Concepts of Object-Oriented Programming

Peter Müller
Chair of Programming Methodology

Autumn Semester 2015
Object Structures

- Objects are the building blocks of object-oriented programming
- However, interesting abstractions are almost always provided by sets of cooperating objects

Definition:

An object structure is a set of objects that are connected via references
Example 1: Array-Based Lists

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

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

    public void setElems( int[ ] ia )
    {
        …
    }
}

array:

next:

list

array:

length:

0:

1:

2:

…
Example 2: Doubly-Linked Lists

- **LinkedList**
  - header: 
  - size: 3

- **Entry**
  - n:
  - p:
  - e:

- **ListItr**
  - next: 
  - nextIndex: 2

- **Object**

---

Peter Müller – Concepts of Object-Oriented Programming
6. Object Structures and Aliasing

6.1 Aliasing
6.2 Problems of Aliasing
6.3 Readonly Types
6.4 Ownership Types
Alias

- **Definition:**
  
  *A name that has been assumed temporarily*

  [WordNet, Princeton University]
Aliasing in Procedural Programming

- var-parameters are passed by reference (call by name)
- Modification of a var-parameter is observable by caller
- Aliasing: Several variables (here: p, q) refer to same memory location
- Aliasing can lead to unexpected side-effects

```plaintext
program aliasTest
procedure assign( var p: int, var q: int );
begin
  { p = 1 ∧ q = 1 }
  p := 25;
  { p = 25 ∧ q = 25 }
end;
begin
  var x: int := 1;
  assign( x, x );
  { x = 25 }
end
end.
```
Aliasing in Object-Oriented Programming

- Definition:

  *An object o is aliased if two or more variables hold references to o.*

- Variables can be
  - Fields of objects (instance variables)
  - Static fields (global variables)
  - Local variables of method executions, including `this`
  - Formal parameters of method executions
  - Results of method invocations or other expressions
Static Aliasing

- Definition:
  An alias is static if all involved variables are fields of objects or static fields.

- Static aliasing occurs in the heap memory

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

- Definition: 
  *An alias is dynamic if it is not static.*

- Dynamic aliasing involves stack-allocated variables

```java
int[] ia = list1.array;
list1.array[0] = 1;
ia[0] = -1;
System.out.println(list1.array[0]);
```
Intended Aliasing: Efficiency

- In OO-programming, data structures are usually **not copied** when passed or modified.

- Aliasing and destructive updates make OO-programming efficient.

```java
class SList {
    SList next;
    Object elem;
    SList rest() { return next; }
    void set(Object e) { elem = e; }
}

void foo(SList slist) {
    SList rest = slist.rest();
    rest.set(“Hello”);
}
```
Intended Aliasing: Sharing

- Aliasing is a direct consequence of object identity
- Objects have state that can be modified
- Objects have to be shared to make modifications of state effective
Unintended Aliasing: 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.

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

```java
class ArrayList {
    private int[] array;
    private int next;
    public int[] getElems()
    {
        return array;
    }
    ...
}
```
6. Object Structures and Aliasing

6.1 Aliasing

6.2 Problems of Aliasing

6.3 Readonly Types

6.4 Ownership Types
Observation

- Many **well-established techniques** of object-oriented programming work for individual objects, but **not for object structures in the presence of aliasing**

  "**The big lie of object-oriented programming is that objects provide encapsulation**"  
  [Hogg, 1991]

- **Examples**
  - Information hiding and exchanging implementations
  - Encapsulation and consistency
Exchanging Implementations

- Interface including contract remains unchanged

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

    // requires ia != null
    // ensures ∀ i. 0 ≤ i < ia.length:
    //     isElem(old( ia[i] ))

    public void setElems(int[] ia) {
        array = ia; next = ia.length;
    }
}
```
Exchanging Implementations (cont’d)

- Aliases can be used to by-pass interface
- Observable behavior is changed!

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

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

    // invariant array != null &&
    // 0<=next<=array.length &&
    // \forall i.0<=i<next: array[i] >= 0

    public void add(int i) { ... }
    public void setElems(int[] ia) {
        ... }
}
```
Consistency of Object Structures (cont’d)

- Aliases can be used to violate invariant
- Making all fields private is not sufficient to encapsulate internal state

```java
int foo( ArrayList list ) { // invariant of list holds
    int[] ia = new int[ 3 ];
    list.setElems( ia ); // invariant of list holds
    ia[ 0 ] = -1; // invariant of list violated
}
```
class Malicious {
    void bad() {
        Identity[] s;
        Identity trusted = java.Security…;
        s = Malicious.class.getSigners();
        s[0] = trusted;
        /* abuse privilege */
    }
}

Identity[] getSigners() {
    return signers;
}
Problem Analysis

- Breach caused by **unwanted alias**
  - Leaking of reference
- Difficult to prevent
  - Information hiding: not applicable to arrays
  - Restriction of Identity objects: not effective
  - Secure information flow: read access permitted
  - Run-time checks: too expensive
Other Problems with Aliasing

- **Synchronization in concurrent programs**
  - Monitor of each individual object has to be locked to ensure mutual exclusion

- **Distributed programming**
  - For instance, parameter passing for remote method invocation

- **Optimizations**
  - For instance, object inlining is not possible for aliased objects
### Alias Control in Java: LinkedList

- **All fields are private**
- **Entry is a private inner class of LinkedList**
  - References are not passed out
  - Subclasses cannot manipulate or leak Entry-objects
- **ListItr is a private inner class of LinkedList**
  - Interface ListIterator provides controlled access to ListItr-objects
  - ListItr-objects are passed out, but in a controlled fashion
  - Subclasses cannot manipulate or leak ListItr-objects
- **Subclassing is severely restricted**
Alias Control in Java: String

- All **fields** are **private**
- References to internal character-array are not passed out
- **Subclassing is prohibited** (final)
6. Object Structures and Aliasing

6.1 Aliasing
6.2 Problems of Aliasing
6.3 Readonly Types
6.4 Ownership Types
# Object Structures Revisited

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

    public String getStreet() { ... }
    public void setStreet(String s) { ... }
    public String getCity() { ... }
    public void setCity(String s) { ... }
}
```

```java
class Person {  
    private Address addr;
    public Address getAddress() {  
        return addr.clone();  
    }
    public void setAddress(Address a) {  
        addr = a.clone();  
    }
  }
```

![Diagram showing object structures and aliasing](image.png)

---

Peter Müller – Concepts of Object-Oriented Programming
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

```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();
    }
    ... }
```

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

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

interface ReadonlyAddress {
    …
    public ReadonlyPhoneNo getPhone( );
}
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

```java
class Person {
    private Address addr;
    public ReadonlyAddress getAddress() {
        return addr;
    }
    public void setAddress(Address a) {
        addr = a.clone();
    }
    ...
}

void m(Person p) {
    ReadonlyAddress ra = p.getAddress();
    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

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

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

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

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

Compile-time errors
Readonly Access in C++: const Functions

- const functions must not modify their receiver object

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

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

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

Compile-time error: Call of const function allowed.
It wouldn’t be C++ …

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

```cpp
class Address {
    string city;

public:
    string getCity( void ) const
    {
        return city;
    }

    void setCity( string s ) const {
        Address* me = ( Address* ) this;
        me->city = s;
    }
};

class Person {
    Address* addr;

public:
    const Address* getAddr( )
    {
        return addr;
    }

    void setAddr( Address a )
    {
        /* clone */
    }
};

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

Call of const function allowed
It wouldn’t be C++ … (cont’d)

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

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

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

void m( Person* p ) {
    const Address* a = p->getAddr();
    Address* ma = ( Address* ) a;
    ma->setCity( “Hagen” );
}
```
Readonly Access in C++: Transitivity

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

```c++
// Phone class
class Phone {
    public:
        int number;
};

// Address class
class Address {
    string city;
    Phone* phone;
    public:
        Phone* getPhone( void ) const
            { return phone; }
    ...;
};

void m( Person* p ) {
    const Address* a = p->getAddr( );
    Phone* p = a->getPhone( );
    p->number = 2331…;
}
```
Transitivity (cont’d)

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

const functions may modify objects other than the receiver
### Readonly Access in C++: Discussion

<table>
<thead>
<tr>
<th><strong>Pros</strong></th>
<th><strong>Cons</strong></th>
</tr>
</thead>
</table>
| - 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 | - 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 be overridden only by pure methods

```java
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 \texttt{readonly} T in programs

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

```java
class Person {
    private Address addr;
    public Readonly Address getAddr() {
        ...}
    }
```

6.3 Object Structures and Aliasing – Readonly Types
Subtype Relation

- **Subtyping** among readwrite and readonly types is defined as in Java
  - S extends or implements T ⇒ \( rw \ S <: \ rw \ T \)
  - S extends or implements T ⇒ \( ro \ S <: \ ro \ T \)

- **Readwrite types** are subtypes of corresponding readonly types
  - \( rw \ T <: \ ro \ T \)

```java
class T { ... }

class S extends T { ... }

S rwS = ...
T rwT = ...
readonly S roS = ...
readonly T roT = ...

