Konzepte objektorientierter Programmierung
– Lecture 7 –

Prof. Dr. Peter Müller

Werner Dietl

Chair of Programming Methodology

Herbstsemester 2008

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Agenda for Today

7. Static Safety and Extended Typing

- 7.1 Type Systems
- 7.2 Readonly Types
- 7.3 Ownership Types

Objectives

- Repetition Readonly Types
- Ownership Types

7. Static Safety and Extended Typing

Readonly Access in Java

```
interface ReadonlyAddress {
  public String getStreet( );
  public String getCity( );
}
```

```
class Address
    implements ReadonlyAddress {
… // as before
}
```

```
class Person {
  private Address addr;
  public ReadonlyAddress
      getAddr(  )
   { return addr; }
  public void setAddr( Address a )
   { addr = a.clone( ); }
  …
}
```

- Address objects are returned as ReadonlyAddress
- Clients use only the methods in this interface

Problems of Java Solution

- Solution does not work for
  - Reused library classes that do not implement a readonly interface
  - Arrays, fields, non-public methods
- Solution is not safe
  - Readwrite aliases can occur, e.g., by capturing
  - Clients can use casts to get full access

```
class Person {
  private Address addr;
  public ReadonlyAddress
      getAddr( )    { return addr; }
  public void setAddr( Address a )
   { addr = a.clone( ); }
  … }
```

```
void m( Person p ) {
  ReadonlyAddress ra =
                          p.getAddr( );
  Address a = (Address) ra;
  a.setCity( “Hagen” );
}
```

Pure Methods

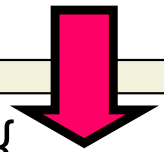
- Tag side-effect free methods as **pure**
- Pure methods
  - Must not contain writing attribute access
  - Must not invoke non-pure methods
  - Must not create objects
  - Can only be overridden by pure methods

```
class Address {
  private String street;
  private String city;
  public pure String
                    getStreet( ) { … }
  public void setStreet( String s )
    { … }
  public pure String
                    getCity( )    { … }
  public void setCity( String s )
    { … }
  …
}
```

Types

- Each class or interface T **introduces two types**
- **Peer type** *peer T*
  - Denoted by T or **peer** T in programs
- **Readonly type** *ro T*
  - Denoted by **readonly** T in programs

```
class Person {
  private Address addr;
  public ReadonlyAddress
         getAddr( ) { return addr; }
  public void setAddr( Address a )
   { addr = a.clone( ); }
  … }
```



```
class Person {
  private Address addr;
  public pure readonly Address
         getAddr( ) { … }
  …
}
```

Subtype Relation

- **Subtyping** among peer and readonly types is **defined as in Java**
  - S extends or implements T ⇒ *peer S* < *peer T*
  - S extends or implements T ⇒ *ro S* < ro *T*
- **peer types** are **subtypes of** corresponding **readonly types**
  - *peer T* < *ro T*

```
class T { … }
```

```
class S extends T { … }
```

```
S peerS = …
T peerT  = …
readonly S roS = …
readonly T roT = …
```

```
peerT    = peerS;
roT      = roS ;
roT      = peerT;
```

```
peerT  = roT ;
```

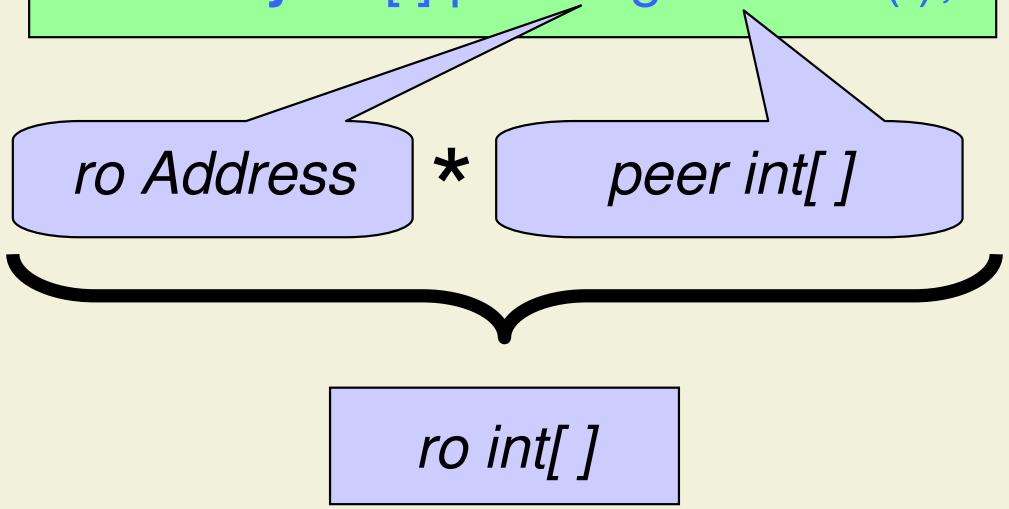
Type Rules: Transitive Readonly (cont'd)

- The type of
  - An attribute access
  - An array access
  - A method invocation

  expression is determined by the type combinator function *

*	*peer T*	*ro T*
peer S	*peer T*	*ro T*
ro S	*ro T*	*ro T*

```
Person p = …
readonly Address a;
a = p.getAddr( );

readonly int[ ] ph = a.getPhone( );
```



Type Rules: Readonly Access

- Expressions of readonly types must not occur
  - As target of a **writing attribute access**
  - As target of a **writing array access**
  - As target of an **invocation** of a **non-pure method**
- Readonly types must not be **cast to peer types**

```
readonly Address roa;
roa.street = “Rämistrasse”;
roa.phone[ 0 ] = 41;
roa.setCity( “Hagen” );
```

```
readonly Address roa;
Address a = ( Address ) roa;
```

Discussion

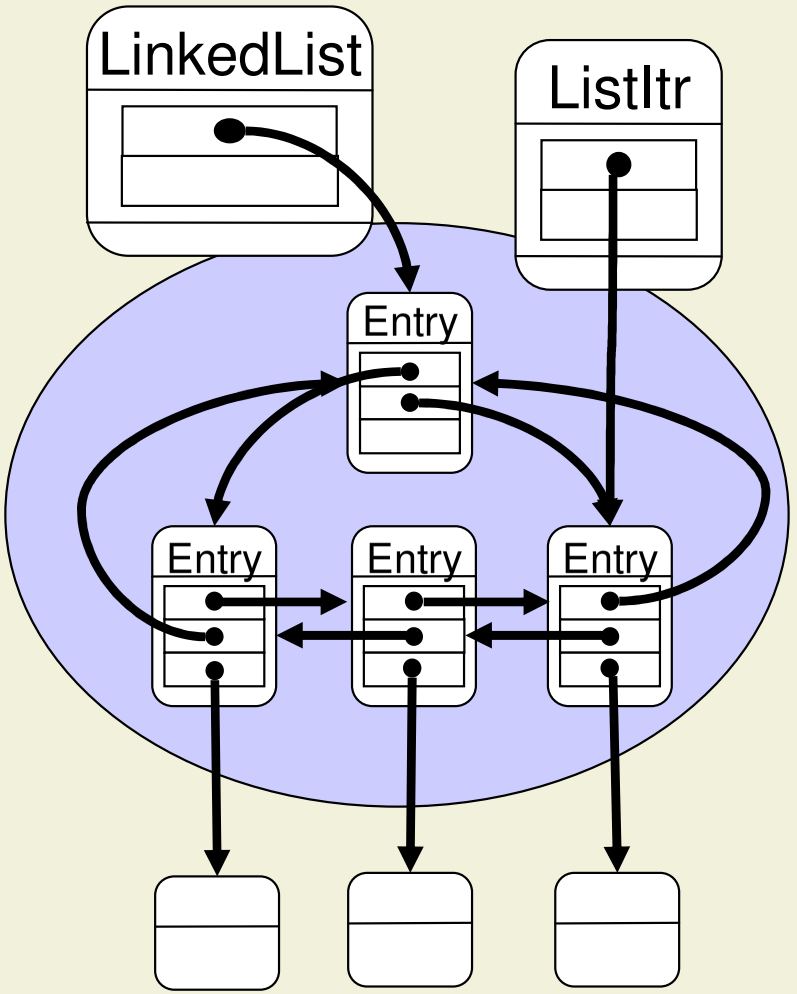
- Readonly types enable **safe sharing of objects**
- All rules for pure methods and readonly types can be **checked statically by a compiler**

