Konzepte objektorientierter Programmierung
– Lecture 7 –

Prof. Dr. Peter Müller
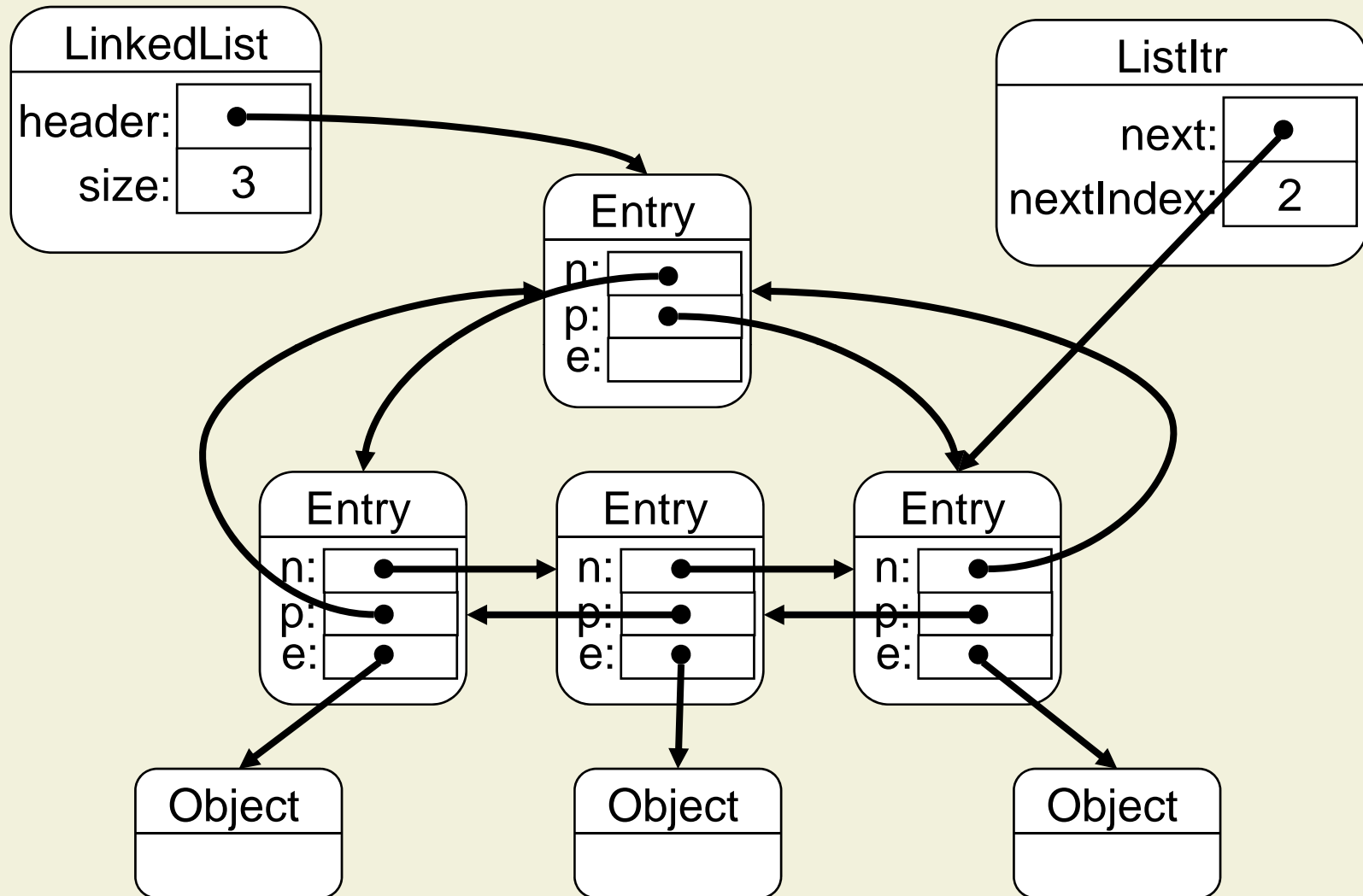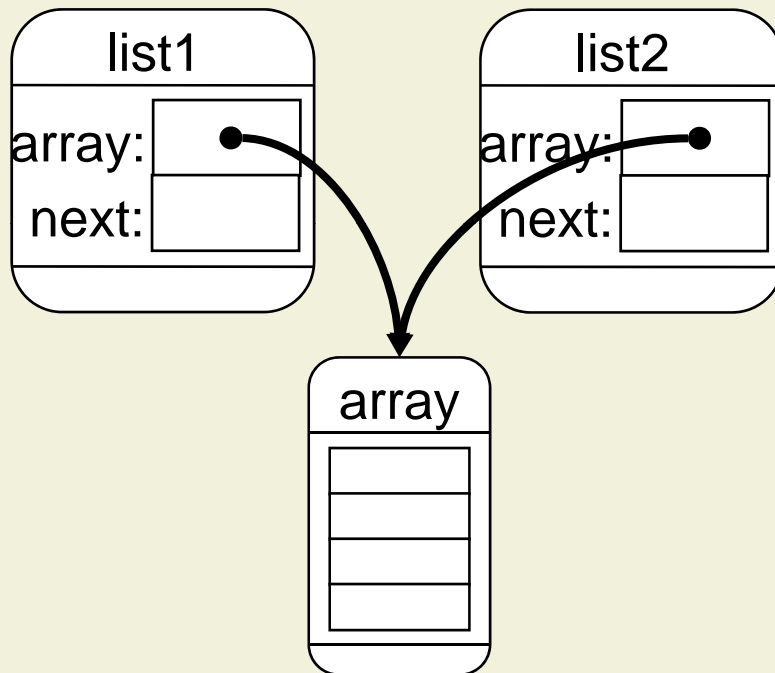
Chair of Programming Methodology

