Concepts of Object-Oriented Programming

Peter Müller

Chair of Programming Methodology

Autumn Semester 2011

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

7. Ownership Types

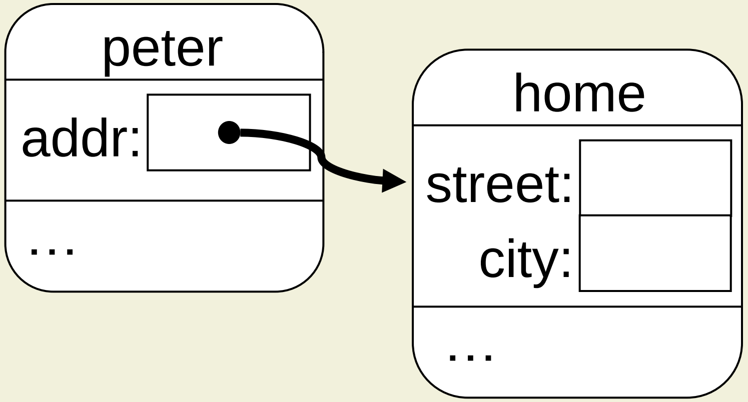
7.1 Readonly Types

7.2 Topological Types

Object Structures Revisited

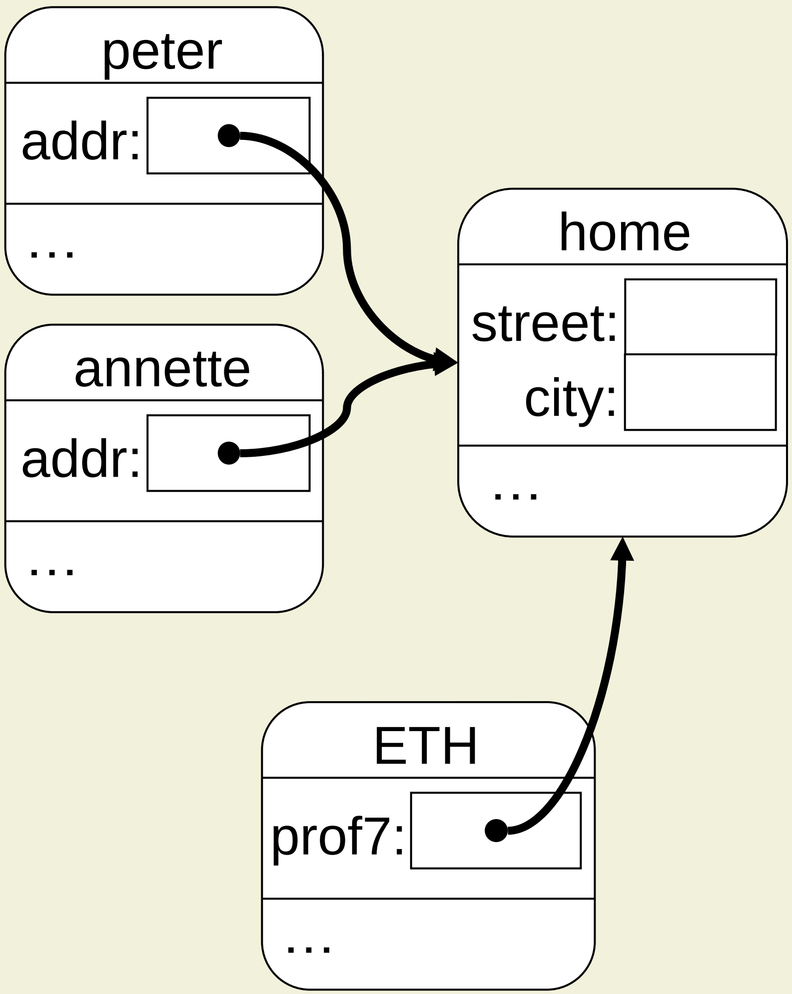
```
class Address {
  private String street;
  private String city;

  public String getStreet( ) { … }
  public void setStreet( String s )
    { … }

  public String getCity( ){ … }
  public void setCity( String s )
    { … }
  …
}
```

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



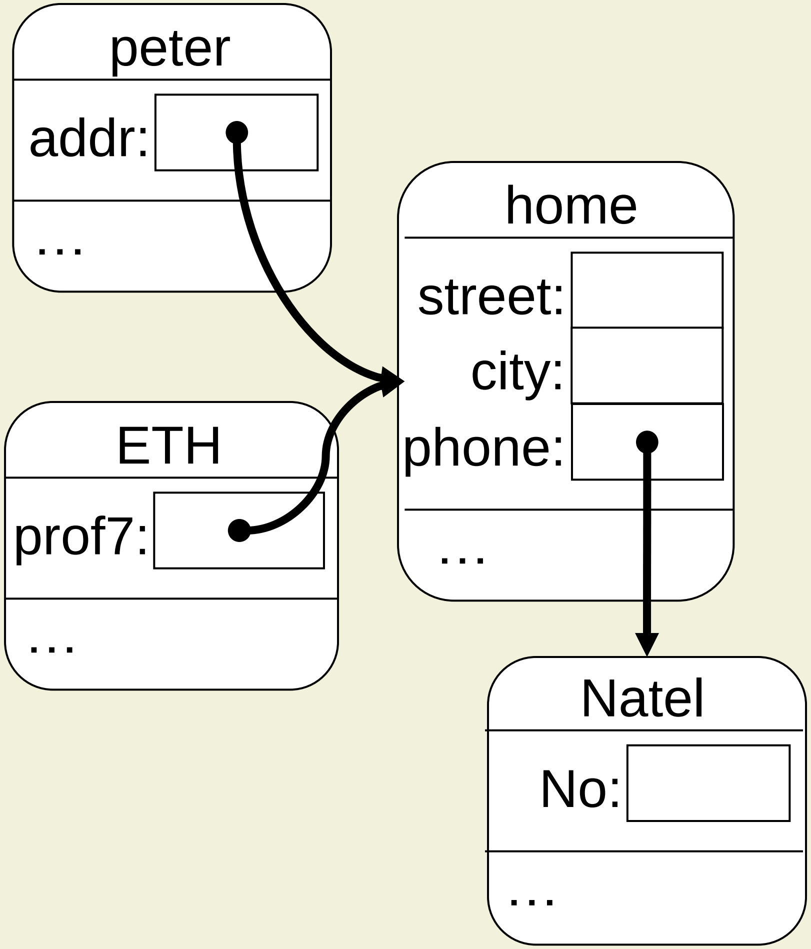
Drawbacks of Alias Prevention

- Aliases are helpful to share side-effects
- Cloning objects is not efficient
- In many cases, it suffices to restrict access to shared objects
- Common situation: grant read access only