- Readonly types solve problems of interface solution
  - Reused library classes
  - Arrays, attributes, and non-public methods
  - Casts
- Readwrite aliases can still occur, e.g., by capturing

7. Static Safety and Extended Typing

Roles in Object Structures

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

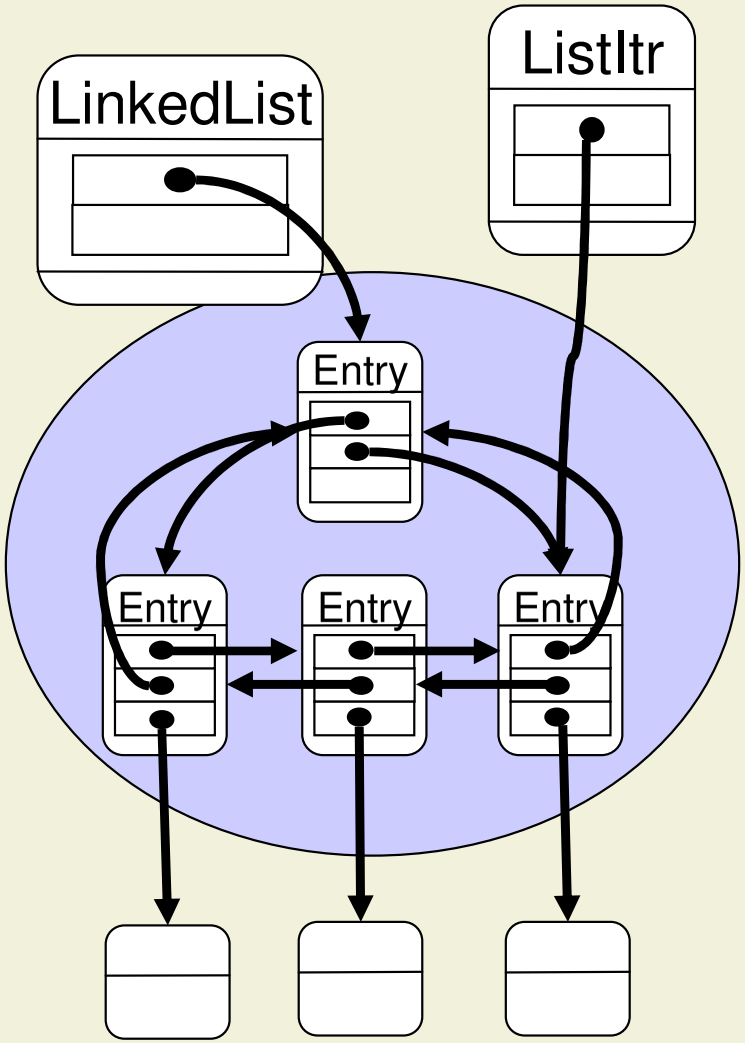


(Simplified) Programming Discipline

- **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

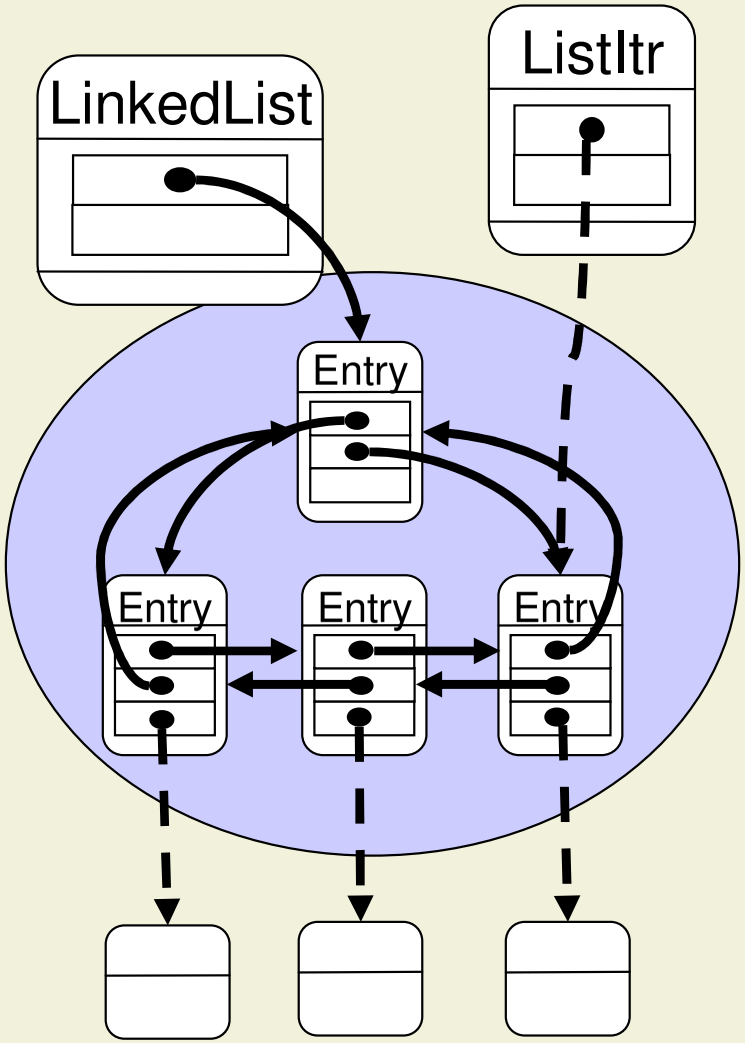
Ownership Model

- The **object store** is partitioned into **contexts**
- Each object **belongs to exactly one context**
- Each context has at most one **owner object**
  - The owner does not belong to the context it owns
- Contexts are **hierarchical**