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Example 2: Doubly-Linked Lists
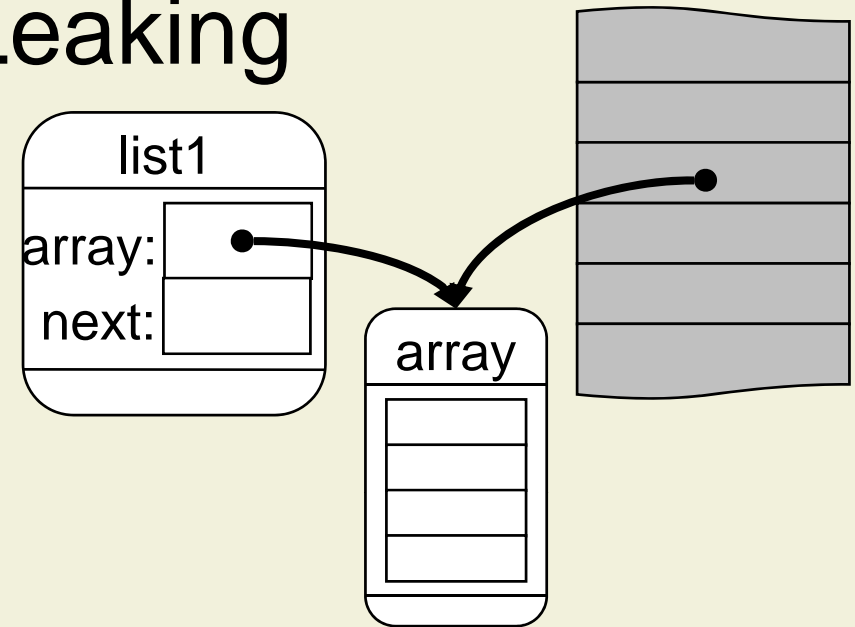
Aliasing

list1
array:
next:

list2
array:
next:

array

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

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

list1

array:

next:

array

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

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

- Type safety
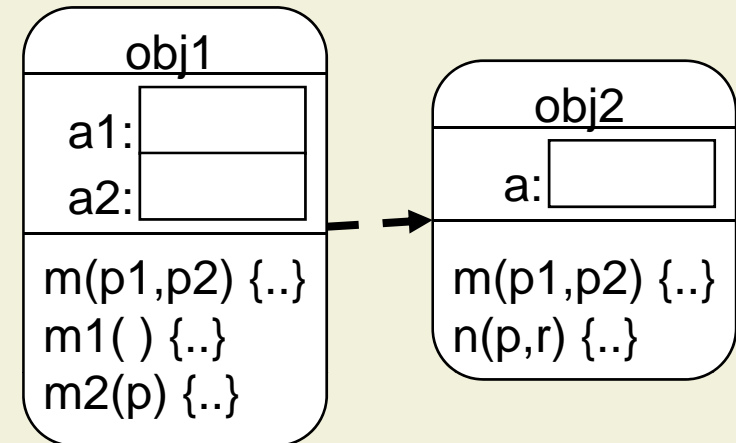- Static checking by type systems

7. Static Safety and Extended Typing

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Static Safety

- Objects access attributes and methods of other objects

- If the receiver object does not have the requested attribute or method, a **runtime error** occurs
- **Type systems** can be used to **detect such errors statically**

obj1

a1:
a2:

m(p1,p2) {..}
m1( ) {..}
m2(p) {..}

obj2

a:

m(p1,p2) {..}
n(p,r) {..}

```
…
r = obj2.m( 0, 1 );
s = obj2.a;
```

```
r = obj2.m( );
r = obj2.anotherMethod( 0, 1 );
s = obj2.anotherField;
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Type Systems

- Definition:

  A type system is a tractable syntactic method for proving absence of certain program behaviors by classifying phrases according to the kinds of values they compute.
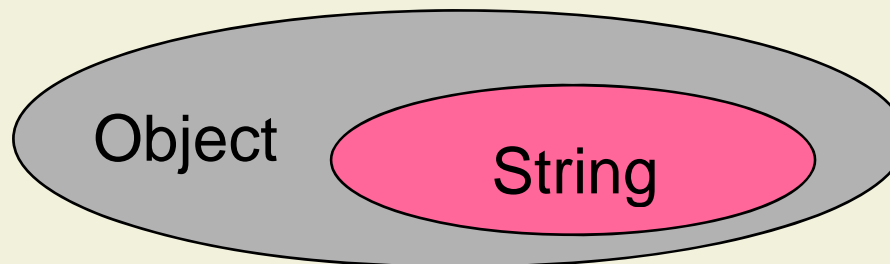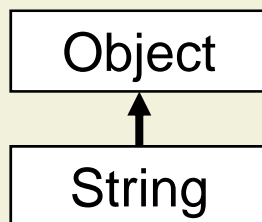
  [B.C. Pierce, 2002]

- *Syntactic*: Rules can be checked by a compiler
- *Phrases*: Expressions, methods, etc. of a program
- *Kinds of values*: Types

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

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Type Checking

- **Each expression** of a program **has a type**

- **Types** of variables and methods are **declared explicitly**

- Types of expressions can be derived from the types of their constituents

- **Type rules** are used to check whether a program is **correctly typed**

"A String"

5+7

---

int a;

boolean equals(Object o)

---

a + 7

"A Number: " + 7

"A String".equals(null)

Type Rules: Assignment

- ■ The assignment

$$v = exp;$$

  is correctly typed if

  - - exp is correctly typed and
  - - The type of exp is a subtype of the declared type of v