Requirements for Readonly Access

- Mutable objects
  - Some clients can mutate the object, but others cannot
  - Access restrictions apply to references, not whole objects
- Prevent field updates
- Prevent calls of mutating methods
- Transitivity
  - Access restrictions extend to references to sub-objects



Readonly Access via Supertypes

```
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( ); }
  … }
```

- Clients use only the methods in the interface
  - Object remains mutable
  - No field updates
  - No mutating method in the interface

Limitations of Supertype Solution

- Reused classes might not implement a readonly interface
  - See discussion of structural subtyping
- Interfaces do not support arrays, fields, and non-public methods

```
class Address
        implements ReadonlyAddress {
  …
  private PhoneNo phone;
  public PhoneNo getPhone( )
   { return phone; } }
```

```
interface ReadonlyAddress {
  …
  public ReadonlyPhoneNo getPhone( );
}
```

- Transitivity has to be encoded explicitly
  - Requires sub-objects to implement readonly interface

Supertype Solution is not Safe

- No checks that methods in readonly interface are actually side-effect free
- Readwrite aliases can occur, e.g., through 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" );
}
```

Readonly Access in Eiffel

- Better support for fields
  - Readonly supertype can contain getters
  - Field updates only on “this” object
- Command-query separation
  - Distinction between mutating and inspector methods
  - But queries are not checked to be side-effect free
- Other problems as before
  - Reused classes, transitivity, arrays, aliasing, downcasts

Readonly Access in C++: const Pointers

```
class Address {
  string city;
public:
  string getCity( void )
    { return city; }
  void setCity( string s )
    { city = s; }
};
```
C++

```
class Person {
  Address* addr;
public:
  const Address* getAddr(  )
    { return addr; }
  void setAddr( Address a )
    { /* clone */ }
};
```
C++

- C++ supports readonly pointers
  - No field updates
  - No mutator calls

```
void m( Person* p ) {
  const Address* a =  p->getAddr( );
  a->setCity( "Hagen" );
  cout << a->getCity( );
}
```

Compile-time errors

Readonly Access in C++: const Functions

```
class Address {
  string city;
public:
  string getCity( void ) const
    { return city; }
  void setCity( string s )
    { city = s; }
};
```
C++

```
class Person {
  Address* addr;
public:
  const Address* getAddr( )
    { return addr; }
  void setAddr( Address a )
    { /* clone */ }
};
```
C++

- const Functions must not modify their receiver object

```
void m( Person* p ) {
  const Address* a = p->getAddr( );
  a->setCity( "Hagen" );
  cout << a->getCity( );
}
```

Call of const function allowed

Compile-time error

It wouldn't be C++ …

```
class Address {
  string city;
public:
  string getCity( void ) const
    { return city; }
  void setCity( string s ) const {
    Address* me = ( Address* ) this;
    me->city = s;
} };
```

C++

```
class Person {
  Address* addr;
public:
  const Address* getAddr( )
    { return addr; }
  void setAddr( Address a )
    { /* clone */ }
};
```

C++

- const-ness can be cast away
  - No run-time check

```
void m( Person* p ) {
  const Address* a = p->getAddr( );
  a->setCity( "Hagen" );
}
```

Call of const function allowed

It wouldn't be C++ … (cont'd)

```
class Address {
  string city;
public:
  string getCity( void ) const
    { return city; }
  void setCity( string s )
    { city = s; }
};
```
C++

```
class Person {
  Address* addr;
public:
  const Address* getAddr( )
    { return addr; }
  void setAddr( Address a )
    { /* clone */ }
};
```
C++

- const-ness can be cast away
  - No run-time check

```
void m( Person* p ) {
  const Address* a = p->getAddr( );
  Address* ma = ( Address* ) a;
  ma->setCity( "Hagen" );
}
```
C++

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Readonly Access in C++: Transitivity

```
class Phone {
public:
  int number;
};
```
C++

```
class Address {
  string city;
  Phone* phone;
public:
  Phone* getPhone( void ) const
    { return phone; }
…
};
```
C++

```
void m( Person* p ) {
  const Address* a =  p->getAddr( );
  Phone* p = a->getPhone( );
  p->number = 2331…;
}
```
C++

- const pointers are not transitive
- const-ness of sub-objects has to be indicated explicitly

Transitivity (cont'd)

```
class Address {
  string city;
  Phone* phone;
public:
  const Phone* getPhone( void ) const {
    phone->number = 2331;
    return phone;
  }
  …
};
```

C++

const functions may modify objects other than the receiver

Readonly Access in C++: Discussion

Pros

- const pointers provide readonly pointers to mutable objects
  - Prevent field updates
  - Prevent calls of non-const functions
- Work for library classes
- Support for arrays, fields, and non-public methods

Cons

- const-ness is not transitive
- const pointers are unsafe
  - Explicit casts
- Readwrite aliases can occur

Pure Methods

- Tag side-effect free methods as **pure**
- Pure methods
  - Must not contain field update
  - 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
- Readwrite type *rw T*
  - Denoted by 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 readonly Address
        getAddr( ) { … }
  …
}
```


ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Subtype Relation

- Subtyping among readwrite and readonly types is defined as in Java
  - S extends or implements T $\Rightarrow$ *rw S <: rw T*
  - S extends or implements T $\Rightarrow$ *ro S <: ro T*
- Readwrite types are subtypes of corresponding readonly types
  - *rw T <: ro T*

```
class T { … }
```

```
class S extends T { … }
```

```
S rwS = …
T rwT  = …
readonly S roS = …
readonly T roT = …
```

```
rwT  = rwS;
roT  = roS;
roT  = rwT;
```

```
rwT  = roT;
```


Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Type Rules: Transitive Readonly

```
class Address {
  …
  private int[ ] phone;
  public int[ ] getPhone( )  { … }
}
```

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

- Accessing a value of a readonly type or through a readonly type should yield a readonly value

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

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

Type Rules: Transitive Readonly (cont'd)

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

  is determined by the type combinator ►

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

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

ro Address ► *rw int[ ]*

ro int[ ]

►	*rw T*	*ro T*
rw S	*rw T*	*ro T*
ro S	*ro T*	*ro T*

Type Rules: Transitive Readonly (cont'd)

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

  is determined by the type combinator ▶

▶	*rw T*	*ro T*
rw S	*rw T*	*ro T*
ro S	*ro T*	*ro T*

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

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

ro Address ▶ *rw int[ ]*

ro int[ ]

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Type Rules: Readonly Access

- Expressions of readonly types must not occur as receiver of
  - a field update
  - an array update
  - an invocation of a non-pure method

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

- Readonly types must not be cast to readwrite types

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

Discussion

- Readonly types enable safe sharing of objects
- Very similar to const pointers in C++, but:
  - Transitive
  - No casts to readwrite types
- All rules for pure methods and readonly types can be checked statically by a compiler
- Readwrite aliases can still occur, e.g., by capturing

