

# 4. Object Structures, Aliasing, and Encapsulation

## Collaboration remark:

The following slides are partly taken from the lecture „Konzepte objektorientierter Programmierung“ by Prof. Peter Müller (ETH Zürich).

## Overview:

- Object Structures and Aliasing
- Immutability
- Alias Control and Encapsulation

## Motivation:

- Understand the possible collaborations of objects
- Object systems need structure and control

„Unless objects are conceptually allowed to **contain other objects** in their entirety, there is little hope to master complexity in a pure object-oriented approach. ... It is, therefore, and quite paradoxically, nontrivial to introduce the notion of **components into object systems**.“

[ David Luckham et al. ]

## 4.1 Object Structures and Aliasing

- Objects are the basic building blocks of object-oriented programming.
- However, interesting **system components** and **program abstractions** are almost always provided by sets of cooperating objects.

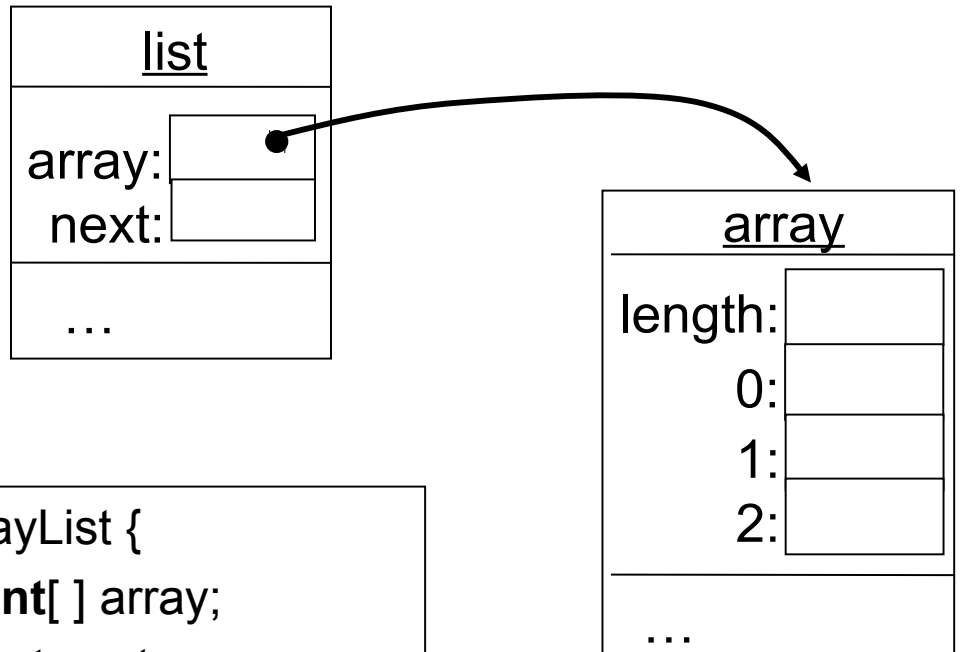
Explanation: (Object structure)

An **object structure** (*Objektgeflecht*) is a set of objects that are connected via references.



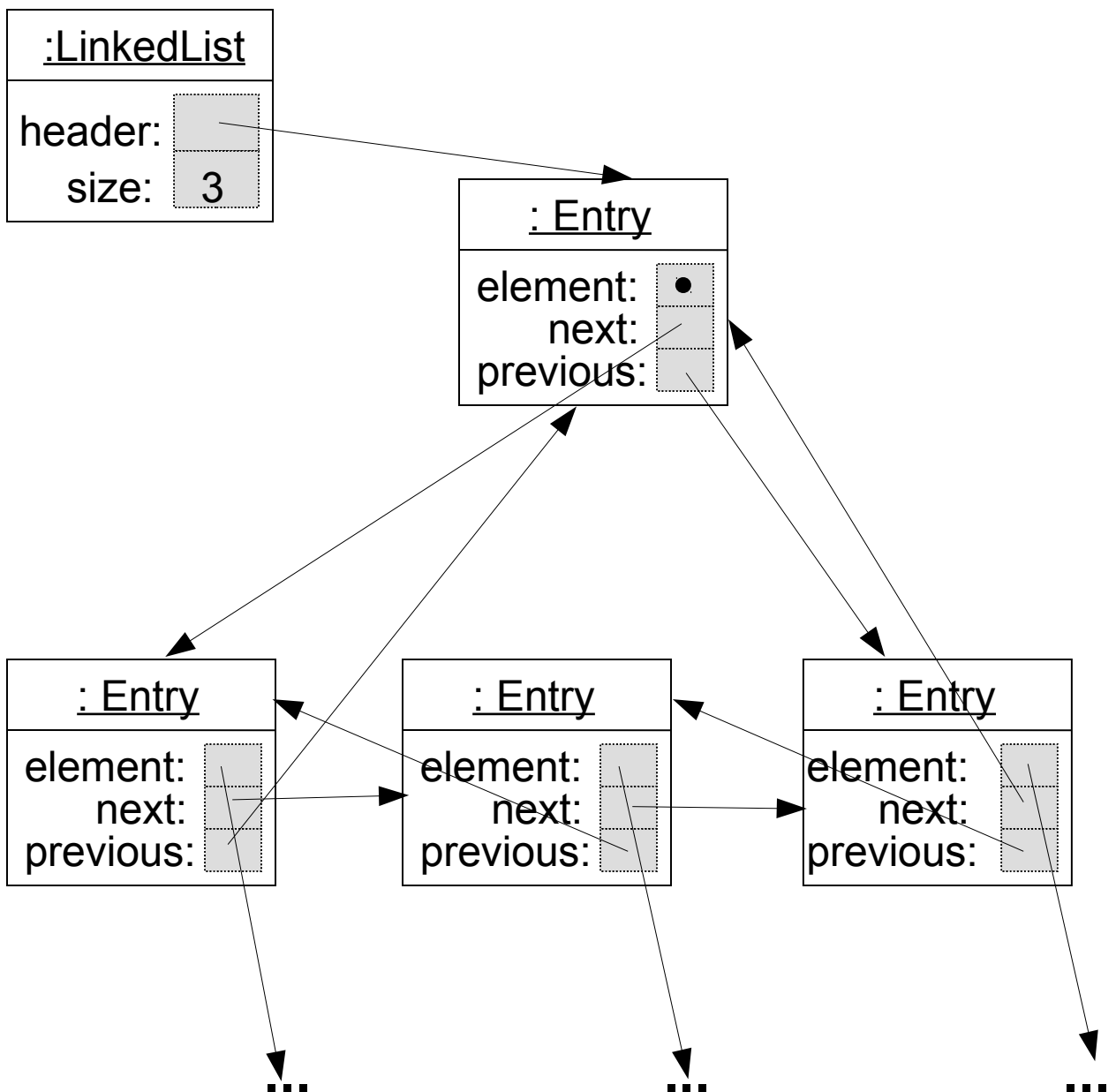
# Examples: (Object structures)

1. A very simple structure consisting of 2 objects:



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

## 2. A simple structure of several objects:



3. More interesting object structures can e.g. be found in graphical user interfaces.



## Problems:

- Structural integrity (→ structural invariants)
- Behavior and state of an object tightly depends on other objects.
- „Multiple references“ to one object (→ aliasing)

## Explanation: (Aliasing in OO programming)

An object  $X$  is **aliased** if two or more variables hold references to  $X$ . A variable can be

- an instance variable
- a class or static variable (global variable)
- a local variable of a method incarnation, including the implicit parameter **this**
- a formal parameter
- the result of a method invocation or her intermediate expression results

Often such a variable is called an **alias** of  $X$ .