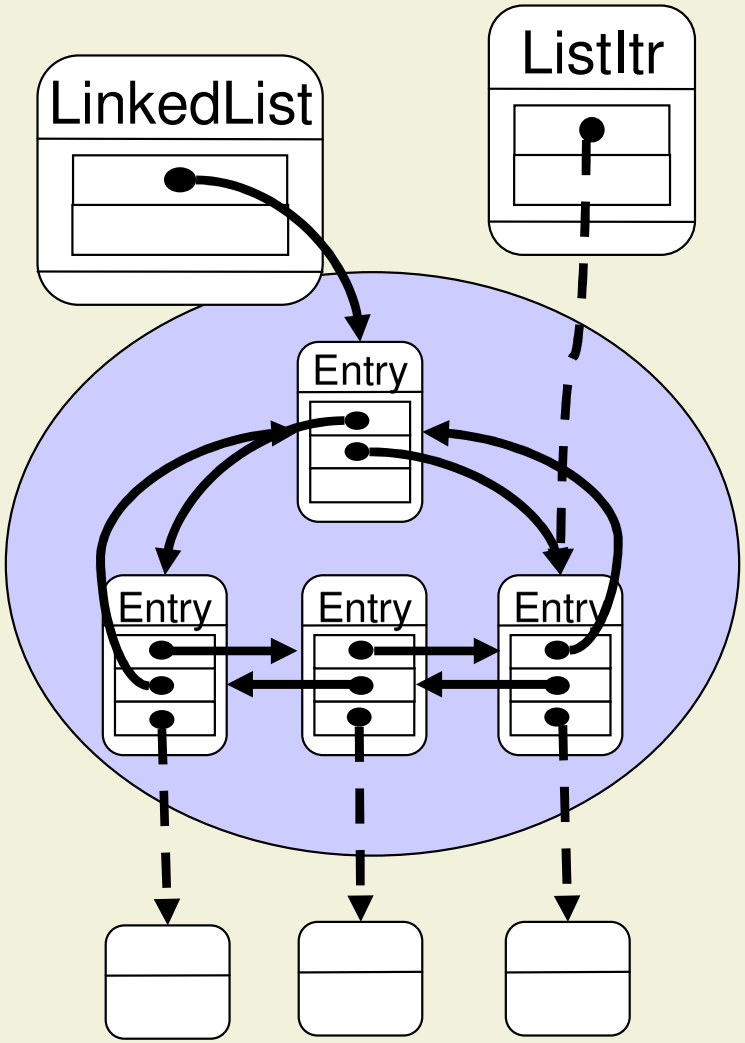
Owner-as-Modifier Discipline

- The owner is the only object outside a context that can have a readwrite reference to objects inside
- Objects inside a context cannot have readwrite references to objects outside



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



Types

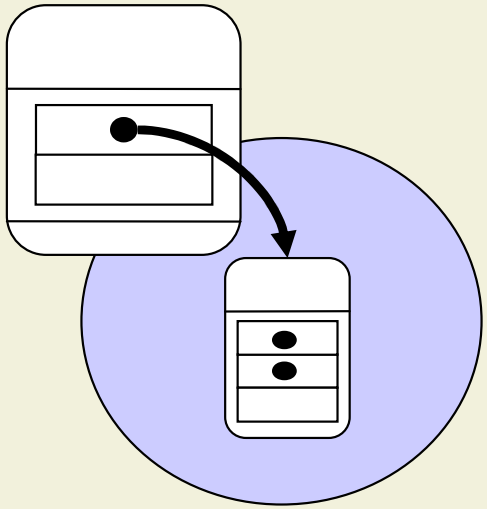
- Each class or interface C introduces **three types**
- Peer and readonly types
- **Rep type** *rep T*
  - Denoted by **rep** T in programs

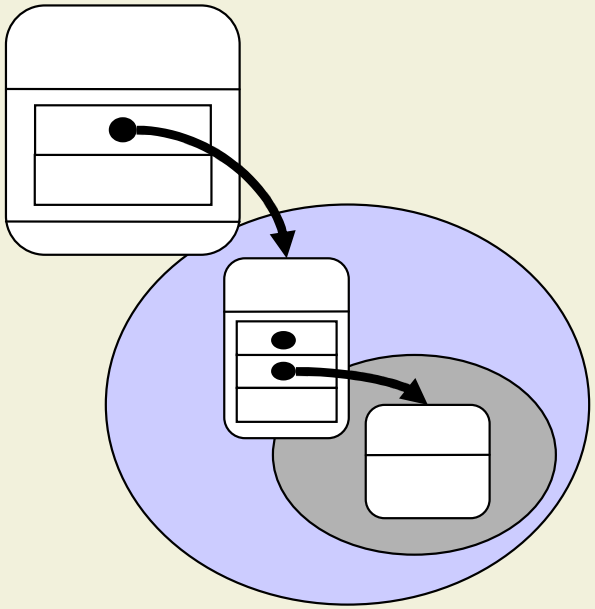
```
class LinkedList {
  private rep Entry header;
  public void add( readonly Object o ) {
    rep Entry newE =
      new rep Entry( o, header, header.previous );
    … }
}
```

```
class Entry {
  private readonly Object element;
  private peer Entry previous, next;
  public Entry( readonly Object o,
                peer Entry p, peer Entry n ) { … }
}
```

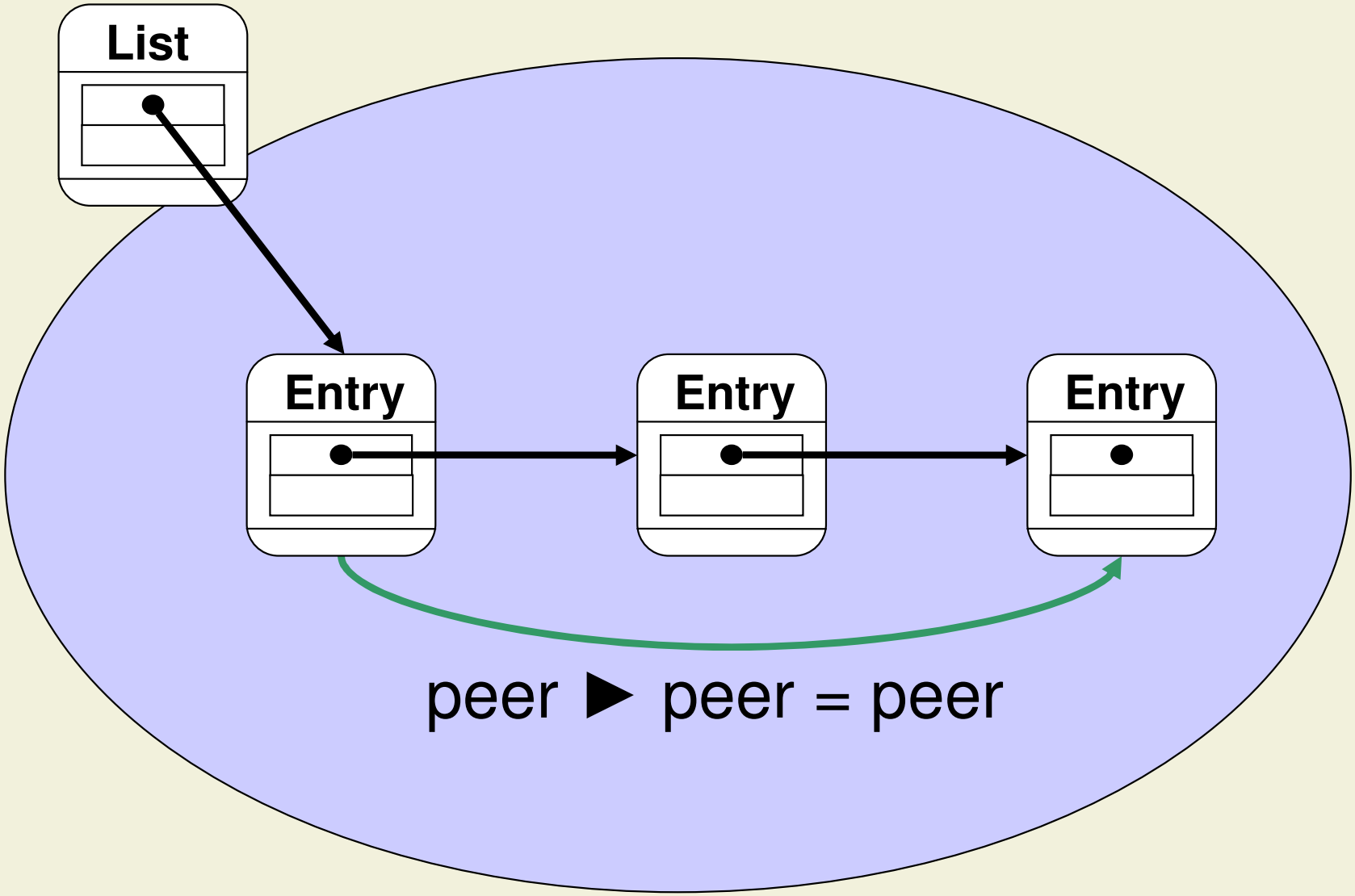
Types Rules: Access to Contexts

- Objects in different **contexts must not be confused**
- A peer type indicates that an object is in the same context as **this**
- A rep type indicates that an object is owned by **this**
- At certain points this viewpoint is changed