```
Object v;
v = new String( );
return v.equals( "Hello" );
```

```
String v;
v = new Object( );
v = v.concat( "Hello" );
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Type Rules: Result Types

- If S is a subtype of T and S has a method m that overrides a method in T then the result type of m in S must be a subtype of the result type of m in T

class T {   Object m( ) { … } }

class T {   String m( ) { … } }

class S **extends** T {
   String m( ) { … }
}

class S **extends** T {
   Object m( ) { … }
}

Object v; T t = **new** S( ); v = t.m( ); **return** v.equals("Hello");

String v;
T t = **new** S( );
v = t.m( );
v = v.concat("Hello");

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Dynamic Type Checking

- Some OO-languages support **type conversions by casts**

- Casts **cannot be type-checked statically**

- **Runtime checks** throw an exception in case of a type error

- **instanceof** can be used to avoid runtime errors

```
Object[ ] oa = new Object[ 10 ];
String s = "A String";


oa[ 0 ] = s;


…
if ( oa[ 0 ] instanceof String )
  s = (String) oa[ 0 ];


s = s.concat( "Another String" );
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Discussion

- Advantages of static type checking

  - Robustness: **Elimination of type errors**

  - Readability: Types are **excellent documentation**

  - Efficiency: Type information allows **optimizations**

- Limitations of **static** type checking

  - Does only work if all clients of a class are **recompiled after changes** are made

  - Does not work if code can be obtained at runtime in **executable format** (dynamic class loading, mobile code)

  - Strong static typing is sometimes **not flexible** enough

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

