

Konzepte objektorientierter Programmierung

Prof. Dr. Peter Müller

Werner Dietl

Software Component Technology

Exercises 7: Aliasing

Wintersemester 03/04

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Nested & Inner Classes

- Method to structure internal, helper, or dependent classes
- Have access to private attributes and methods of the surrounding class and vice versa
- Either associated with a class or with an instance
- Visibility restrictions possible using access modifiers, but only to a certain degree
- Details will not be part of the exam

Homework – Exercise 1

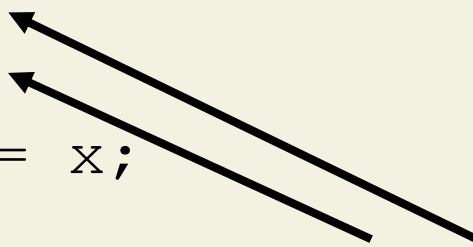
```
class Zimmer {  
    private Mensch m;  
    void betreten( Mensch m ) {}  
    Mensch verlassen() {}  
}  
  
public class Haus {  
    private Zimmer[] raeume;  
    void betreten( Mensch m ) {  
        raeume[0].betreten( m ); }  
    Mensch verlassen() {  
        raeume[0].verlassen( m ); }  
}
```

Homework – Exercise 1

```
class Zimmer {  
    private /*arg*/ Mensch m;  
    void betreten(/*arg*/ Mensch m ) {}  
    /*arg*/ Mensch verlassen() {}  
}  
  
public class Haus {  
    private /*rep*/ Zimmer[] raeume;  
    void betreten( /*arg*/ Mensch m ) {  
        raeume[0].betreten( m ); }  
    /*arg*/ Mensch verlassen() {  
        raeume[0].verlassen( m ); }  
}
```

Homework – Exercise 2

```
public class Assignments {  
    private /* rep */ Integer attr_x;  
    private /* arg */ Integer attr_y;  
    private Integer attr_z;  
    public void m( /* arg */ Integer a ) {  
        /* rep */ Integer x = new /* rep */  
                                Integer(4);  
        /* arg */ Integer y = new Integer(3);  
        Integer z = new Integer(2);  
  
        x = y;      y = x;  
        x = z;      y = z;  
        attr_x = x; attr_y = a;  
    }  
}
```



Compilation Error!

(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

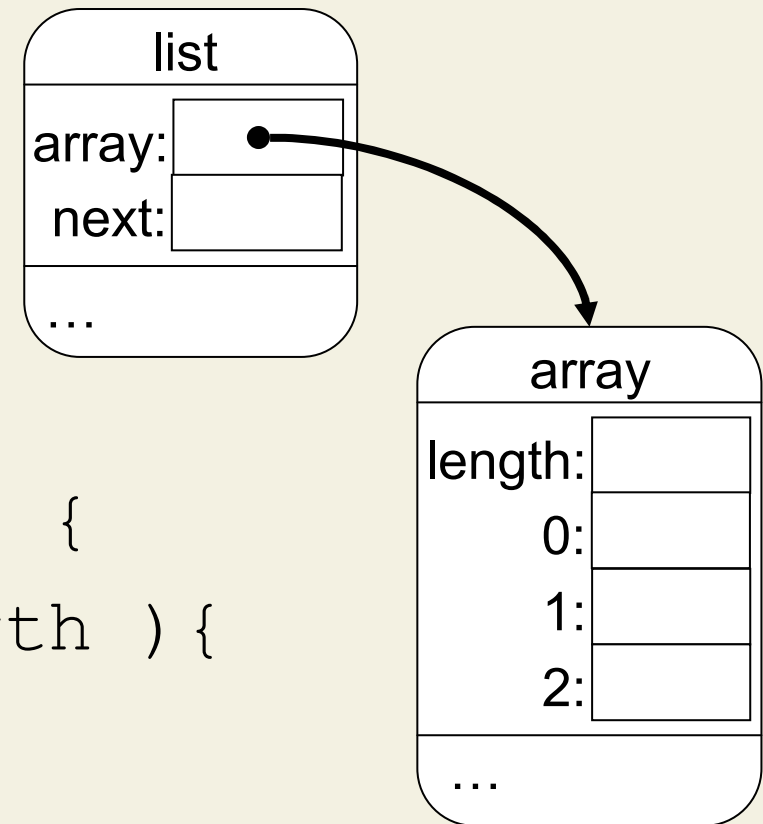
- rep-mode must not occur in an object's interface
- Methods must not take or return rep-objects
- Fields with rep-mode may only be accessed on **this**

