Konzepte objektorientierter Programmierung

Prof. Dr. Peter Müller

Werner Dietl

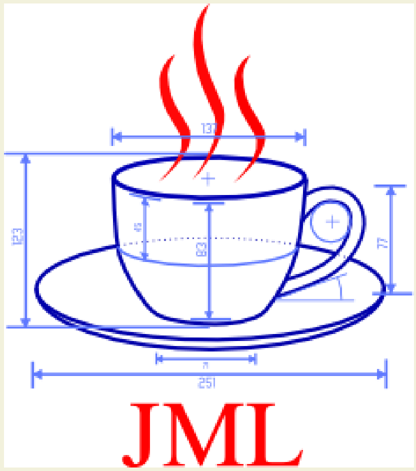
Software Component Technology

Exercises 8: Aliasing

Wintersemester 06/07

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Java Modeling Language JML



- Available at:
  `http://jmlspecs.org/`

- Gary T. Leavens, Yoonsik Cheon.
  Design by Contract with JML.
  `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`

- “The Java Modeling Language (JML) is a behavioral interface specification language that can be used to specify the behavior of Java modules.”

JML Notation

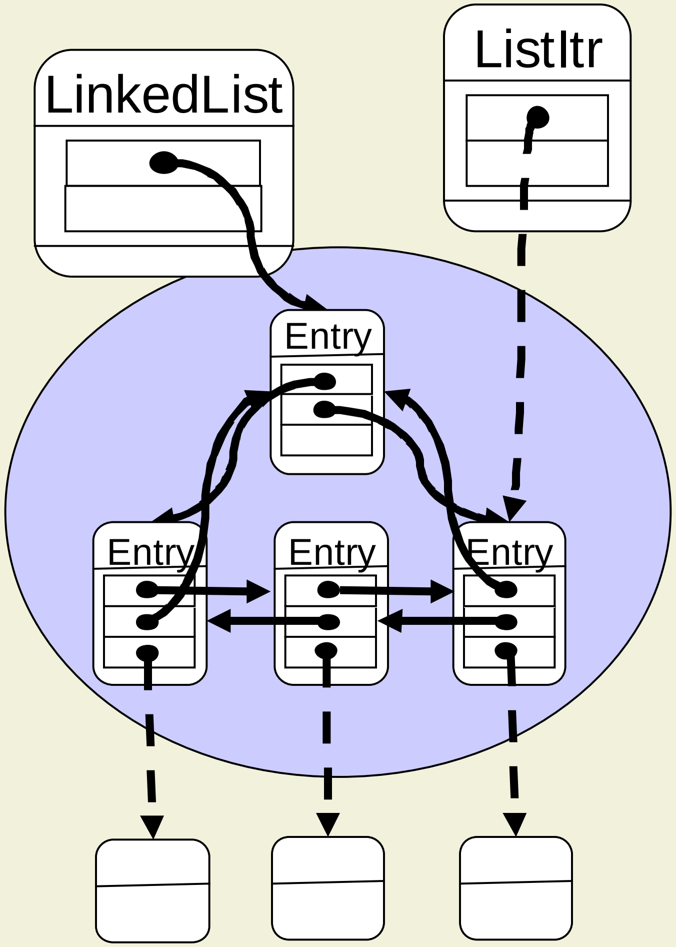
- Additional stuff in special comments:

```
//@ till end of line

/*@ requires x > 0;
  @ ensures x == 0;
  @*/
```

- In the lecture examples, we sometimes leave out the comments and sometimes the @
- More later

Alias Control by Extended Typing

- We introduce different types for the different roles of objects
  - peer types for objects in the **same context as this** (interface objects)
  - rep types for representation objects in the **context owned by this**
  - readonly types for argument objects **in any context**
- Type rules replace the programming discipline