7. Static Safety and Extended Typing
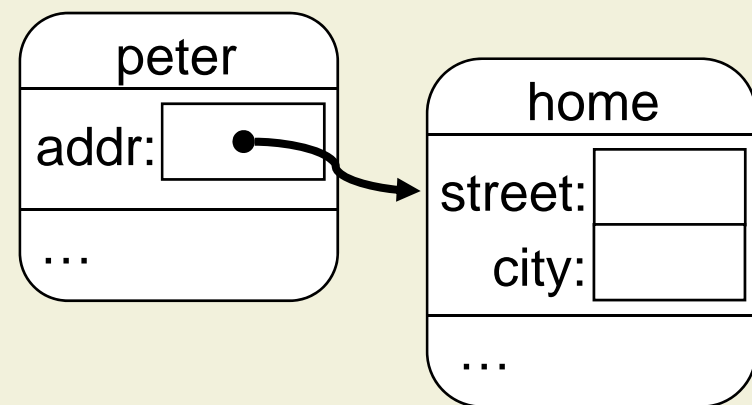
7.1 Type Systems

7.2 Readonly Types

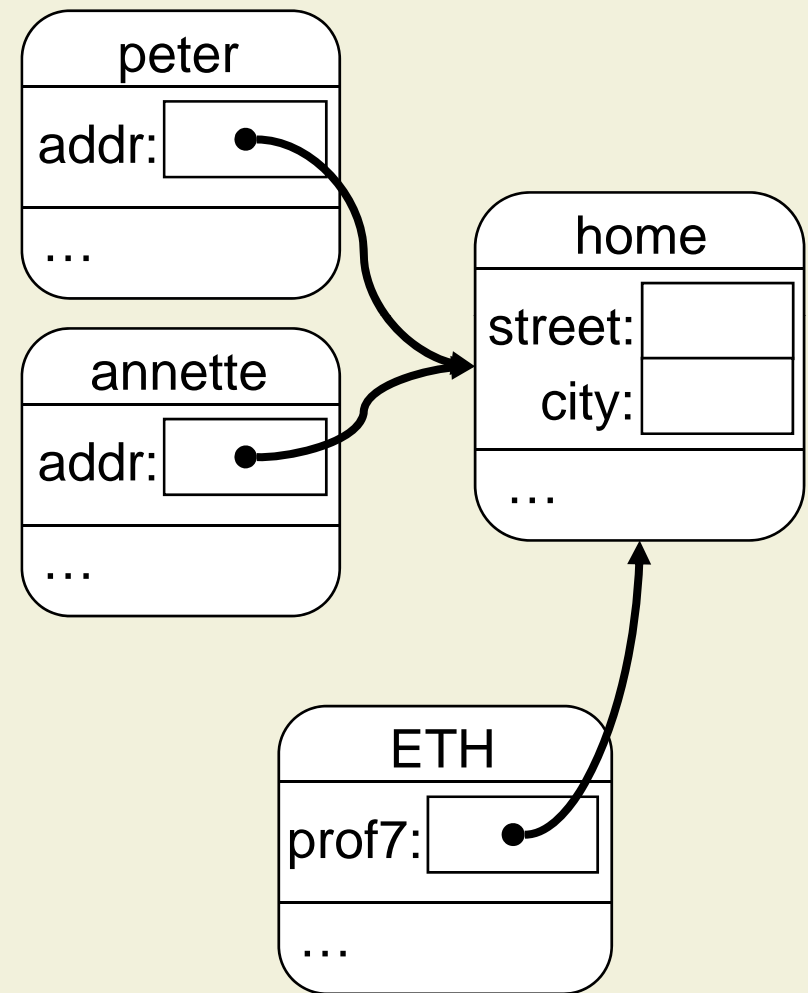## 7.3 Ownership Types

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 Control

- Aliases are helpful to **share side-effects**

- Cloning objects is not efficient

- In many cases, it would suffice to **restrict access** to shared objects

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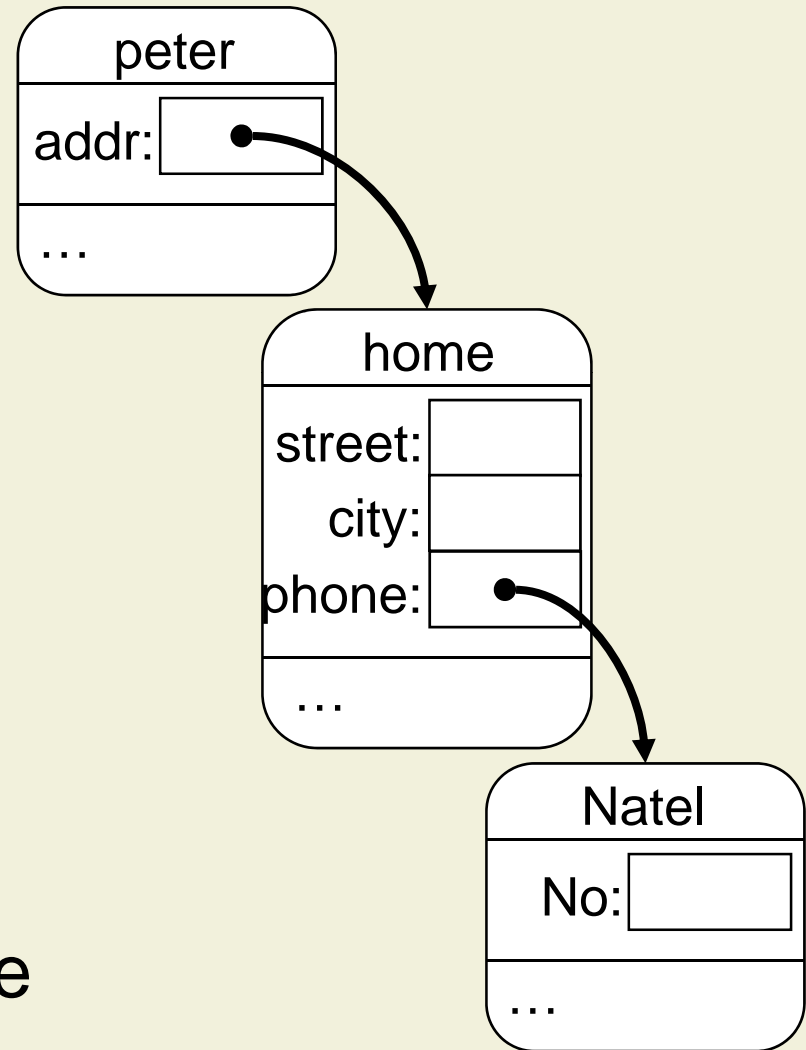
```
void m( Person p ) {
  ReadonlyAddress ra =
                    p.getAddr( );
  Address a = (Address) ra;
  a.setCity( "Hagen" );
}
```

Transitive Readonly Access

```
class Address {

  …

  private PhoneNo phone;

  public PhoneNo getPhone( )  { … }

}
```

```
interface ReadonlyAddress {

  …

  public ReadonlyPhoneNo getPhone( );

}
```

- Problem: objects structures
- **const** in C++ is not transitive

peter
addr:
…

home
street:
city:
phone:
…

Natel
No:
…

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zürich
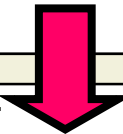
Pure Methods

- 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 )
    { … }
  …
}
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

```
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 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)*

---

class T { … }

class S **extends** T { … }

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

---

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

---

peerT = roT ;

ETH
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( );
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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( );


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

ro(Address) ***** *peer(int[ ])*

ro(int[ ])

