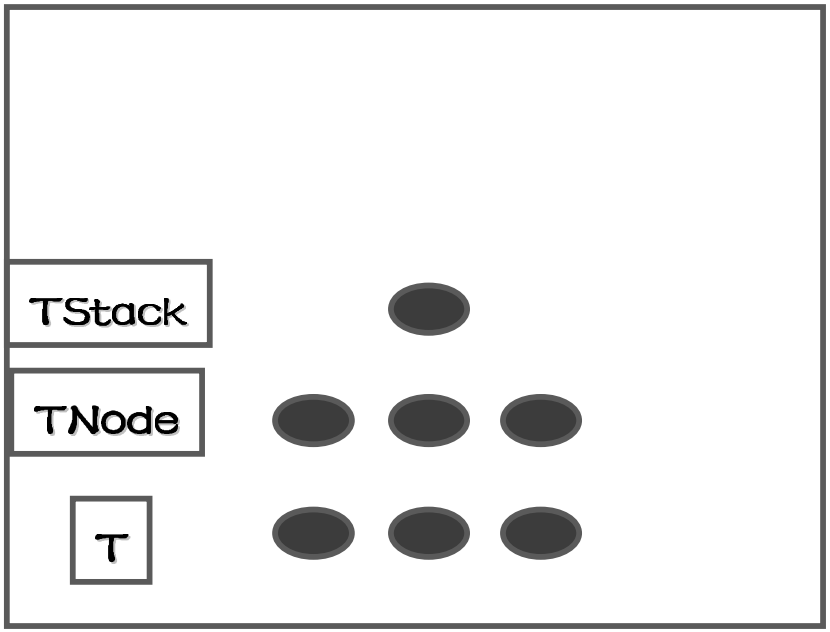
TStack Example (With Owners)

```
class TStack⟨stackOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
}
class TNode⟨nodeOwner, TOwner⟩ {
    TNode⟨nodeOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
}
class T ⟨TOwner⟩ {…}
```



TStack Example

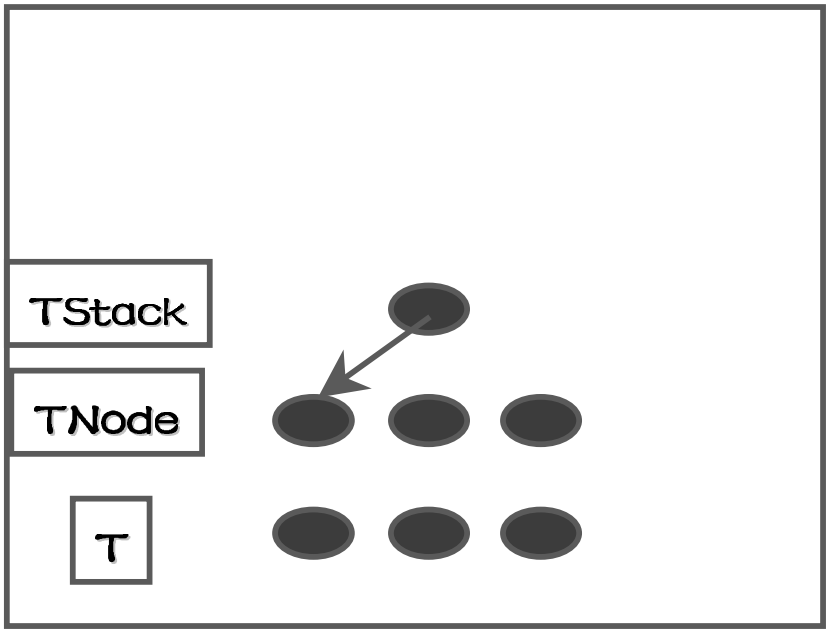
```
class TStack⟨stackOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
}
class TNode⟨nodeOwner, TOwner⟩ {
    TNode⟨nodeOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
}
class T ⟨TOwner⟩ {…}
```