7. Ownership Types

7.1 Readonly Types

7.2 Topological Types

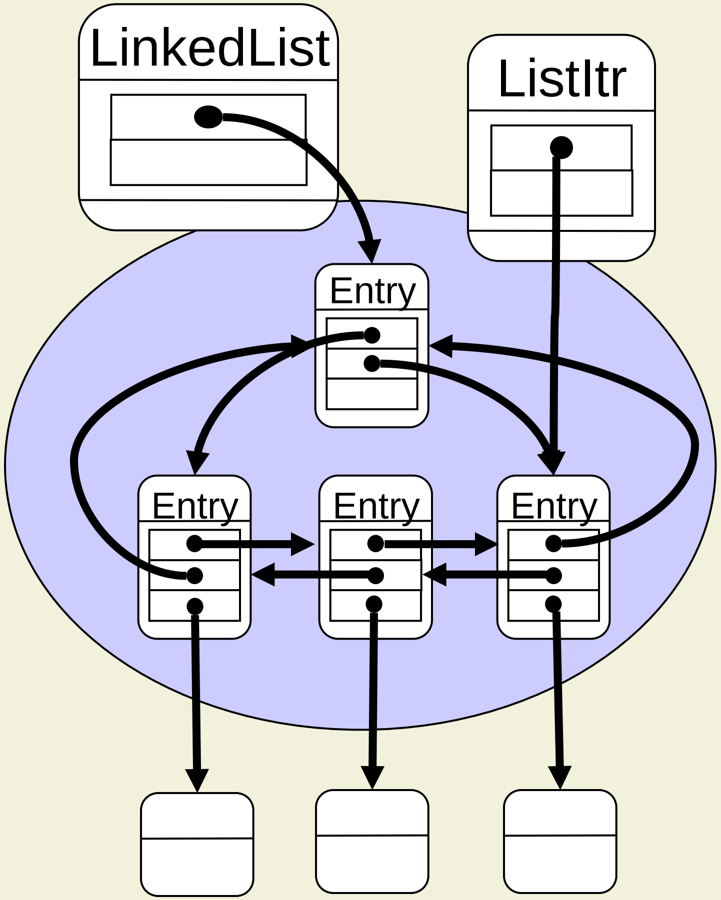
Object Topologies

- Read-write aliases can still occur, e.g., by capturing or leaking
- We need to distinguish “internal” references from other references

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

  public Company getEmployer( )
    { return employer; }
  public void setEmployer( Company c )
    { employer = c; }
  …
}
```

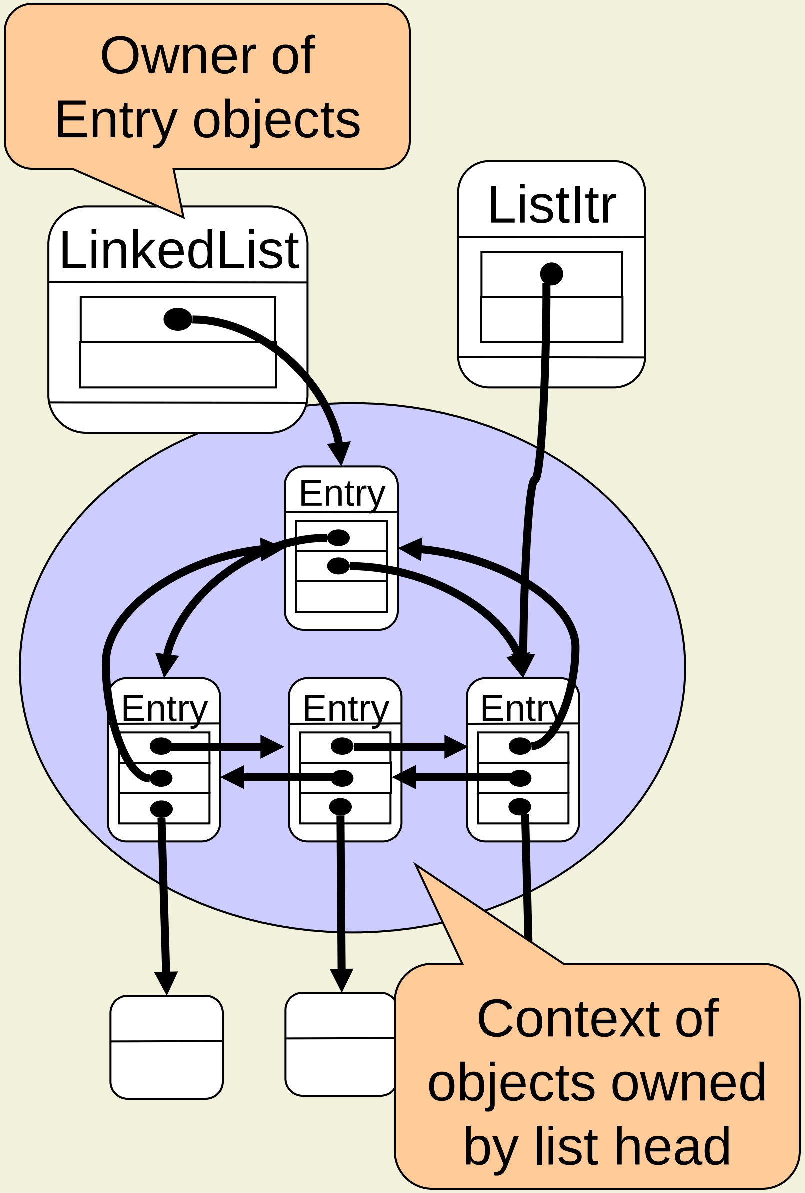
Roles in Object Structures

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



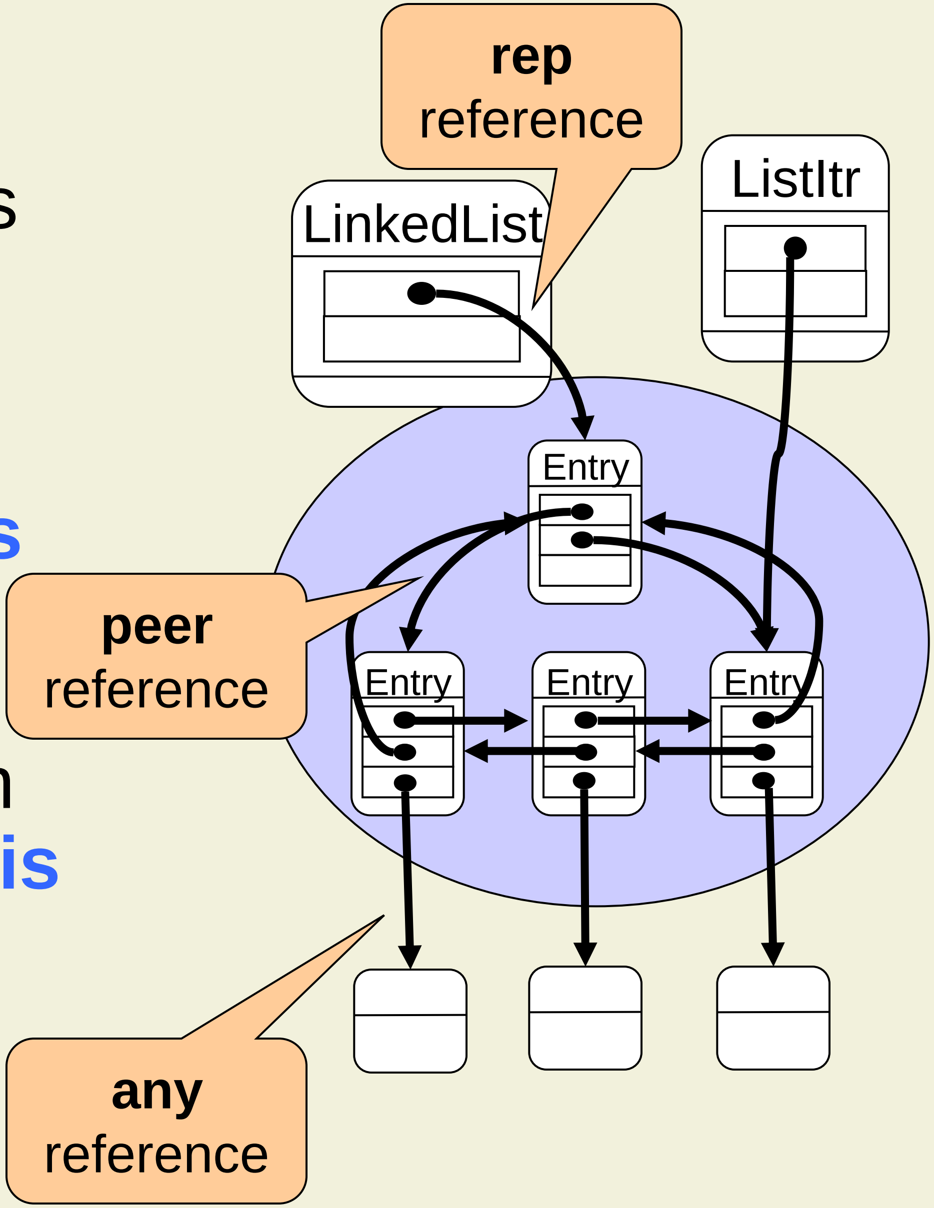
Ownership Model

- Each object has zero or one owner objects
- The set of objects with the same owner is called a context
- The ownership relation is acyclic
- The heap is structured into a forest of ownership trees



OwnershipTypes

- We use types to express ownership information
- **peer** types for objects in the same context as **this**
- **rep** types for representation objects in the context owned by **this**
- **any** types for argument objects in any context



Example