Aliasing is called **static** if all involved variables are instance or static variables, i.e. belong to the heap. Otherwise, it is called **dynamic**.



## Desirable Aspects of Aliasing:

- Consistent view to shared objects
- Multiple cursors/iterators into object structures
- Time and space efficiency

## Examples: (Desirable aliasing)

1. Address objects that capture the address of a person or institution:
  - Several objects may reference an address *A*.
  - If *A* is modified, the referencing objects keep a consistent and up-to-date view.
2. Consider a complex tree or graph structure:
  - The structure can be traversed by several iterators.
  - Cursors can point to nodes of current interest.
3. Data structures can share objects (e.g. singly linked lists):
  - avoids to copy data structures when they are enlarged or reduced in size;
  - saves a lot of memory space.



## Undesirable Aspects of Aliasing:

1. Side-effects may become difficult to understand and control
2. Inconsistent access to objects
3. Aliases allow to by-pass interface operations
4. Optimizations and formal techniques become more complex

## Examples: (Undesirable aliasing)

1. Violation of invariants through aliases:
  - A set class uses a list class to implement the collection. Invariant: no duplicates.
  - Alias to the list can lead to invariant violation.
2. Inconsistent access:
  - An algorithm reads in different steps the attributes of an address object  $X$  (name, street, town).
  - Someone else modifies  $X$  while the algorithm reads its information.

### 3. By-passing interfaces:

#### Security breach in Java JDK 1.1.1:

- Each Class-object stores array of signers
- Only trusted signers get extended access rights to the host system
- Through *leaking* of the array of signers, a malicious applet can modify the list of signers to get extended rights
- Again, access modifiers cannot prevent the problem, because arrays are mutable

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

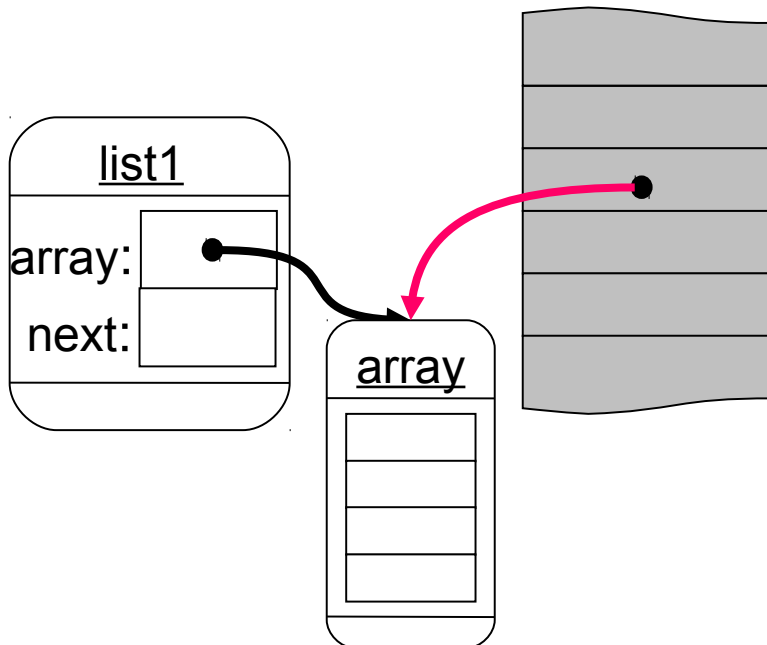




# Forms of Alias Creation:

## Capturing:

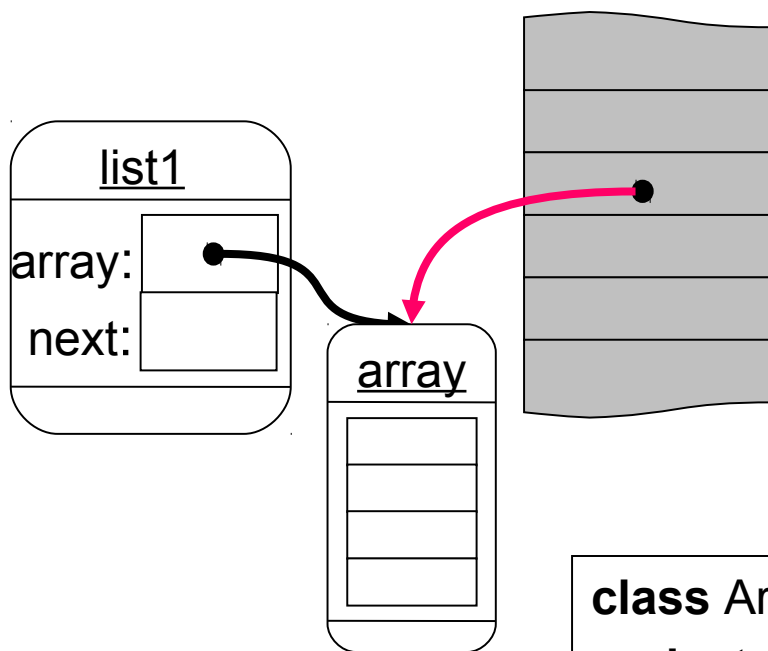
- Capturing occurs when objects are **passed to a data structure and then stored** by the data structure.
- Capturing often occurs **in constructors** (e.g., streams in Java).
- Problem: Alias can be used to **by-pass interface** of data structure.



```
class ArrayList {  
    private int[ ] array;  
    private int next;  
    public void addElems( int[ ] ia )  
        { array = ia; next = ia.length; }  
    ...  
}
```

## Leaking:

- Leaking occurs when data structure **passes 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.



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

General approaches to solve alias problems:

- Support aliasing, but disable modifications (avoids undesirable aspects (1.), (2.) and (4.) from above)
- Control aliasing and access

## 4.2 Immutability

**Referential transparency** means that the holder of an object reference  $R$

- does not have to care about aliases of  $R$ ,
- in particular, cannot observe whether aliases of  $R$  exist (by using  $R$ ).

