

Konzepte objektorientierter Programmierung – Lecture 6 –

Prof. Dr. Peter Müller

Software Component Technology

Wintersemester 03/04

ETH

Edgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Information Hiding

- Definition

Information hiding is a technique for reducing the dependencies between modules:

- *The intended client is provided with all the information needed to use the module **correctly**, and **with nothing more***
- *The client uses only the (publicly) available information*

Exchanging Implementations

- Observable behavior must be preserved
- Exported attributes limit modifications severely
 - Use getter and setter methods instead
 - Uniform access in Eiffel
- Modifications are critical
 - Fragile baseclass problem
 - **Object structures**

```
class Coordinate {  
    private float x,y;  
  
    ...  
  
    public float distOrigin( )  
        { return Math.sqrt( x*x + y*y ); }  
}
```

```
class Coordinate {  
    private float radius, angle;  
  
    ...  
  
    public float distOrigin( )  
        { return radius; }  
}
```

Encapsulation

- Definition

Encapsulation is a technique for structuring the state space of executed programs. Its objective is to guarantee data and structural consistency by establishing capsules with clearly defined interfaces.

Levels of Encapsulation

- Capsules can be
 - **Individual objects**
 - Object structures
 - **A class (with all of its objects)**
 - All classes of a subtype hierarchy
 - A package (with all of its classes and their objects)
 - Several packages
- Encapsulation requires a definition of the boundary of a capsule and the interfaces at the boundary

Invariants for Java (Simple Solution)

- Assumption: The invariants of object X may only refer to **private attributes** of X

- For each invariant, we have to show
 - That all exported method **and constructors of class T** preserve the invariants **of all objects of T and T's subclasses**
 - That all constructors **in addition** establish the invariants of the new object

Agenda for Today

6. Object Structures and Aliasing

6.1 Object Structures

6.2 Aliasing

6.3 Problems of Aliasing

6.4 Encapsulation of Object Structures

Objectives

- Aliasing
- Encapsulation of object structures

6. Object Structures and Aliasing

6.1 Object Structures

6.2 Aliasing

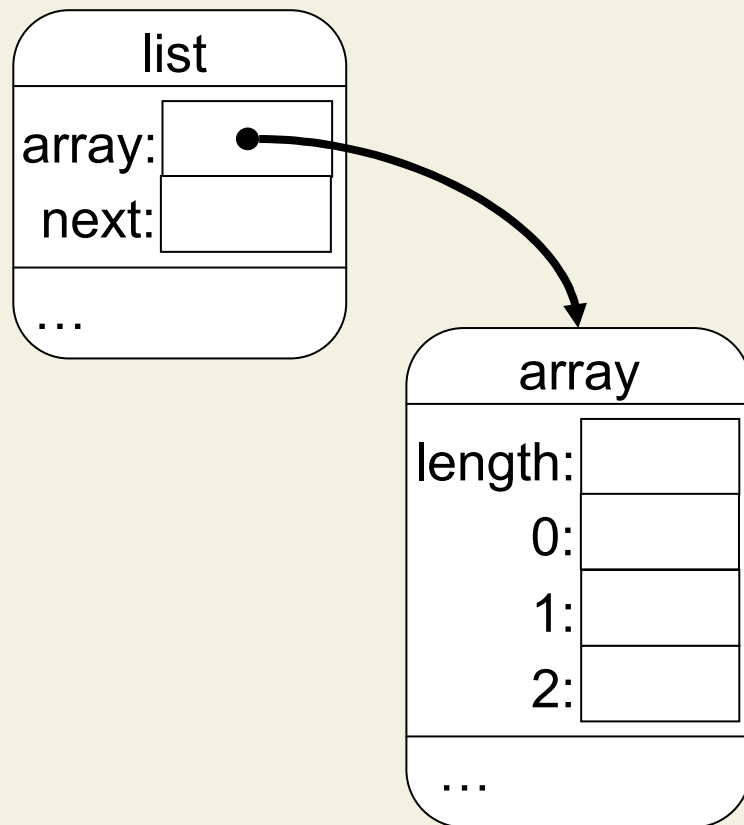
6.3 Problems of Aliasing

6.4 Encapsulation of Object Structures

Object Structures

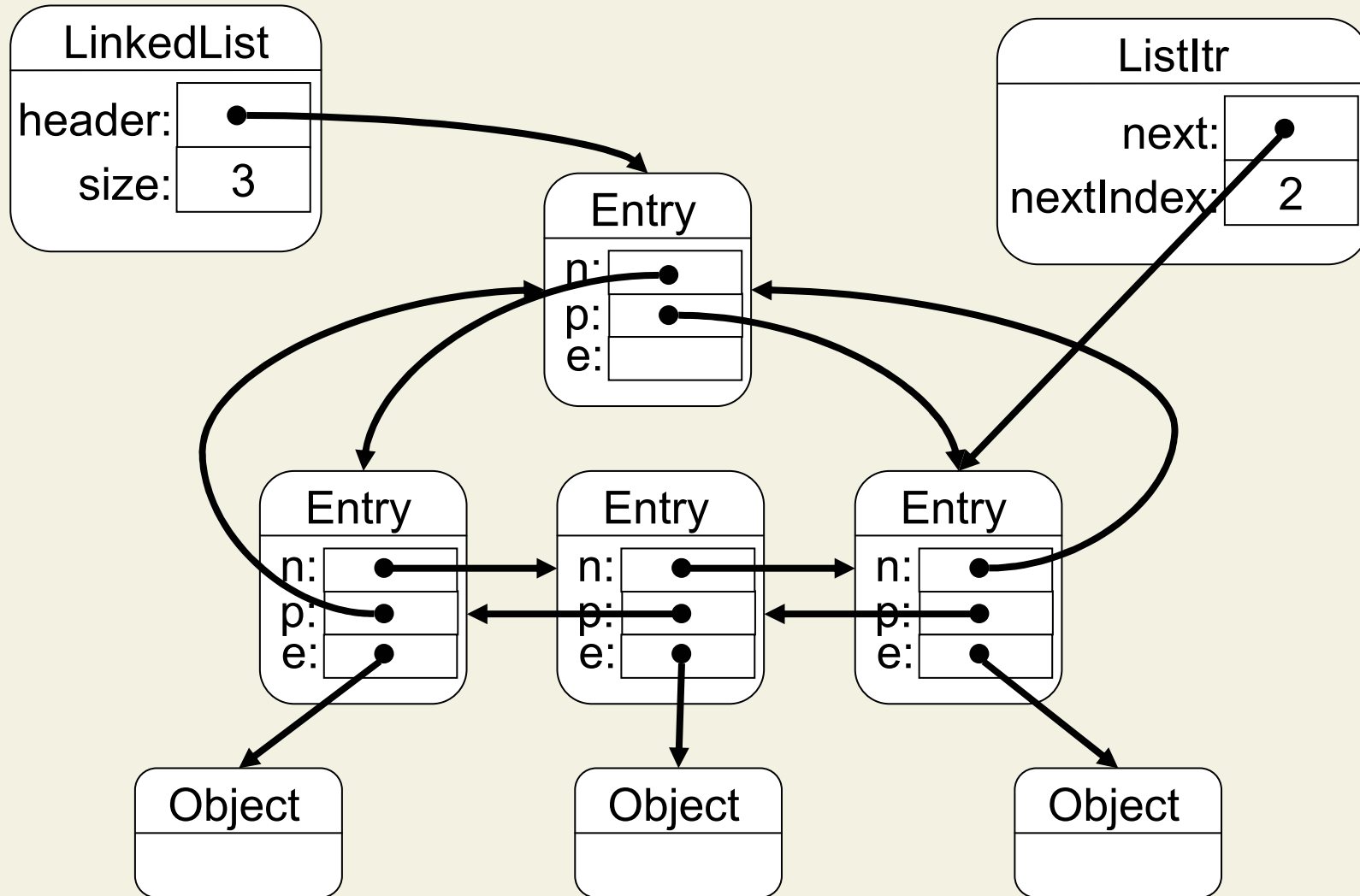
- Objects are the building blocks of object-oriented programming
- However, interesting abstractions are almost always provided by sets of cooperating objects
- Definition:
An object structure is a set of objects that are connected via references