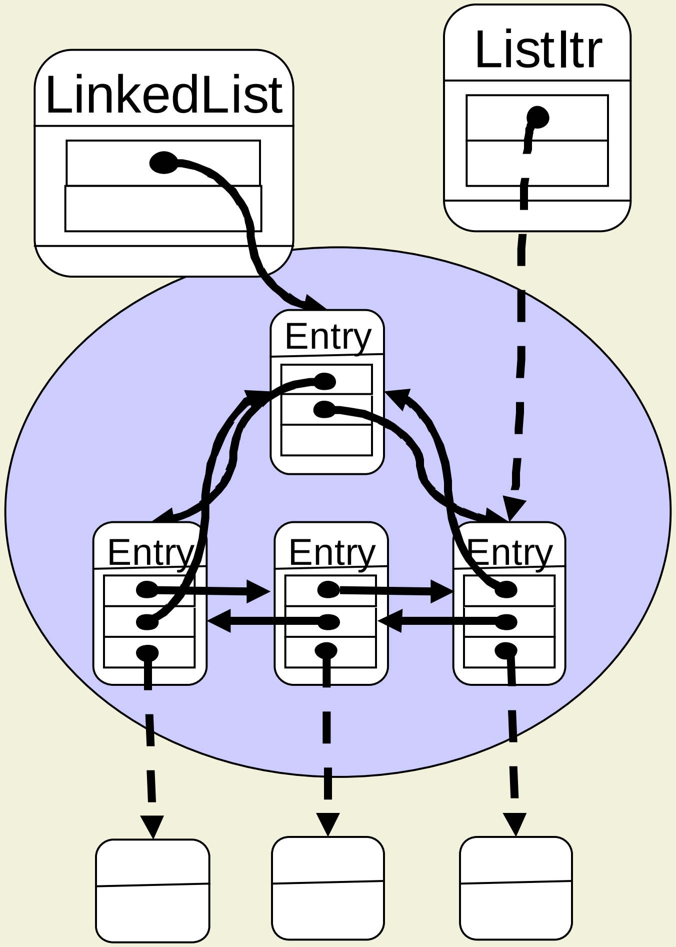
Linked List Example

```
public class LinkedList {
  public void add( readonly Object o ) {...}

  public readonly Object getFirst() {...}
  ...
```

```
LinkedList l = new peer LinkedList();
l.add( new peer Integer(3) );
l.add( new peer Object() );

peer Object o = (peer Object) l.getFirst();
```

Alias Control by Extended Typing

- We introduce different types for the different roles of objects
  - peer types for objects in the **same context as this** (interface objects)
  - Rep types for representation objects in the **context owned by this**
  - Readonly types for argument objects **in any context**
- Type rules replace the programming discipline



Homework – Exercise 1

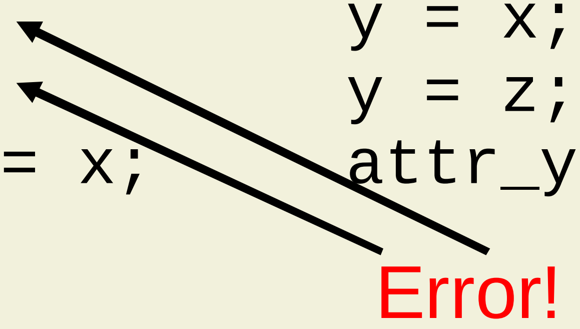
```
class Room {
    private Human m;
    void enter( Human m ) {}
    Human leave() {}
}
public class Building {
    private Room[] rooms;
    void enter( Human m ) {
    rooms[0].enter( m ); }
    Human leave() {
    rooms[0].leave( m ); }
}
```

Homework – Exercise 1

```
class Room {
    private /*arg*/ Human m;
    void enter(/*arg*/ Human m ) {}
    /*arg*/ Human leave() {}
}
public class Building {
    private /*rep*/ Room[] rooms;
    void enter( /*arg*/ Human m ) {
    rooms[0].enter( m ); }
    /*arg*/ Human leave() {
    rooms[0].leave( 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;
  }
}
```