Techniques to achieve referential transparency:

- immutability
- uniqueness

### Assumption:

To keep the following definitions simpler, we assume that reading and writing to instance variables of an object  $X$  can only be done by methods of  $X$ 's class (possibly inherited methods).

That is, objects can only be observed by method invocation.



## Explanation: (Immutability)

We assume that equality for the primitive types is defined by „==“ and for reference types by some reasonably defined method `equals`.

An object  $X$  is called (*observationally*) **immutable** if after termination of its constructor call any two invocations

$X.m(p_1, \dots, p_n)$  and  $X.m(q_1, \dots, q_n)$

with  $p_i$  equals  $q_i$  ( $1 \leq i \leq n$ ) either

- yield equal results or
- throw equal exceptions or
- both do not terminate.

A *class*  $C$  is called *immutable* if all instances of  $C$  in any program execution are immutable. ■

## Remark:

- Immutability is usually defined by prohibiting state changes and dependency of „external“ state.
- About 660 concrete classes in Java's standard library are immutable (about 20%) ■

## Examples: (Immutability)

### 1. Immutable class:

```
class ImmutableList {
    private int head;
    private ImmutableList tail;

    boolean isEmpty(){ return tail == null; }

    int head(){
        if( isEmpty() )
            throw new NoSuchElementException();
        return head;
    }

    ImmutableList tail(){
        if( isEmpty() )
            throw new NoSuchElementException();
        return tail;
    }

    ImmutableList cons( int i ) {
        ImmutableList aux = new ImmutableList();
        aux.head = i;
        aux.tail = this;
        return aux;
    }

    boolean equals( Object that ) {
        ...
    }
}
```

Objects of class `ImmutableList` are immutable for an appropriate method `equals`.

```

class ImmutableList { // continued
    ...
    boolean equals( Object that ) {
        if( that == null ||
            !(that instanceof ImmutableList )
            return false;
        ImmutableList tl = (ImmutableList) that;
        if( isempty() ) {
            return that.isempty();
        } else if( that.isempty() ) {
            return false;
        } else {
            return ( this.head == that.head )
                && tail().equals( that.tail() );
        }
    }
}

```

## 2. Immutability and inheritance:

There may be scenarios in which objects *of type* `ImmutableList` are not immutable:

```

...
static boolean somemethod( ImmutableList il )
{
    if( !il.isempty() ) {
        return il.head() == il.head();
    } else return true;
}

```

```

class NotImmutableList extends ImmutableList
{
    private boolean flag;

    boolean isEmpty(){ return false; }

    int head(){
        flag = !flag;
        if( flag ){
            return 7;
        } else return 9;
    }

    ImmutableList cons( int i ) {
        return new NotImmutableList();
    }
}

```

```

...
ImmutableList il = new NotImmutableList();
il.cons(7).cons(9);
System.out.println( somemethod(il) );
...

```

The example demonstrates that

→ subclasses can breach immutability.

### 3. Immutable objects must not depend on global variables:

```
class Global {
    public static int a = 5;
    public static int getA() {
        return a;
    }
}

class Immutable2 {
    public int alwaysTheSame() {
        return Global.getA();
    }
}
```

**Immutable2-objects are not immutable!**

The example demonstrates that

→ mutability can depend on global state.



#### 4. Immutable objects with varying I/O-behavior:

```
final class Immutable3 {
    private int state;
    public int alwaysTheSame() {
        System.out.println(state++);
        return 47;
    }
}
```

#### 5. Immutable objects where state changes can make sense:

Typical example: Initialization of instance variables on demand, i.e., not by the constructor, but prior to first use (***memoization***)



### Techniques for Realizing Immutability:

Immutability is implemented using different techniques:

- Access and inheritance restrictions
- Immutable state (local and reachable state)
- Prohibition of access to global state (direct or via methods)

## Sufficient criteria for immutability of an object $X$ :

- Instance variables of  $X$  cannot be modified after termination of its constructor.
- $X$  is not exposed during construction.
- Objects referenced by  $X$  are immutable according to these criteria.
- Constructors can only take immutable objects as parameters.
- Methods do not depend on global variables, i.e.,
  - They do not access global variables.
  - They do not invoke methods that depend on global variables.
- Methods do not create new objects.

## Remarks:

- The above criteria are still difficult to check:
  - It is difficult to show that no modifications can occur.
  - Inheritance has to be controlled.
- Classes like String are (almost) immutable, although they modify the state and do not satisfy the above criteria.
- Restricted forms of immutability can be checked by tools.



## 4.3 Alias Control and Encapsulation

Explanation: (Alias control, encapsulation)

Techniques for ***alias control*** avoid or alleviate alias problems by preventing certain forms of aliasing and by guaranteeing structural invariants.

Implementation techniques for alias control are:

- Alias modes
- Access restrictions
- Read-only references/methods
- Encapsulation

***Encapsulation techniques*** structure the state space of executing programs in a way that allows

- to guarantee data and structural consistency by
- establishing *boundaries / capsules* with well-defined interfaces.



Alias control and encapsulation techniques are another approach to reduce aliasing problems:

1. Side-effects are simpler to control.
2. Consistent access to objects can be achieved.
3. By-passing of interface operations can be avoided

Overview:

- Notions of alias control
- Type-based encapsulation

### 4.3.1 Notions of Alias Control

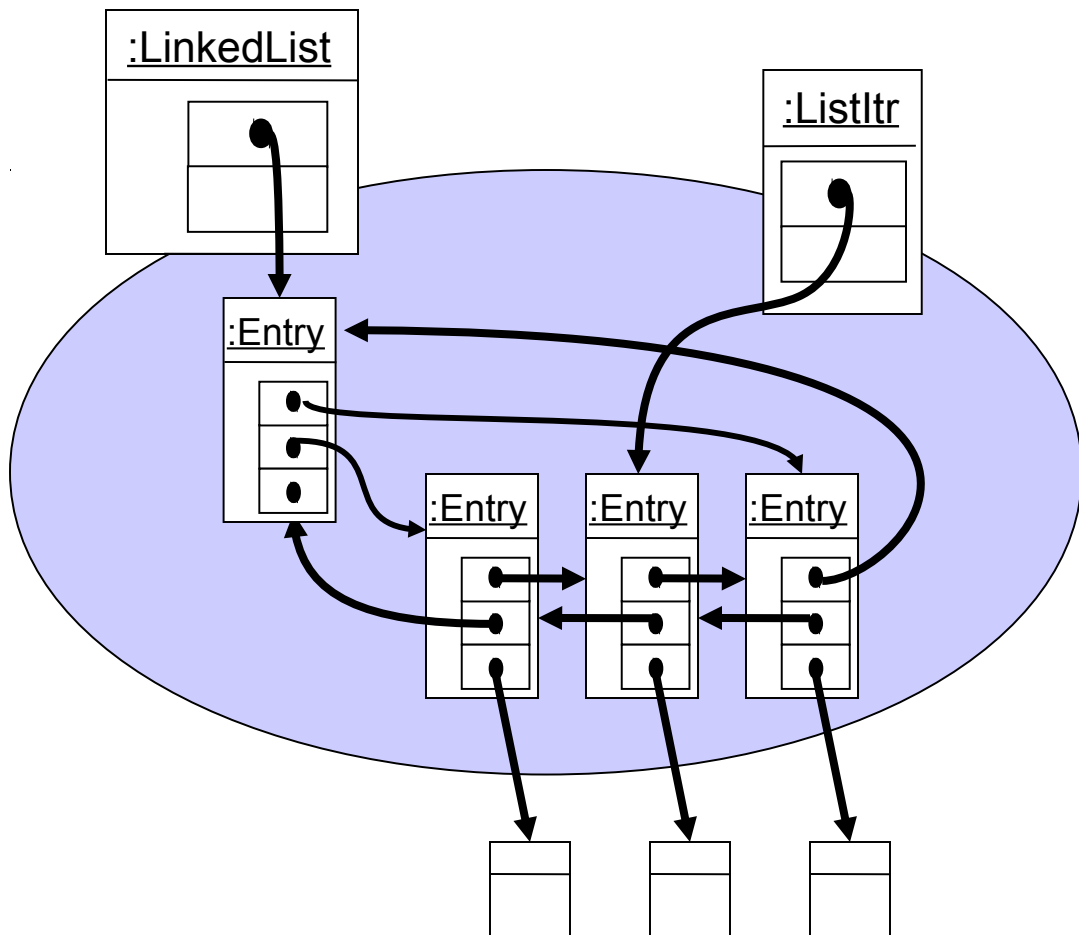
We need **better control** over the objects in an object structure to avoid aliasing problems.

Approach: ***Alias modes***:

- Define **roles** of objects in object structures
- Assign a tag (**alias mode**) to every expression to indicate the role of the referenced object
- Impose **programming rules** to guarantee that objects are only used according to their alias modes

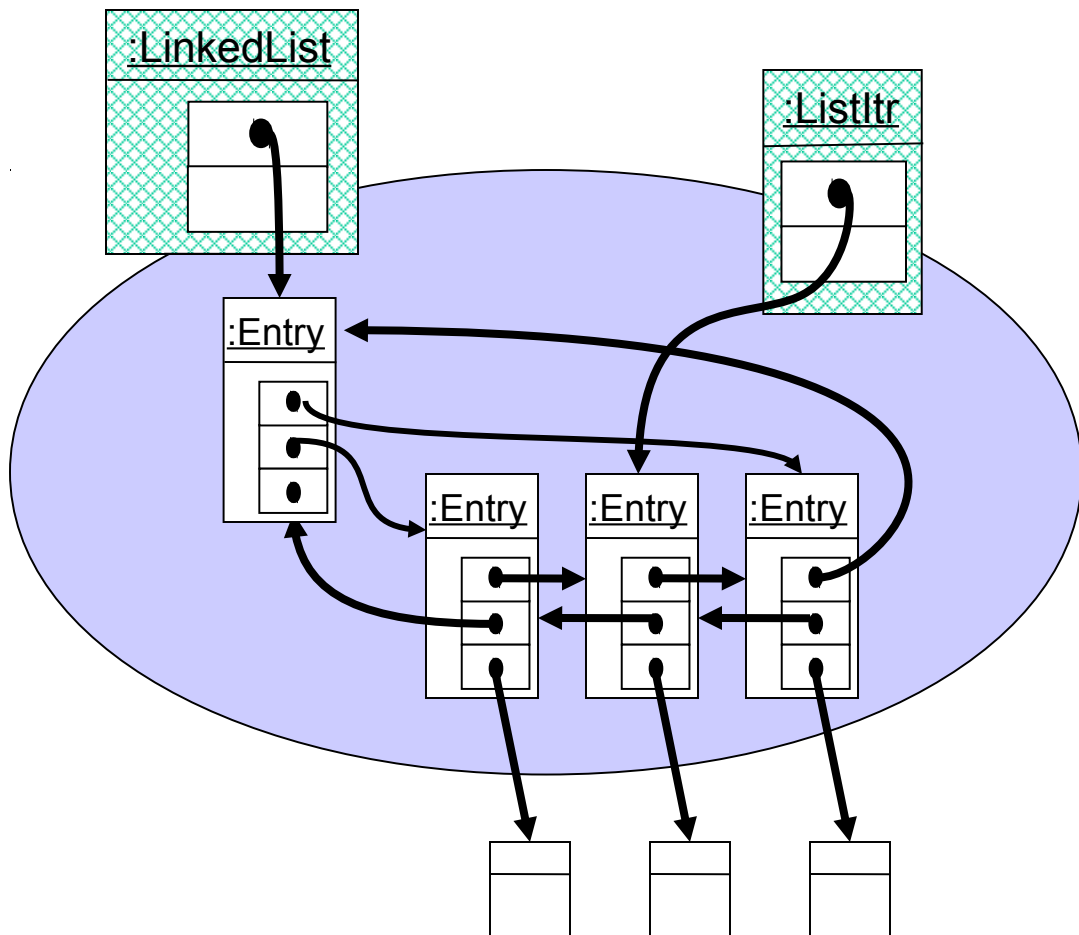
## Roles in Object Structures:

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



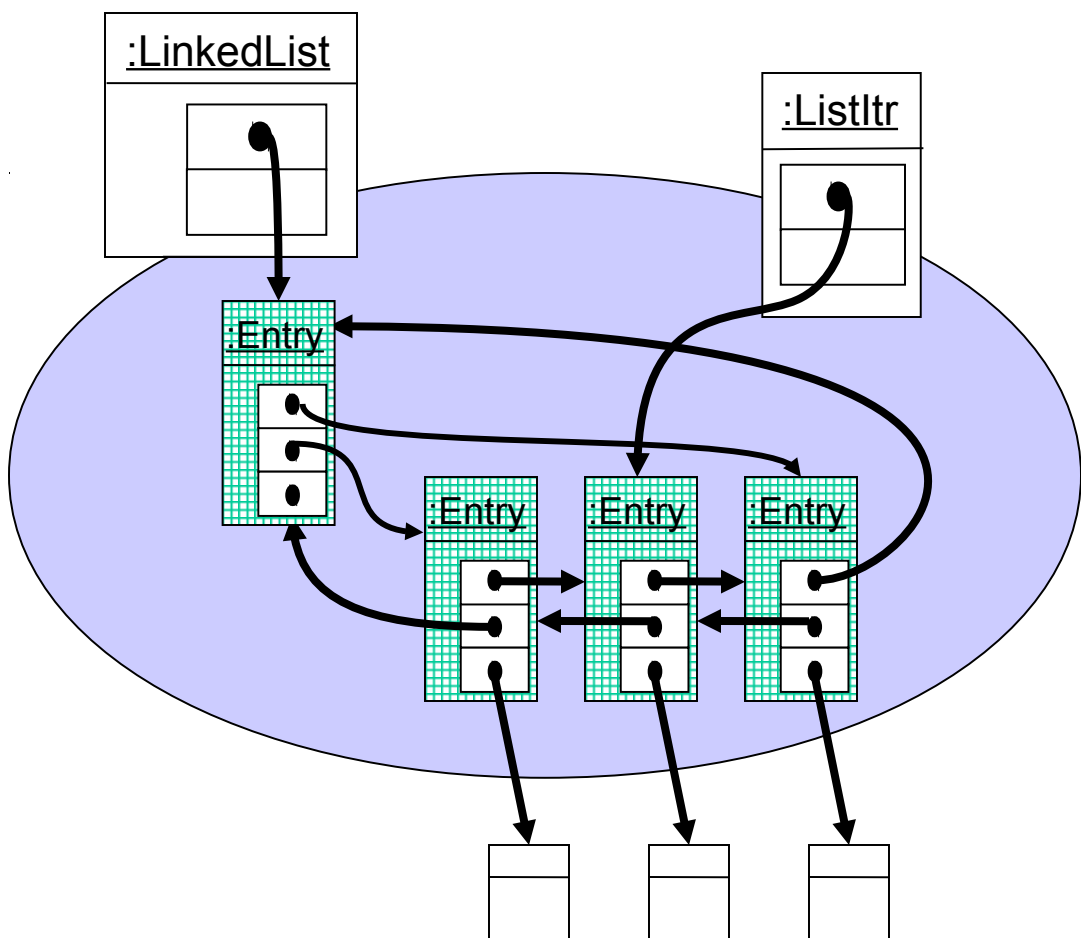
## 1. Interface objects (default mode):

- Interface objects are used to **access the structure**
- Interface objects can be **used in any way** objects are usually used (passed around, changed, etc.)



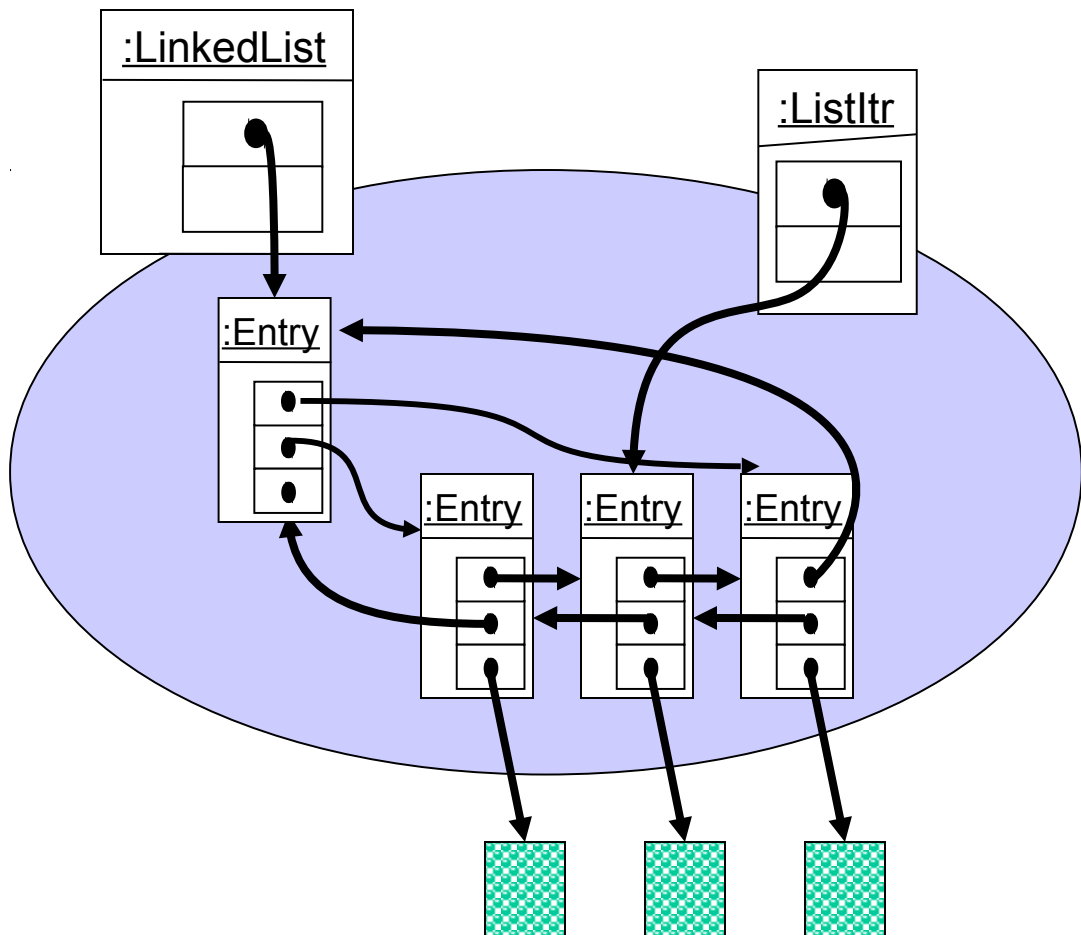
## 2. Representations (rep mode):

- Expressions with mode “rep” hold references to the **representation** of the object structure
- Objects referenced by rep-expressions can **be changed**
- Rep-objects **must not be exported** from the object structure



### 3. Arguments (arg mode):

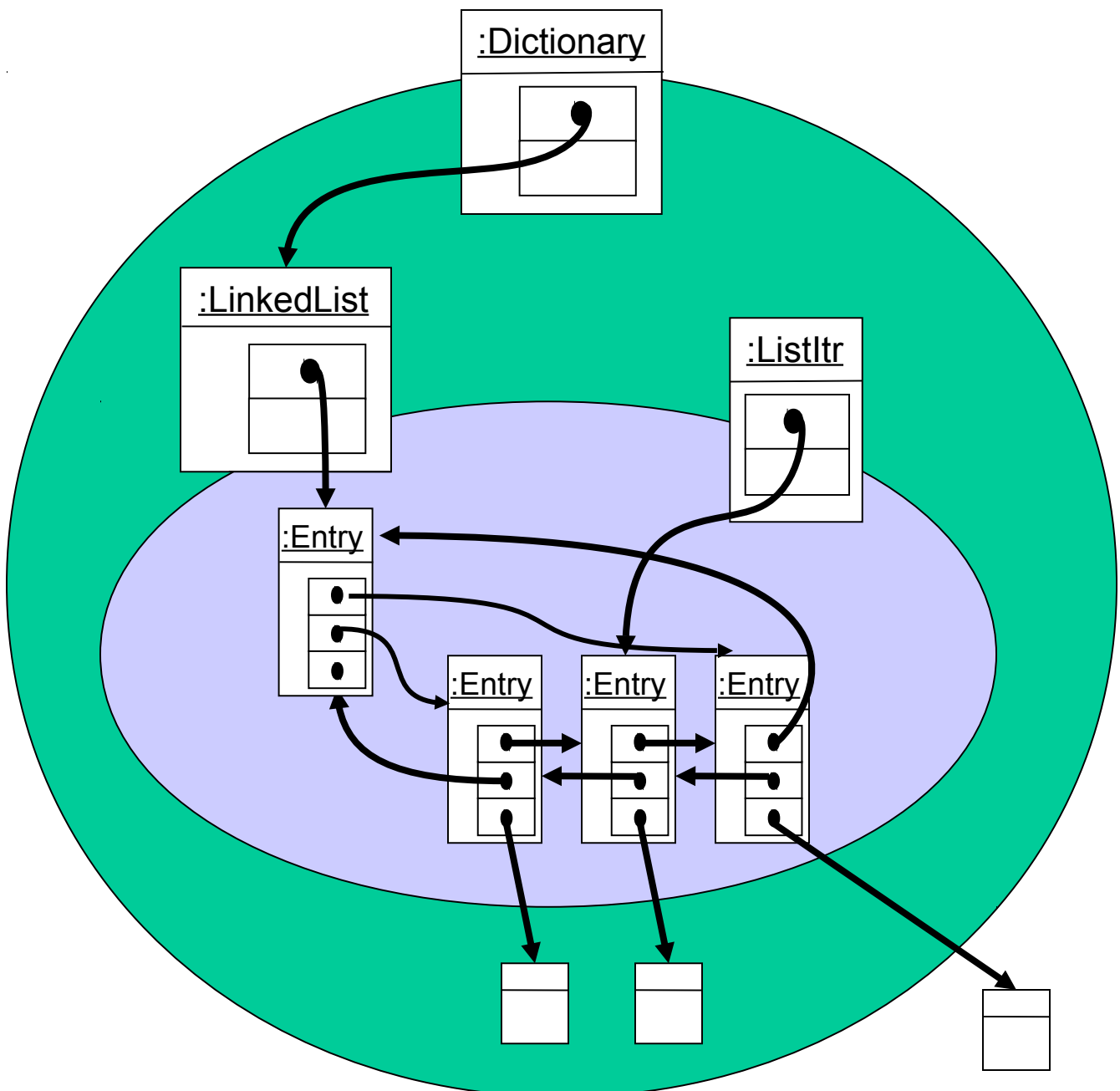
- Expressions with mode “arg” hold references to **arguments** of the object structure.
- Objects **must not be changed** through arg-references.
- Arg-objects can be **passed around** and aliased freely.





## Meaning of alias modes:

- Alias modes describe the role of an object relative to an interface object
- Informally: references with
  - **default mode** stay in the same bubble
  - **rep-mode** go from an interface object into its bubble
  - **arg-mode** may go to any bubble.



## Example: (Alias modes as annotations)

In programs, alias modes can be expressed by comments/annotations tagging types:

```
class LinkedList {
  private /* rep */ Entry header;
  private int size;

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

```
class Entry {
  private /* arg */ Object element;
  private Entry previous, next;

  public Entry( /* arg */ Object o, Entry p, Entry n ) { ... }
}
```



## Programming Discipline (simplified):

### **Rule 1: No role confusion**

- Expressions 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

The following examples illustrate these rules.

Literature:

J. Noble, J. Vitek, J. M. Potter:  
Flexible Alias Protection. ECOOP '98

D. Clarke, J. Noble, T. Wrigstad:  
Aliasing in Object-Oriented Programming.  
LNCS 7850; Springer-Verlag, 2013

## Example: (Role confusion)

- Array is **internal representation** of the list
- Method addElems **confuses alias modes**

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

  public void addElems( int[ ] ia ) {
    array = ia;
    next = ia.length;
  }
  ...
}
```

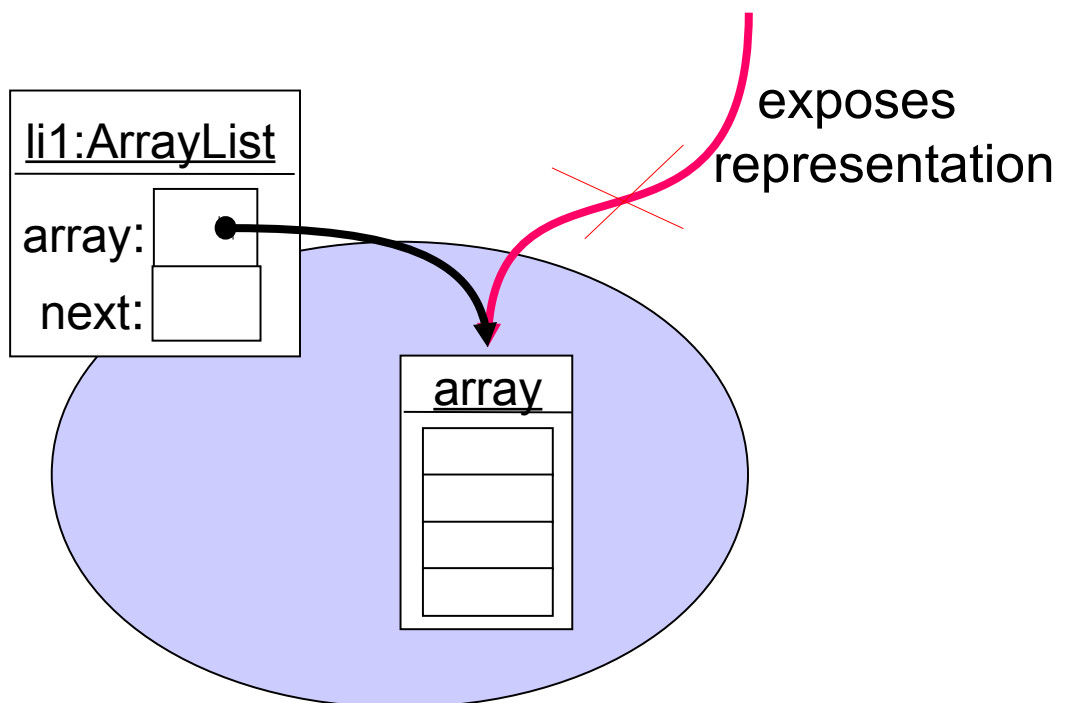
Clean solution requires **array copy**:

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



## Example: (Representation exposure)

- Rep-objects can only be referenced
  - by their interface objects
  - by other rep-objects of the same object structure
- Rep-objects can only be modified
  - by methods executed on their interface objects
  - by methods executed on rep-objects of the same object structure
- Rep-objects are **encapsulated** inside the **object structure**



## Example: (Argument dependencies)

```
class Task {
    int prio;
    ...
}

class PriorityQueue {
    Vector tasks = new Vector();

    /*@ invariant
       @ (\forall int i,j;
       @   0<=i && i<j && j<tasks.size();
       @   tasks.get(i).prio <=
       @   tasks.get(j).prio );
    @*/

    void insertTask(/*arg*/ Task t ){...}

    Task nextTask() {
        return
            (Task) tasks.get(tasks.size()-1);
    } }
}
```

PriorityQueue depends on the prio-attribute of the Task-arguments.