```
class LinkedList {
  private rep Entry header;
  …
}
```

A list owns its nodes

Lists store elements with arbitrary owners

```
class Entry {
  private any Object element;
  private peer Entry previous, next;
  …
}
```

All nodes have the same owner

Type Safety

- Run-time type information consists of
  - The class of each object
  - The owner of each object

- Type invariant: the static ownership information of an expression e reflects the run-time owner of the object o referenced by e's value
  - If e has type *rep T* then o's owner is **this**
  - If e has type *peer T* then o's owner is the owner of **this**
  - If e has type *any T* then o's owner is arbitrary

An existential type

Subtyping and Casts

- For types with identical ownership modifier, subtyping is defined as in Java
  - *rep S <: rep T*
  - *peer S <: peer T*
  - *any S <: any T*
- **rep** types and **peer** types are subtypes of corresponding **any** types
  - *rep T <: any T*
  - *peer T <: any T*

```
class T { … }
```

```
class S extends T { … }
```

```
peer T peerT  = …
any T anyT = …
rep S repS = …
rep T repT = …
```

```
repT    = repS;
anyT    = repT;
peerT   = ( peer T ) anyT;
repT    = ( rep T ) anyT;
```

Run-time checks

```
repT    = peerT;
peerT   = repT;
repT    = anyT;
```

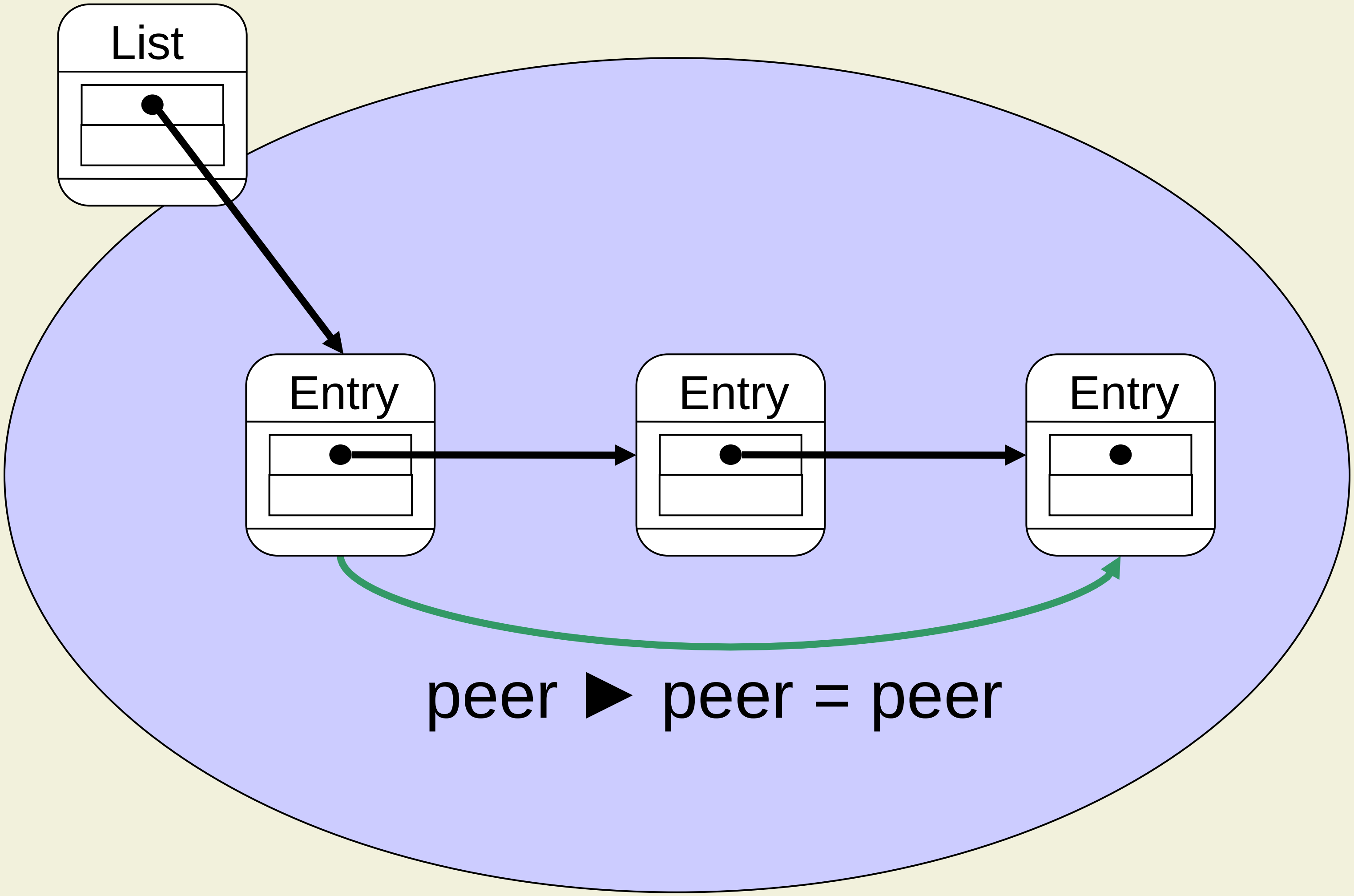
Example (cont'd)

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

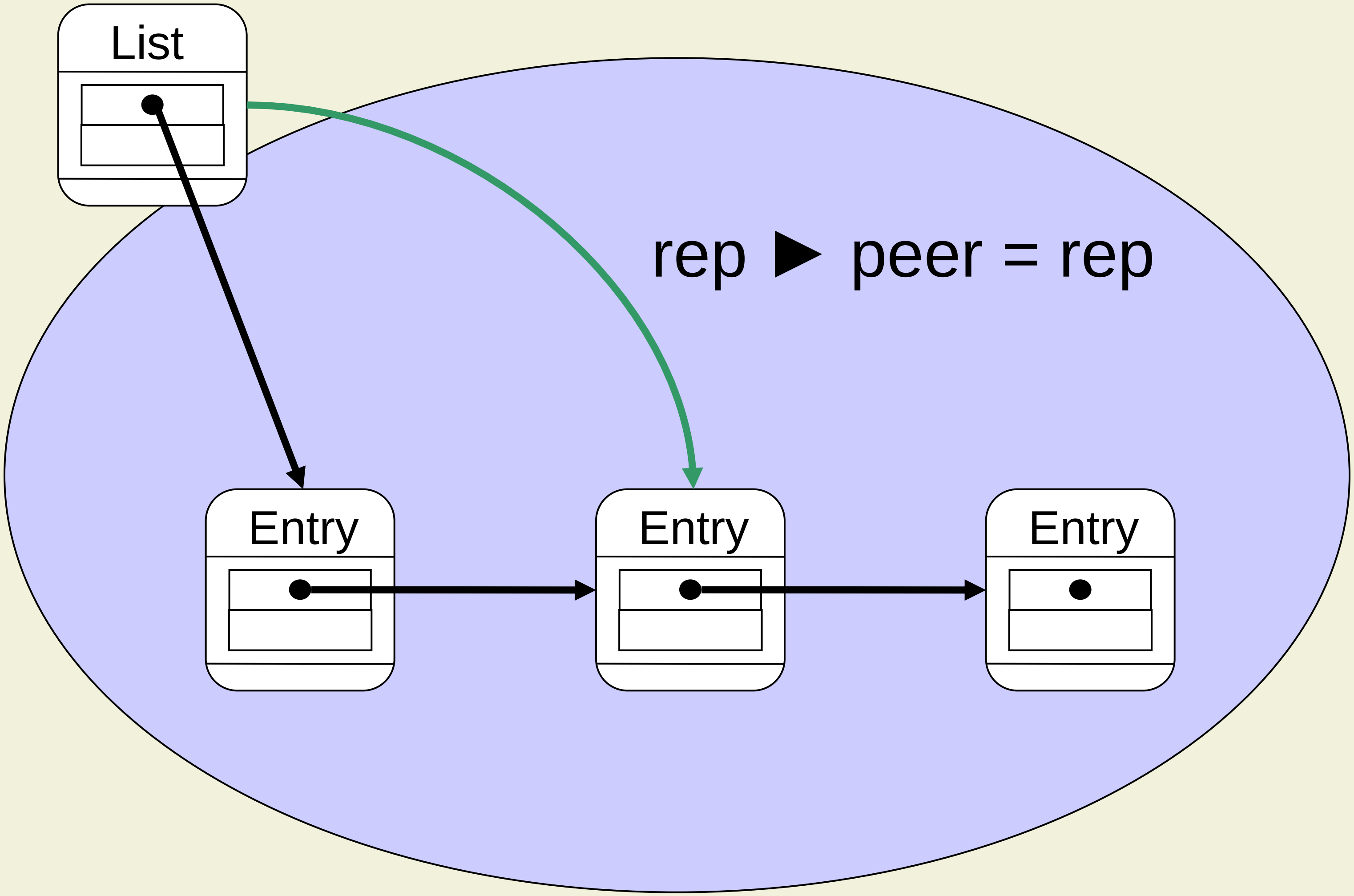
```
class Entry {
  private any Object element;
  private peer Entry previous, next;
  public Entry( any Object o, peer Entry p, peer Entry n ) { … }
}
```

Ownership information is relative to **this** reference (viewpoint)

Viewpoint Adaptation: Example 1



Viewpoint Adaptation: Example 2



Viewpoint Adaptation

▶	*peer T*	*rep T*	*any T*
peer S	*peer T*	*?*	*any T*
rep S	*rep T*	*?*	*any T*
any S	*?*	*?*	*any T*

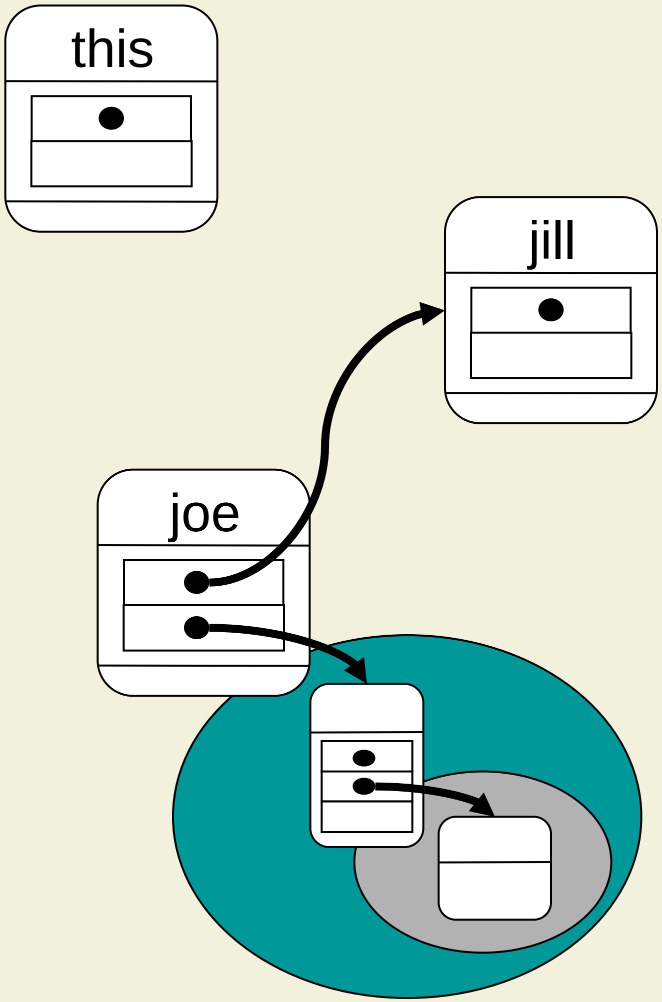
$\boxed{v = e.f;}$

$\tau(e) \blacktriangleright \tau(f) <: \tau(v)$

$\boxed{e.f = v;}$

$\tau(v) <: \tau(e) \blacktriangleright \tau(f)$

Read vs. Write Access




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

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

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