■ Rule 3: No Argument Dependence

- Implementations must not depend on the state of argument objects

ArrayList

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



ArrayList

```
public void setElems( int[] ia ) {  
    array = ia;  
}
```

```
public int[] getElems() {  
    return array;  
}
```

- setElems is *Capturing* an external array
- getElems is *Leaking* the internal representation

ArrayList – Main Program

```
public static void main( String[] args ) {  
    int[] myarr = new int[10];  
  
    for(int i=0; i < myarr.length; ++i)  
        myarr[i] = i;  
  
    ArrayList al = new ArrayList();  
  
    al.setElems( myarr );  
  
    myarr[0] = 42;  
}
```

Annotated ArrayList

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

- Array is part of the internal representation
- Marking it as `/* rep */` makes this explicit for the programmer

ArrayList – setElems

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

- The input array is of default type and can not directly be assigned to the `/* rep */ array`
- We have to create our own copy that can be used as internal representation → no *Capturing*

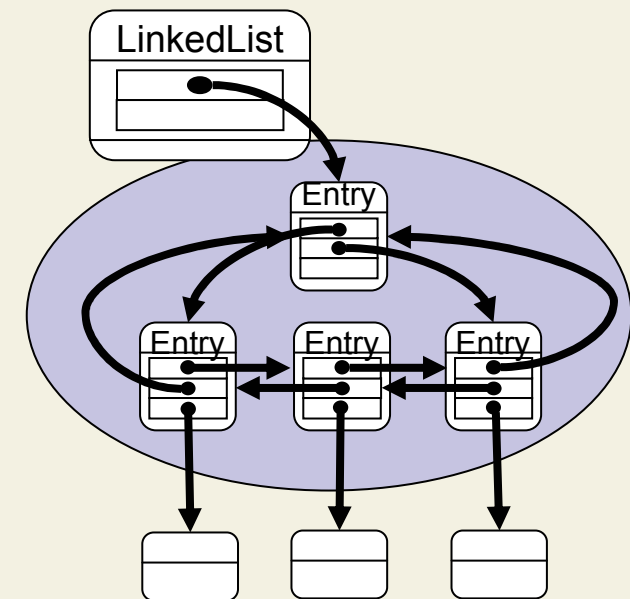
ArrayList – getElems

```
public int[] getElems() {  
    return (int[]) array.clone();  
}
```

- The `/* rep */` array can not be passed out directly as default array
- We create a clone of the internal array and return it as result
- The internal representation is still encapsulated → no *Leaking*

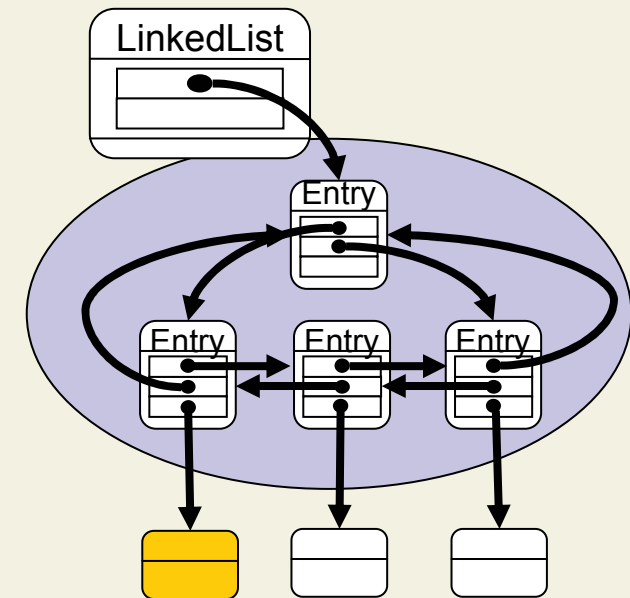
Annotated LinkedList – Setup

```
class LinkedList {  
    private /* rep */ Entry header;  
    private int size;  
  
    public LinkedList() {  
        header = new /* rep */  
            Entry(null, null, null);  
        header.next = header;  
        header.previous = header;  
  
        size = 0;  
    }  
}
```