Modifying the attribute could violate the invariant of priority queues and cause malfunction of nextTask. ■

## 4.3.2 Type-based Encapsulation

Type systems can be used to guarantee **encapsulation invariants**:

- Encapsulation at the package level: *confined types*
- Further structuring techniques

### Encapsulation at the Package Level:

Goal:

Guarantee that objects of particular type can only be accessed and manipulated by the classes of one package.

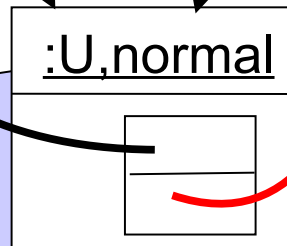
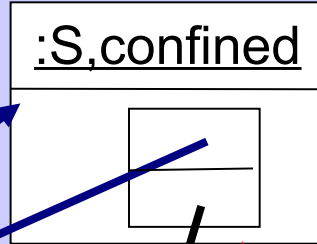
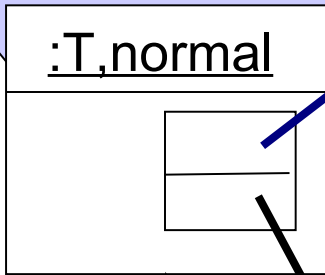
Approach:

1. Consider a package P as a capsule containing the objects of the classes in P. The capsules are disjoint.
2. Mark a type T (interface/class) of P as *confined* if objects of type T should only be manipulated by program code in P. We say as well that these objects are confined.
3. Define rules guaranteeing that references of confined objects of P are only stored in instance variables, parameters and local variables of objects of P.

outside P

package P

inside P



encapsulation error

package Q



## Connection with type systems:

- Each variable gets as extended type information:
  - the package to which it belongs (implicit)
- Each object gets as extended type information:
  - the package to which it belongs (implicit)
  - information whether confined or normal
- Encapsulation errors (analogous to type errors) occur if a variable  $v$  of package  $P$  references a confined object  $X$  of a package  $Q$  different from  $P$ . Invariant:

If  $v$  references some confined  $X$ , then  $v$  and  $X$  belong to the same package.

- Encapsulation errors are excluded by statically checkable rules (analogous to type conditions/rules).

We consider an extension to Java by confined types:

J. Vitek, B. Bokowski: Confined types in Java.

Software: Practice & Experience, 31(6):507-532, 2001.

## Remark:

The study of the encapsulation rules is of general interest for the development of OO-programs.



## Scenarios in which references are exported (1-6):

```
package inside;
```

```
public class C extends outside.B {  
    public void putReferences() {  
        C c = new C();  
/*1*/    outside.B.c1 = c;  
/*2*/    outside.B.storeReference(c);  
/*3*/    outside.B.c3s = new C[]{c};  
/*4*/    calledByConfined();  
/*5*/    implementedInSubclass();  
/*6*/    throw new E();  
    }  
  
    public void implementedInSubclass() {}  
/*7*/    public static C f = new C();  
/*8*/    public static C m(){ return new C();}  
/*9*/    public static C[] fs  
        = new C[]{new C()};  
/*10*/    public C() { }  
}  
  
public class E extends RuntimeException{}
```

## Scenarios in which references are imported (7-10):

```
package outside;

public class B {
    /*1*/ public static inside.C c1;
    /*2*/ public static void
        storeReference( inside.C c2 ){
        // store c2
    }
    /*3*/ public static inside.C[] c3s;
    /*4*/ public void calledByConfined(){
        // store this
    }
    static void getReferences() {
    /*7*/     inside.C c7 = inside.C.f;
    /*8*/     inside.C c8 = inside.C.m();
    /*9*/     inside.C[] c9s = inside.C.fs;
    /*10*/    inside.C c10 = new inside.C();
        D d = new D();
        try {
            d.putReferences();
    /*6*/    } catch( inside.E ex ){
        // store ex
    }
    }
}

class D extends inside.C {
    /*5*/ public void implementedInSubclass(){
        // store this
    }
}
```

## Explanation: (confined, anonymous)

A type is called **confined**

- if it is declared with the keyword `confined` or
- if it is an array type with a confined component type.

Methods and constructors can be declared as **anonymous** if the following properties hold:

- A1: The reference `this` can only be used for accessing fields and calling anonymous methods of the current instance or for object reference comparison.
- A2: Anonymity of methods and constructors must be preserved when overriding methods.
- A3: Constructors called from an anonymous constructor must be anonymous.
- A4: Native methods may not be declared anonymous.



## Remark:

- The behavior of anonymous methods only depends on the actual parameters and the values of the instance variables of the implicit parameter.
- Anonymous methods cannot introduce new aliases to the current receiver object.
- Anonymous methods allow confined types to use methods inherited from unconfined supertypes.



## Static Encapsulation Rules:

The following rules **guarantee** the encapsulation:

### Confinement in declarations:

C1: A confined class or interface must not be declared public and must not belong to the unnamed global package.

C2: Subtypes of a confined type must be confined.

## Preventing widening:

- C3: Widening of references from a confined type to an unconfined type is forbidden in assignments, method call arguments, return statements, and explicit casts.
- C4: Methods invoked on a confined object must either be non-native methods defined in a confined class or be anonymous methods.
- C5: Constructors called from the constructors of a confined class must either be defined by a confined class or be anonymous constructors.

## Preventing transfer from the inside and outside:

C6: Subtypes of `java.lang.Throwable` and `java.lang.Thread` must not be confined.

C7: The declared type of public and protected fields in unconfined types must not be confined.

C8: The return type of public and protected methods in unconfined types must not be confined.

## Further Structuring Techniques:

Different structuring techniques have been discussed in the literature in the last years. Often cited techniques:

- Balloon types:

P. S. Almeida: Balloon Types: Controlling Sharing of State in Data Types. ECOOP '97.

- Ownership:

D. G. Clarke, J. M. Potter, J. Noble: Ownership Types for Flexible Alias Protection. OOPSLA '98.

Simple Ownership Types for Object Containment. ECOOP '01.

P. Müller, A. Poetzsch-Heffter: A Type System for Controlling Representation Exposure in Java. Formal Techniques for Java Programs '00.

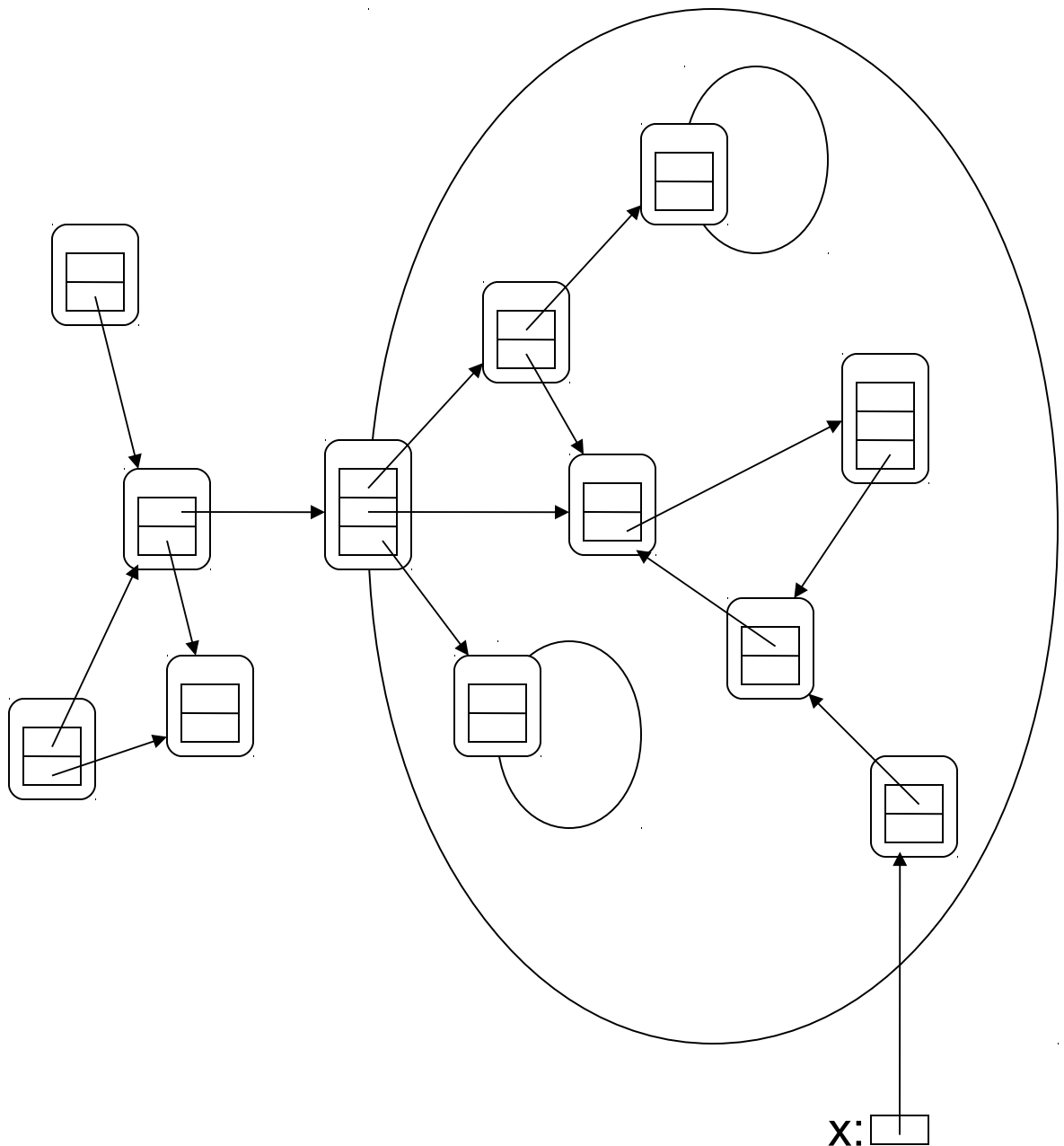
Other approaches are:

- Preventing aliasing
- Write protection and readonly modes

## Balloon types:

A type can be marked as balloon type.  
Objects of balloon types are called *balloon objects*.

### Idea:





## Cluster:

Let  $G$  be the undirected graph with

- all objects as nodes and
- all references between non-balloon objects and from balloon objects to non-balloon objects as edges.

A **cluster** is a connected subgraph of  $G$  which is not contained in a larger connected subgraph.

## Internal objects:

An object  $O$  is said to be *internal* to a balloon object  $B$  if and only if:

- $O$  is a non-balloon object in the same cluster as  $B$  or
- $O$  is a balloon object referenced by  $B$  or by some non-balloon object in the same cluster as  $B$  or
- there exists a balloon object  $B'$  internal to  $B$  and  $O$  is internal to  $B'$ .

## External objects:

An object is said to be *external* to a balloon object  $B$  iff it is neither  $B$  nor internal to  $B$ .

## Balloon invariant:

If B is an object of a balloon type then:

- B is referenced by at most one instance variable.
- If such a stored reference exists it is from an object external to B.
- No object internal to B is referenced by any object external to B.

## Remark:

- The invariant allows that objects referenced only by dynamic aliases reference internal objects. Such objects are internal to the balloon.

That is why a balloon B can contain more objects than just the set of all objects reachable from B.

- To guarantee the balloon invariant, Almeida uses the following techniques:
  - Assignment of references to instance variables of balloon objects is not always allowed.
  - Data flow analysis



## Ownership:

### Idea:

- Introduce an ownership relation between objects:  
object X owns object Y.
- Define encapsulation based on ownership:  
Only the owner may directly access his objects.
- Declare the ownership relation as extended type information.

### Example: (Ownership annotations)

```
class Engine {
    void start() { ... }
    void stop() { ... }
}

class Driver { ... }

class Car {
    rep Engine engine; // belongs to representation
    Driver driver;     // not part of representation
    Car() {
        engine = new rep Engine();
        driver = null;
    }
}
```

```

rep Engine getEngine() { return engine; }
void setEngine( rep Engine e ){ engine=e; }
void go () {
    if(driver!=null) engine.start();
}
}

```

```

class Main {
    void main() {
        Driver bob = new Driver(); // root owner
        Car car = new Car(); // root owner
        car.driver = bob;
        car.go();

        car.engine.stop(); // fails
        car.getEngine().stop(); // fails

        rep Engine e = new rep Engine();
        car.setEngine(e); // fails
    }
}

```



The **ownership relation** is a binary relation between objects where objects are either normal objects or the special owner `root`.

The owner of an object `X` is determined when `X` is created:

- If `X` is created by `new T()`, then `root` is the owner.
- If `X` is created by `new rep T()`, then the current receiver object will be the owner of `X`.

### Ownership invariant:

All reference paths from a global object, that is, an object owned by `root`, to some object `X` with owner `B` go through `B`.

### Remark:

- Ownership invariant is more general than the balloon invariant. It allows references to leave the set of owned objects.
- The ownership relation in the described form is still quite restrictive, as it only allows one owner.
- Recently, a number of different ownership models have been developed and investigated.