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

rwT = roT;
```
Type Rules: Transitive Readonly

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

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>rw T</th>
<th>ro T</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw S</td>
<td>rw T</td>
<td>ro T</td>
</tr>
<tr>
<td>ro S</td>
<td>ro T</td>
<td>ro T</td>
</tr>
</tbody>
</table>
The type of
- A field access
- An array access
- A method invocation
is determined by the type combinator ▶

<table>
<thead>
<tr>
<th>▶</th>
<th>rw T</th>
<th>ro T</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw S</td>
<td>rw T</td>
<td>ro T</td>
</tr>
<tr>
<td>ro S</td>
<td>ro T</td>
<td>ro T</td>
</tr>
</tbody>
</table>

Person p = ...
```java
readonly Address a;
a = p.getAddr();
```
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 types must not be cast to readwrite types

```java
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**
- 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
6. Object Structures and Aliasing

6.1 Aliasing
6.2 Problems of Aliasing
6.3 Readonly Types
6.4 Ownership Types
Object Topologies

- Read-write aliases can still occur, e.g., by capturing or leaking

- We need to distinguish “internal” references from other references

```java
class Person {
    private Address addr;
    private Company employer;
    public readonly Address getAddress() {
        return addr;
    }
    public void setAddress(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
  - Must not be exposed to clients
- **Arguments** of the object structure
  - Must not be modified
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

```java
class LinkedList {
    private rep Entry header;
    ...
}
```

A list owns its nodes

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

Lists store elements with arbitrary owners

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 $\text{rep } T$ then $o$’s owner is $\text{this}$
  - If $e$ has type $\text{peer } T$ then $o$’s owner is the owner of $\text{this}$
  - If $e$ has type $\text{any } T$ then $o$’s owner is arbitrary
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 \( T \)
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 ) { … }
}
Viewpoint Adaptation: Example 1

```
List
Entry
Entry
Entry
```

peer ➤ peer = peer
Viewpoint Adaptation: Example 2

rep ➤ peer = rep
## Viewpoint Adaptation

<table>
<thead>
<tr>
<th></th>
<th>peer $T$</th>
<th>rep $T$</th>
<th>any $T$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>peer $S$</strong></td>
<td>peer $T$</td>
<td>?</td>
<td>any $T$</td>
</tr>
<tr>
<td><strong>rep $S$</strong></td>
<td>rep $T$</td>
<td>?</td>
<td>any $T$</td>
</tr>
<tr>
<td><strong>any $S$</strong></td>
<td>?</td>
<td>?</td>
<td>any $T$</td>
</tr>
</tbody>
</table>

\[
\tau(e) \triangleright \tau(f) <: \tau(v)
\]

\[
v = e.f;
\]

\[
\tau(v) <: \tau(e) \triangleright \tau(f)
\]

\[
e.f = v;
\]
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

```java
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: Details

<table>
<thead>
<tr>
<th></th>
<th>peer T</th>
<th>rep T</th>
<th>any T</th>
</tr>
</thead>
<tbody>
<tr>
<td>peer S</td>
<td>peer T</td>
<td>lost T</td>
<td>any T</td>
</tr>
<tr>
<td>rep S</td>
<td>rep T</td>
<td>lost T</td>
<td>any T</td>
</tr>
<tr>
<td>any S</td>
<td>lost T</td>
<td>lost T</td>
<td>any T</td>
</tr>
<tr>
<td>lost S</td>
<td>lost T</td>
<td>lost T</td>
<td>any T</td>
</tr>
</tbody>
</table>

- Subtyping
  - \( rep \ T <: lost \ T \)
  - \( peer \ T <: lost \ T \)
  - \( lost \ T <: any \ T \)

Another existential type
Type Rules: Field Access

- **The field read**
  
  \[ v = e.f; \]
  
  is correctly typed if
  - \( e \) is correctly typed
  - \( \tau(e) \triangleright \tau(f) <: \tau(v) \)

- **The field write**
  
  \[ e.f = v; \]
  
  is correctly typed if
  - \( e \) is correctly typed
  - \( \tau(v) <: \tau(e) \triangleright \tau(f) \)
  - \( \tau(e) \triangleright \tau(f) \) 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

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

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

peer Person joe;

joe.addr = new rep Address();

this.addr = new rep Address();
```
The self Modifier: Details

<table>
<thead>
<tr>
<th></th>
<th>peer T</th>
<th>rep T</th>
<th>any T</th>
</tr>
</thead>
<tbody>
<tr>
<td>peer S</td>
<td>peer T</td>
<td>lost T</td>
<td>any T</td>
</tr>
<tr>
<td>rep S</td>
<td>rep T</td>
<td>lost T</td>
<td>any T</td>
</tr>
<tr>
<td>any S</td>
<td>lost T</td>
<td>lost T</td>
<td>any T</td>
</tr>
<tr>
<td>lost S</td>
<td>lost T</td>
<td>lost T</td>
<td>any T</td>
</tr>
<tr>
<td>self S</td>
<td>peer T</td>
<td>rep T</td>
<td>any T</td>
</tr>
</tbody>
</table>

- Subtyping
  - self T <: peer T

\[ v = e.f; \]

\[ \tau(e) \gg \tau(f) <: \tau(v) \]

\[ e.f = v; \]

\[ \tau(v) <: \tau(e) \gg \tau(f) \]

\[ \tau(e) \gg \tau(f) \text{ does not have lost modifier} \]
Example: Sharing

```java
class Person {
    public rep Address addr;
    ...
}
```

- Different Person objects have different Address objects
  - No unwanted sharing
Example: Internal vs. External Objects

```java
class Person {
    private rep Address addr;

    public rep Address getAddr() {
        return addr;
    }

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

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

- Address is part of Person’s internal representations
- 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;
    }
}
Owner-as-Modifier Discipline

- Based on the topological type system we can strengthen encapsulation with extra restrictions
  - 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 \( \tau( e ) \) is self, peer, or rep
  - Method call e.m(...) is valid only if \( \tau( e ) \) is self, peer, or rep, or called method is pure
A method may modify only objects directly or indirectly owned by the owner of the current `this` object.
Internal vs. External Objects Revisited

```java
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
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 private inner or final classes, etc.

```java
class ArrayList {
    protected rep int[] array;
    private int next;
    ...
}
```
```java
class MyList extends ArrayList {
    public peer int[] leak() {
        return array;
    }
}
```
Exchanging Implementations

- Interface including contract remains unchanged

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

    // requires ia != null
    // ensures ∀i. 0 <= i < ia.length:
    // isElem( old( ia[ i ] ) )

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

    ...
}
```

```java
class ArrayList {
    private Entry header;

    // requires ia != null
    // ensures ∀i. 0 <= i < ia.length:
    // isElem( old( ia[ i ] ) )

    public void setElems( int[] ia )
    {
        ... /* create Entry for each element */
    }

    ...
}
```
Exchanging Implementations (cont’d)

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

    // requires ia != null
    // ensures ∀i. 0<=i<ia.length:
    //              isElem( old( ia[ i ] ) )

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

Accidental capturing is prevented

```java
class ArrayList {
    private rep Entry header;

    // requires ia != null
    // ensures ∀i. 0<=i<ia.length:
    //              isElem( old( ia[ i ] ) )

    public void setElems( any int[ ] ia )
        { … /* create Entry for each element */ }
    ...
}
```
6.4 Object Structures and Aliasing – Ownership Types

Exchanging Implementations (cont’d)

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

    public any int[ ] getElems( )
    { return array; }
    ...
}
```

```
class ArrayList {
    private rep Entry header;

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

- Observable behavior is changed

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

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

    // invariant array != null &&
    // 0<=next<=array.length &&
    // \forall i. 0<=i<next: array[i] >= 0

    public void add(int i) { ... }
    public void setElems(int[] ia) {
        ... }
    ...
}
```
Invariants for Object Structures

- The invariant of object o may depend on
  - Encapsulated fields of o
  - Fields of objects (transitively) owned by o

- Interface objects have full control over their rep-objects

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

    // invariant array != null &&
    // 0<=next<=array.length &&
    // \forall i.0<=i<next: array[ i ] >= 0

    public void add( int i ) { ... }
    public void setElems( any int[ ] ia ) { ... }
}
```
class Malicious {

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

Identity[] getSigners() {
    return signers;
}

System
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
References