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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( );
```

ro(Address) ***** peer(int[ ])

ro(int[ ])

Type Rules: Readonly Access

- **Expressions of readonly types must not occur**
  - As target of an **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;
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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
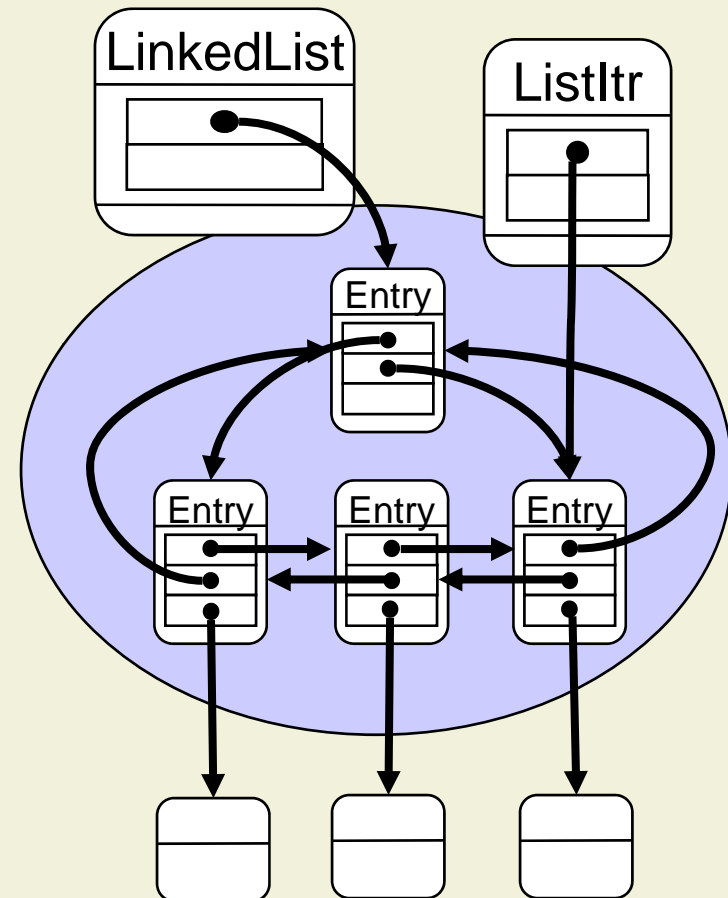- Readwrite aliases can still occur, e.g., by capturing

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

7. Static Safety and Extended Typing

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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
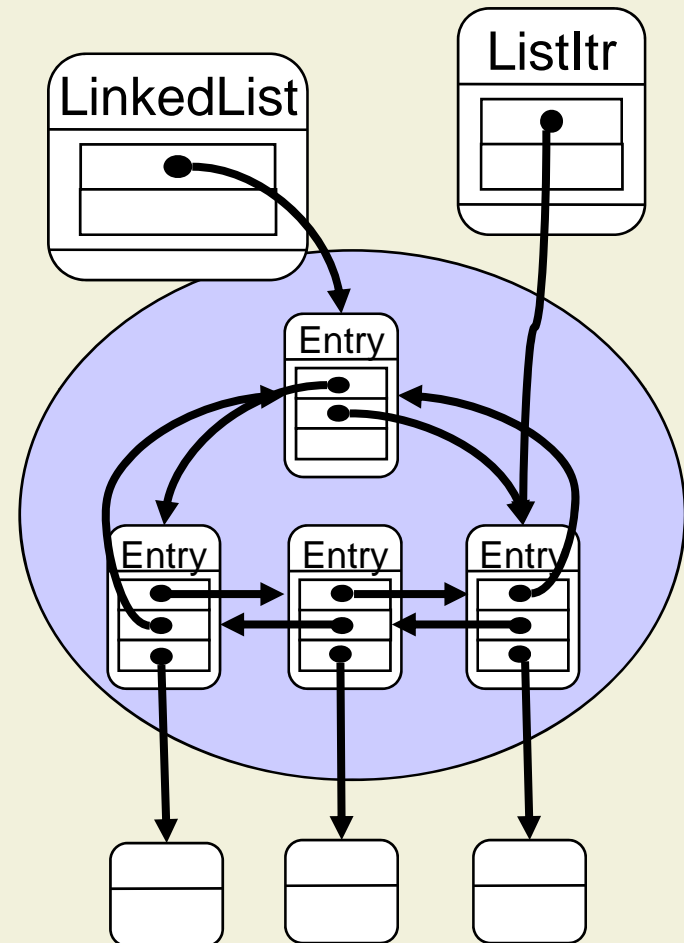
- **Rule 2: No Representation Exposure**

  - 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
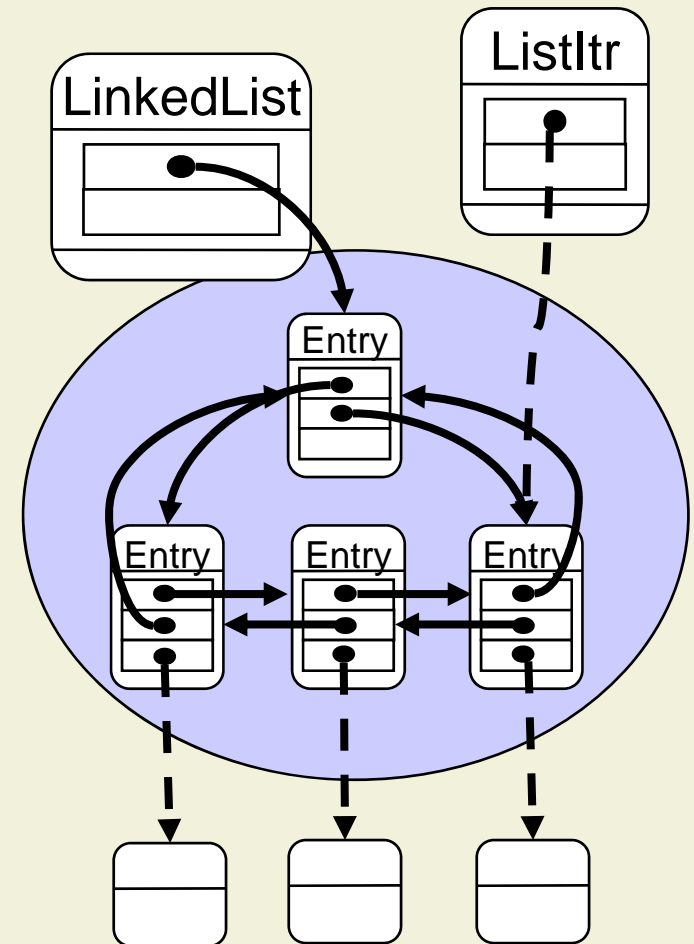
Ownership Model

- The **object store** is partitioned into **contexts**

