
Trust the Clones
KaWai Cheng, Sophia Drossopoulou
James Noble

Trust the clones – What?

- Cloning of objects requires cloning of some (not all) fields of that object – sheep cloning.

- Therefore, programmer needs to write custom cloning code for each class; this is tedious, error-prone boilerplate code.

- We propose annotations of the field declarations, and that be used to generate the code.

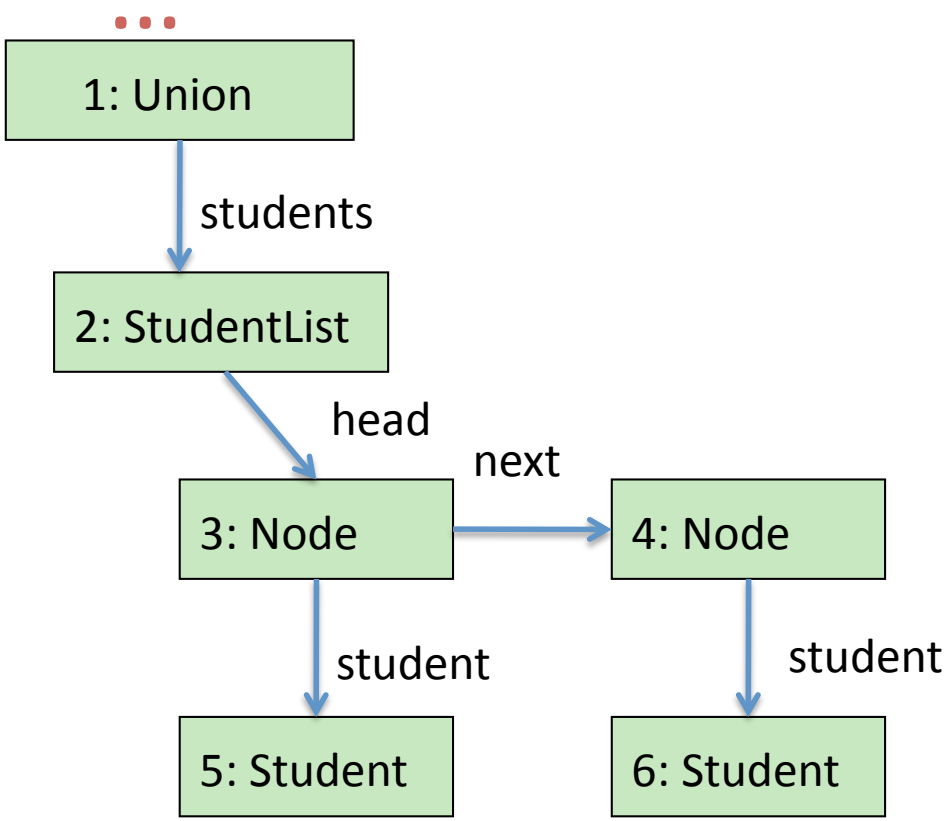
Contents of the talk

- Example of sheep cloning
- Annotating code with cloning information
  - Path Types
  - Cloning Domains
- Generation of the cloning methods
  - The cloning method for the Node example
  - The cloning method in general
- Properties of Cloning Methods

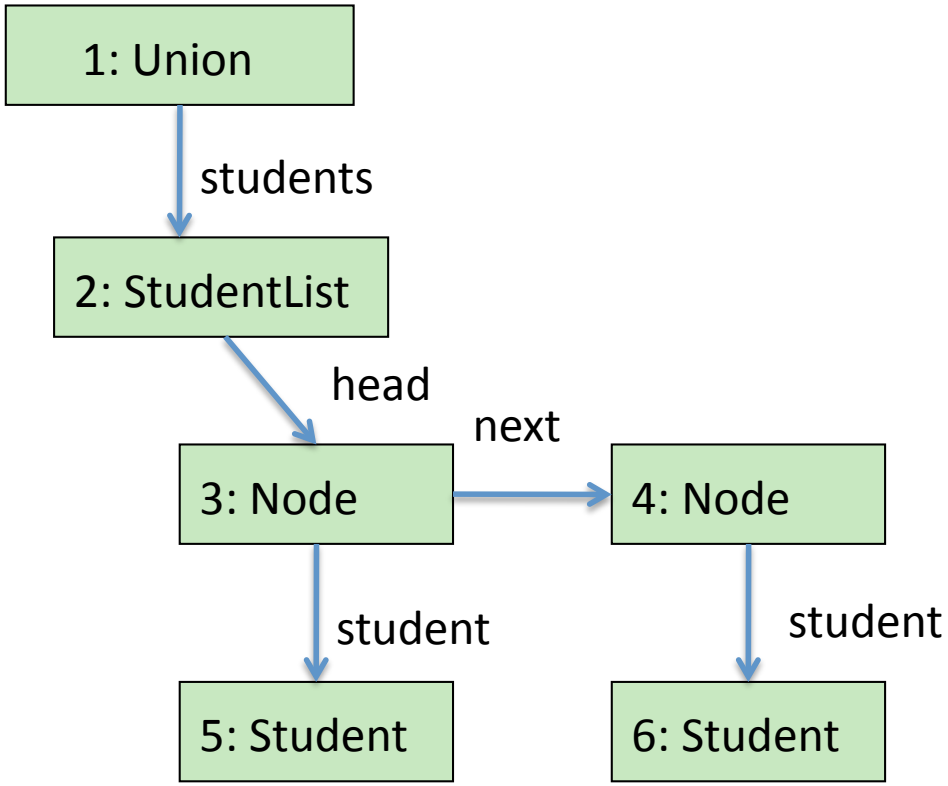
Contents of the talk

- Example of sheep cloning
- Annotating code with cloning information
  - Path Types
  - Cloning Domains
- Generation of the cloning methods
  - The cloning method for the Node example
  - The cloning method in general
- Properties of cloning methods

Example: assuming following objects

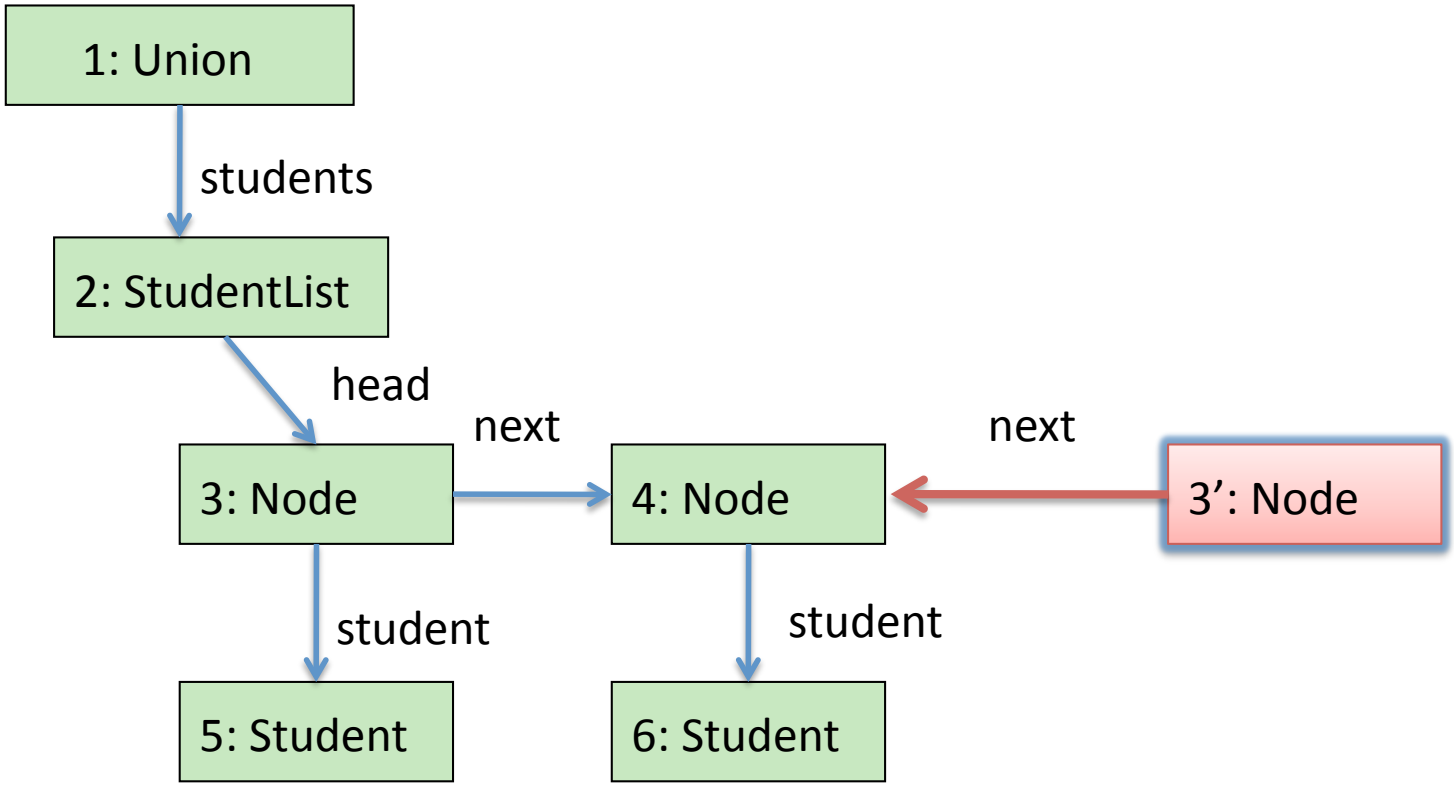


Example: … described by following classes

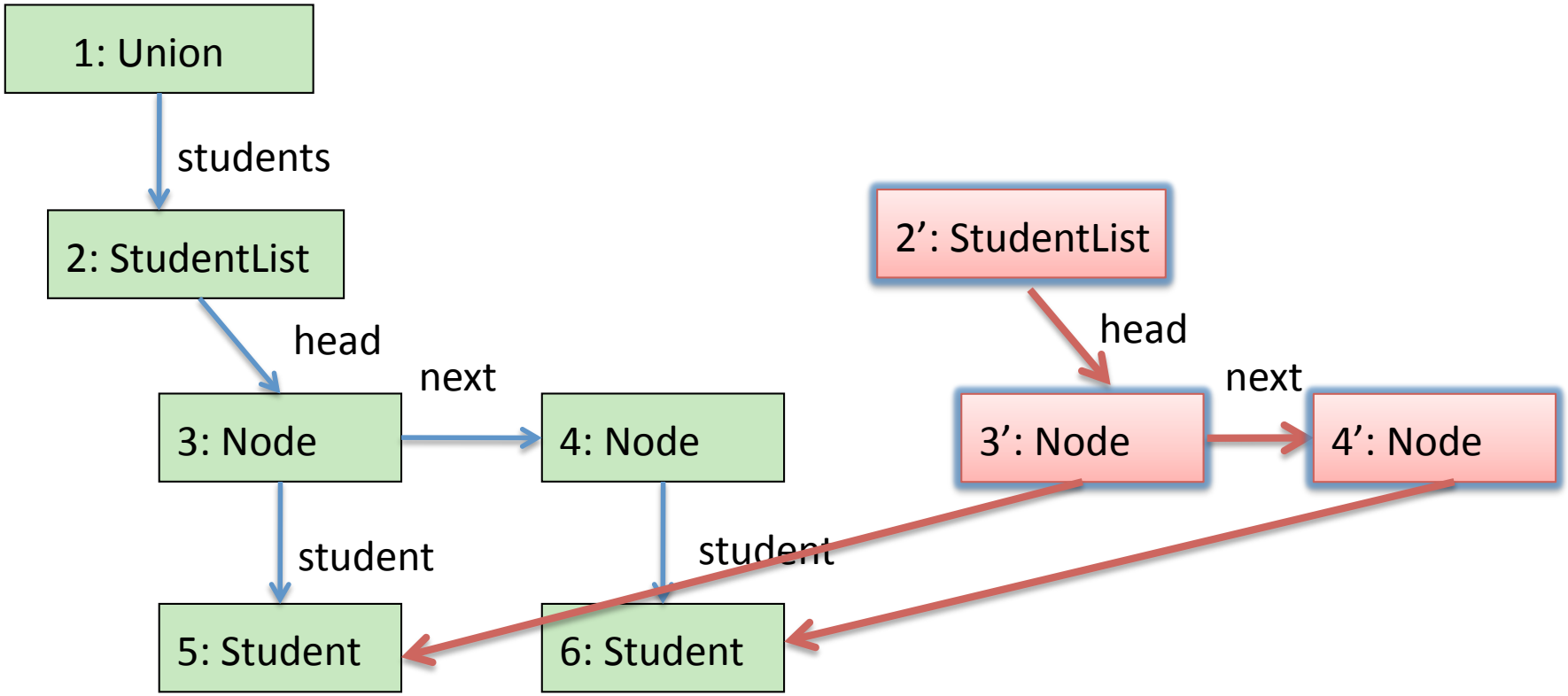



```
class Union{
    StudentList students;
    Union clone(){ ??? }
}

class StudentList{
    Node head;
    StudentList clone()
          { ??? }
}

class Node{
     Node next;
     Student student;
     Node clone(){ ??? }
}

class Student{
     Student clone(){ ??? }
}
```

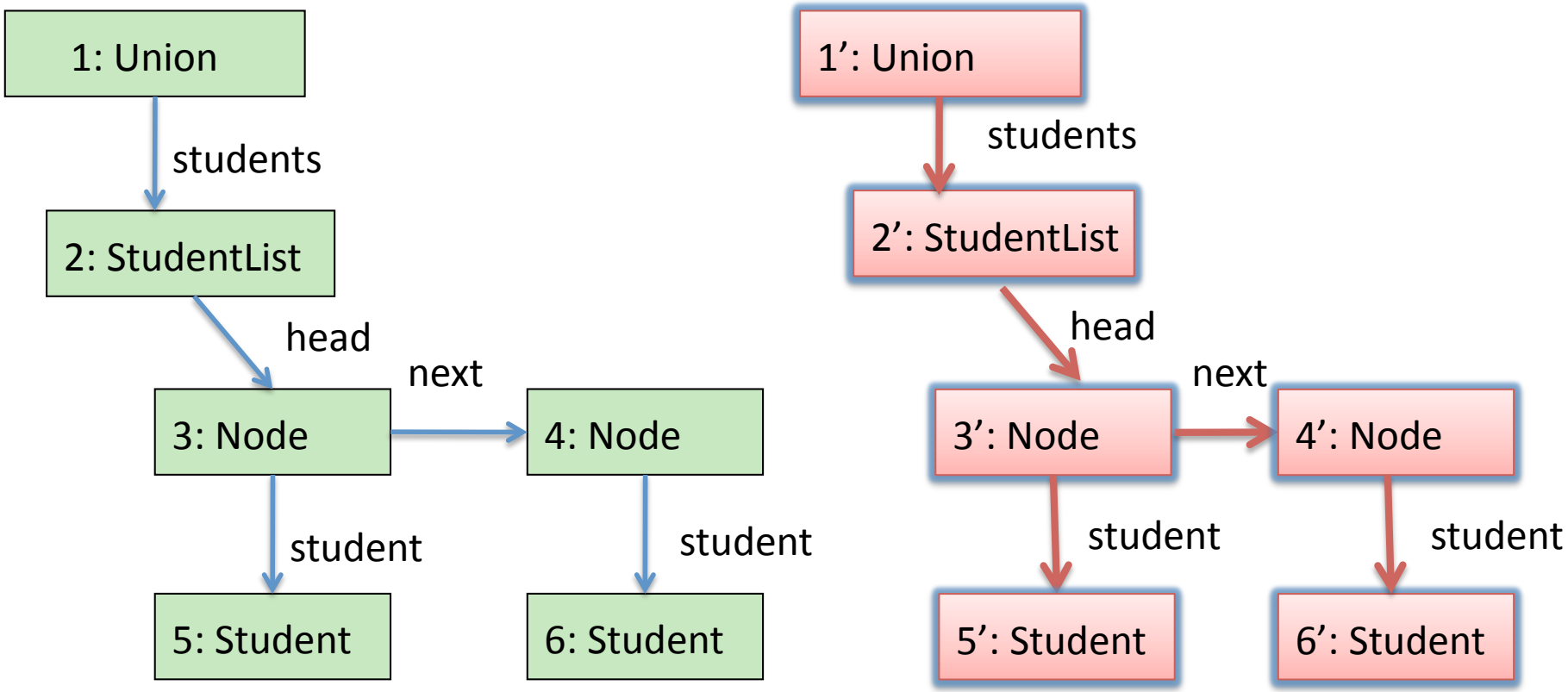
Example: assume that cloning 3, just duplicates 3



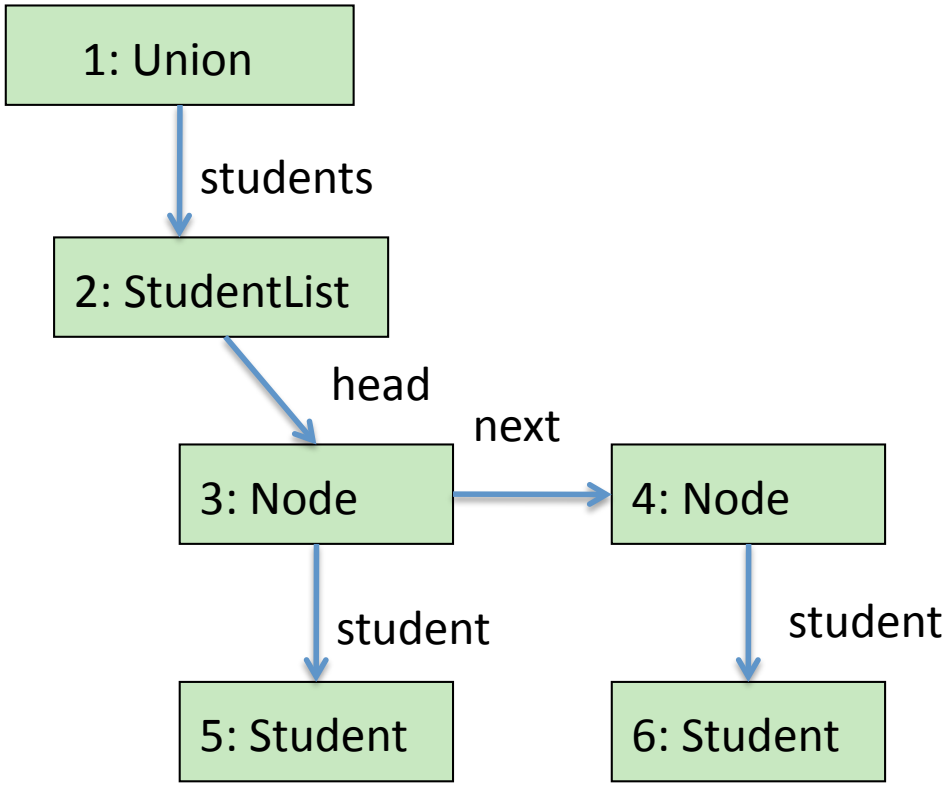
Example: assume that cloning 2, duplicates 2, 3, 4



Example: assume that cloning 1, duplicates 1, 2, 3, 4, 5, 6

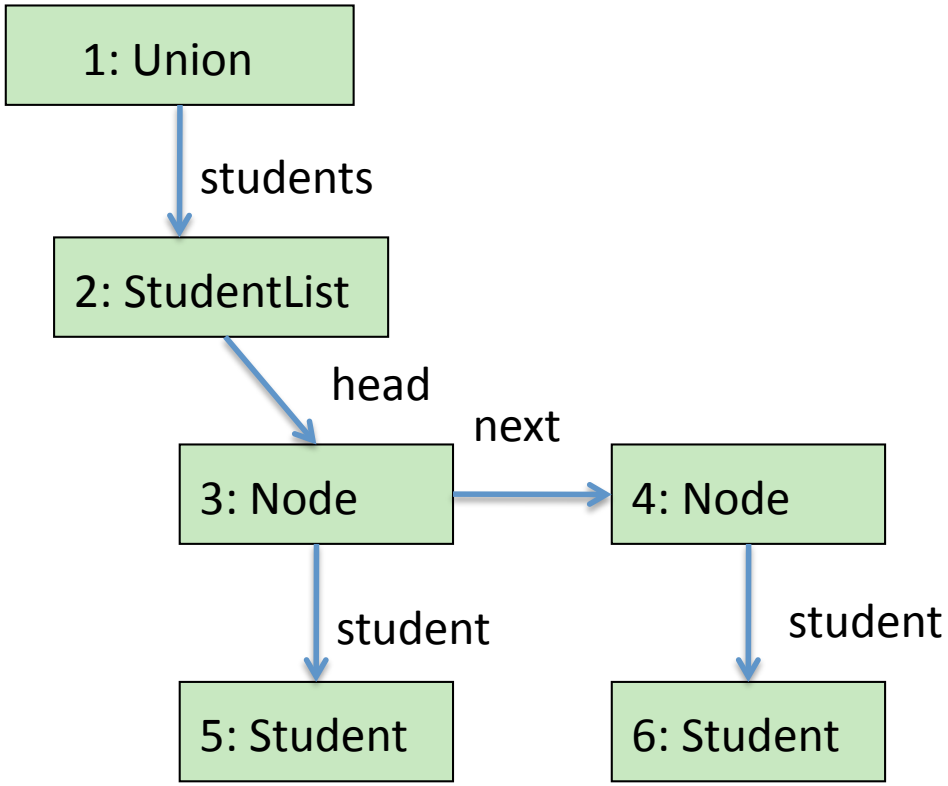


So, how can we write `clone()` so that …



- `1.clone()`
  duplicates `1, 2, 3, 4, 5, 6`
- `2.clone()`
  duplicates `2, 3, 4`
- `3.clone()`
  duplicates `3,` and
- `4.clone()`
  duplicates `4`
- `5.clone()`
  duplicates `5,` and
- `6.clone()`
  duplicates `6`

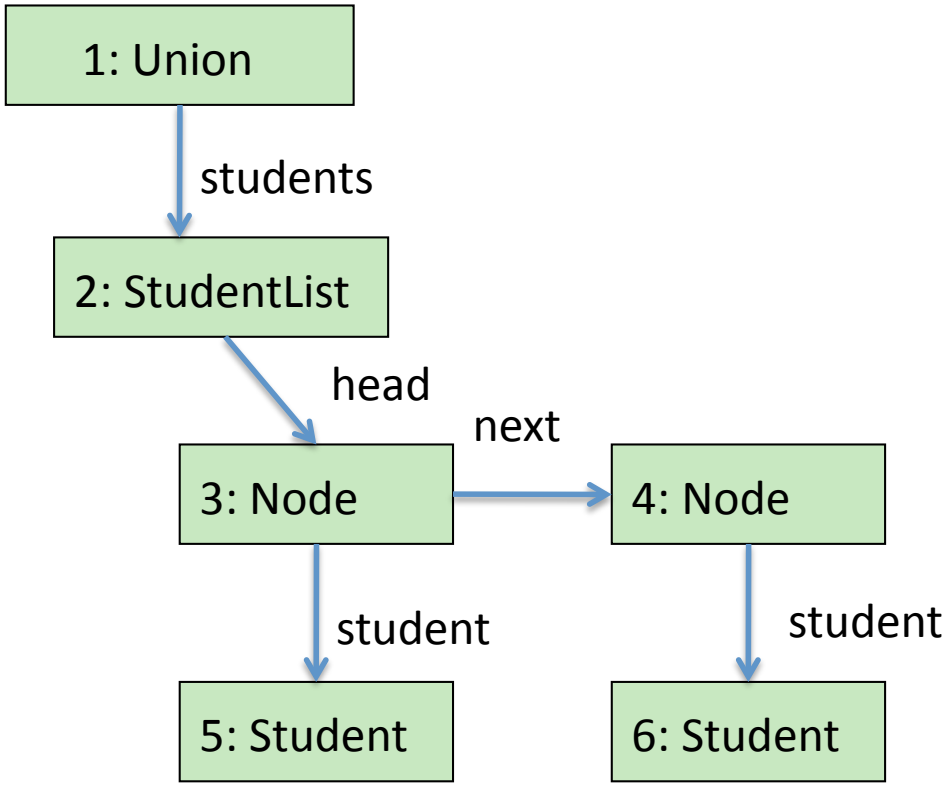
So, how can we write `clone()`?



"Naïve" Approach:

Write a different `clone()` method per class which "navigates" the structure, as needed for cloning.

So, how can we write `clone()`?



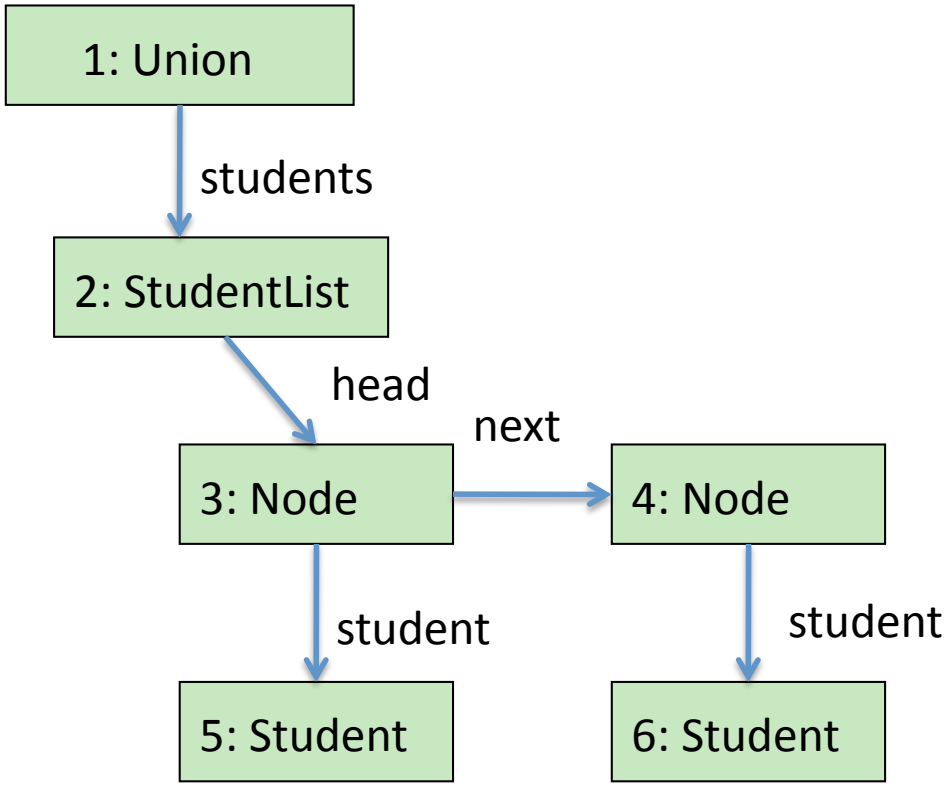
"Naïve" Approach:

Write a different `clone()` method per class which "navigates" the structure, as needed for cloning.

Therefore, the method `clone()` from class `Union` would read the `Node`'s fields. This exposes the internals of class `Node` to class `Union`.

☹ ☹ ☹

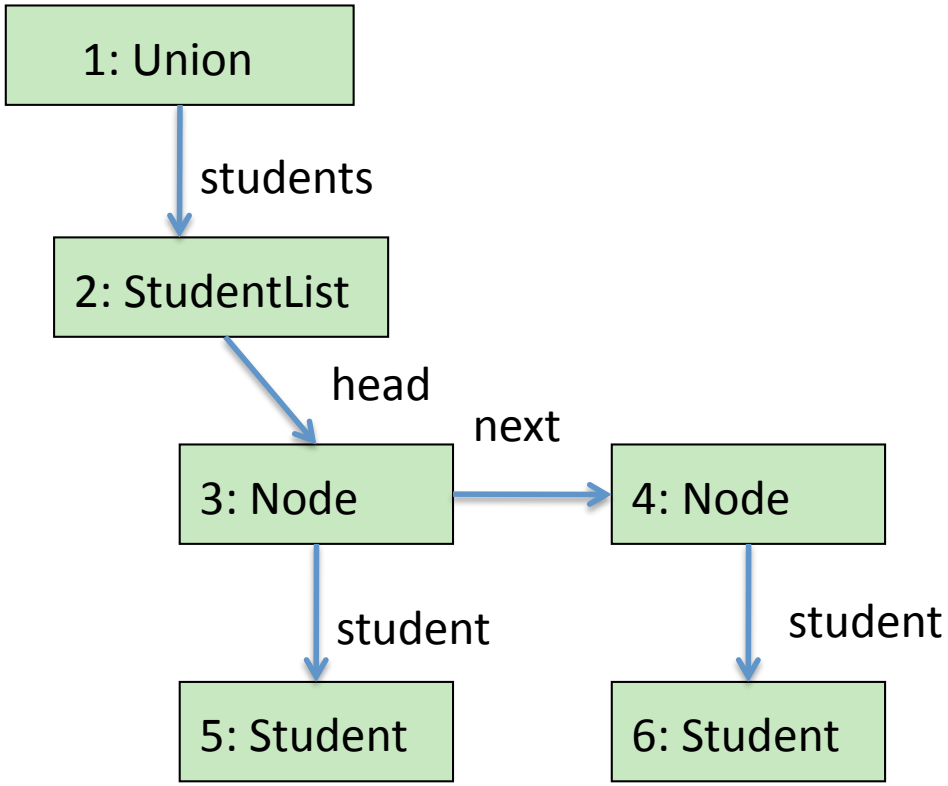
So, how can we write `clone()`?



"Better" Approach:

Write one `clone()` method per class and provide it with some context, so that it knows how much to duplicate. This method will contain further calls of `clone()` on some of object's fields and provide appropriate context.

So, how can we write `clone()`?

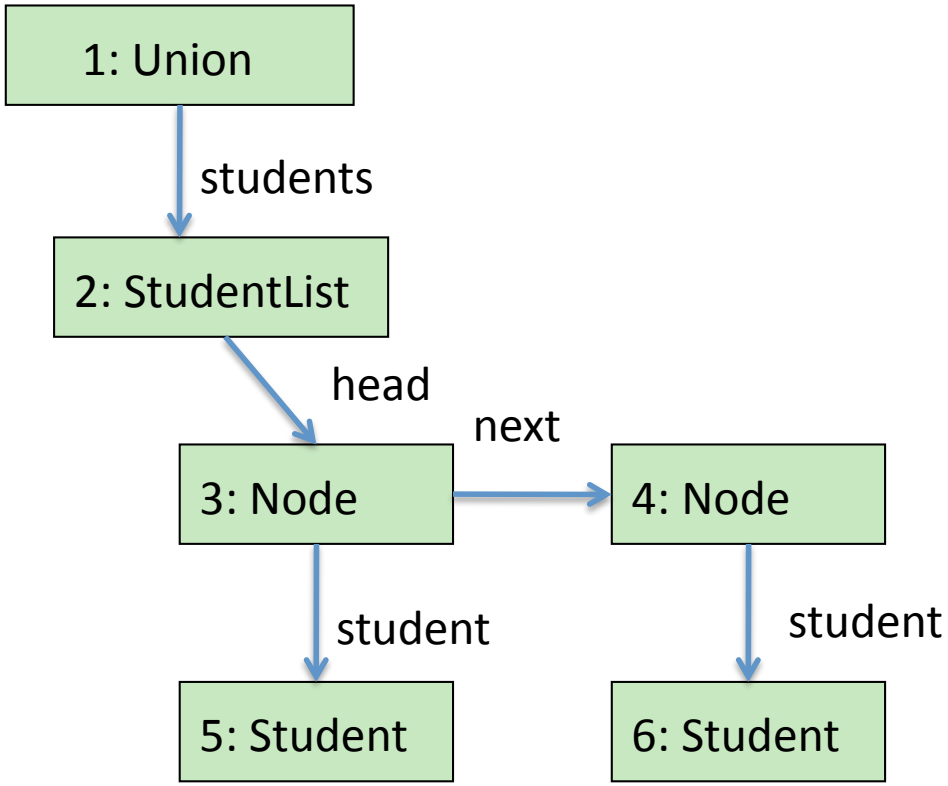


"Better" Approach:

Write one `clone()` method per class and provide it with some context, so that it knows how much to duplicate. This method will contain further calls of `clone()` on some of object's fields and provide appropriate context.