```
any Address a = joe.addr;
```

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

The lost Modifier

- Some ownership relations cannot be expressed in the type system
- Internal modifier **lost** for fixed, but unknown owner
- Reading locations with lost ownership is allowed
- Updating locations with lost ownership is unsafe

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

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

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

lost Address

```
any Address a = joe.addr;
```

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

lost Address

The lost Modifier: Details

▶	*peer T*	*rep T*	*any T*
peer S	*peer T*	*lost T*	*any T*
rep S	*rep T*	*lost T*	*any T*
any S	*lost T*	*lost T*	*any T*
lost S	*lost T*	*lost T*	*any T*

Another existential type

- Subtyping
  - *rep T <: lost T*
  - *peer T <: lost T*
  - *lost T <: any T*

Type Rules: Field Access

- The field read

  $\boxed{v = e.f;}$

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

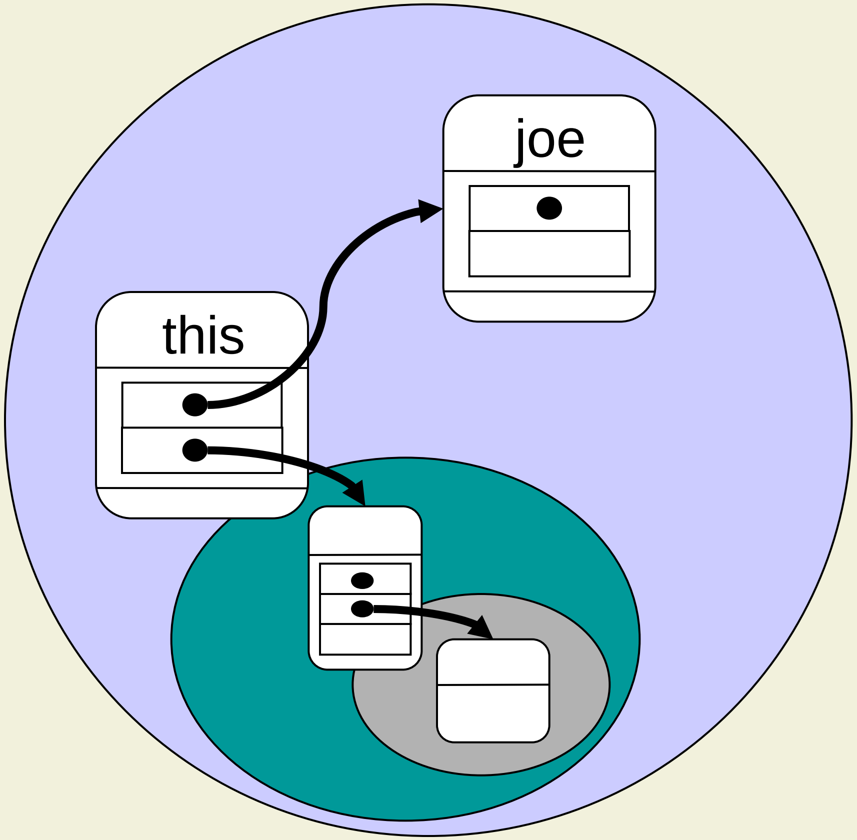
- The field write

  $\boxed{e.f = v;}$

  is correctly typed if
  - e is correctly typed
  - $\tau(v) <: \tau(e) \blacktriangleright \tau(f)$
  - $\tau(e) \blacktriangleright \tau(f)$ does not have **lost** modifier

- Analogous rules for method invocations
  - Argument passing is analogous to field write
  - Result passing is analogous to field read

The self Modifier




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

peer Person joe;

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

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

- Internal modifier **self** only for the **this** literal

The self Modifier: Details

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

- Subtyping
  - *self T <: peer T*

v = e.f;

τ(e) ▶ τ(f) <: τ(v)

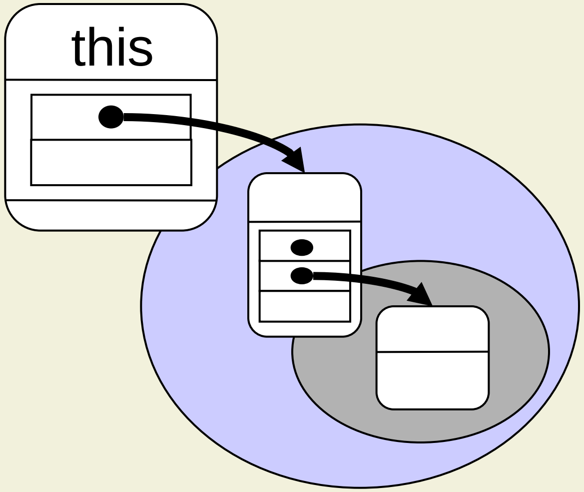
e.f = v;

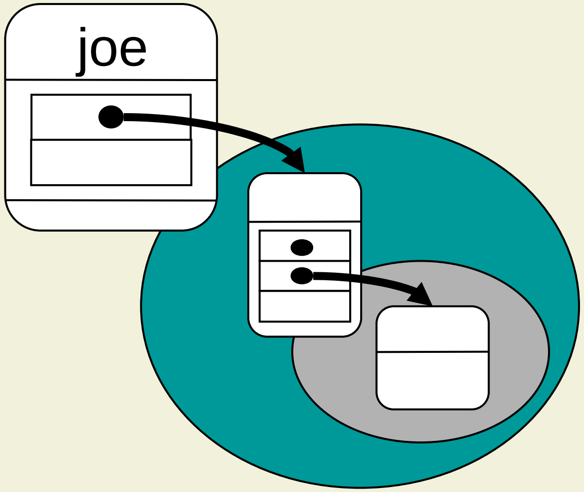
τ(v) <: τ(e) ▶ τ(f)

τ(e) ▶ τ(f) does not have **lost** modifier

Example: Sharing

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





- Different Person objects have different Address objects
  - No unwanted sharing

Example: Internal vs. External Objects