Example 1: Array-Based Lists



```
class ArrayList {  
    private int[ ] array;  
    private int next;  
  
    public void add( int i ) {  
        if (next==array.length) resize( );  
        array[ next ] = i;  
        next++;  
    }  
  
    public void addElems( int[ ] ia )  
        { ... }  
  
    ...  
}
```

Example 2: Doubly-Linked Lists



6. Object Structures and Aliasing

6.1 Object Structures

6.2 Aliasing

6.3 Problems of Aliasing

6.4 Encapsulation of Object Structures

Alias

- Definition:

A name that has been assumed temporarily

[WordNet, Princeton University]

Aliasing in Procedural Programming

- var-parameters are passed **by reference** (call by name)
- Modification of a var-parameter is observable by caller
- Aliasing: **Several variables** (here: p, q) refer to **same memory cell**
- Aliasing can lead to **unexpected side-effects**

```
program asliasTest
  procedure assign( var p: int, var q: int );
  begin
    { p = 1  $\wedge$  q = 1 }
    p := 25;
    { p = 25  $\wedge$  q = 25 }
  end;
  begin
    var x: int := 1;
    assign( x, x );
    { x = 25 }
  end
end.
```

Aliasing in Object-Oriented Programming

- Definition:

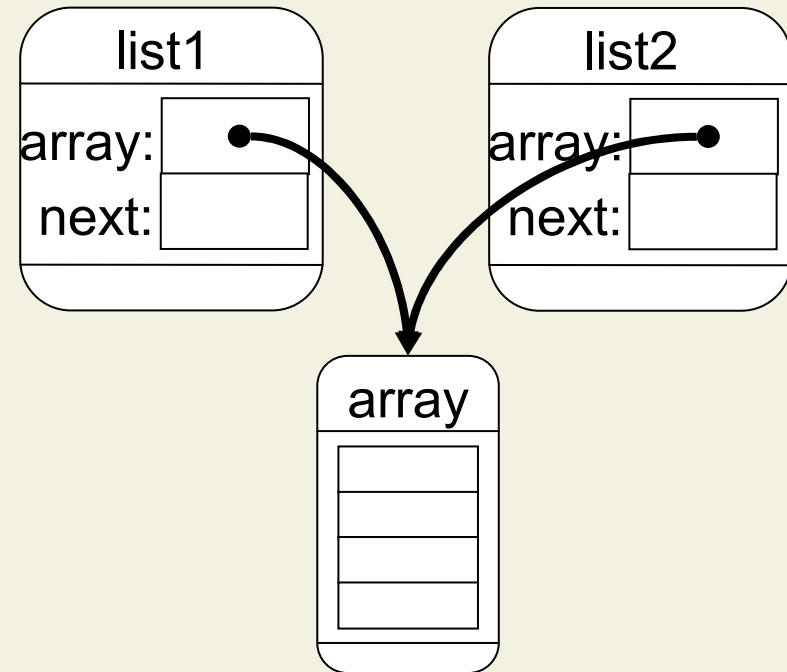
An object X is aliased if two or more variables hold references to X .

- Variables can be

- Fields of objects (instance variables)
- Static fields (global variables)
- Local variables of method executions, including **this**
- Formal parameters of method executions
- Results of method invocations or other expressions