First owner owns the "this" object

TStack Example

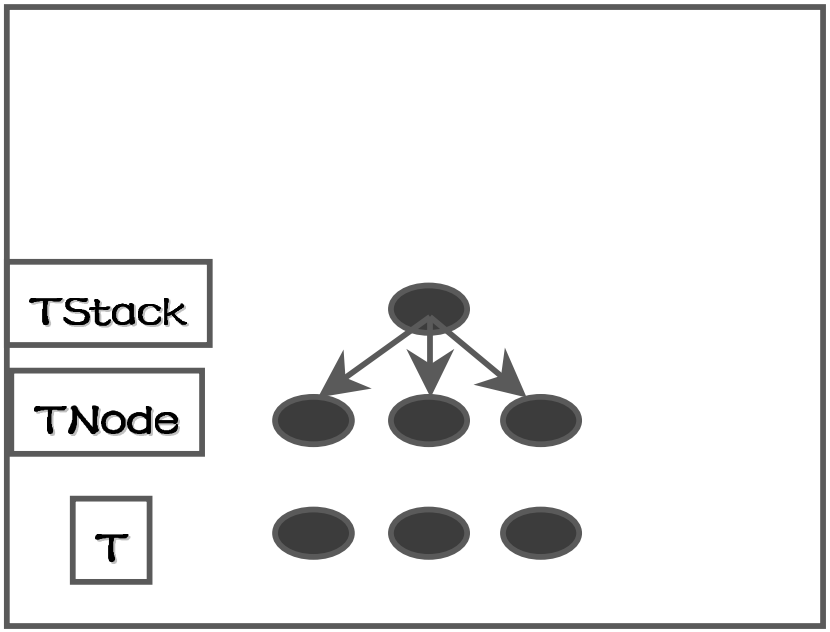
```
class TStack⟨stackOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
}
class TNode⟨nodeOwner, TOwner⟩ {
    TNode⟨nodeOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
}
class T ⟨TOwner⟩ {…}
```



TStack owns the "head" TNode

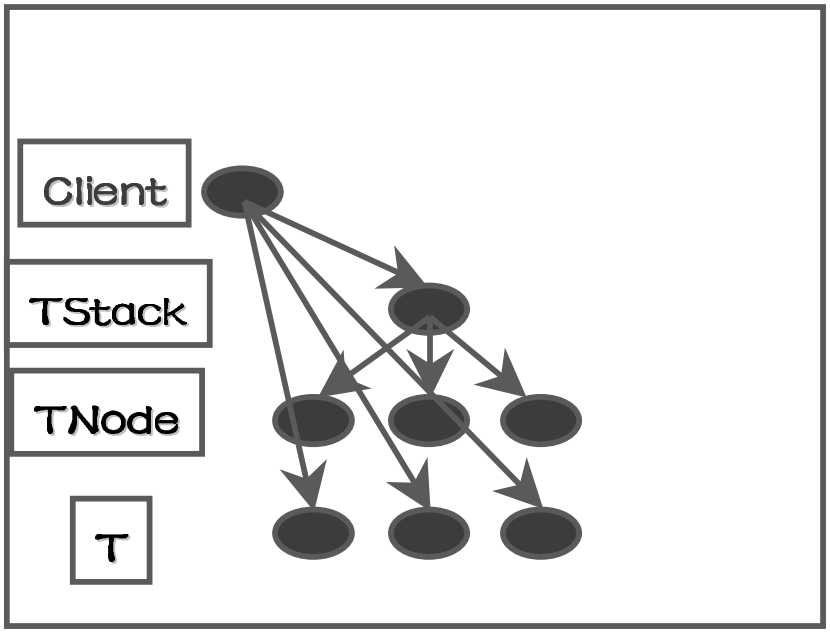
TStack Example

```
class TStack⟨stackOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
}
class TNode⟨nodeOwner, TOwner⟩ {
    TNode⟨nodeOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
}
class T ⟨TOwner⟩ {…}
```