LinkedList – add

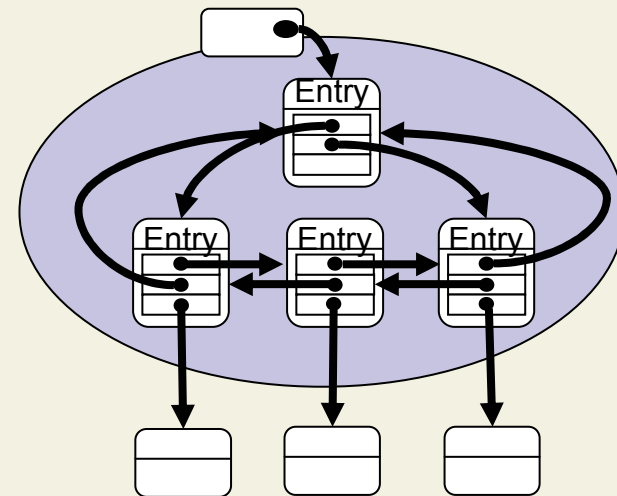
```
public void add( /* arg */ Object o ) {  
    /* rep */ Entry newE =  
        new /* rep */ Entry( o, header,  
                             header.next );  
  
    header.next.previous = newE;  
    header.next = newE;  
  
    ++size;  
}
```



```
public /* arg */ Object get( int idx ) { ... }
```

Helper Class Entry

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

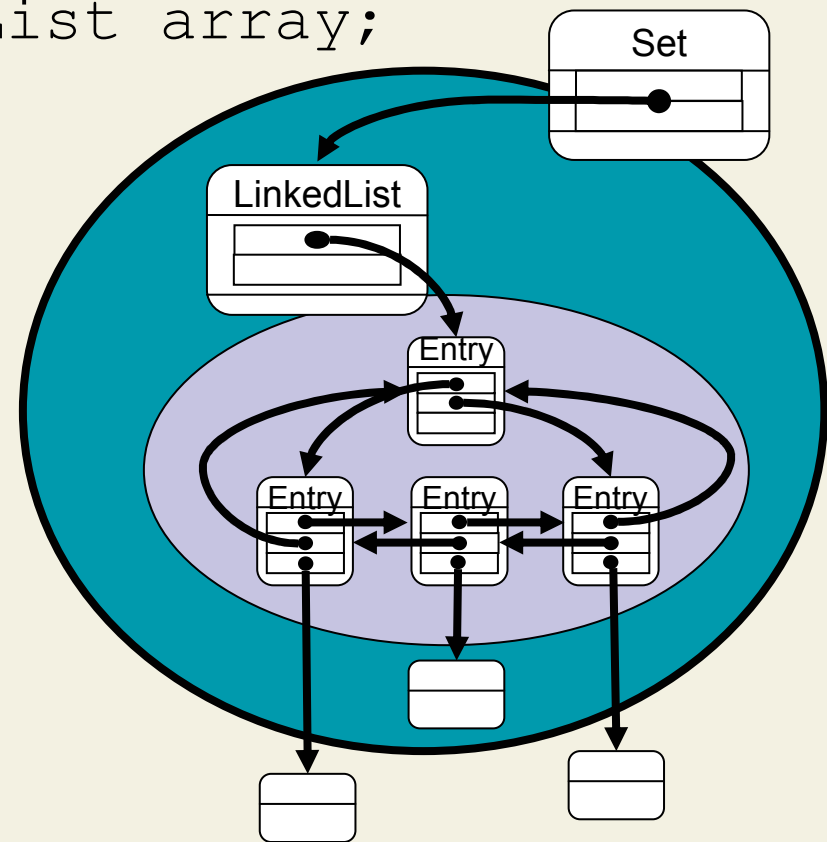


Some Main Program

```
public static void main( String[] args ) {  
  
    LinkedList ll = new LinkedList();  
  
    ll.add( new Integer(20) );  
    ll.add( new Float(2.2f) );  
  
    System.out.println("Element[0]: " +  
        ll.get(0) );  
    System.out.println("Element[1]: " +  
        ll.get(1) );  
}
```