Static Aliasing

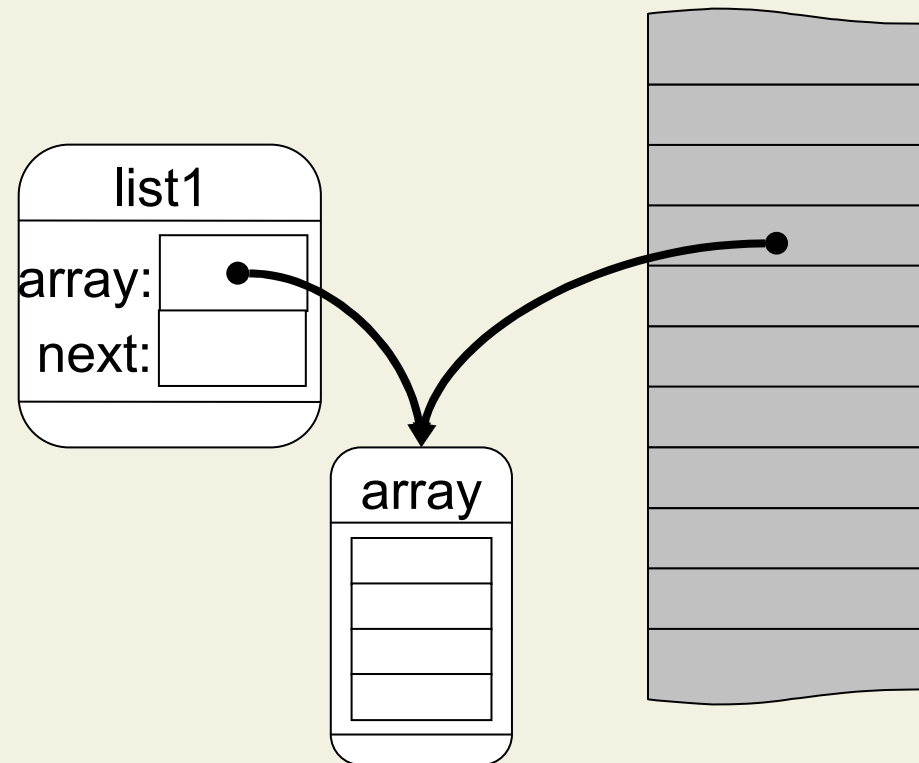
- Definition:
An alias is static if all involved variables are fields of objects or static fields.
- Static aliasing occurs in the heap memory



```
list1.array[ 0 ] = 1;  
list2.array[ 0 ] = -1;  
System.out.println( list1.array[ 0 ] );
```


Dynamic Aliasing

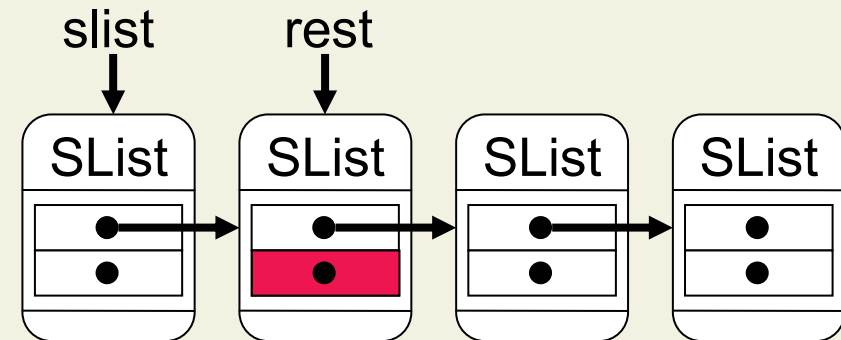
- Definition:
An alias is dynamic if it is not static.
- Static aliasing involves stack-allocated variables



```
int[ ] ia = list1.array;  
list1.array[ 0 ] = 1;  
ia[ 0 ] = -1;  
System.out.println( list1.array[ 0 ] );
```

Intended Aliasing: Efficiency

- In OO-programming, data structures are usually **not copied** when passed or modified
- Aliasing and **destructive updates** make OO-programming efficient

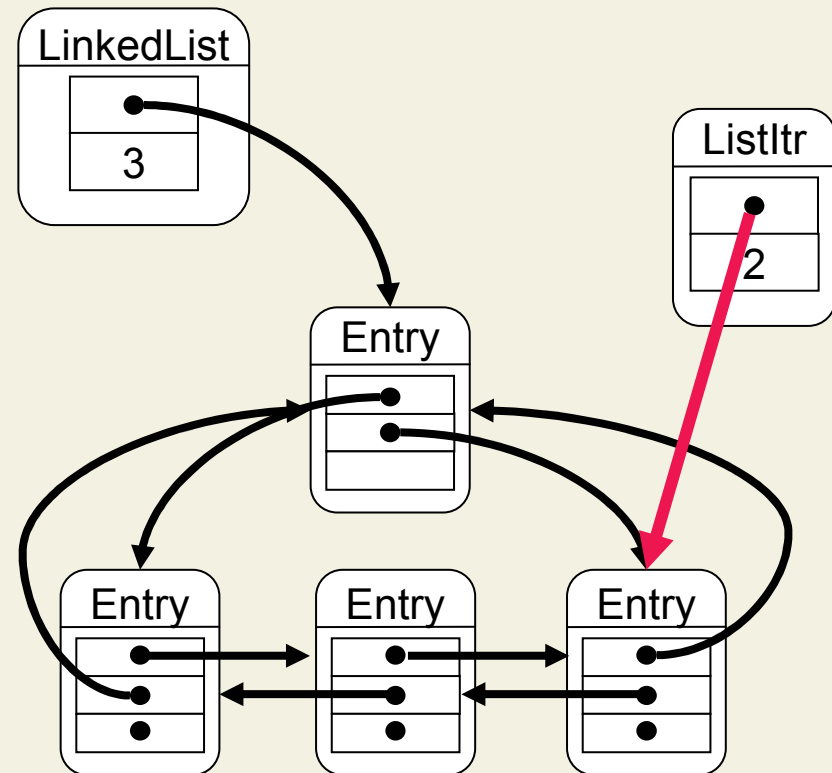


```
class SList {  
    SList next;  
    Object elem;  
    void rest( ) { return next; }  
    void set( Object e ) { elem = e; }  
}
```

```
void foo( SList slist ) {  
    SList rest = slist.rest( );  
    rest.set( "Hello" );  
}
```

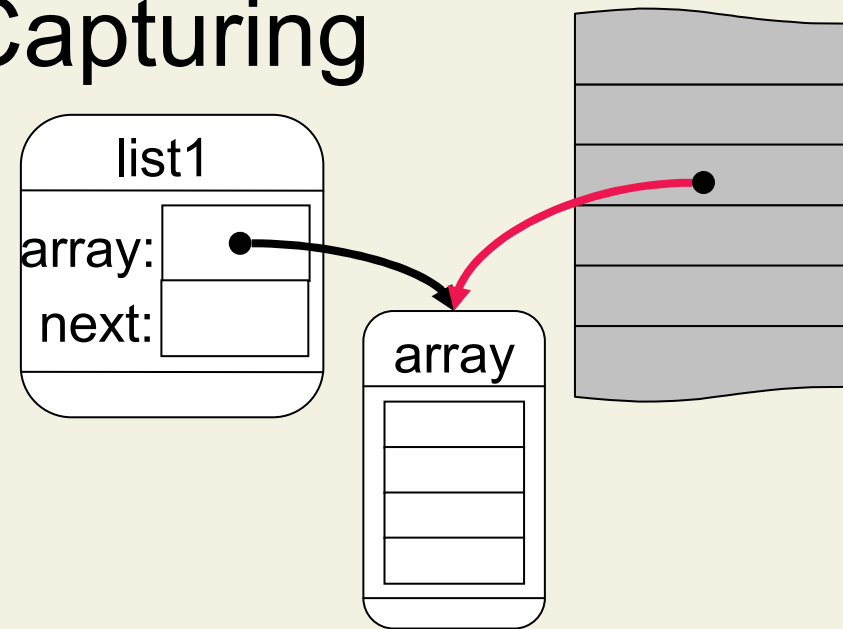
Intended Aliasing: Sharing

- Aliasing is a direct **consequence of object identity**
- Objects have **state** that can be modified
- Objects have to be **shared** to make modifications of state effective



Unintended Aliasing: Capturing

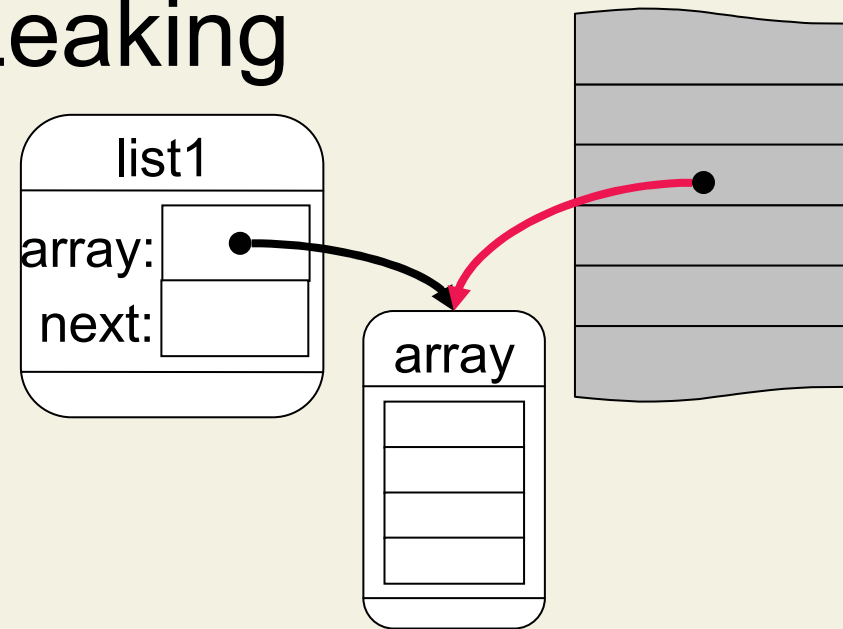
- Capturing occurs when objects are **passed to a data structure and then stored** by the data structure
- Capturing often occurs **in constructors** (e.g., streams in Java)
- Problem: Alias can be used to **by-pass interface** of data structure



```
class ArrayList {  
    private int[ ] array;  
    private int next;  
    public void addElems( int[ ] ia )  
        { array = ia; next = ia.length; }  
    ...  
}
```

Unintended Aliasing: Leaking

- Leaking occurs when data structure **pass a reference** to an object, which is **supposed to be internal** to the outside
- Leaking **often** happens **by mistake**
- Problem: Alias can be used to **by-pass interface** of data structure



```
class ArrayList {  
    private int[ ] array;  
    private int next;  
    public int[ ] getElems( )  
        { return array; }  
    ...  
}
```

6. Object Structures and Aliasing

6.1 Object Structures

6.2 Aliasing

6.3 Problems of Aliasing

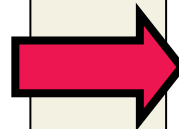
6.4 Encapsulation of Object Structures

Observation

- Many **well-established techniques** of object-oriented programming work for individual objects, but **not for object structures in the presence of aliasing**
- Examples
 - Information hiding and exchanging implementations
 - Encapsulation and consistency

Exchanging Implementations

```
class ArrayList {  
    private int[ ] array;  
    private int next;  
  
    // requires ia != null  
    // ensures  $\forall i. 0 \leq i < \text{ia.length}:$   
    //         isElem( old( ia[ i ] ) )  
    public void addElems( int[ ] ia )  
    { array = ia; next = ia.length; }  
  
    ...  
}
```



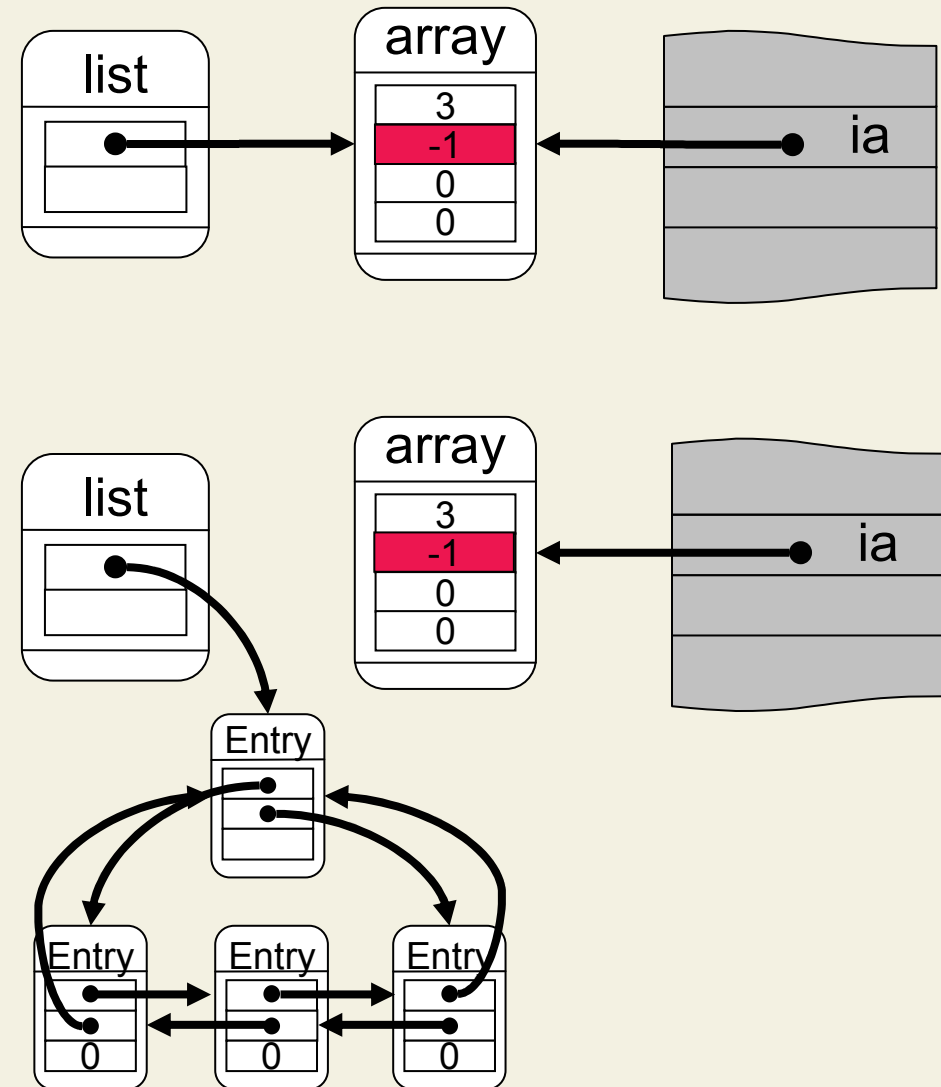
```
class ArrayList {  
    private Entry header;  
  
    // requires ia != null  
    // ensures  $\forall i. 0 \leq i < \text{ia.length}:$   
    //         isElem( old( ia[ i ] ) )  
    public void addElems( int[ ] ia )  
    { ... /* create Entry for each  
        element */ }  
  
    ...  
}
```

- Interface including contract remains unchanged

Exchanging Implementations (cont'd)

```
int foo( ArrayList list ) {
    int[ ] ia = new int[ 3 ];
    list.addElems( ia );
    ia[ 0 ] = -1;
    return list.getFirst( );
}
```

- Aliases can be used to by-pass interface
- **Observable behavior is changed!**



Consistency of Object Structures

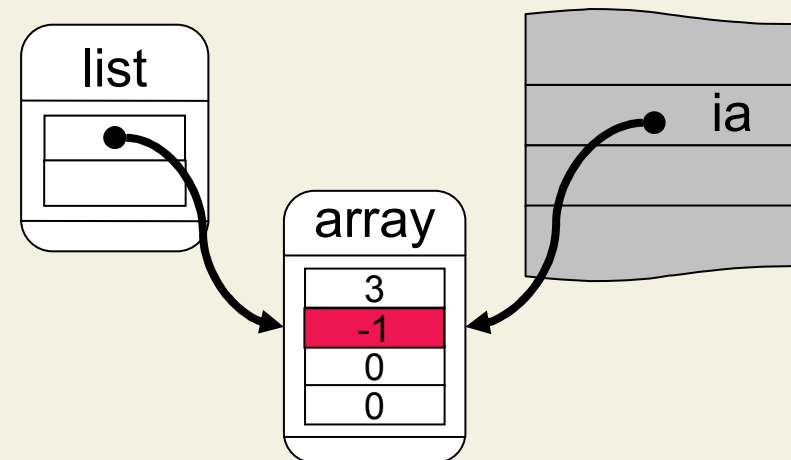
- Consistency of object structures depends on **fields of several objects**
- **Invariants** are usually specified as part of the contract **of those objects** that represent the **interface of the object structure**

```
class ArrayList {  
    private int[ ] array;  
    private int next;  
  
    // invariant array != null    &&  
    //  0<=next<=array.length  &&  
    //   $\forall i. 0 \leq i < \text{next}: \text{array}[i] \geq 0$   
  
    public void add( int i )    { ... }  
    public void addElems( int[ ] ia )  
        { ... }  
  
    ...  
}
```

Consistency of Object Structures (cont'd)

```
int foo( ArrayList list ) {      // invariant of list holds
    int[ ] ia = new int[ 3 ];
    list.addElems( ia );        // invariant of list holds
    ia[ 0 ] = -1;               // invariant of list violated
}
```

- **Aliases can be used to violate invariant**
- Making all attributes private is not sufficient to encapsulate internal state



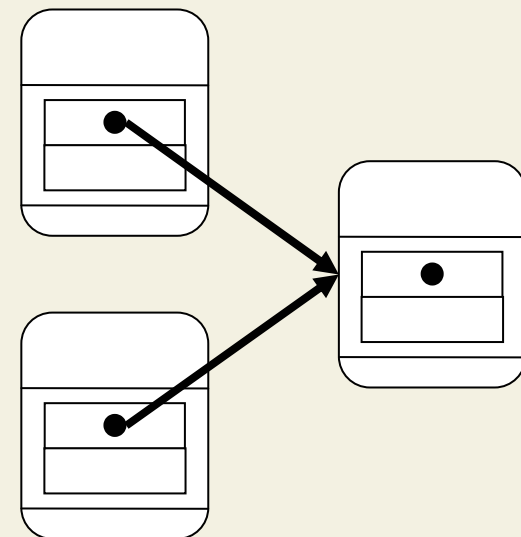
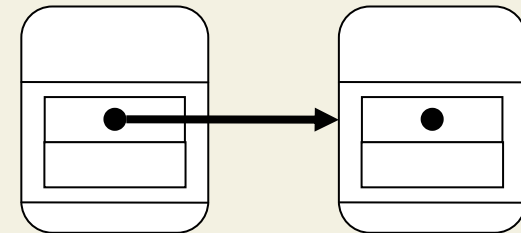
Security Breach in Java JDK 1.1.1

- Each Class-object stores array of signers
- Only trusted signers get extended access rights to the host system
- Through leaking, a malicious applet can modify the list of signers to get extended rights
- Again, access modifiers cannot prevent the problem because arrays are mutable

```
private Identity[ ] signers;  
...  
public Identity[ ] getSigners( ) {  
    return signers;  
}
```

Other Problems with Aliasing

- Synchronization in concurrent programs
 - Monitor of each individual object has to be locked to ensure mutual exclusion
- Distributed programming
 - For instance, parameter passing for remote method invocation
- Optimizations
 - For instance, object inlining is not possible for aliased objects



6. Object Structures and Aliasing

6.1 Object Structures

6.2 Aliasing

6.3 Problems of Aliasing

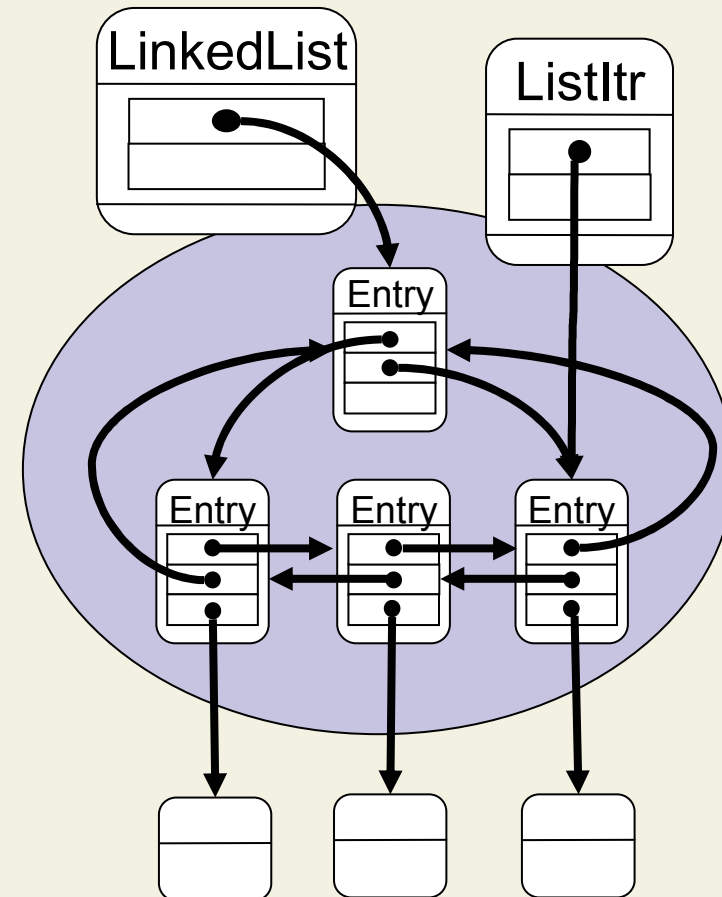
6.4 Encapsulation of Object Structures

Alias Modes

- We need **better control** over the objects in an object structure to avoid the problems with aliasing
- Approach
 1. Define **roles** of objects in object structures
 2. Assign a tag (**alias mode**) to every expression to indicate the role of the referenced object
 3. Impose **programming rules** to guarantee that objects are only used according to their alias modes

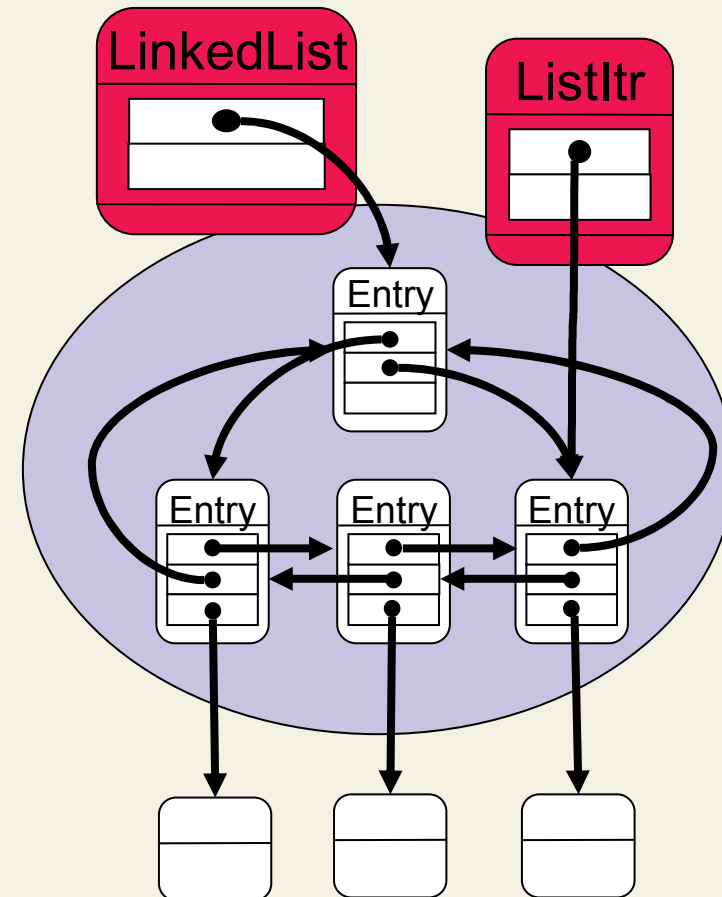
Roles in Object Structures

- **Interface objects** that are used to access the structure
- **Internal representation** of the object structure
- **Arguments** of the object structure



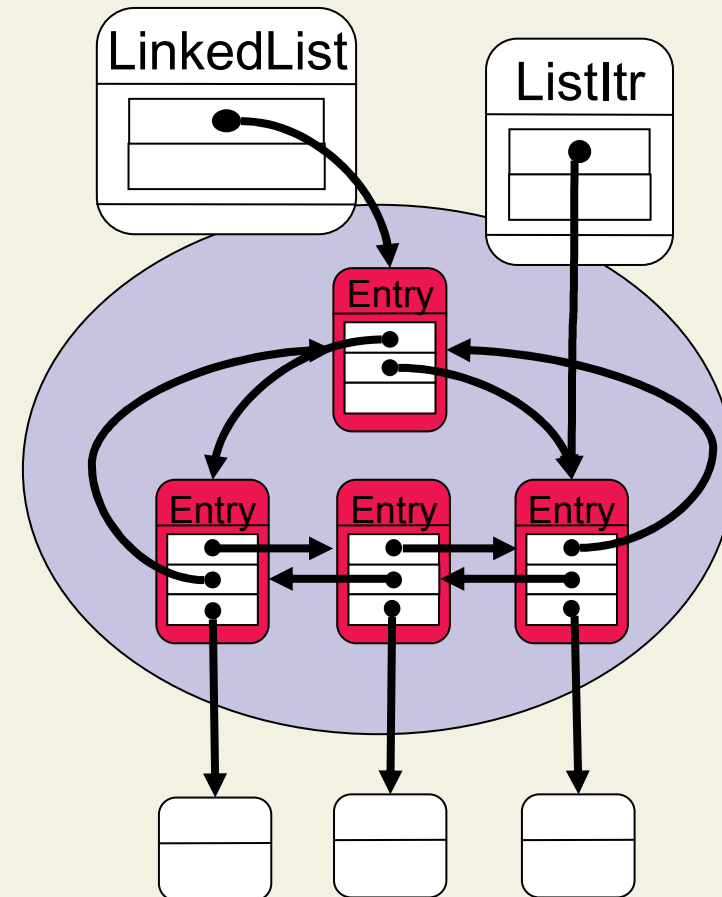
Interface Objects (default Mode)

- Interface objects are used to **access the structure**
- Interface objects can be **used in any way** objects are usually used (passed around, changed, etc.)



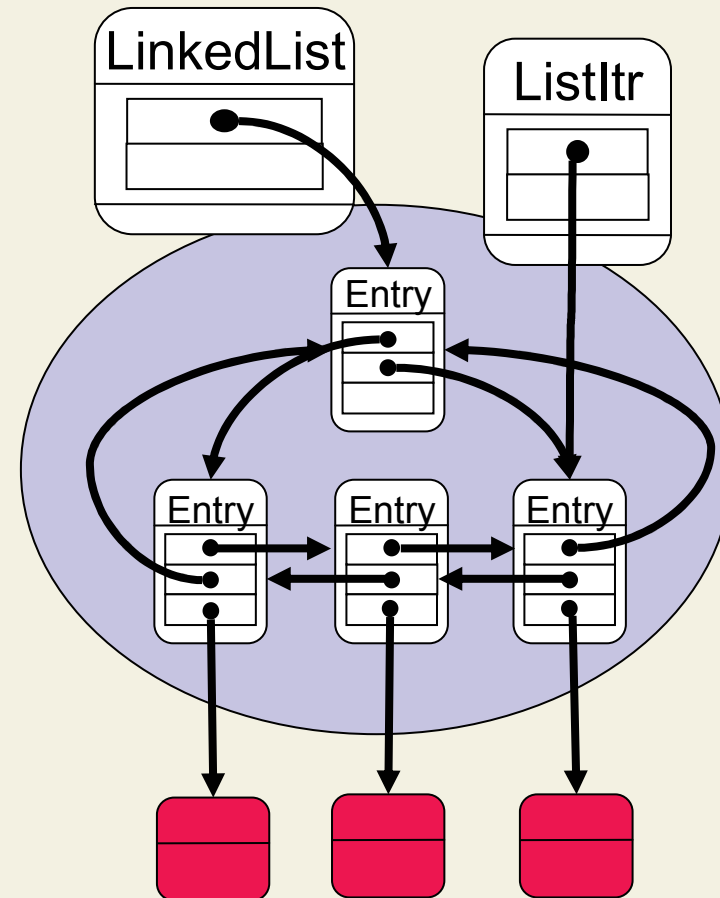
Representations (rep Mode)

- Expressions with mode “rep” hold references to the **representation** of the object structure
- Objects referenced by rep-expressions can **be changed**
- Rep-objects **must not be exported** from the object structure



Arguments (arg Mode)

- Expressions with mode “arg” hold references to **arguments** of the object structure
- Objects **must not be changed** through arg-references
- Arg-objects can be **passed around** and aliased freely



(Simplified) Programming Discipline

■ Rule 1: No Role Confusion

- Expression with one alias mode must not be assigned to variables with another mode

■ Rule 2: No Representation Exposure

- rep-mode must not occur in an object's interface
- Methods must not take or return rep-objects
- Fields with rep-mode may only be accessed on **this**

■ Rule 3: No Argument Dependence

- Implementations must not depend on the state of argument objects

Example 1: LinkedList with Alias Modes

```
class LinkedList {  
    private /* rep */ Entry header;  
    private int size;  
  
    public void add( /* arg */ Object o ) {  
        /* rep */ Entry newE = new /* rep */ Entry( o, header, header.previous );  
        ... }  
}
```

```
class Entry {  
    private /* arg */ Object element;  
    private Entry previous, next;  
  
    public Entry( /* arg */ Object o, Entry p, Entry n ) { ... }  
}
```

Example 2: ArrayList with Alias Modes

- Array is **internal representation** of the list
- Method addElems **confuses alias modes**
- Clean solution requires **array copy**

```
class ArrayList {  
    private /* rep */ int[ ] array;  
    private int next;  
  
    public void addElems( int[ ] ia ) {  
        array = ia;  
  
        next = ia.length;  
    }  
    ...  
}
```

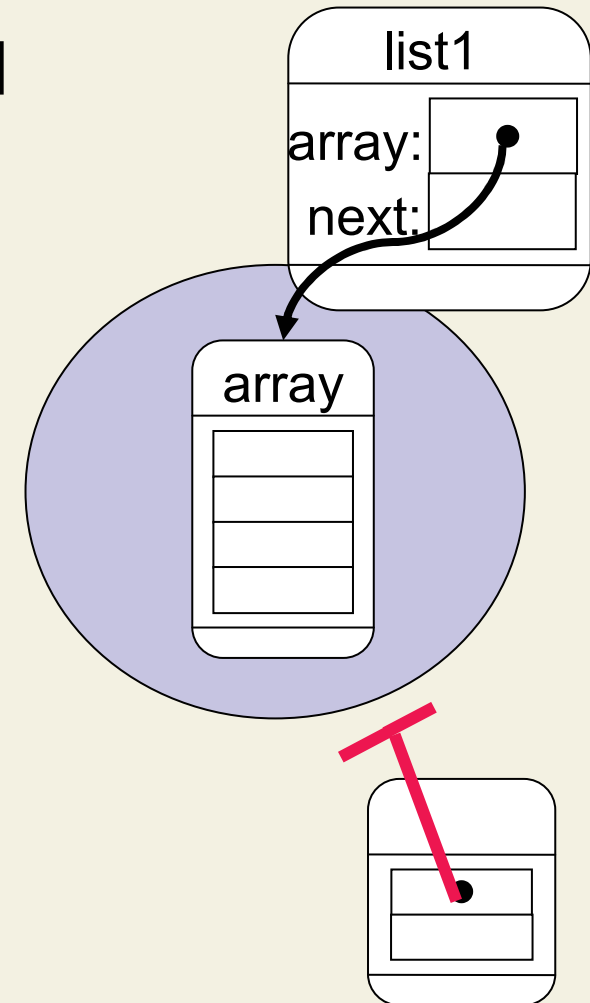
Example 2: ArrayList with Alias Modes

- Array is **internal representation** of the list
- Method addElems **confuses alias modes**
- Clean solution requires **array copy**

```
class ArrayList {  
    private /* rep */ int[ ] array;  
    private int next;  
  
    public void addElems( int[ ] ia ) {  
        array = new /* rep */ int[ ia.length ];  
        System.arraycopy  
            (ia, 0, array, 0, ia.length );  
        next = ia.length;  
    }  
    ...  
}
```

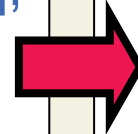
No Representation Exposure

- rep-objects can only be referenced
 - By their interface objects
 - By other rep-objects of the same object structure
- rep-objects can only be modified
 - By methods executed on their interface objects
 - By methods executed on rep-objects of the same object structure
- Rep-objects are **encapsulated** inside the **object structure**



Implementation Exchange Revisited

```
class ArrayList {  
    private /* rep */ int[ ] array;  
    private int next;  
    public void addElems( int[ ] ia ) {  
        array = new /* rep */ int[ ia.length ];  
        System.arraycopy  
            (ia, 0, array, 0, ia.length );  
        next = ia.length;  
    }  
    ...  
}
```



```
class ArrayList {  
    private /* rep */ Entry header;  
  
    public void addElems( int[ ] ia )  
    { ... /* create Entry for each  
        element */ }  
    ...  
}
```

- **Observable behavior remains unchanged!**

Invariants for Object Structures

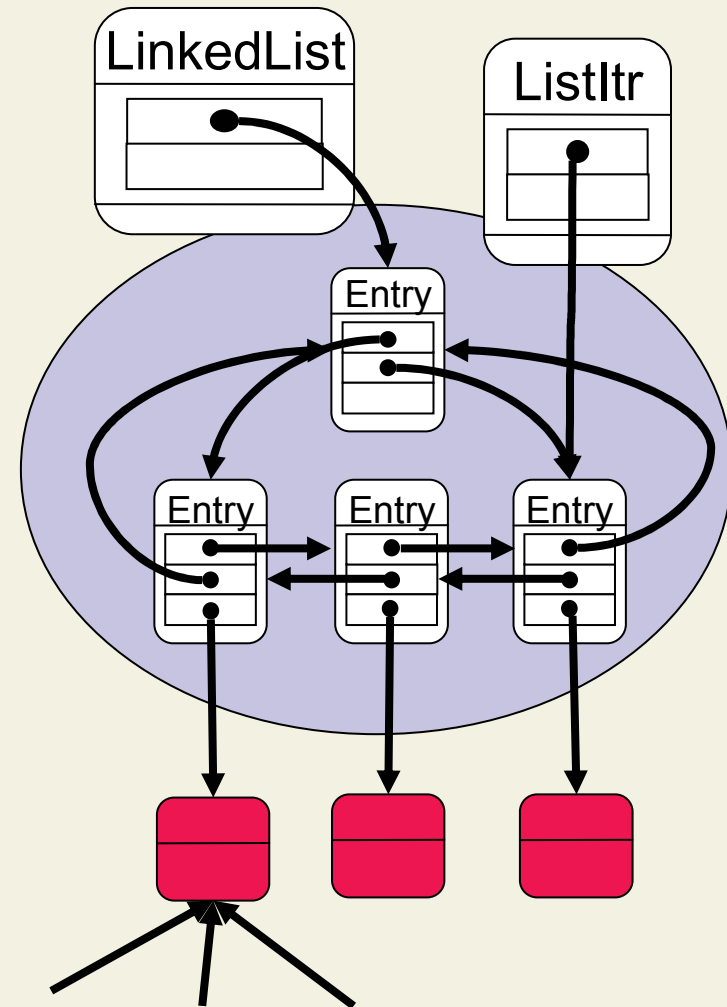
- The invariant of object X **may depend on**
 - Encapsulated fields of X
 - Fields of X's rep-objects
- Interface objects have **full control** over their rep-objects

```
class ArrayList {  
    private /* rep */ int[ ] array;  
    private int next;  
  
    // invariant array != null    &&  
    //  0<=next<=array.length  &&  
    //  ∀i.0<=i<next: array[ i ] >= 0  
  
    public void add( int i )    { ... }  
    public void addElems( int[ ] ia )  
        { ... }  
  
    ...  
}
```

No Argument Dependence

- Argument objects may be **freely aliased**
- Object structures have **no control over the state** of their argument objects
- Invariants **must not depend** on fields of argument objects, but can depend on their identity

```
private /* arg */ T v, w;  
// invariant v != w      -- legal  
// invariant v.f != w.f  -- illegal
```



Alias Control in Modular Programs

- Rules for rep-mode can in general **not** be **checked modularly**
- **Subclasses** can add new methods or override methods
- In Java, rep exposure can be prevented by
 - Access modifiers
 - Final
 - Inner Classes

```
class ArrayList {  
    protected /* rep */ int[ ] array;  
    private int next;  
    ...  
}
```

```
class MyList extends ArrayList {  
    public int[ ] leak( ) {  
        return array;  
    }  
}
```

Alias Control in Java: LinkedList

- All **fields** are **private**
- Entry is a **private inner class** of LinkedList
 - References are not passed out
 - Subclasses cannot manipulate or leak Entry-objects
- Listltr is a **private inner class** of LinkedList
 - Interface ListIterator provides controlled access to Listltr-objects
 - Listltr-objects are passed out, but in a controlled fashion
 - Subclasses cannot manipulate or leak Listltr-objects
- **Subclassing is severely restricted**

Alias Control in Java: String

- All **fields** are **private**
- References to internal character-array are not passed out (no representation exposure)
- **Subclassing is prohibited** (final)