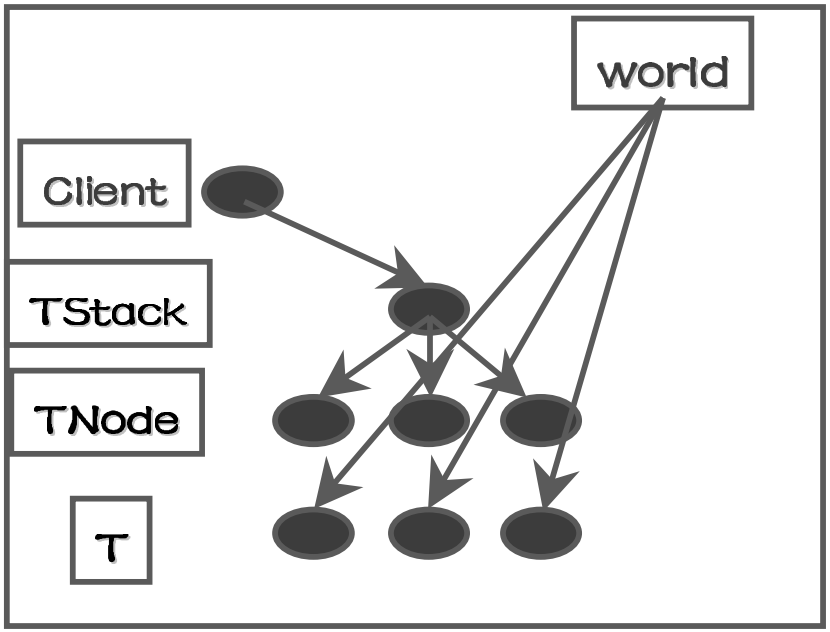
TStack Example

```
class TStack⟨stackOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
}
class TNode⟨nodeOwner, TOwner⟩ {
    TNode⟨nodeOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
}
class Client⟨clientOwner⟩ {
    TStack⟨this,  this⟩  s1;
    TStack⟨this,  world⟩ s2;
    TStack⟨world, world⟩ s3;
}
```



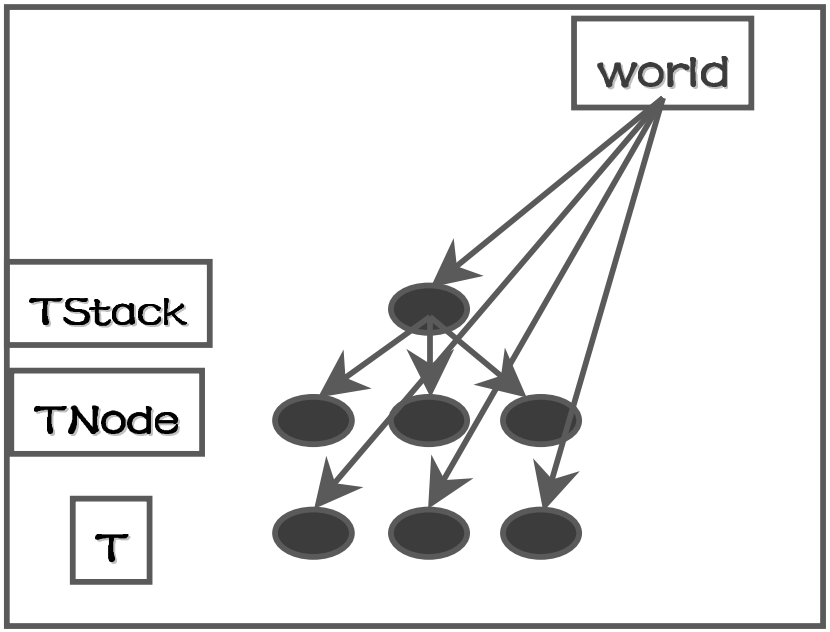
TStack Example

```
class TStack⟨stackOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
}
class TNode⟨nodeOwner, TOwner⟩ {
    TNode⟨nodeOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
}
class Client⟨clientOwner⟩ {
    TStack⟨this,    this⟩    s1;
    TStack⟨this,    world⟩ s2;
    TStack⟨world, world⟩ s3;
}
```



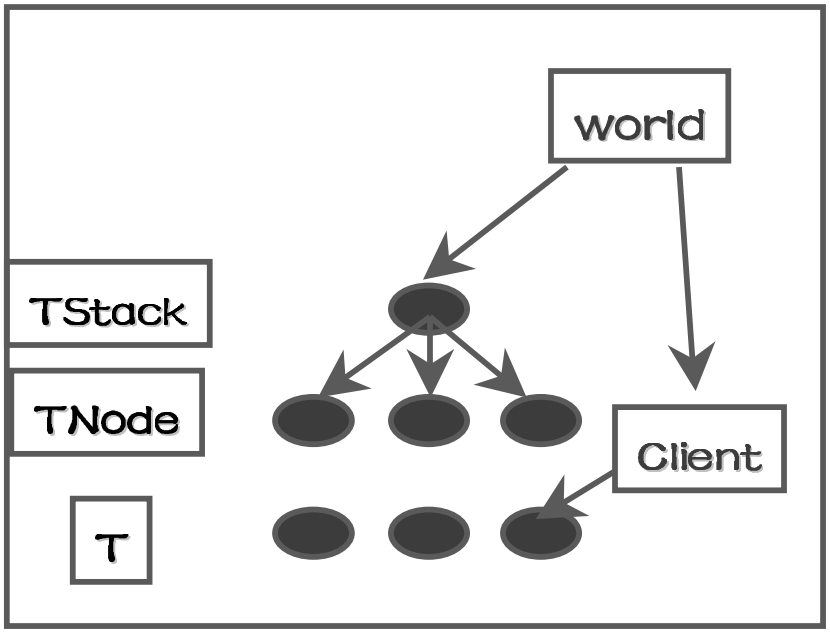
TStack Example

```
class TStack⟨stackOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
}
class TNode⟨nodeOwner, TOwner⟩ {
    TNode⟨nodeOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
}
class Client⟨clientOwner⟩ {
    TStack⟨this,    this⟩    s1;
    TStack⟨this,    world⟩ s2;
    TStack⟨world, world⟩ s3;
}
```



TStack Example

```
class TStack⟨stackOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
}
class TNode⟨nodeOwner, TOwner⟩ {
    TNode⟨nodeOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
}
class Client⟨clientOwner⟩ {
    TStack⟨this,  this⟩  s1;
    TStack⟨this,  world⟩ s2;
    TStack⟨world, world⟩ s3;
    TStack⟨world, this⟩  s4;  // illegal
}
```



The first owner <=The second owner

Subtyping

- The first owner parameter of the supertype must be the same as the subtype
- Thus T(TOwner) is not a subtype of Object(World)!!!

Problem! Iterator

- Consider an Iterator i over Stack s
- If i is encapsulated within s
  - Then i cannot be implemented by extending the existing (general) Iterators outside s
  - i can't be used outside s
- If i is not encapsulated within s
  - Then i cannot access representation of s

- Motivation
- Ownership Type
- Subtyping and the problem
- Solution--Inner Class (major contribution)
- Effect
- Application & Summary

Solution

- Inner Class
  - Previous ownership type combine the inner class with the ownership
- An inner class is parameterized with owners just like a regular class, but it is not necessarily the same as the container class
- Thus the Iterator in stack s can extends the existing iterators outside s

Innerclass

- The inner class must explicitly include the outer class parameter in its declaration in order to use it inside.
- Theorem: X can access an object owned by o only if
  - 1) x<=o or
  - 2) x is an inner class object of o

Proof

- Because the outer class can access the object instantiated from the inner class, so the they should prove that the inner class's direct owner is the outer class's ancestor
- Confusion:
  - What is exactly enumOwner?
  - f<=o, why? The point is that C.this can access o, so f<=o or f directly own o

- Motivation
- Ownership Type
- Subtyping and the problem
- Solution--Inner Class (major contribution)
- Effect
- Application & Summary

Effect

- Reads (r) writes (w)
  - The method can write an object x only if x <=w
  - The method can read an object x only if x <=r
- Ownership types and effects can be used to locally reason about the side effects of method calls
- Not contribution of this paper

- Motivation
- Ownership Type
- Subtyping and the problem
- Solution--Inner Class (major contribution)
- Effect
- Application & Summary

Application of Ownership Type

- Lazy Modular Upgrades in Persistent Object Stores
  - Boyapati, Liskov, Shrira, Moh, Richman(OOPSLA '03)
- Ownership Types for Safe Programming: Preventing Data Races and Deadlocks
  - Boyapati, Lee, Rinard (OOPSLA '01) (OOPSLA '02)
- Ownership Types for Safe Region-Based Memory Management in Real-Time Java
  Boyapati, Salcianu, Beebee, Rinard (PLDI '03)

Summary(from the Author)

- Ownership types capture dependencies
- Extension for inner class objects allows iterators and wrappers
- Approach provides expressive power, yet ensures modular reasoning
- Effects clauses enhance modular reasoning