For example, `Node` will have three versions of `clone()`: one when the "originator" is it`self`; one when the originator is a `StudentList`; and one when the originator is a `Union`.

So, how can we write `clone()`?

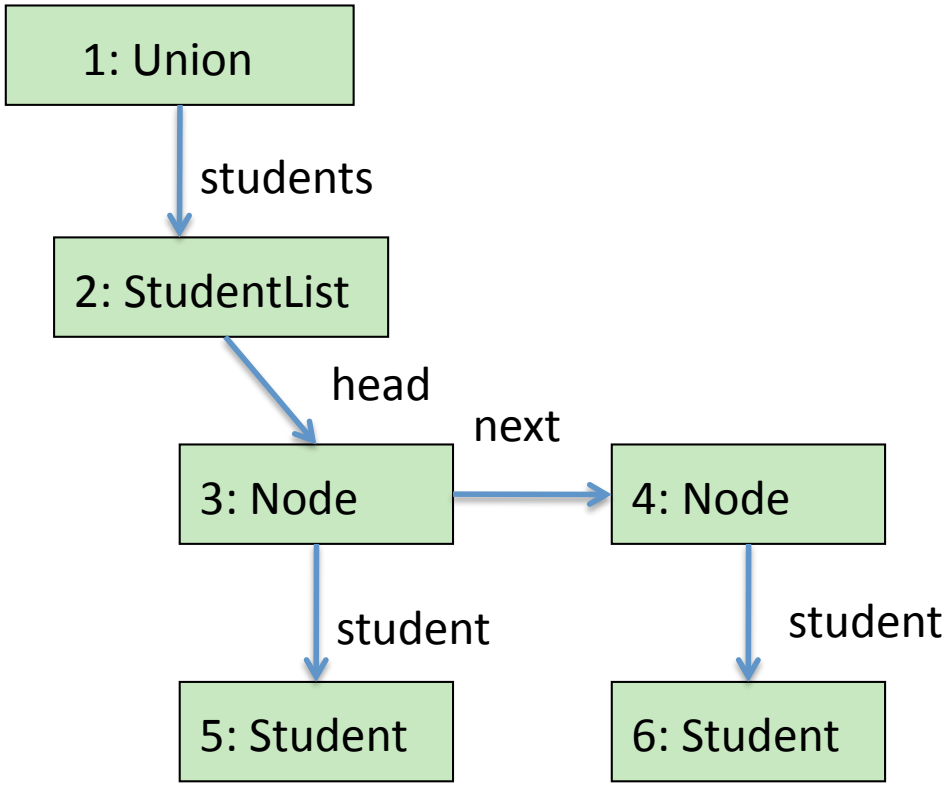


Our Approach:

A mechanism for the *generation* of clone methods.

Use annotations to indicate object membership to cloning domains.

So, how can we write `clone()`?



Our Approach:

A mechanism for the *generation* of clone methods.

Use type annotations to indicate object membership to cloning domains.

In other words, *put objects into boxes.*

Objects into Boxes started 14 years ago …

Flexible Alias Protection

James Noble[1], Jan Vitek[2], and John Potter[1]

[1] Microsoft Research Institute, Macquarie University, Sydney
kjx,potter@mri.mq.edu.au
[2] Object Systems Group, Université de Genève, Geneva.
Jan.Vitek@cui.unige.ch

Abstract. Aliasing is endemic in object oriented programming. Because an object can be modified via any alias, object oriented programs are hard to understand, maintain, and analyse. *Flexible alias protection* is a conceptual model of inter-object relationships which limits the visibility of changes via aliases, allowing objects to be aliased but mitigating the undesirable effects of aliasing. Flexible alias protection can be checked statically using programmer supplied *aliasing modes* and imposes no run-time overhead. Using flexible alias protection, programs can incorporate mutable objects, immutable values, and updatable collections of shared objects, in a natural object oriented programming style, while avoiding the problems caused by aliasing.

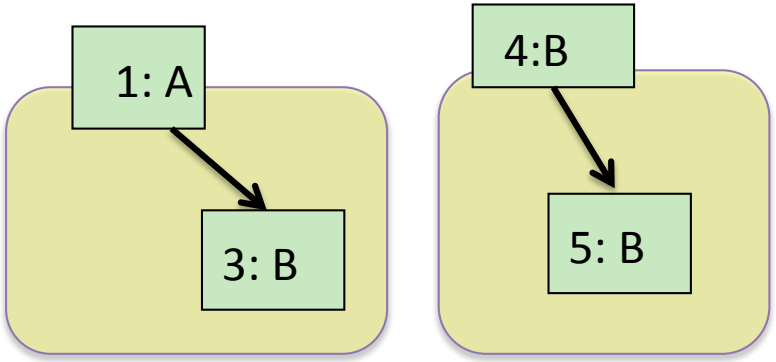
1 Introduction

I am who I am; I will be who I will be.

Object identity is the foundation of object oriented programming. Objects are useful for modelling application domain abstractions precisely because an object's identity always remains the same during the execution of a program —

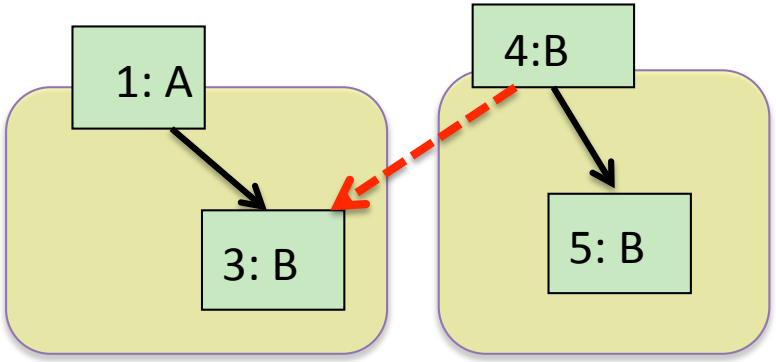
Boxes owned by objects…

- Each object is owned by another object.
- Each owner has a set of objects it owns.
- The ownership relation therefore introduces a tree hierarchy.



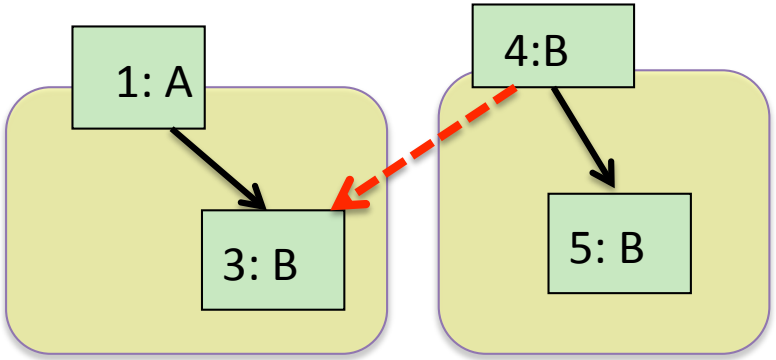
The flavours of owners …

- Owners as *dominators* (for encapsulation, garbage collection)
- Owners as *modifiers* (for reasoning; invariant depends on owned objects)



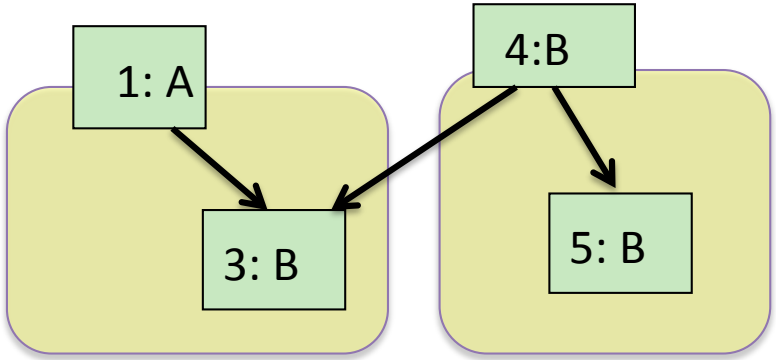
The flavours of owners …

- Owners as *dominators*
  (for memory management, encapsulation)
- Owners as *modifiers*
  (for reasoning; invariant depends on owned objects)



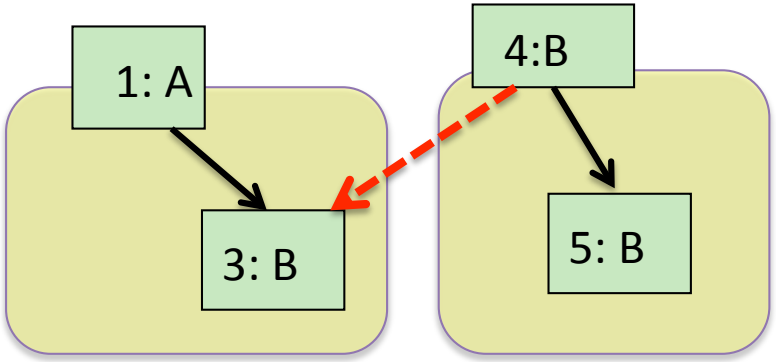
- Owners for software *architecture*

- Owners for *effects*
  (for concurrency, reasoning)

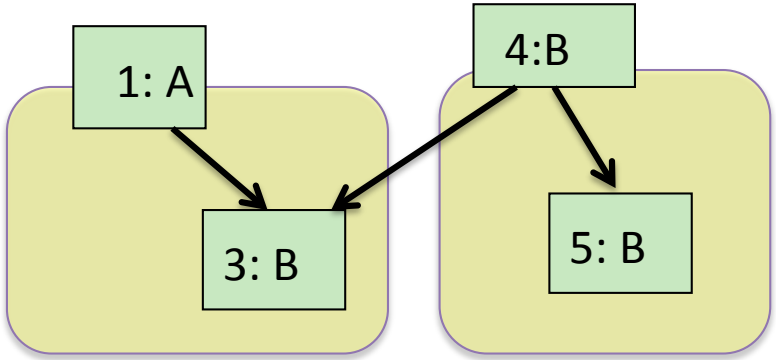


The flavours of owners …

- Owners as *dominators*
  (for memory management, encapsulation)
- Owners as *modifiers*
  (for reasoning; invariant depends on owned objects)