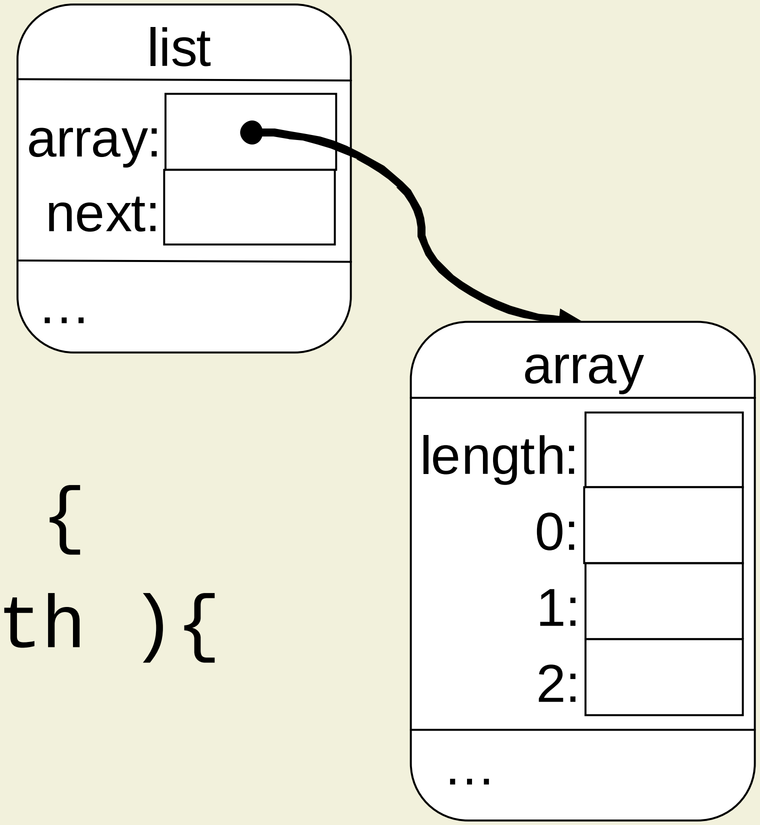
Error!

(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*

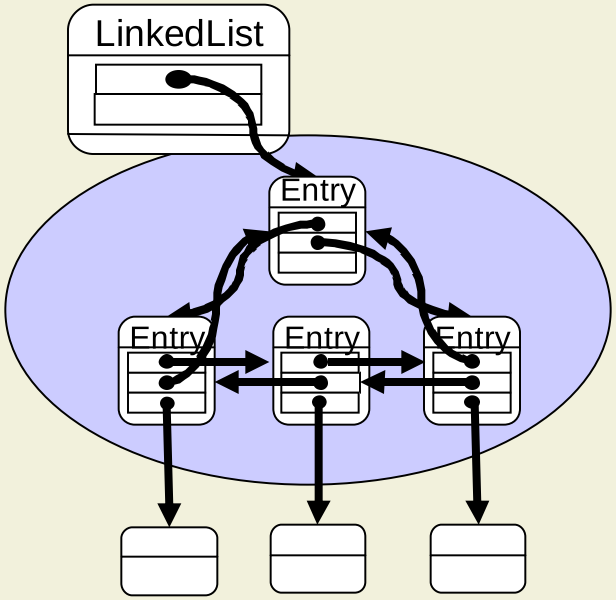
ArrayList – getElems

```
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*

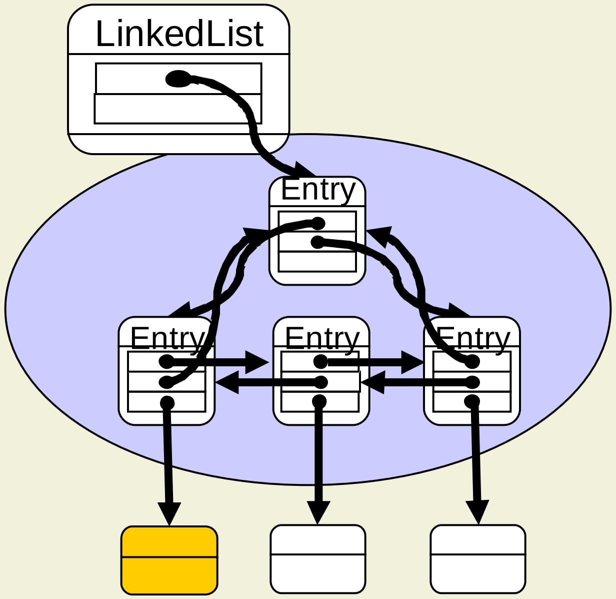
Annotated LinkedList – Setup