Class Set as user of ArrayList

```
public class Set {  
  
    private /* rep */ ArrayList array;  
  
    public Set() {  
        array = new /* rep */  
            ArrayList();  
    }  
  
    ...  
}
```



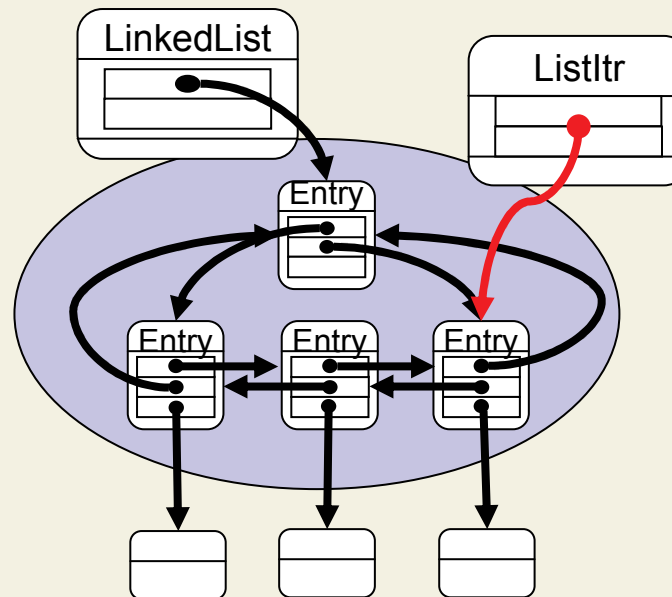
Problems with Annotations

- No help from the compiler, just comments
- Other developers do not have to follow the annotation
- Subclasses do not have to follow the annotation

```
public class BadArrayList  
    extends ArrayList {  
        public int[] leakArray() {  
            return array;  
        }  
    }  
}
```

Just one Object can be the Owner

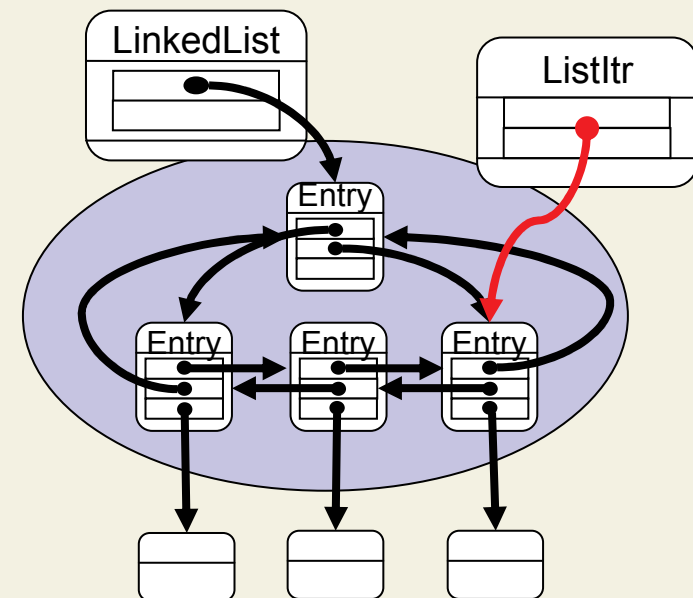
- Sometimes we would like to allow multiple objects access to the internal representation
- E.g. an iterator access to a list