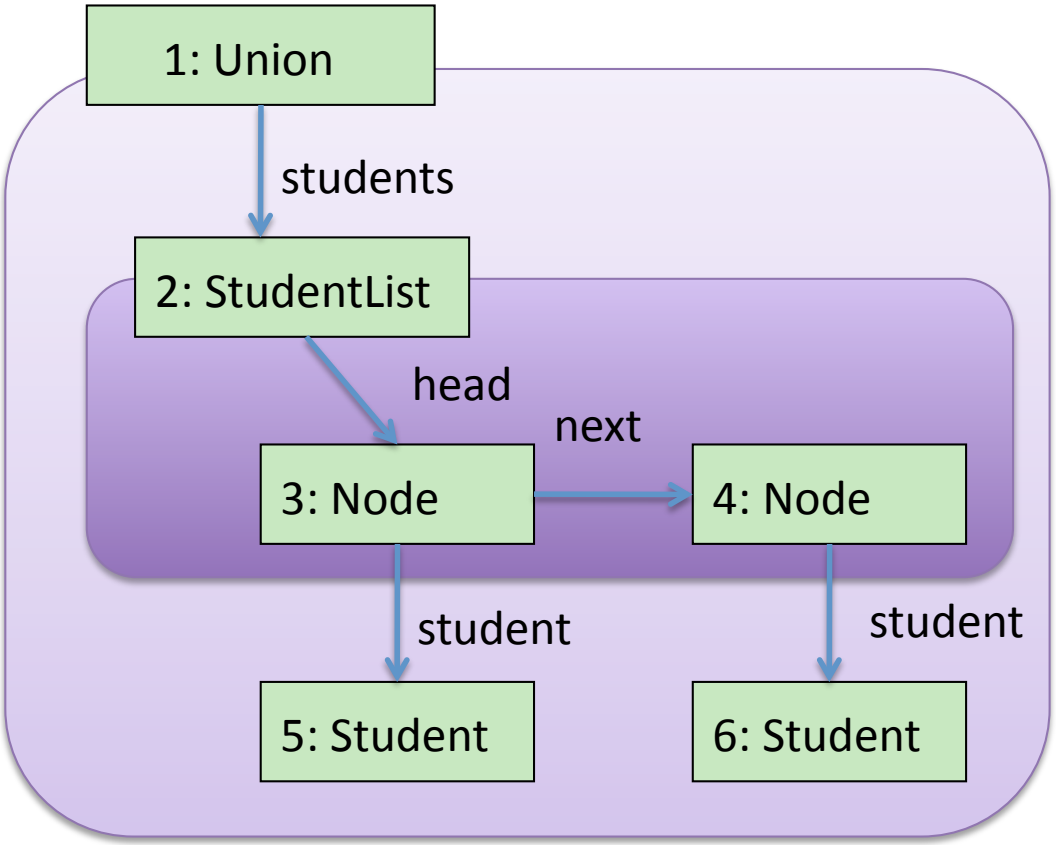
- Owners for software *architecture*
- Owners for *effects*
  (for concurrency, reasoning)
- and more …



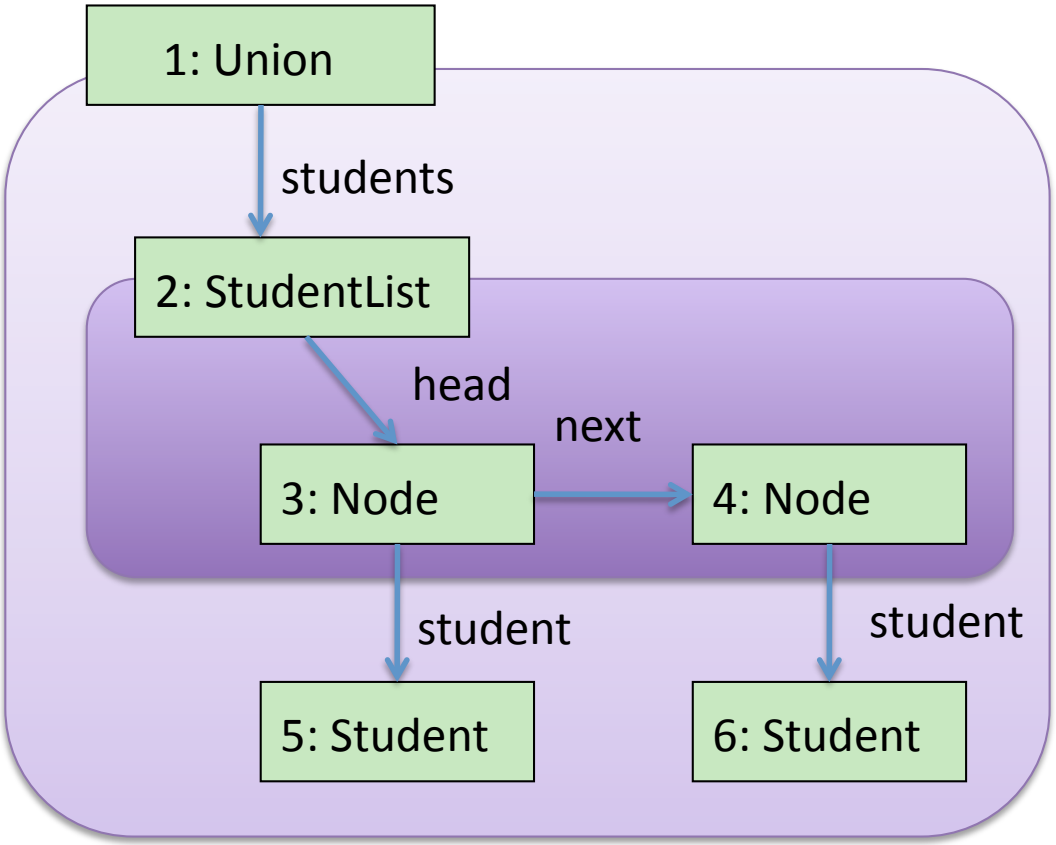
Contents of the talk

- Example of sheep cloning
- Annotating code with cloning information
  - Path Types
  - Cloning Domains
- Generation of the cloning methods
  - The cloning method for the Node example
  - The cloning method in genera;
- Properties of cloning methods
- What the underlying language must provide
- Conclusions

Putting objects into boxes …



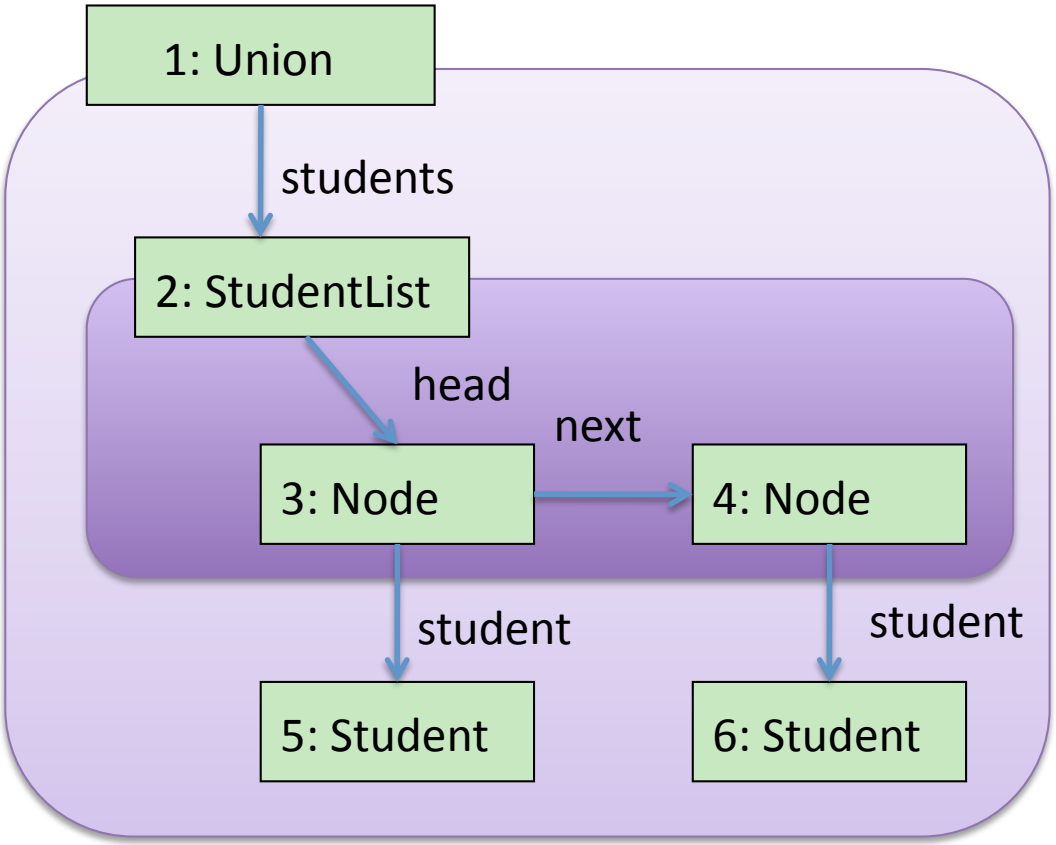
Putting objects into boxes …



means that

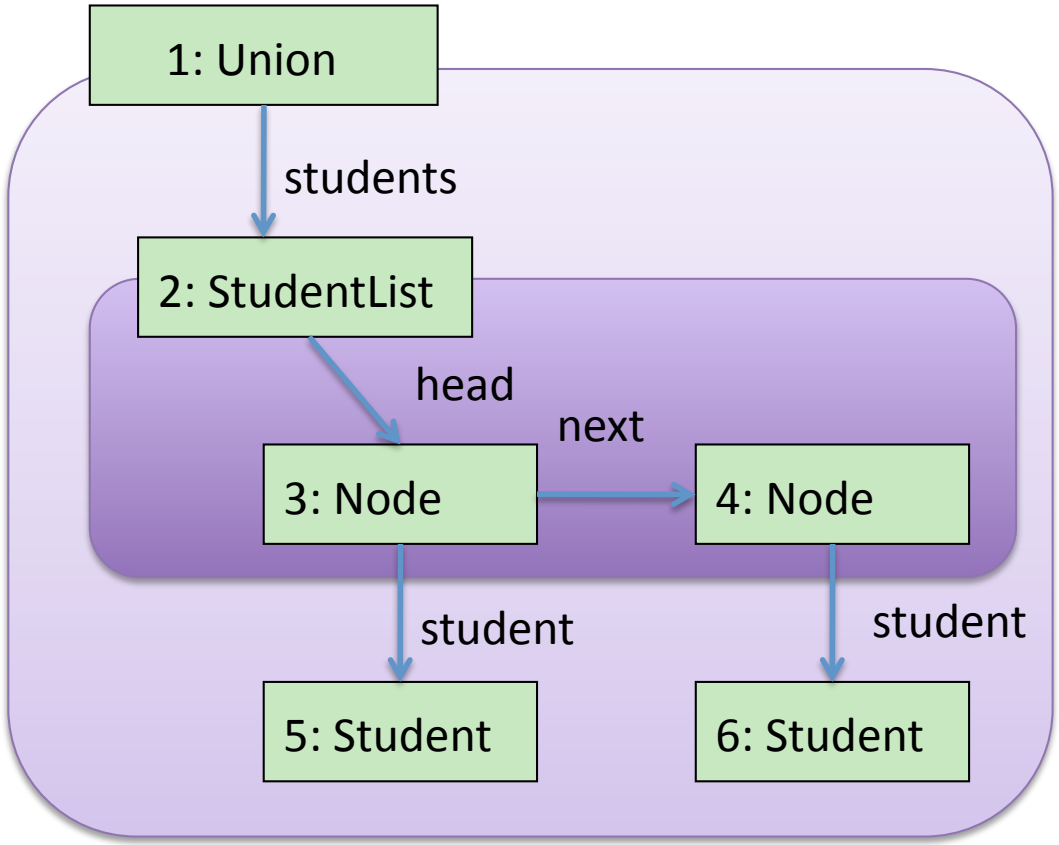
- 3 and 4 belong to the *cloning domain* of 2.
- 2, 3, 4, 5 and 6 belong to the *cloning* domain of 1.

Putting objects into boxes …




```
class Union<c>{
    StudentList<this> students;
    Union clone(){ ??? }
}

class StudentList<c>{
    Node<this,c> head;
    StudentList clone()
          { ??? }
}

class Node<c1,c2>{
     Node<c1,c2> next;
     Student<c2> student;
     Node clone(){ ??? }
}

class Student<c> {
     Student clone(){ ??? }
}
```

Putting objects into boxes …

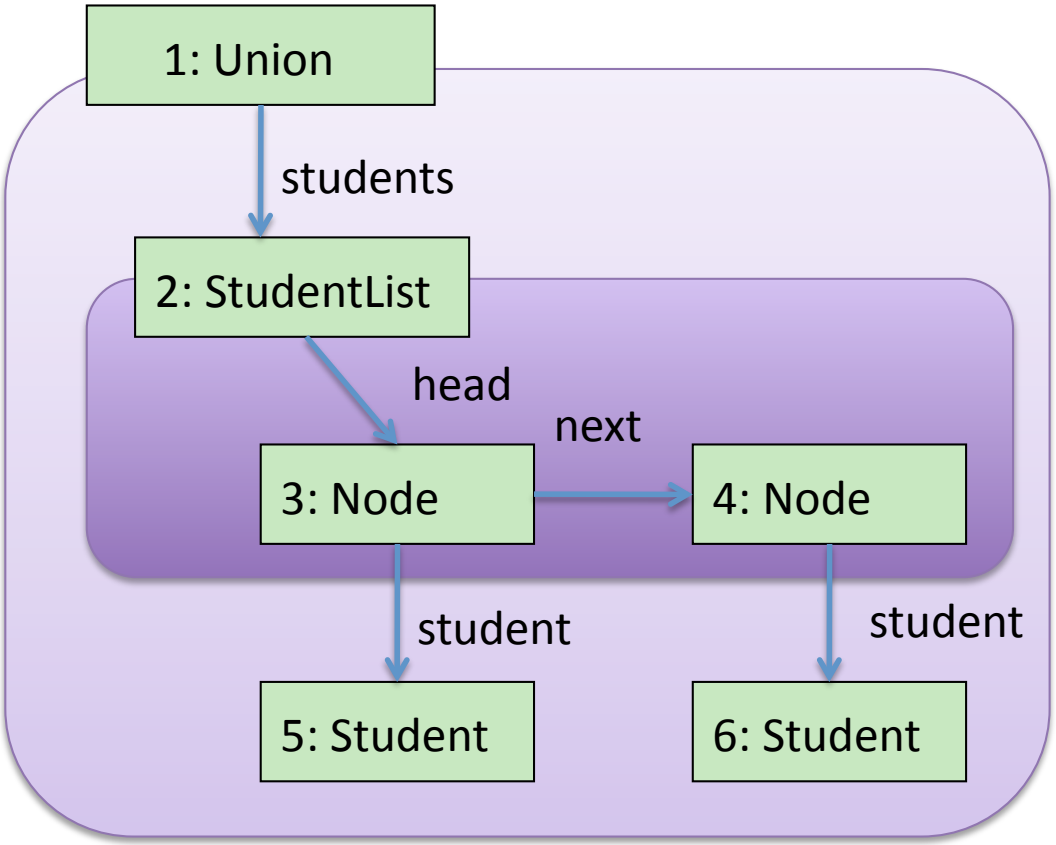



```
class Union<c>{
    StudentList<this> students;
    Union clone(){ ??? }
}
class StudentList<c>{
    Node<this,c> head;
    StudentList clone()
          { ??? }
}
class Node<c1,c2>{
     Node<c1,c2> next;
     Student<c2> student;
     Node clone(){ ??? }
}
class Student<c> {
     Student clone(){ ??? }
}
```

means that

- this.students is inside the cloning domain of this, when in Union.

Putting objects into boxes …




```
class Union<c>{
    StudentList<this> students;
    Union clone(){ ??? }
}
class StudentList<c>{
    Node<this,c> head;
    StudentList clone()
          { ??? }
}
class Node<c1,c2>{
     Node<c1,c2> next;
     Student<c2> student;
     Node clone(){ ??? }
}
class Student<c> {
     Student clone(){ ??? }
}
```

means that

- this.students is inside the cloning domain of this, when in Union.
- Therefore, when 1 is cloned, 2 has to be cloned too.

Path Types - definition

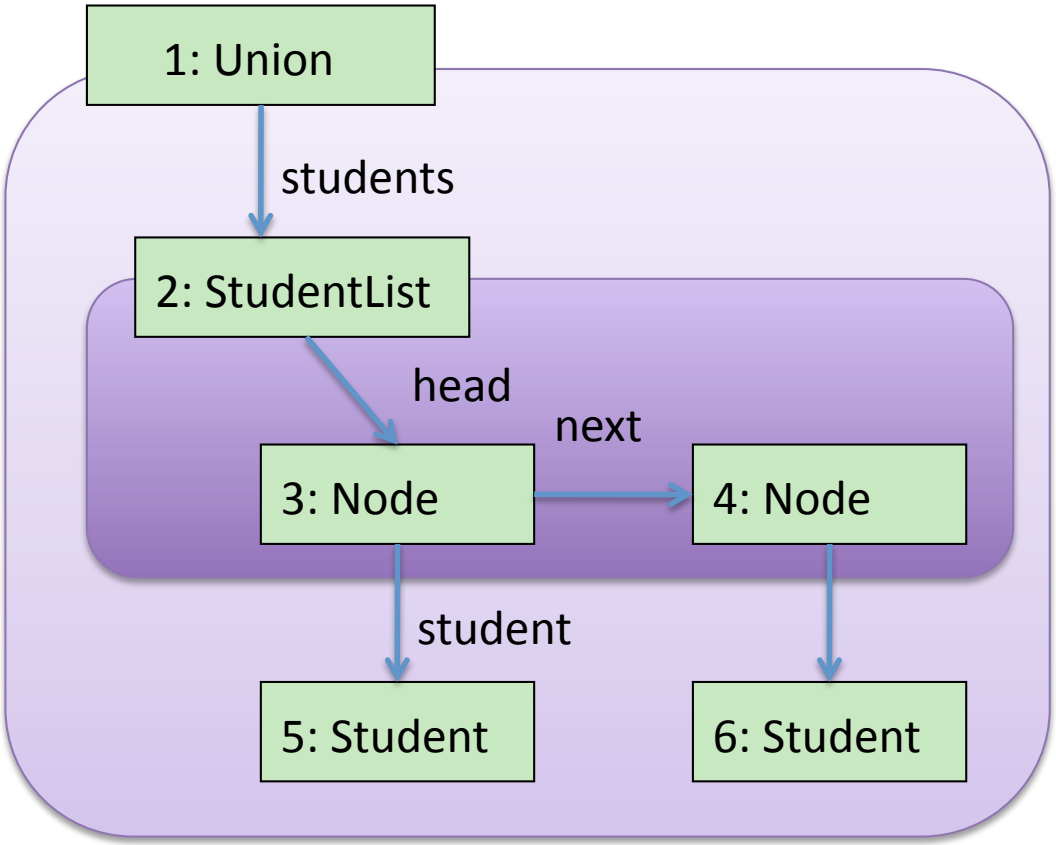
$$\frac{\texttt{class C}\langle c_1, \ldots c_n\rangle\{\ \ldots\ \}}{\texttt{C} \vdash \texttt{this} : \texttt{C}\langle c_1, \ldots c_n\rangle}$$

$$\frac{\begin{array}{c}\texttt{class D}\langle c_1, \ldots c_m\rangle\{\ \ldots\ \texttt{E}\langle ap_1, \ldots ap_n\rangle\ \texttt{f}\ \ldots\ \} \\ \texttt{C} \vdash \texttt{this}.\overline{\texttt{f}} : \texttt{D}\langle cap_1, \ldots cap_m\rangle\end{array}}{\texttt{C} \vdash \texttt{this}.\overline{\texttt{f}}.\texttt{f} : \texttt{E}\langle\, ap_1, \ldots ap_n[cap_1, \ldots cap_m/c_1, \ldots c_m][\texttt{this}.\overline{\texttt{f}}/\texttt{this}]\,\rangle}$$

Fig. 4. Path types for paths

- A path has the form this.f1.....fn.
- Its type gives the path that leads to its owner (and the remaining cloning parameters).

Path Types – example



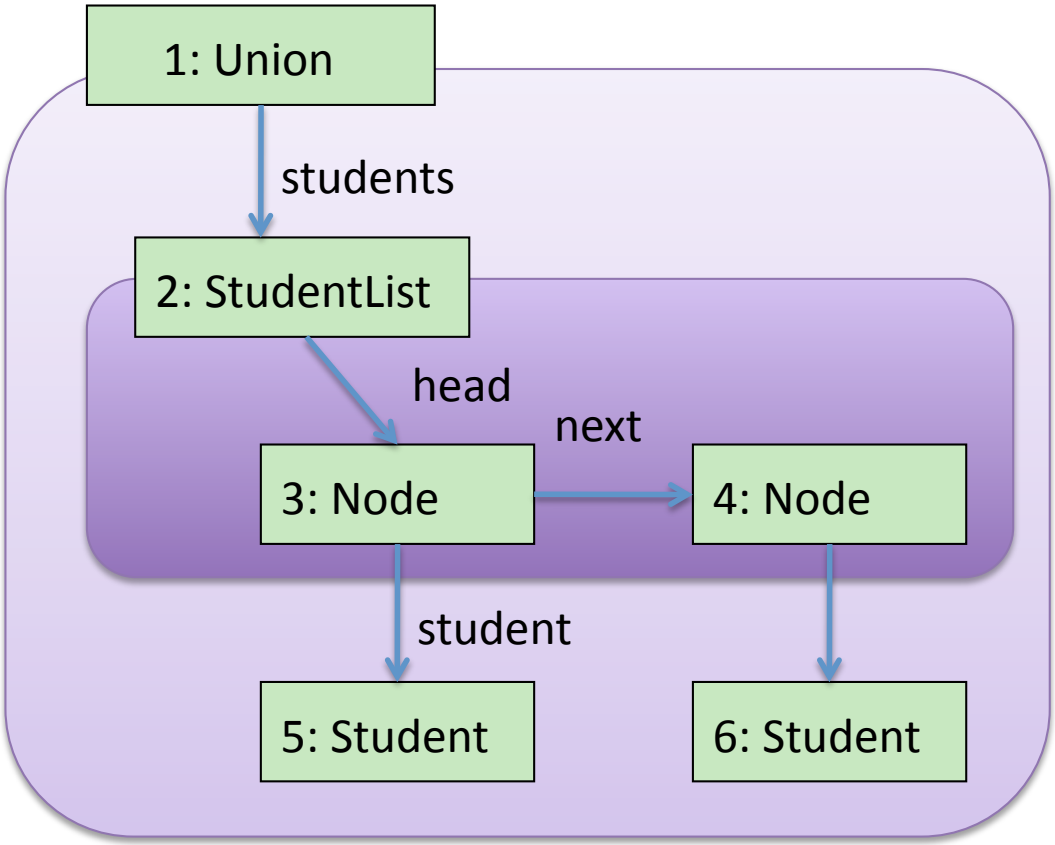

```
Union |- this.students : StudentList<this>
Union |- this.studentList.head.student : Student<this>
```

Cloning domains – definition

$$ClnDom(\mathtt{C}) = \{\, \mathtt{this} \,\} \cup \{\, \mathtt{p.f} \mid \mathtt{C} \vdash \mathtt{p.f} : \mathtt{D}\langle \mathtt{p}', \ldots \rangle \text{ for a class } \mathtt{D}, \text{ and a path } \mathtt{p}', \text{ and where } \mathtt{p} \in ClnDom(\mathtt{C}) \,\}$$

- The cloning domain of a class is the (infinite) set of paths whose type indicates that the corresponding object has to be cloned when this is cloned.

Cloning Domains – example



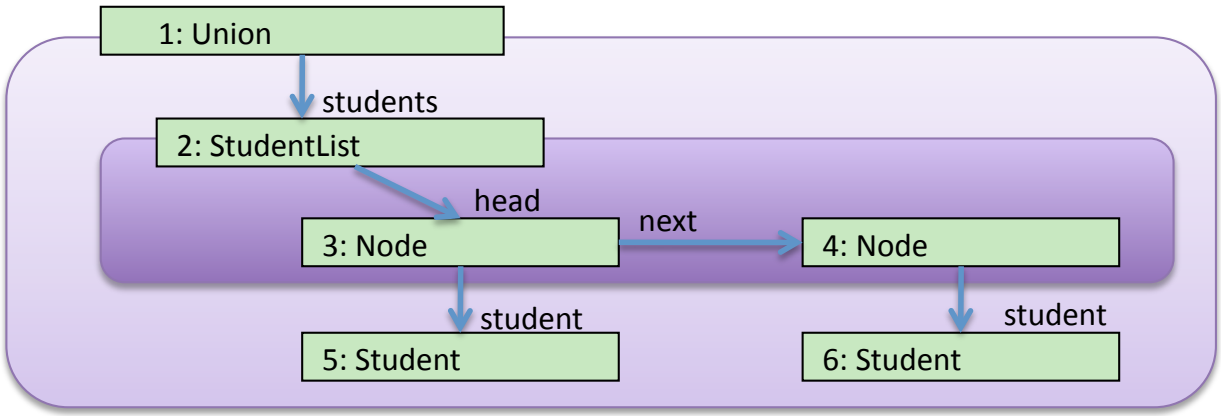

```
ClnDom(Union)=
{ this, this.students,
  this.students.head,
  this.head.studetns.next*,
  this.head.studetns.next*.student   }
```

Contents of the talk

- Example of sheep cloning
- Annotating code with cloning information
  - Path Types
  - Cloning Domains
- Generation of the cloning methods
  - The cloning method for the Node example
  - The cloning method in general
- Conclusions

The `Node::clone()` method

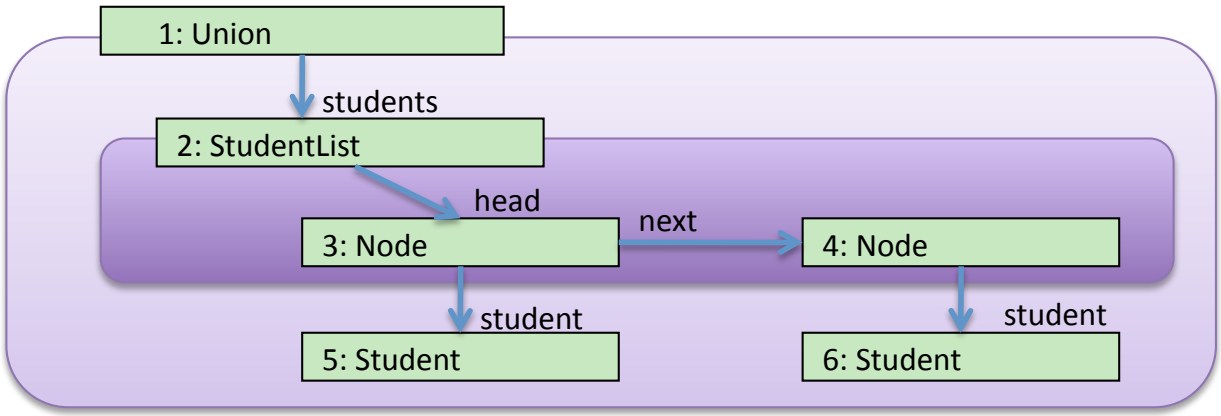
```
class Node<c1,c2>{
    Node<c1,c2> next; Student<c2> student;

}
```



The Node::clone() method

```
class Node<c1,c2>{
    Node<c1,c2> next; Student<c2> student;
    Node clone(){
       this.clone(false, false, new IdentityHashMap()
    };
```

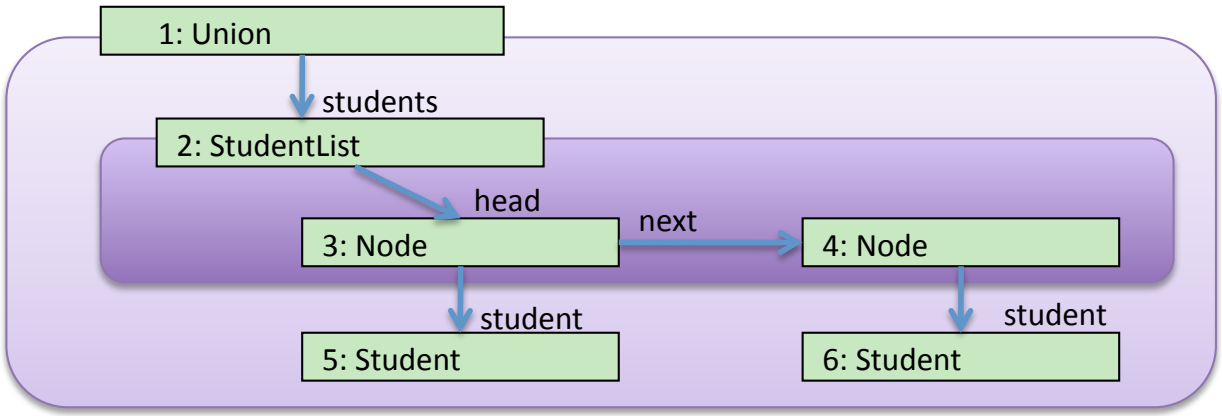
}



The Node::clone() method

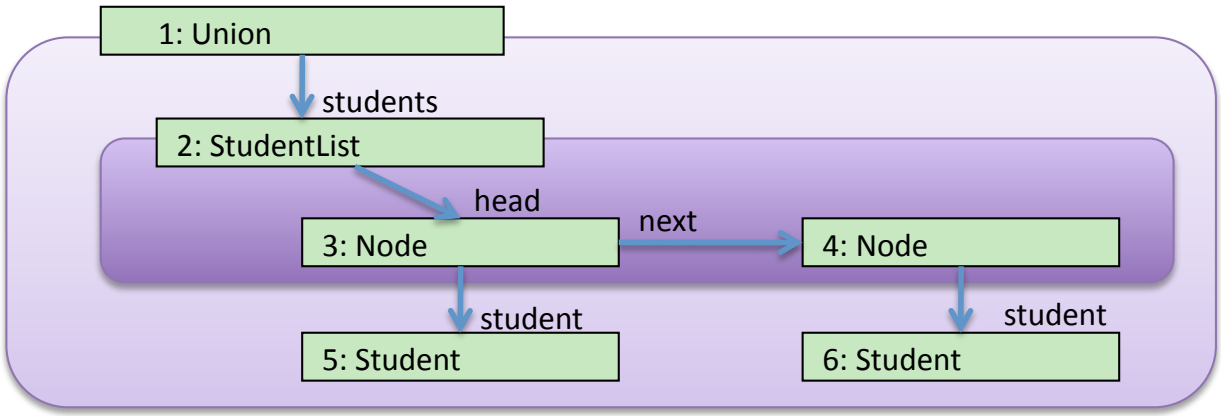
```
class Node<c1,c2>{
    Node<c1,c2> next; Student<c2> student;
    Node clone(){
       this.clone(false, false, new IdentityHashMap()
    };
    Node clone(bool s1, bool s2, Map m){
       Object n=m.get(this);
       if (n!=null){ return (Node)n; }
```

```
}
```



The Node::clone() method

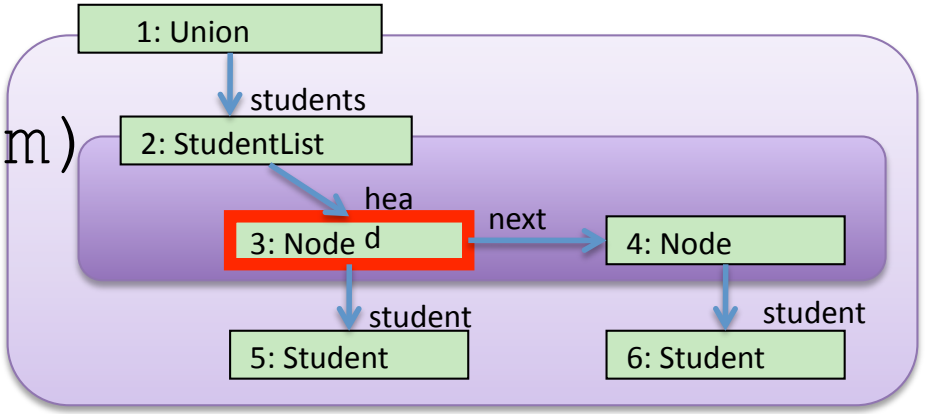
```
class Node<c1,c2>{
    Node<c1,c2> next; Student<c2> student;
    Node clone(){
       this.clone(false, false, new IdentityHashMap()
    };
    Node clone(bool s1, bool s2, Map m){
       Object n=m.get(this);
       if (n!=null){ return (Node)n; }
       Node clone = new Node(); m.put(clone);
       clone.next= s1? this.next.clone(s1,s2,m) : this.next;
       clone.student =
           s2? this.student.clone(s2,m) : this.student;
       return clone;
}
```



The Node::clone() method

```
class Node<c1,c2>{
    Node<c1,c2> next; Student<c2> student;
     . . . .

    Node clone(bool s1, bool s2, Map m){
       …

       Node clone = new Node(); m.put(clone);
       clone.next= s1? this.next.clone(s1,s2,m) : this.next;
       clone.student =
           s2? this.student.clone(s2,m) : this.student;
       return clone;
}
```

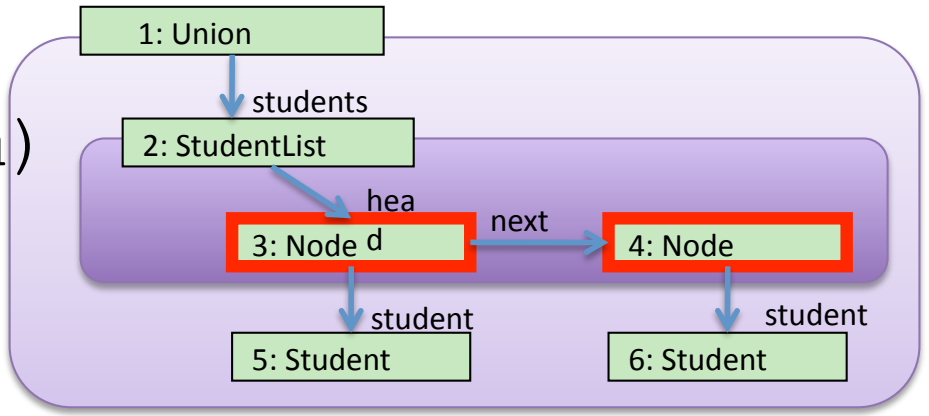
3.clone(false,false,m)
duplicates 3



The Node::clone() method

```
class Node<c1,c2>{
    Node<c1,c2> next; Student<c2> student;
      . . . .

    Node clone(bool s1, bool s2, Map m){
       …

       Node clone = new Node(); m.put(clone);
       clone.next= s1? this.next.clone(s1,s2,m) : this.next;
       clone.student =
           s2? this.student.clone(s2,m) : this.student;
       return clone;
}
```

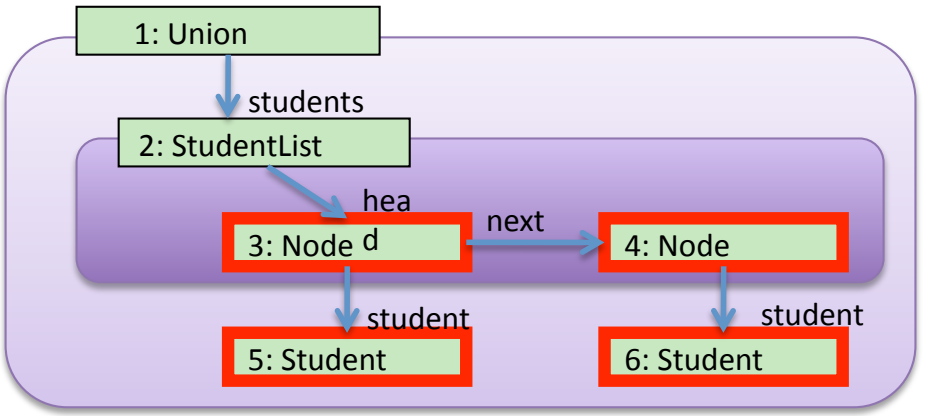
3.clone(true,false,m) duplicates 3 and 4.



The Node::clone() method

```
class Node<c1,c2>{
    Node<c1,c2> next; Student<c2> student;
     . . . .

    Node clone(bool s1, bool s2, Map m){
       …

       Node clone = new Node(); m.put(clone);
       clone.next= s1? this.next.clone(s1,s2,m) : this.next;
       clone.student =
           s2? this.student.clone(s2,m) : this.student;
       return clone;
}
```

3.clone(true,true,m) duplicates 3, 4, 5 and 6.



The `C::clone()` method - in general

```
C clone(Boolean s_1...Boolean s_q, Map m){
    Object o = m.get(this)
    if o ≠ null then
        return (C)o;
    else{
        C o' = new C();
        m.put(this, o');
        o'.f_1 = s_1,1 ? this.f_1.clone(s_1,1...s_1,k1, m) : this.f_1;
        ... = ...
        o'.f_n = s_n,1 ? this.f_n.clone(s_n,1...s_n,kn, m) : this.f_n;
        return o';
    }
}
```

where

$\{f_1, ...f_n\}$ are the fields defined in class C

and where, for all $i \in 1..n$:

$(fType(C\langle s_1...s_n\rangle, f_i))[true/this] = C_i\langle s_{i,1}...s_{i,ki}\rangle$

for some classes C_1,.. C_n.

Contents of the talk

- Example of sheep cloning
- Annotating code with cloning information
  - Path Types
  - Cloning Domains
- Generation of the cloning methods
  - The cloning method for the Node example
  - The cloning method in genera;
- Properties of cloning methods
- Conclusions

Guarantees of cloning

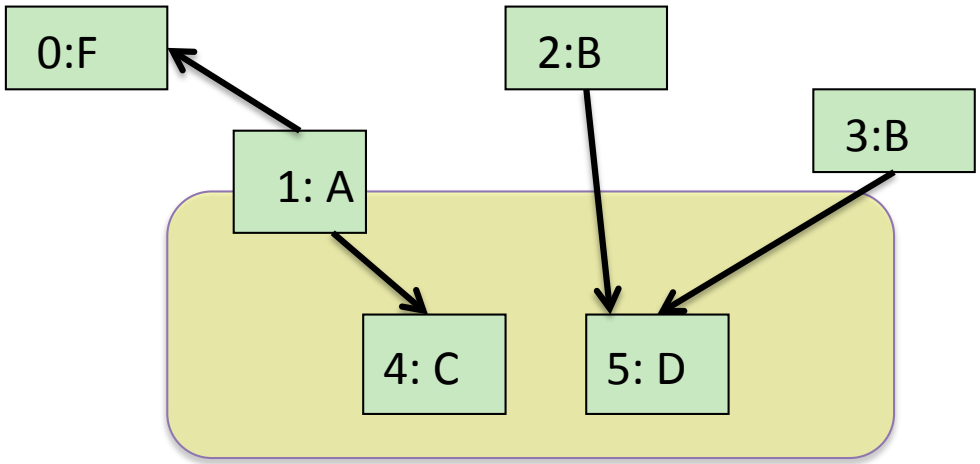
✓**Termination**: any call of `o.clone()` terminates.

✓**Soundness**: all objects from the cloning domain are duplicated.

✓**Isomorphism** the new objects form a structure that is isomorphic to that in the cloning domain, provided that owners are “points of no return.”

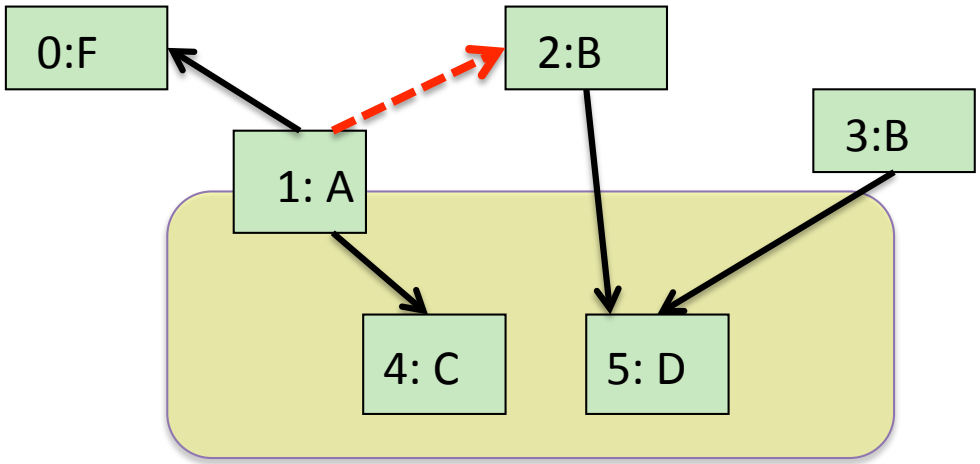
Owner as “point of no return”

- Pointers into a box are *allowed*.



Owner as “point of no return”

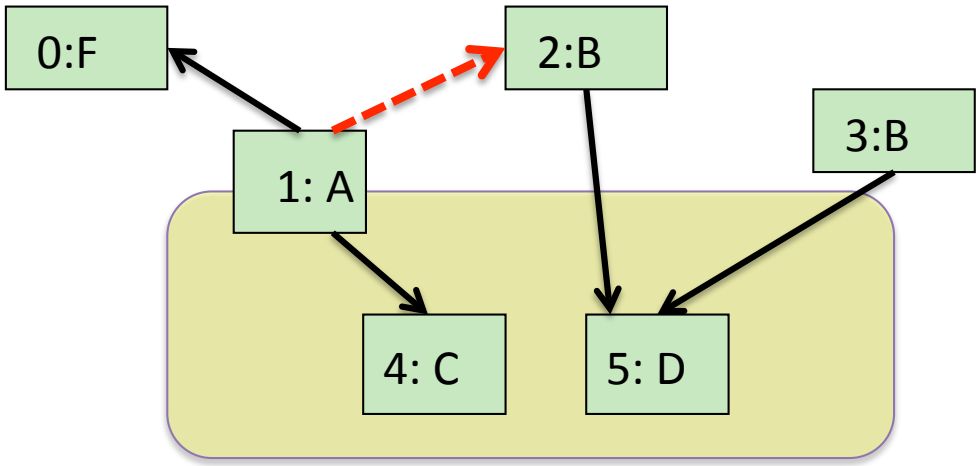
- Pointers into a box are allowed, provided no path exits and then re-enters a box.



- We can check this statically.

Owner as “point of no return”

- Pointers into a box are allowed, provided no path exits and then re-enters a box.



✓**Isomorphism**: the new objects form a structure isomorphic to that in the cloning domain, provided owners are “points of no return.”

Contents of the talk

- Example of sheep cloning
- Annotating code with cloning information
  - Path Types
  - Cloning Domains
- Generation of the cloning methods
  - The cloning method for the Node example
  - The cloning method in genera;
- Properties of cloning methods
- Conclusions

Conclusions

- Ownership-like types are cool.
- Cloning methods can be generated.
- “Point of no return” new flavour of OTs.

Conclusions

- Ownership-like types are cool.
- Cloning methods can be generated.
- “Point of no return” new flavour of OTS