- Each object **belongs to exactly one context**

- Each context has at most one **owner object**

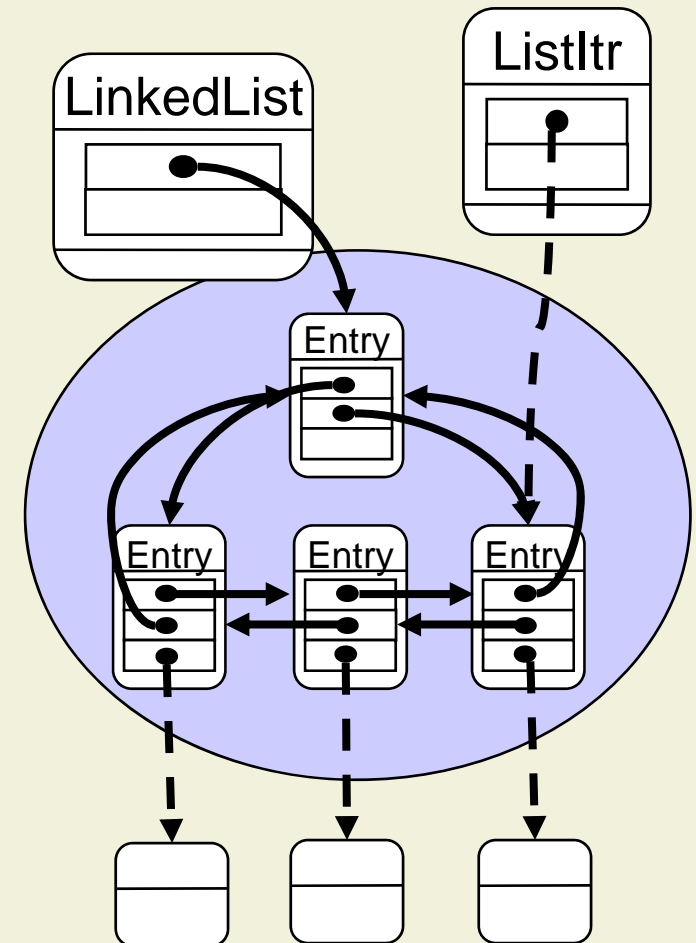 - The owner does not belong to the context it owns

- Contexts are **hierarchical**

Ownership Invariant

- **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 T 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 Entry previous, next;
  public Entry( readonly Object o,
                Entry p, Entry n ) { … }
}
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Subtype Relation

- **Subtyping** among rep types is **defined as in Java**

  - S extends or implements T $\Rightarrow$ *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 { … }
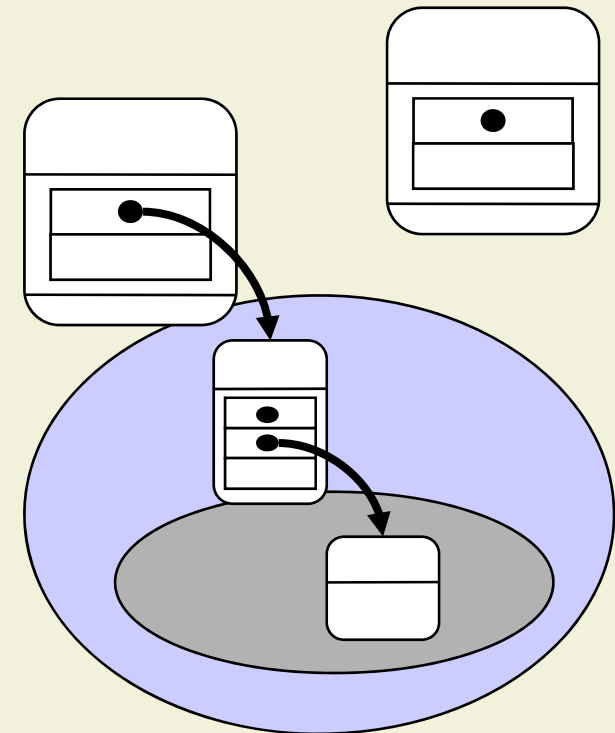
---

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;

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

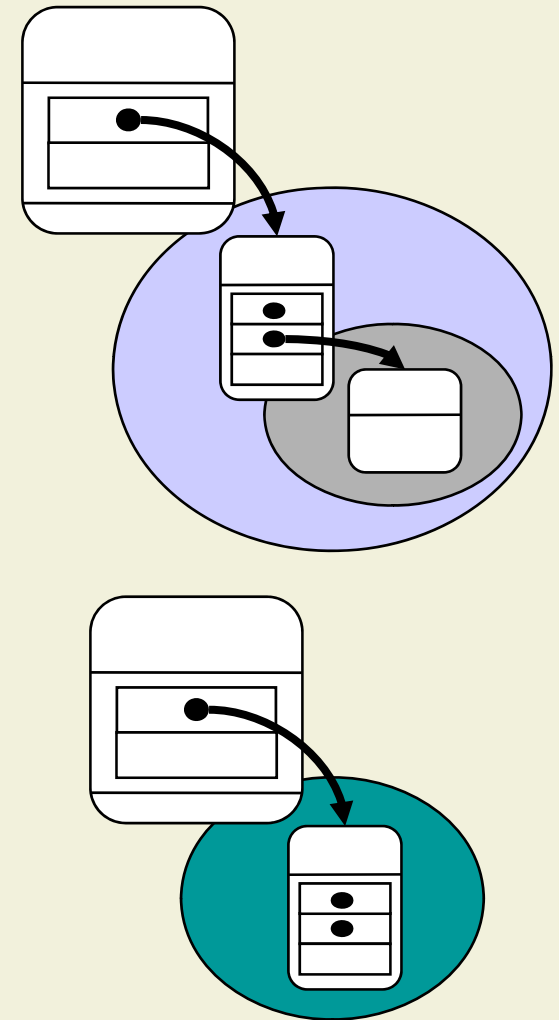# Type Rules: The Type Combinator