```
class Person {
  private rep Address addr;

  public rep Address getAddr( ) {
    return addr;
  }

  public void setAddr( rep Address a ) {
    addr = a;
  }

  public void setAddr( any Address a ) {
    addr = new rep Address( a );
  }
}
```

Address is part of Person’s internal represenations

Clients receive a lost-reference

Cannot be called by clients

Cloning necessary

Internal vs. External Objects (cont'd)

```
class Person {
  private any Company employer;

  public any Company getEmployer( ) {
    return employer;
  }

  public void setEmployer( any Company c ) {
    employer = c;
  }
}
```

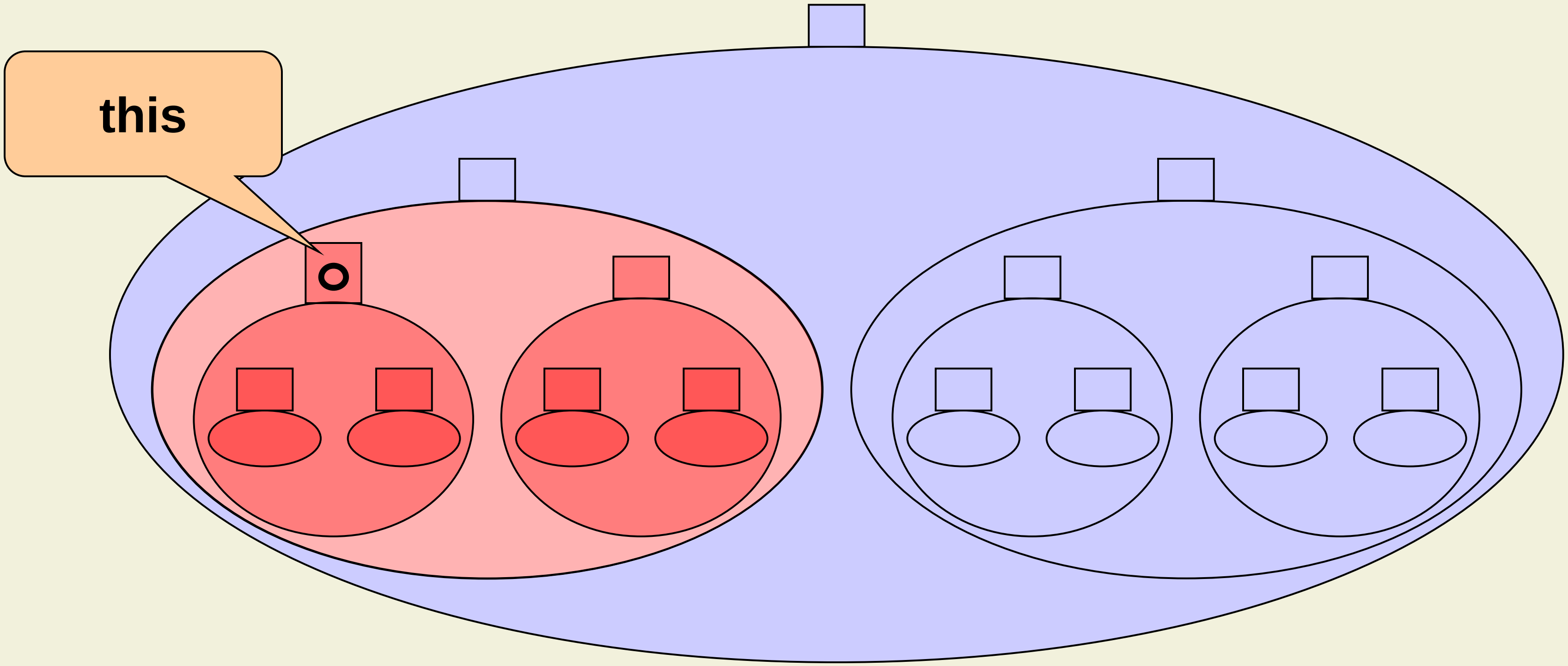
Company is shared between many Person objects

Can be called by clients

Owner-as-Modifier Discipline

- Topological type system can be used to strengthen encapsulation
  - Prevent modifications of internal objects
  - Treat **any** and **lost** as readonly types
  - Treat **self**, **peer**, and **rep** as readwrite types
- Additional rules enforce owner-as-modifier
  - Field write e.f = v is valid only if τ(e) is **self**, **peer**, or **rep**
  - Method call e.m(…) is valid only if τ(e) is **self**, **peer**, or **rep**, or called method is **pure**

Owner-as-Modifier Discipline (cont'd)



- A method may modify only objects directly or indirectly owned by the owner of the current **this** object

Internal vs. External Objects Revisited

```
class Person {
  private rep Address addr;
  private any Company employer;

  public rep Address getAddr( )  { return addr;  }

  public void setAddr( any Address a ) {
    addr = new rep Address( a );
  }

  public any Company getEmployer( )  { return employer; }

  public void setEmployer( any Company c ) { employer = c; }
}
```

Company is shared; cannot be modified

Clients receive (transitive) readonly reference

Accidental capturing is prevented

(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

Different types for different roles

Viewpoint adaptation for **rep** types

Like with programming discipline

Achievements

- **rep** and **any** 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 peer 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)

```
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( any int[ ] ia )
   { System.arraycopy(…);
     next = ia.length; }
  …
}
```

Accidental capturing is prevented

```
class ArrayList {
  private rep Entry header;


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

Exchanging Implementations (cont'd)

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

  public any int[ ] getElems( )
   { return ia; }
  …
}
```

Leaking is still possible

```
class ArrayList {
  private rep Entry header;


  public void any int[ ] getElems( )
   { /* create new array */ }
  …
}
```

```
peer  ArrayList list  = new peer ArrayList( );
list.prepend( 0 );
any int[ ] ia = list.getElems( );
list.prepend( 1 );
assert ia[ 0 ] == 1;
```

- 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   &&
  //    ∀i.0<=i<next: array[ i ] >= 0

  public void add( int i )   { … }
  public void addElems( int[ ] ia )
    { … }

  …
}
```

Invariants for Object Structures

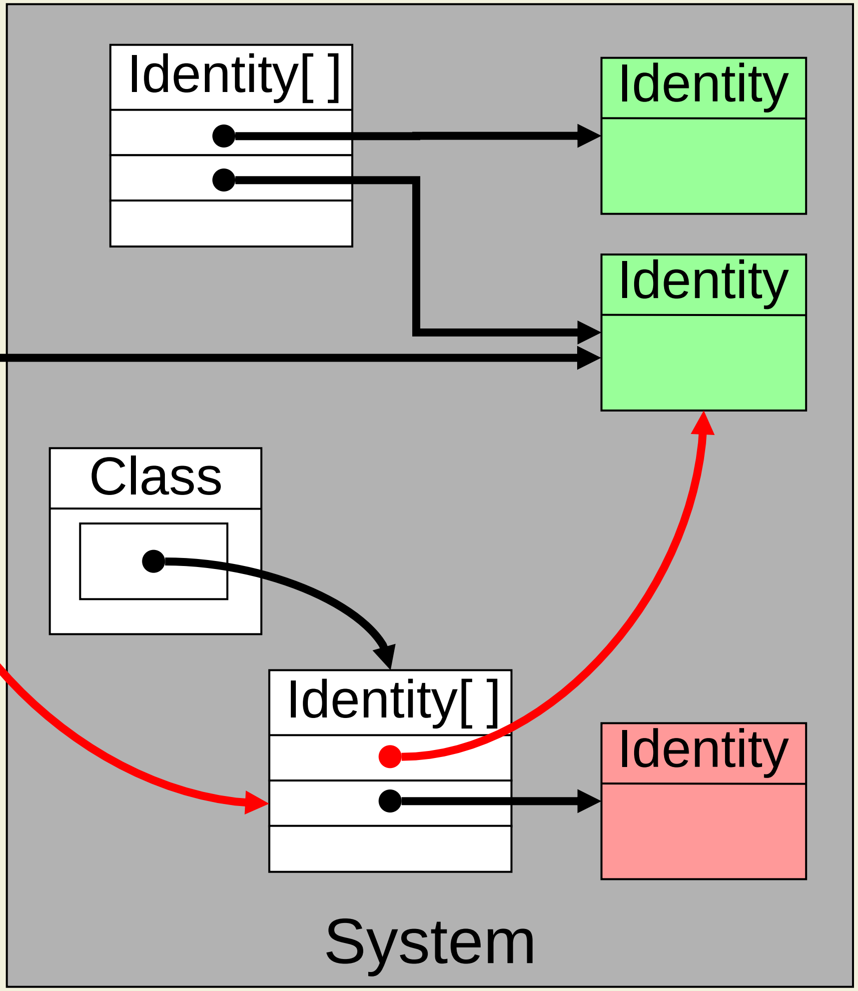
- The invariant of object o may depend on
  - Encapsulated fields of o
  - Fields of objects o references through rep-references
- Interface objects have full control over their rep-objects

```
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
       ( any int[ ] ia )    { … }

  …
}
```

Security Breach in Java 1.1.1

```
class Malicious  {

 void bad( ) {
  Identity[ ] s;
  Identity trusted = java.Security…;
  s = Malicious.class.getSigners( );
  s[ 0 ] = trusted;
  /* abuse privilege */
 }

}
```

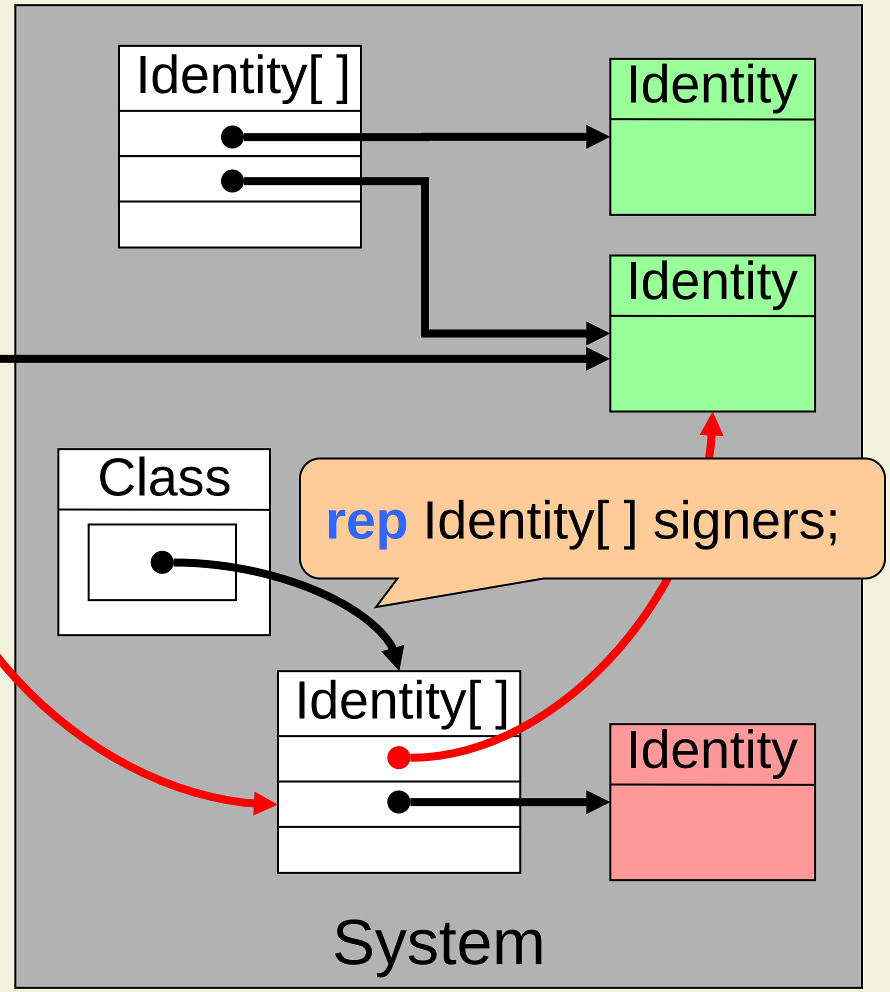
```
Identity[ ] getSigners( )
  { return signers; }
```



Security Breach in Java 1.1.1 (cont'd)

```
class Malicious {

 void bad( ) {
  any Identity[ ] s;
  Identity trusted = java.Security…;
  s = Malicious.class.getSigners( );
  s[ 0 ] = trusted;
 }

}
```

```
rep Identity[ ] getSigners( )
  { return signers; }
```



Ownership Types: Discussion

- Ownership types express heap topologies and enforce encapsulation
- Owner-as-modifier is helpful to control side effects
  - Maintain object invariants
  - Prevent unwanted modifications
- Other applications also need restrictions of read access
  - Exchange of implementations
  - Thread synchronization
- Ownership types are an area of current research