Problems with two Owners

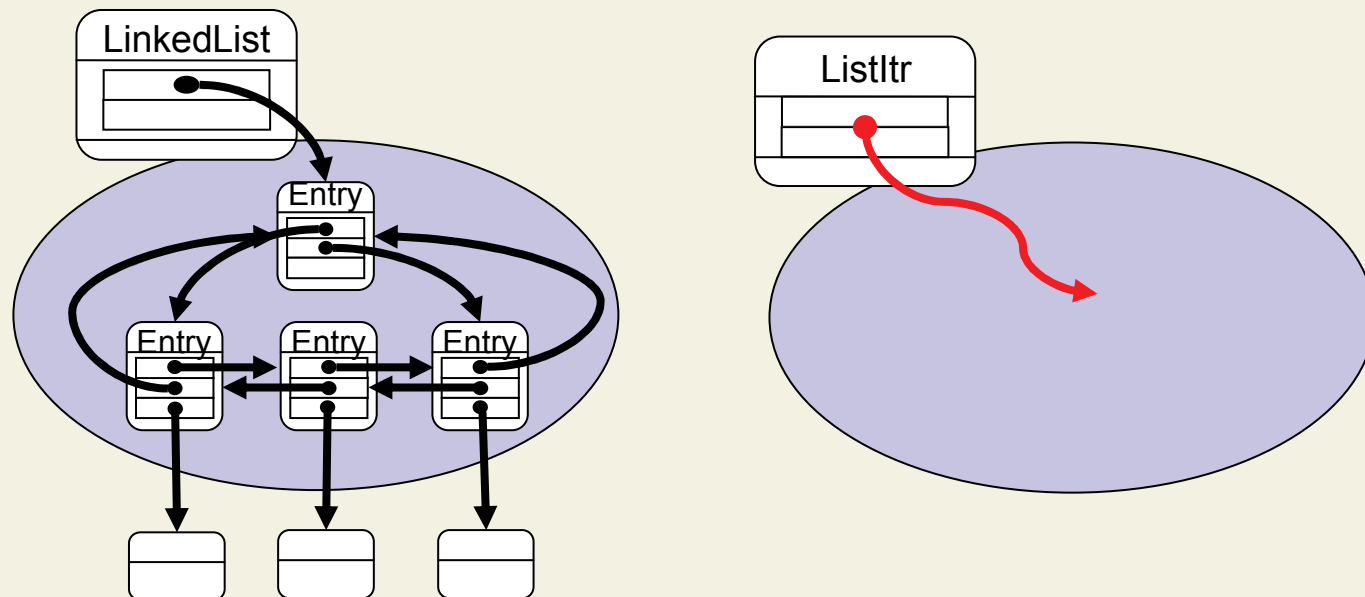
```
ListItr getItr() {  
    ListItr res = new ListItr();  
    res.current = header.next;    }
```

```
class ListItr {  
    /*rep*/ Entry current;  
    /*arg*/ Entry current;  
    Entry current;  
}
```



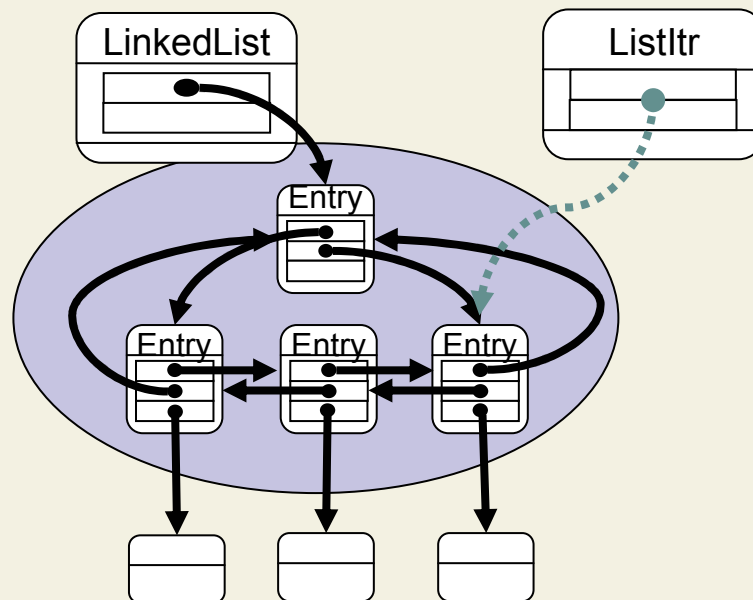
Problems with two Owners

`/* rep */ Entry current;`
would denote an Entry object in Listltr's own context!



Problems with two Owners

`/* arg */ Entry current;`
would denote a readonly reference to an Entry object in the LinkedList context, which might be OK!



Problems with two Owners

`Entry current;`
would denote an `Entry` object on `ListItr`'s level!