*	peer(T)	rep(T)	ro(T)
peer(S)	peer(T)	ro(T)	ro(T)
rep(S)	rep(T)	ro(T)	ro(T)
ro(S)	ro(T)	ro(T)	ro(T)

Types Rules: Access to Contexts

- **Objects in different contexts must not be confused**

- A rep type indicates that an object is owned by **this**

- Attributes of **rep types** and methods that have rep types as parameter or result types can **safely be accessed**

  - On **readonly targets**
  - On target **this**

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Type Rules: Attribute Access

- The attribute access

$$v = exp.f;$$

  is correctly typed if

  - exp is correctly typed
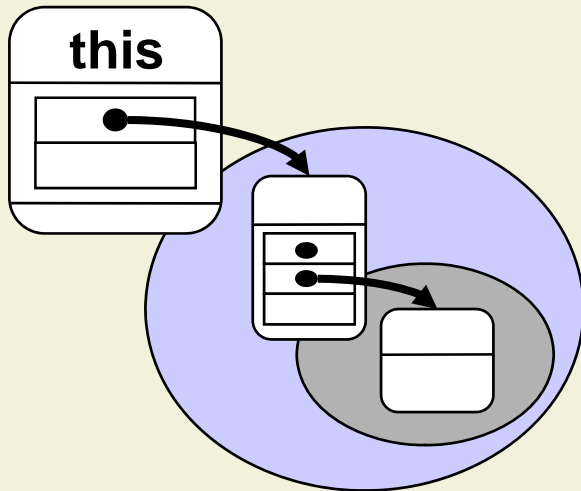 - $\tau(exp) * \tau(f) \leq \tau(v)$

- The attribute access

$$v = \mathbf{this}.f;$$

  is correctly typed if
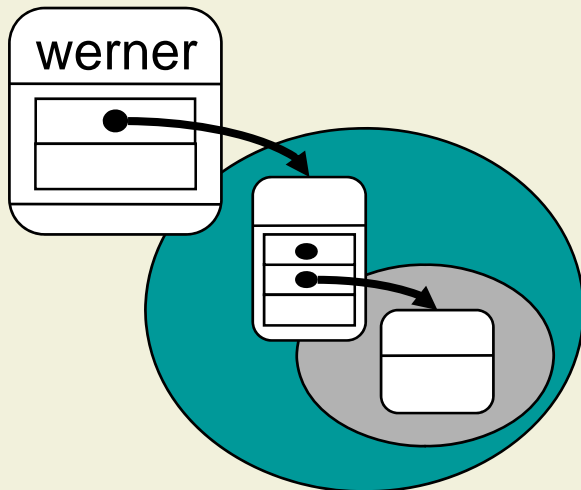
  - $\tau(f) \leq \tau(v)$

- Analogous rules are used for method invocations

Examples: Attribute Access


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