```
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

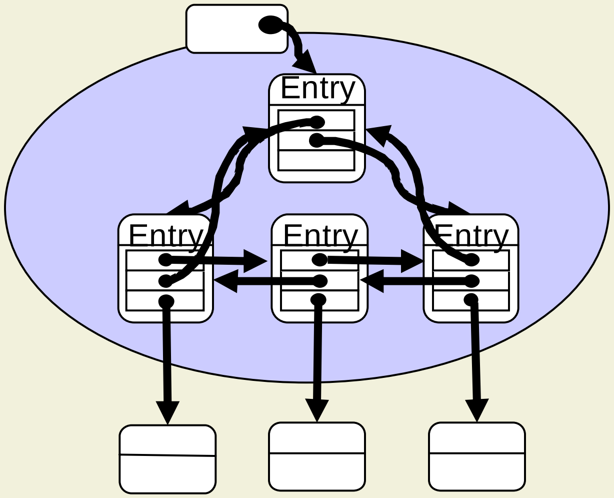
```
public void add( /* arg */ Object o ) {
  /* rep */ Entry newE =
      new /* rep */ Entry( o, header,
        header.next );

  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;
  }
}
```

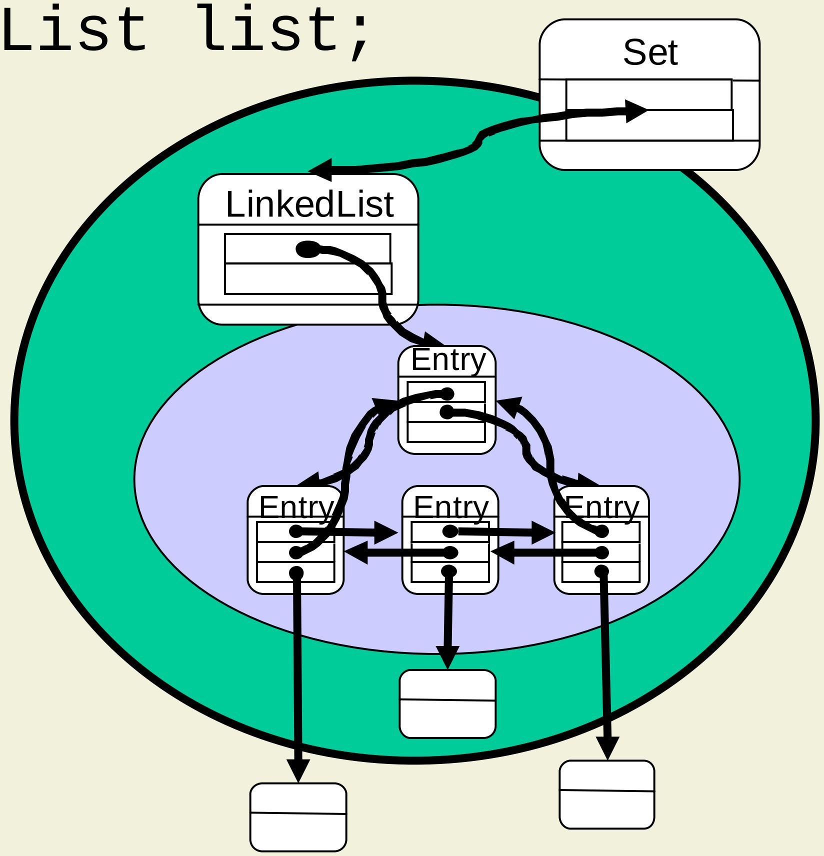


Some Main Program

```
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 LinkedList

```
public class Set {

  private /* rep */ LinkedList list;

  public Set() {
    list = new /* rep */
               LinkedList();
  }

  …

}
```



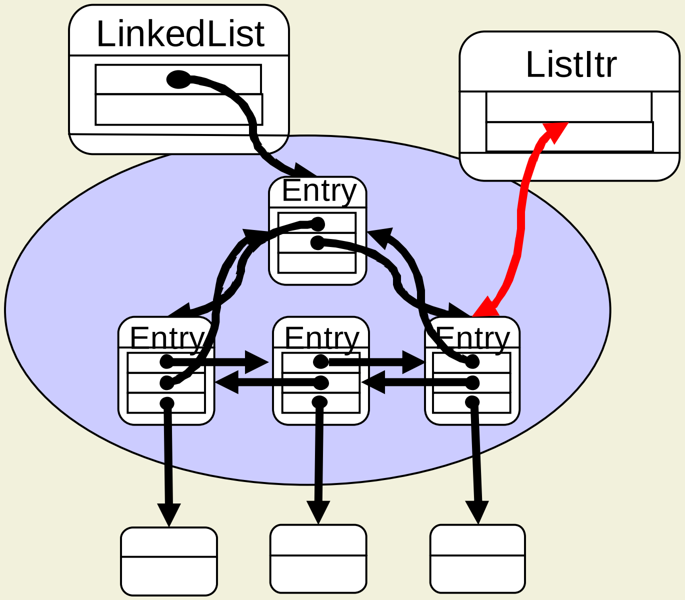
Problems with Annotations

- 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;
     }
}
```

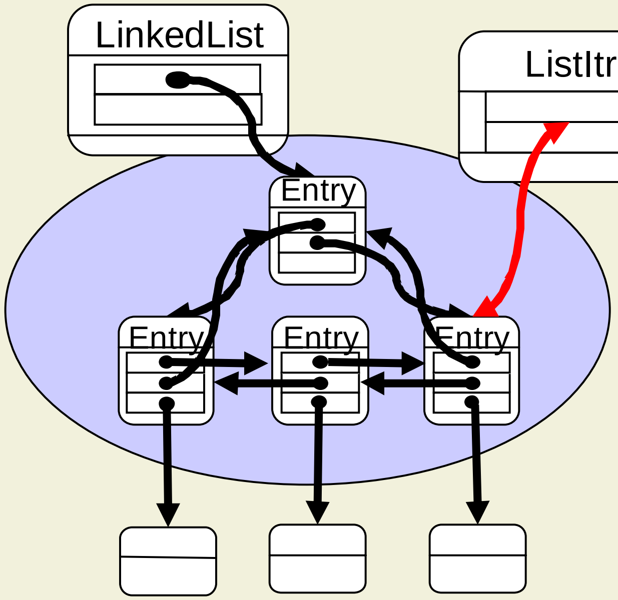
Just one Object can be the Owner

- 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

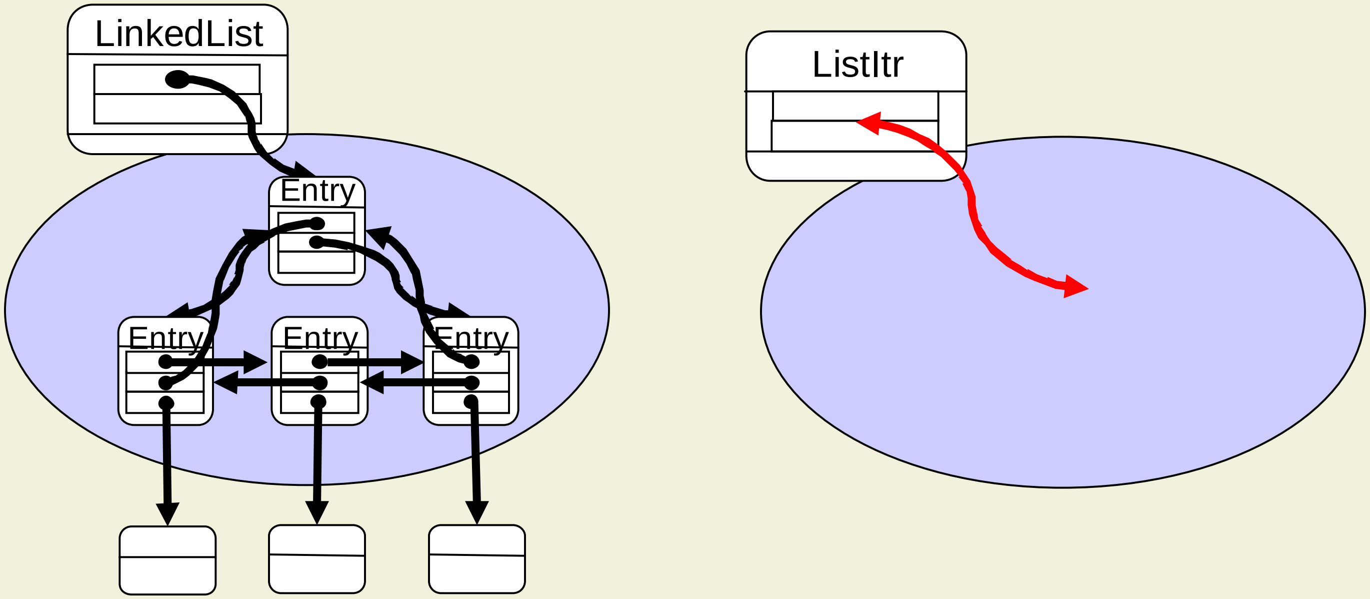
```
ListItr getItr() {
   ListItr res = new ListItr();
   res.current = header.next;    }


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



Problems with two Owners

`/* rep */ Entry current;`

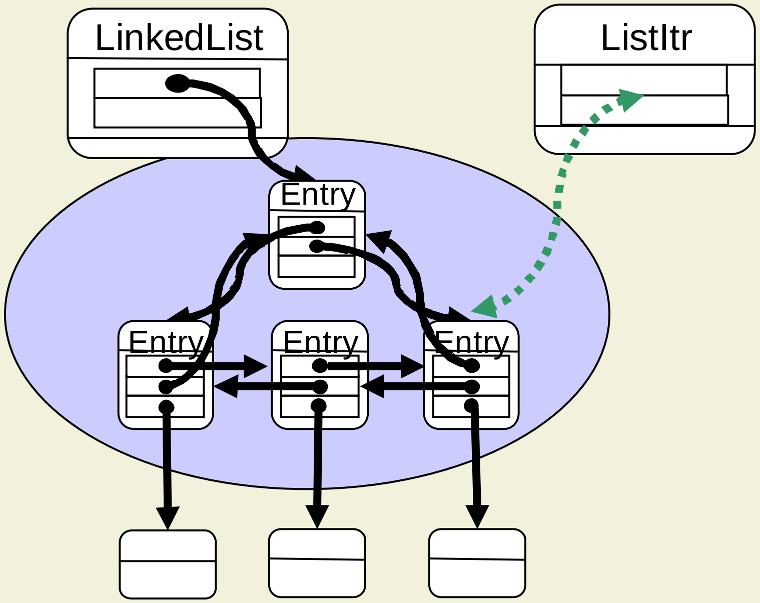
would denote an Entry object in ListItr’s own context!



Problems with two Owners

`/* arg */ Entry current;`

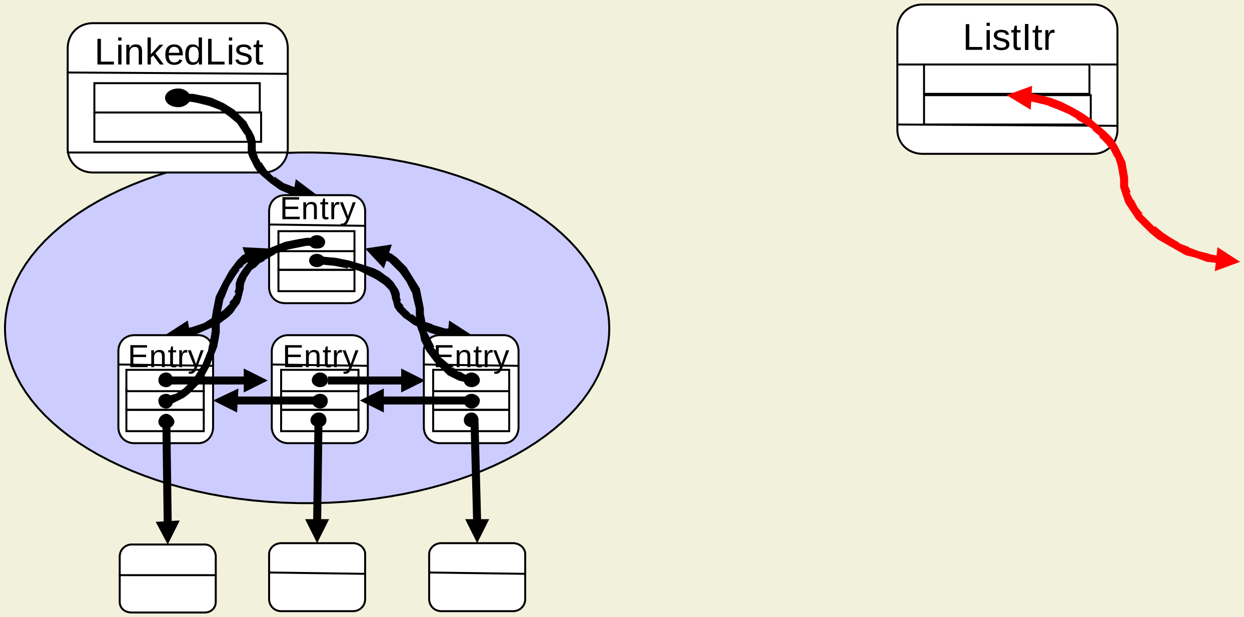
would denote a readonly reference to an Entry object in the LinkedList context, which might be OK!



Problems with two Owners

```
Entry current;
```

would denote an Entry object on ListItr’s level!



C++ `const`

- “When using a pointer, two objects are involved: the pointer itself and the object pointed to. ‘Prefixing’ a declaration of a pointer with `const` makes the object, but not the pointer, a constant. To declare a pointer itself, rather than the object pointed to, to be a constant, we use the declarator operator `*const` instead of plain `*`.”
- Bjarne Stroustrup: “*The C++ Programming Language, Third Edition*”, page 94

C++ `const` is not transitive

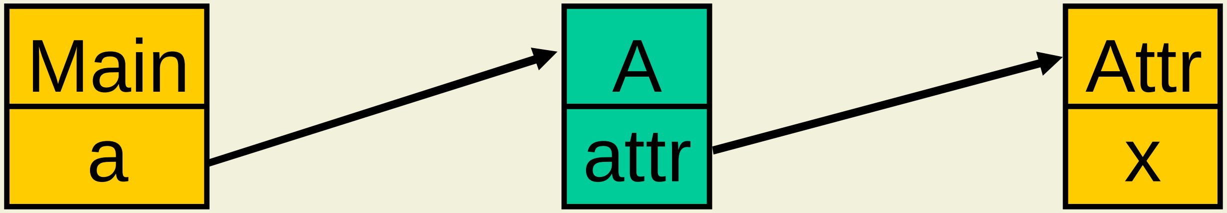
```
class Attr {
  Attr(int i) {x = i;}
  int x;
};

class A {
  A() { attr =
      new Attr(5); }
  Attr *attr;
};
```

```
class Main {
  Main() { a =
       new A(); }

  const A *a;
};
```

C++ `const` is not transitive

```
Main *var = new Main();


var->a->attr->x = 9;
var->a->attr = new Attr(9);
var->a = new A();
```

Compilation Error!



Readonly Access in Java

- Can be explicitly modeled by using read-only super-interfaces for all classes
- Only return read-only interface to clients
- Problems:
  - Not transitive, we need to change all involved interfaces
  - Hard to reuse existing libraries
  - Not safe, the representation can still leak and it is easy to cast the reference to the read-write type

Programming Discipline

- The programming discipline was just comments for the programmer
- No enforcement by the compiler
- No support for multiple developers
- Easy to make errors even for one developer

Extended Types as a Solution

- We extend the type system of our language to support additional modifiers on types
- Not just annotations in comments, but real changes to the language, new keywords and a different type system
- This provides us with direct support from the compiler in checking whether programs are correct
- We have an experimental Java compiler with support for extended types → not standard Java

Types

- Each class or interface `T` **introduces two types**
- **Peer type** *peer(T)*
    - Denoted by `T` in programs
- **Readonly type** *ro(T)*
    - Denoted by **`readonly`** `T` in programs
- For each variable, attribute and parameter we can decide whether it should be read-write or read-only

Subtype Relation

- **Subtyping** among peer and readonly types is **defined as in Java**
  - S extends or implements T

    $\Rightarrow$ *peer(S)* < *peer(T)*
  - S extends or implements T

    $\Rightarrow$ *ro(S)* < ro*(T)*
- **Peer types** are **subtypes of** corresponding **readonly types**
  - *peer(T)* < *ro(T)*

Type Scheme Combinator

- Usually, an access of the form `a.b` has the type of `b`
- Now we need to combine the extended types of a and b to get the type of the result

a * b	*peer(T)*	*ro(T)*
peer(S)	*peer(T)*	*ro(T)*
ro(S)	*ro(T)*	*ro(T)*

- By using this combinator readonly becomes transitive

When do we need the combinator?

- When accessing attributes of an object:

  `class A:  B b;`
  `a.b` ➔ result has type a `* b`

- When invoking methods:

  `class A:  B m();`
  `a.m()` ➔ result has type a `* B`

Questions?