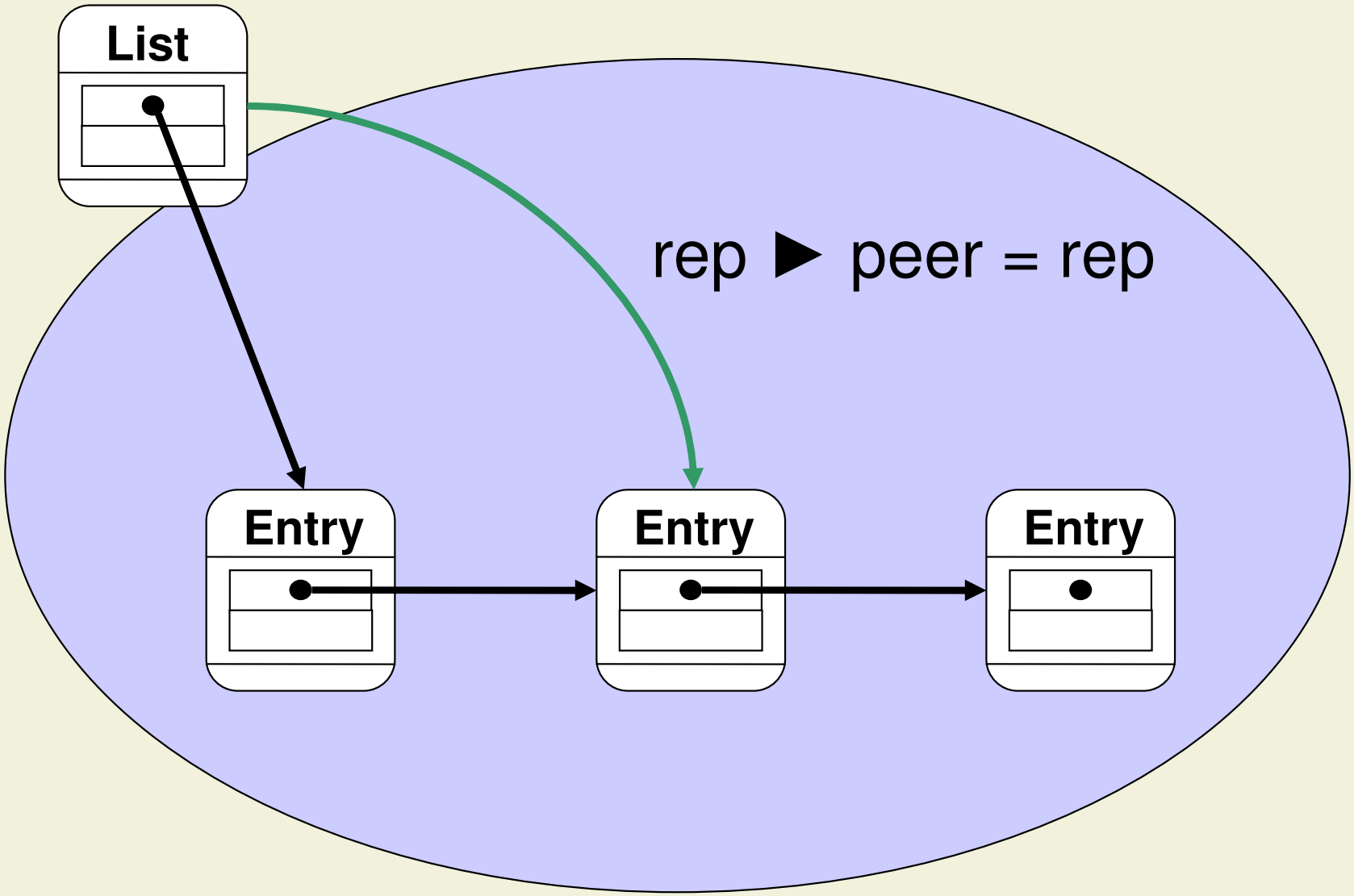


Viewpoint Adaptation

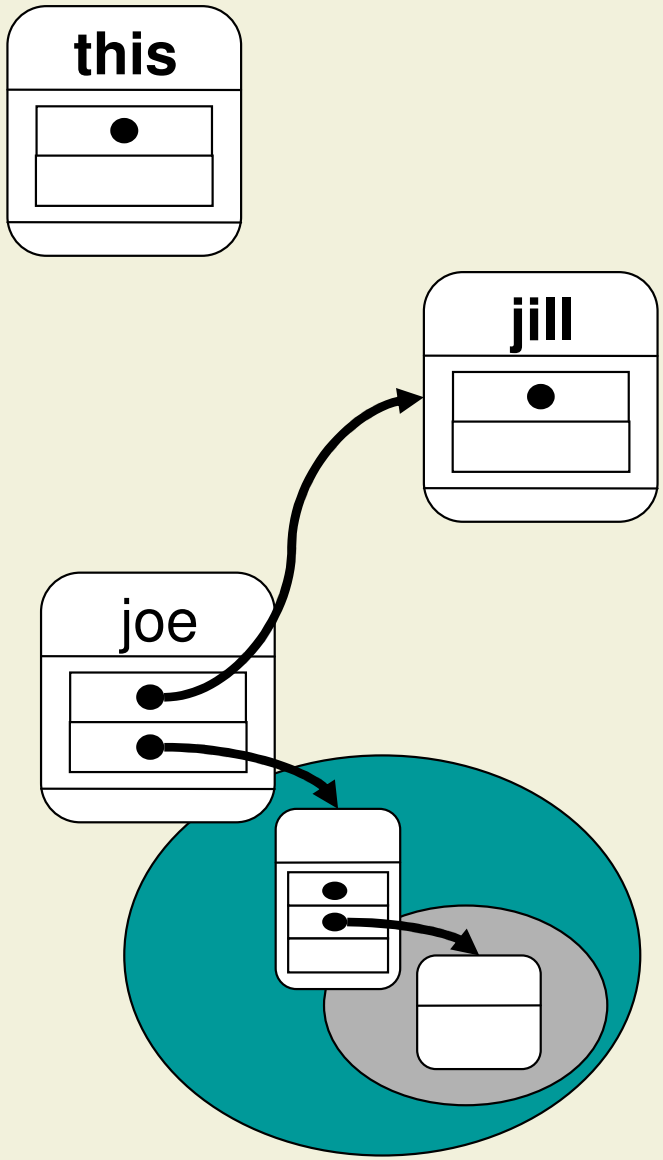


Viewpoint Adaptation

u ▶ ro = ro



Read vs. Write Access




```
class Person {
  public rep Address addr;
  public peer Person spouse;

  … }
```

```
peer Person joe, jill;
```

```
joe.spouse = jill;
```

```
readonly Address roa = joe.addr;
```

```
joe.addr = new rep Address();
```

Read vs. Write Access – lost Modifier

- Only two ownership relations expressible statically
- Internal modifier **lost** for unknown owner
- Reading fields with unknown ownership allowed
- Updating them not

```
class Person {
  public rep Address addr;
  public peer Person spouse;

  … }
```

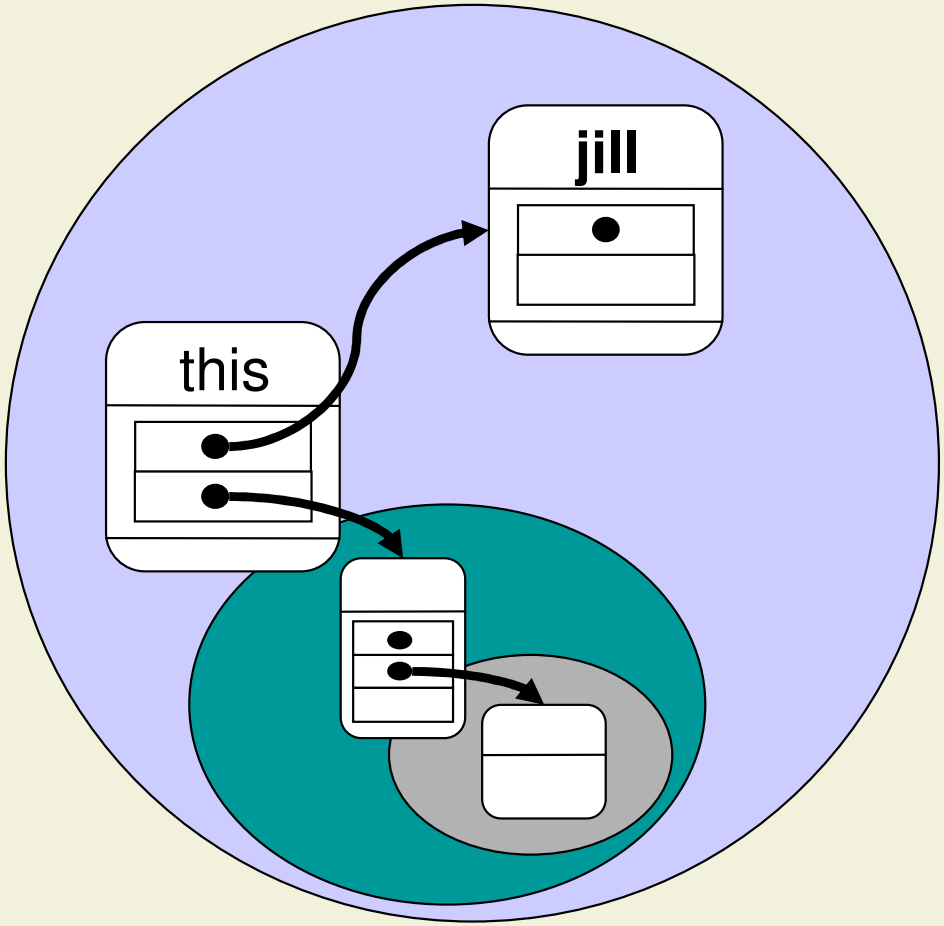
```
peer Person joe, jill;
```

```
joe.spouse = jill;
```

```
readonly Address roa = joe.addr;
```

```
joe.addr = new rep Address();
```

Current object is special




```
class Person {
  public rep Address addr;
  public peer Person spouse;

  … }
```

```
peer Person joe, jill;
```

```
joe.addr = new rep Address();
```

```
this.addr = new rep Address();
```

- Internal modifier **self** only for the **this** literal
- No case distinction on expressions needed

Type Rules: The Type Combinator

▶	*peer T*	*rep T*	*lost T*	*ro T*
self S	*peer T*	*rep T*	*lost T*	*ro T*
peer S	*peer T*	*lost T*	*lost T*	*ro T*
rep S	*rep T*	*lost T*	*lost T*	*ro T*
lost S	*lost T*	*lost T*	*lost T*	*ro T*
ro S	*lost T*	*lost T*	*lost T*	*ro T*

Subtype Relation – rep Types

- **Subtyping** among rep types is **defined as in Java**
  - S extends or implements T ⇒ *rep S* < rep *T*
- **Rep types** are **subtypes of** corresponding **readonly types**
  - *rep T* < *ro T*
- **No subtype** relation between **peer** and **rep types**

```
class T { … }
```

```
class S extends T { … }
```

```
T peerT  = …
readonly T roT = …
rep S repS = …
rep T repT = …
```

```
repT     = repS;
roT      = repT ;
```

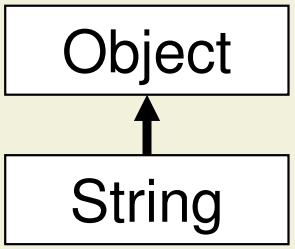
```
repT     = peerT;
peerT    = repT ;
repT     = roT;
```

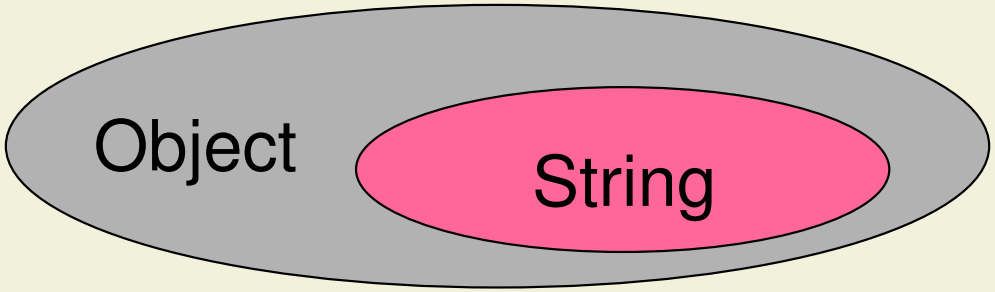
Subtype Relation – self and lost Types

- **self types** are **subtypes of** corresponding **peer types**
  - *self T < peer T*
- **peer** and **rep types** are **subtypes of** corresponding **lost types**
  - *peer T < lost T*
  - *rep T < lost T*
- **all types** are **subtypes of** corresponding **readonly type**
  - *u T <: ro T*

Types

- Definition:
  A type is a set of values sharing some properties. A value v has type T if v is an element of T.
- *Properties*: Available methods, attributes, etc.
- The **subtype relation** corresponds to the **subset relation**





- Usually, each class or interface of a program defines a type

Type Rules: Attribute Access

- The field read

`v = exp.f;`

is correctly typed if

- exp is correctly typed
- $\tau(exp) \blacktriangleright \tau(f) <: \tau(v)$

- The field write

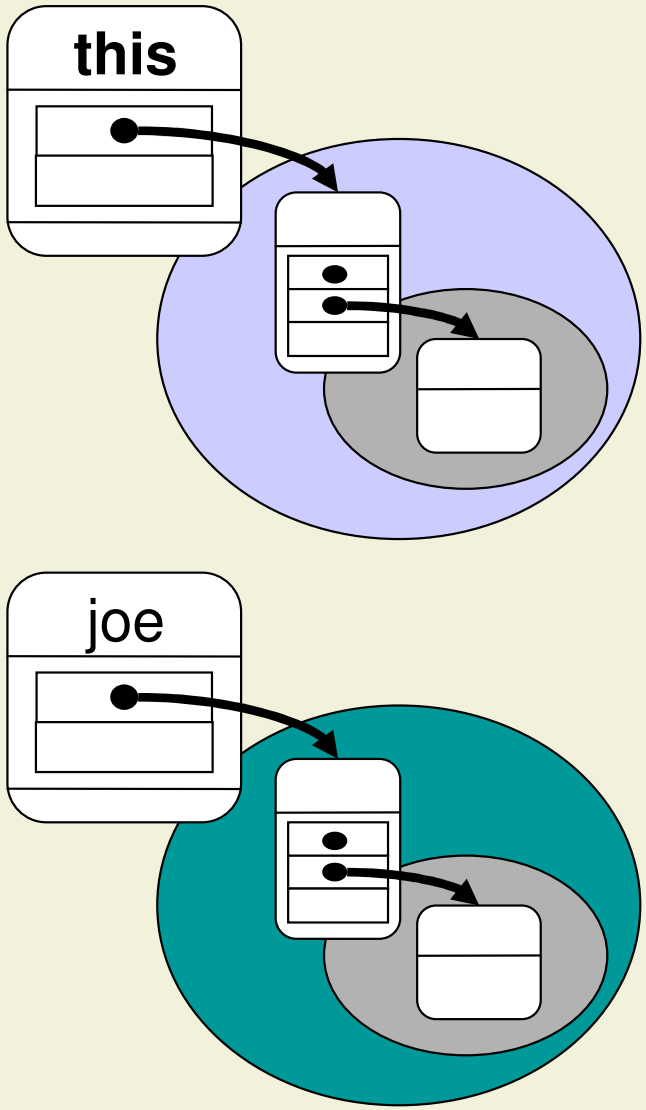
`exp.f = v;`

is correctly typed if

- exp is correctly typed
- $\tau(v) <: \tau(exp) \blacktriangleright \tau(f)$
- **lost** not in $\tau(exp) \blacktriangleright \tau(f)$

- Analogous rules are used for method invocations

Examples: Attribute Access



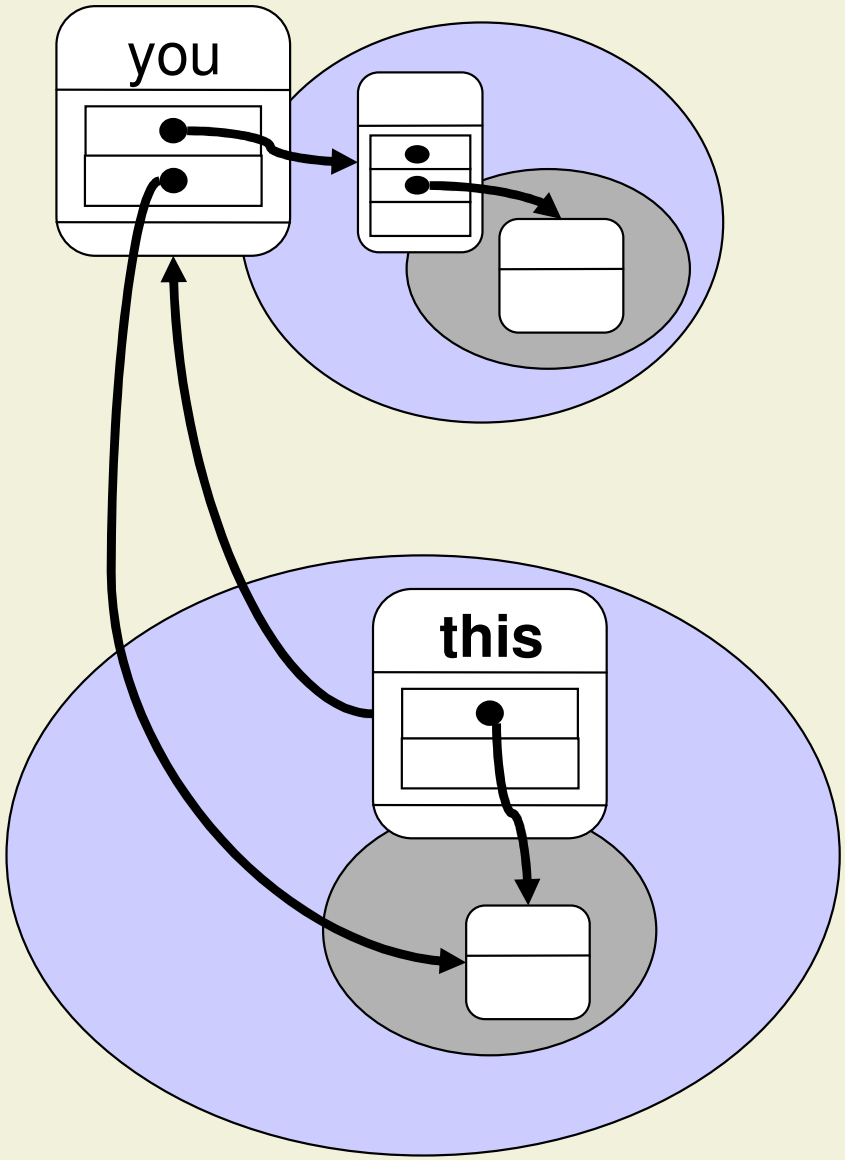

```
class Person {
  public rep Address addr;
  … }
```

```
class Address {
  public rep int[ ] phone;
  … }
```

```
rep Address a = this.addr;
peer Person joe = …;
readonly Address roa = joe.addr;
```

```
rep int[ ] no = this.addr.phone;
rep Address a = joe.addr;
```

Examples: Write on readonly target




```
class Person {
  public rep Address addr;
  public readonly Address favorite;
  … }
```

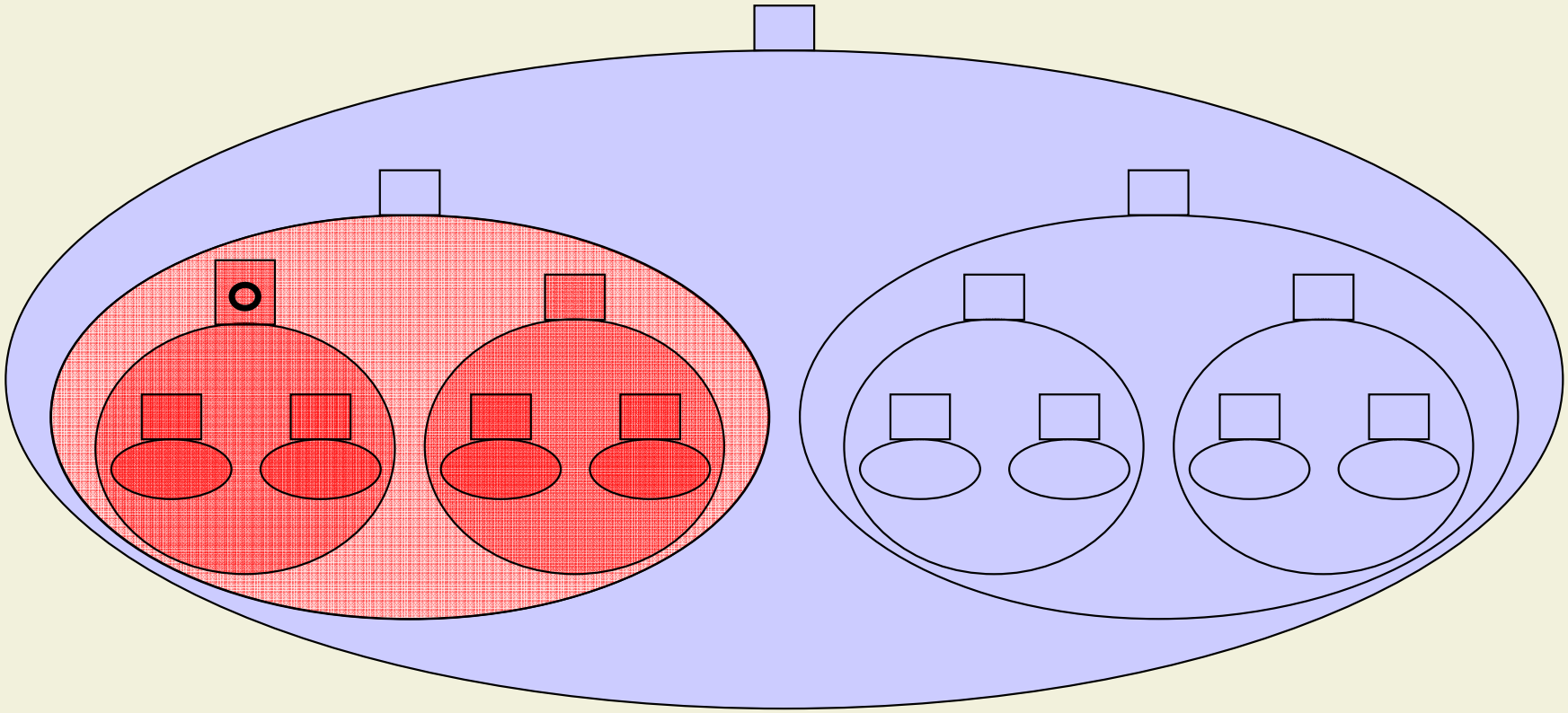
```
readonly Person you = …;
you.favorite = new rep Address();
```

```
you.addr = new rep Address();
you.addr = null;
```

Owner-as-Modifier Discipline

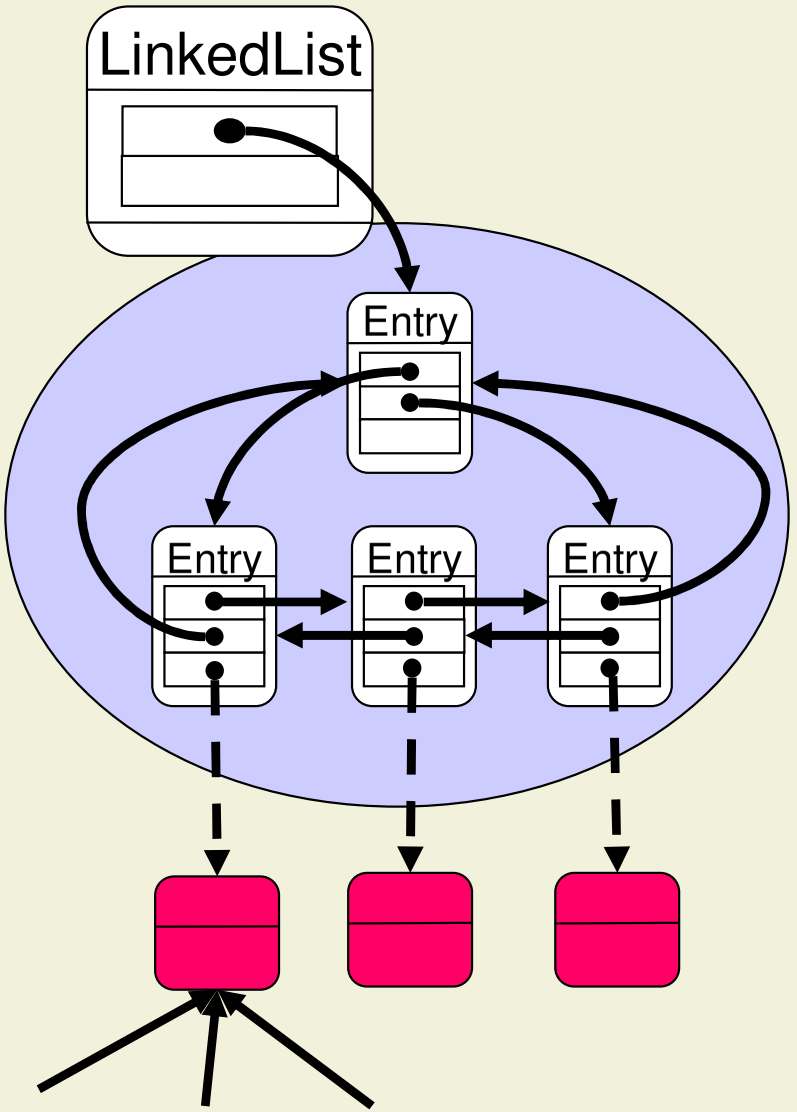
- Rules allow update of readonly fields on any target
- Separate rules enforce owner-as-modifier:
  - field write is only valid if τ(exp) is **peer** or **rep**
  - method call is only valid if
    - τ(exp) is **peer** or **rep** or
    - called method is **pure**
- Only objects directly or indirectly owned by the owner of the current object can be modified

Owner-as-Modifier Discipline (cont'd)



No Argument Dependence

- Argument objects have **readonly types**
- Argument objects may be **freely aliased**
- Invariants **must not depend** on fields of objects referenced through readonly types




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

Dynamic Types

- At compile time, each class or interface C introduces five types
  - *self C, peer C, rep C, lost C, ro C*
- At runtime, the dynamic type of an object consists of its class and its context
- Information about dynamic types can be used to cast readonly types with dynamic checks

```
readonly Address roa = …;
/* dynamic check whether this and roa belong to the same context */
peer Address a = ( peer Address ) roa;
```

Implementation

- Universe Type System is part of JML
  http://jmlspecs.org/
- Syntax:
  - top level: **rep** C f;
  - JML comments: /*@ rep @*/ C f;
  - JML comments 2: /*@ \rep @*/ C f;
- Type checker, runtime system, bytecode storage
- Eclipse plug-in for easy use

http://pm.inf.ethz.ch/research/universes/

Static Type Safety

- Definition:
  A programming language is called type-safe if its design prevents type errors.

- Type-safe object-oriented languages guarantee the following type invariant:
  In every execution state, the type of the value held by variable v is a subtype of the declared type of v

- Type safety guarantees the absence of certain runtime errors

Achievements

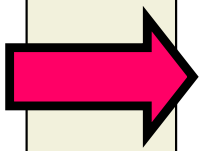
- Rep and readonly types enable **encapsulation of whole object structures**
- Encapsulation **cannot be violated** by subclasses, via casts, etc.
- The technique **fully supports subclassing**
  - In contrast to solutions with final, private inner classes, etc.

```
class ArrayList {
  protected rep int[ ] array;
  private int next;
  …
}
```

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

Exchanging Implementations

```
class ArrayList {
  private int[ ] array;
  private int next;

  // requires ia != null
  // ensures ∀i. 0<=i<ia.length:
  //                isElem( old( ia[ i ] ) )
  public void addElems( int[ ] ia )
    { array = ia; next = ia.length; }

  …
}
```

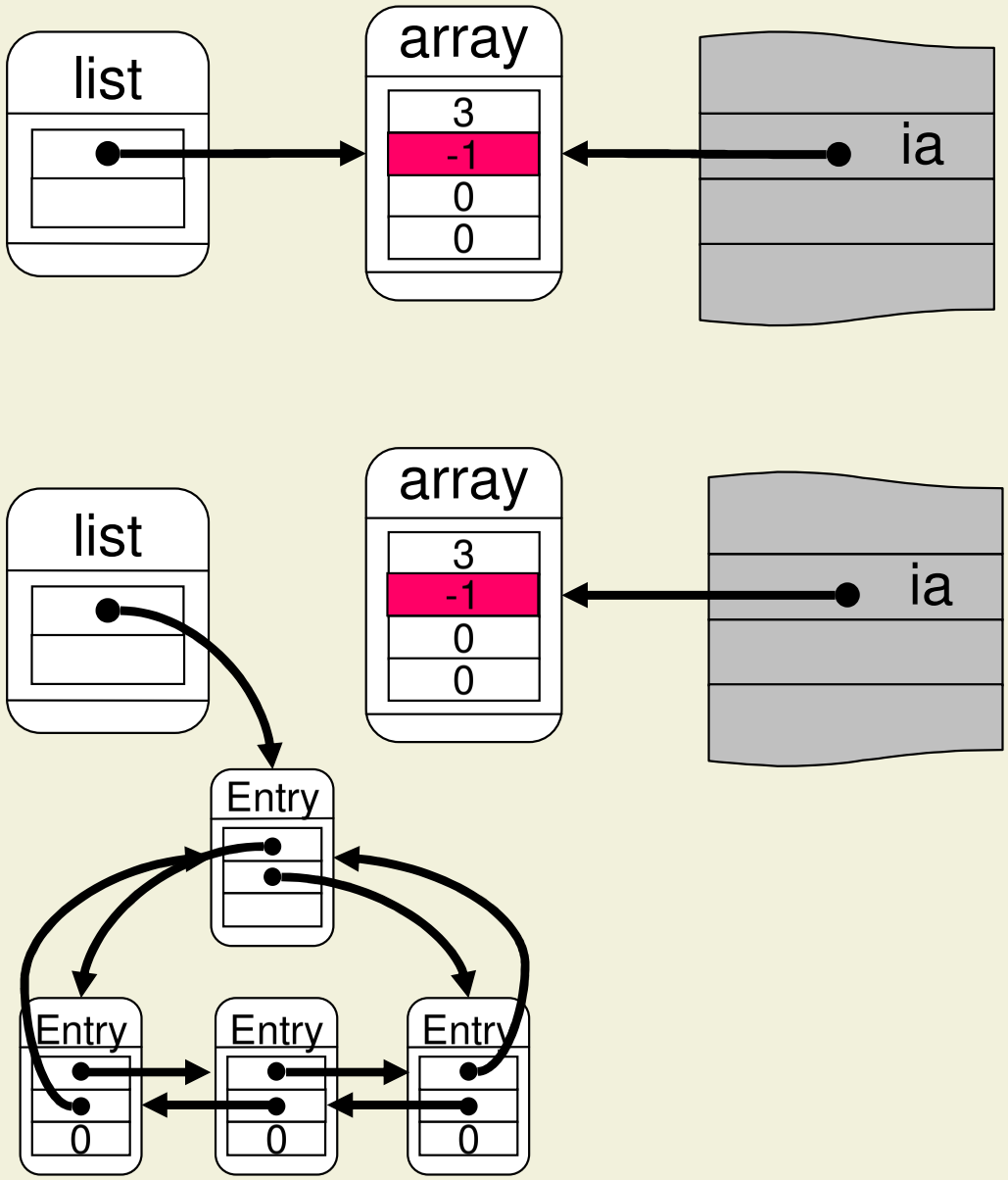


```
class ArrayList {
  private Entry header;

  // requires ia != null
  // ensures ∀i. 0<=i<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!**

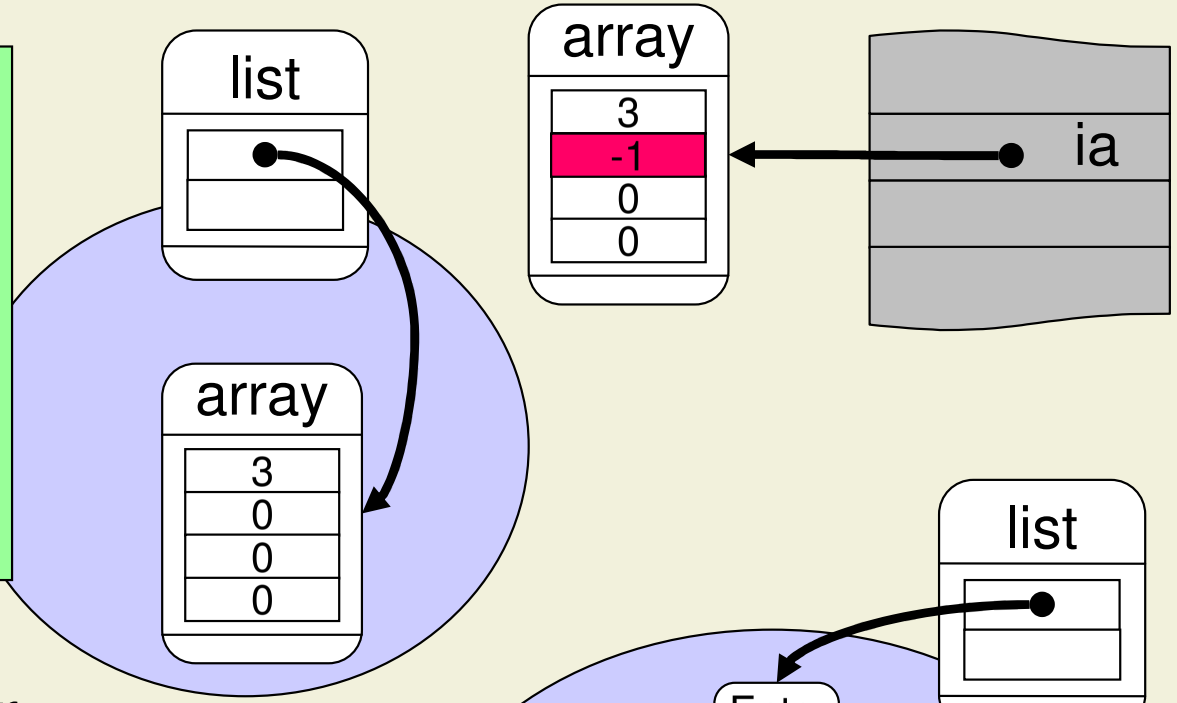
Exchanging Implementations – UTS

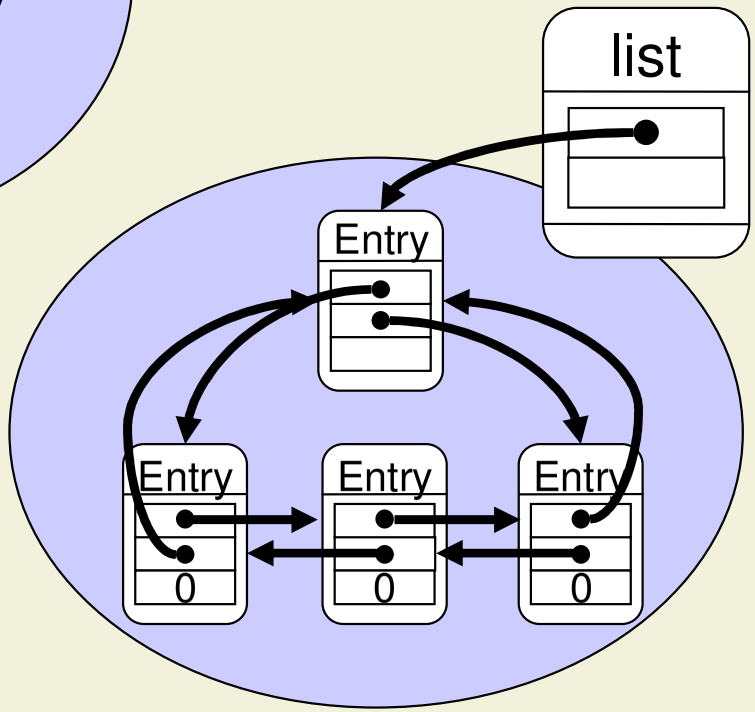
```
class ArrayList {
  private rep int[ ] array;
  private int next;

  // requires ia != null
  // ensures ∀i. 0<=i<ia.length:
  //                isElem( old( ia[ i ] ) )
  public void
  addElems( readonly int[ ] ia )
    { System.arraycopy(…);
      next = ia.length; }
  …
}
```

```
class ArrayList {
  private rep Entry header;


  // requires ia != null
  // ensures ∀i. 0<=i<ia.length:
  //                isElem( old( ia[ i ] ) )
  public void
  addElems( readonly int[ ] ia )
    { … /* create Entry for each
           element */ }
  …
}
```

Exchanging Implementations – UTS (c'd)

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





- Observable behavior **did not** change!
- In general, aliases can **still** be used to leak representation

Open Problems

- Ownership types are an area of current research activities
- Current topics
  - **Several owners** sharing a common representation, e.g., a list header and iterators; see MOJO
  - **Transfer** of objects from one context to another, e.g., for capturing; see UTT
  - **Application** of ownership types to all areas where aliasing leads to problems, e.g., thread synchronization