```
class Address {
  public rep int[ ] phone;
  … }
```
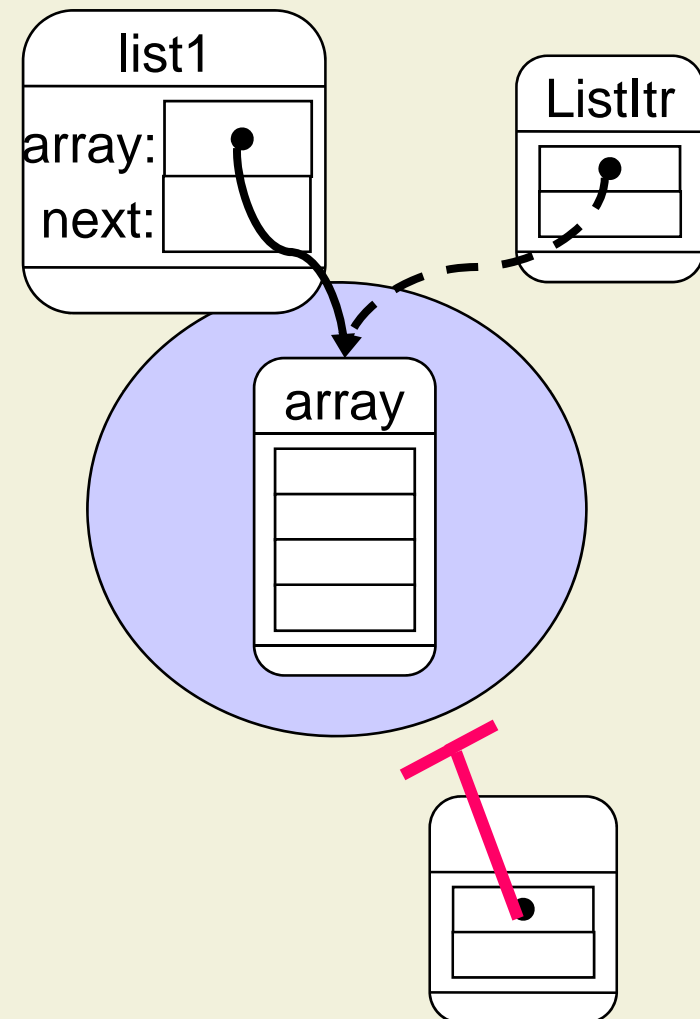
```
rep Address a = this.addr;
readonly Person row = werner;
readonly Address roa = row.addr;
```

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

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

No Representation Exposure

- Capturing or leaking **cannot lead to rep exposure**
  - Interface objects and representation objects have different types

- **Representations** of different interface objects **cannot be confused**
  - Rep fields and methods can only be accessed on **this** and readonly targets
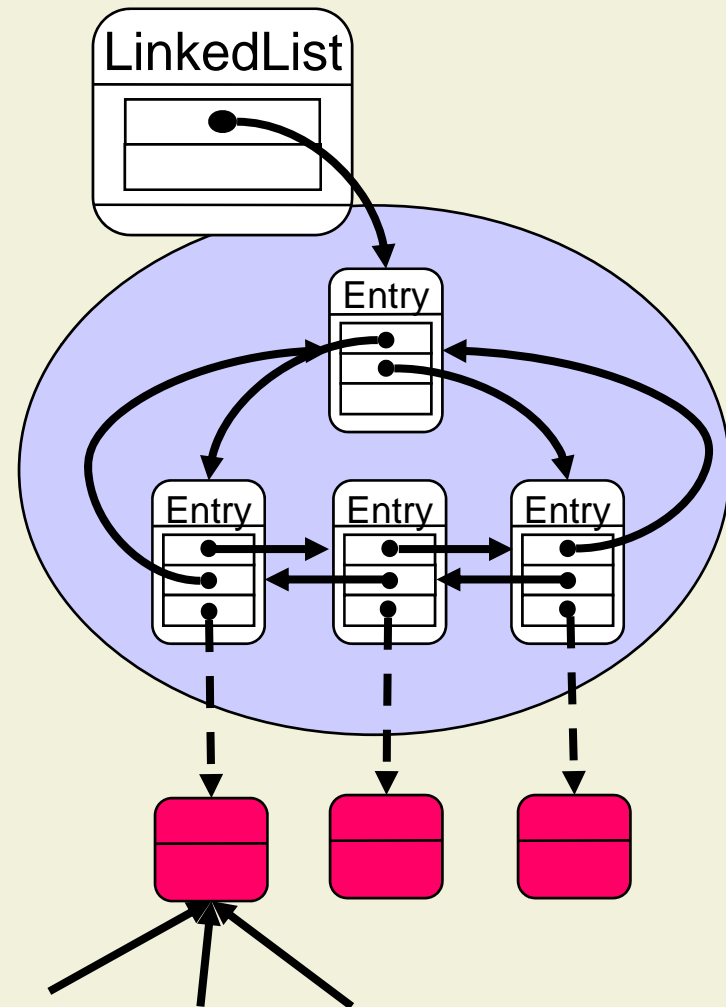
No Argument Dependence

- **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 T introduces three types

  - *peer(T), rep(T), ro(T)*

- ### At runtime, the dynamic type of an object consists of its class and its context

  - rep and readonly are only used to classify references

- ### Information about dynamic types can be used to cast readonly types with dynamic checks

```
readonly Address roa = …;

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

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zürich

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;
  }
}
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Open Problems

- Ownership types are an area of current research activities

- Open issues
  - **Several owners** sharing a common representation, e.g., a list header and iterators
  - **Transfer** of objects from one context to another, e.g., for capturing
  - **Application** of ownership types to all areas where aliasing leads to problems, e.g., thread synchronization

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich