

Concepts of Object-Oriented Programming

Peter Müller

Programming Methodology Group

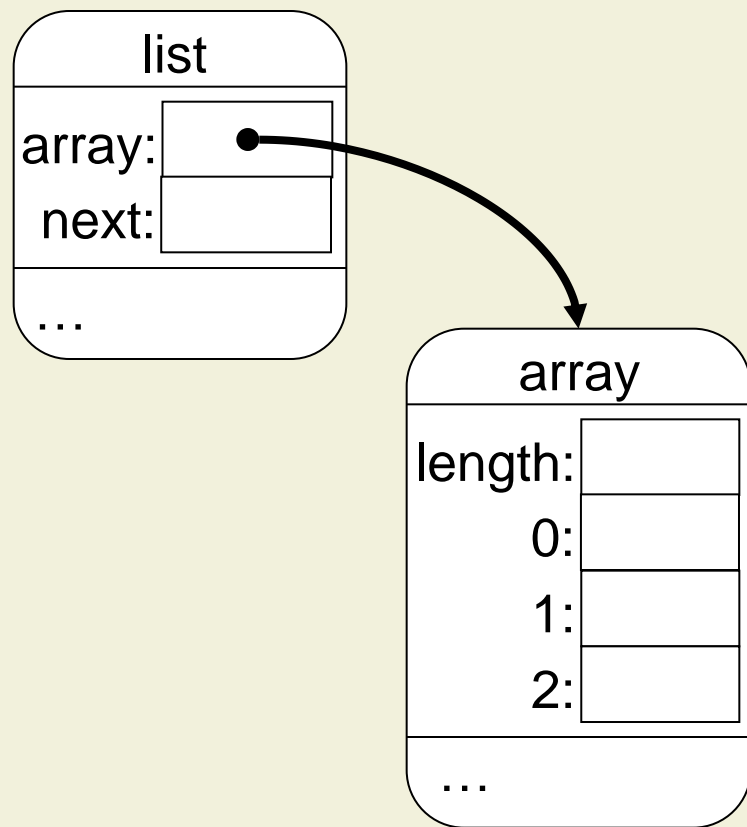
Autumn Semester 2024

ETH zürich

Object Structures

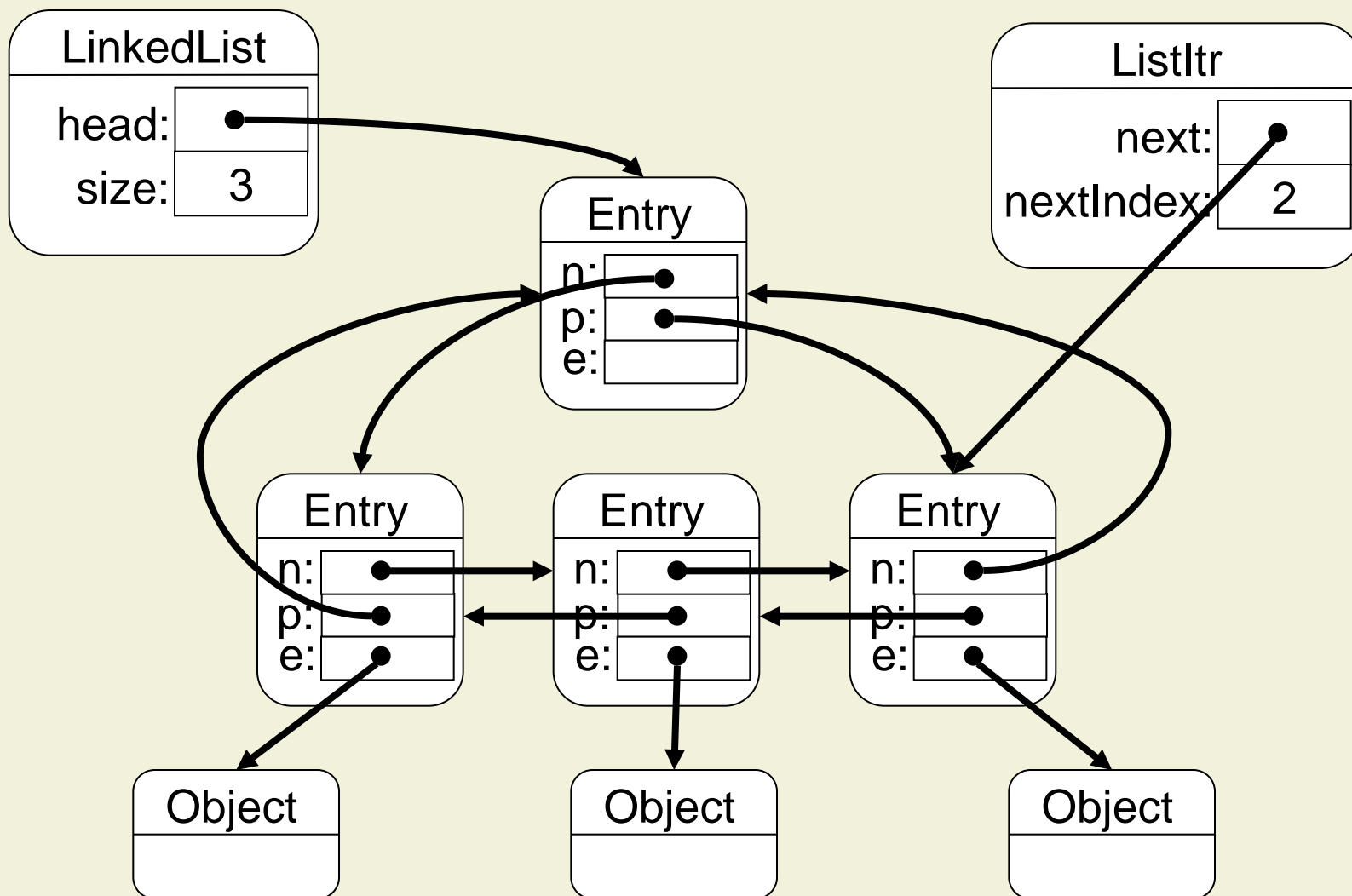
- Objects are the building blocks of object-oriented programming
- However, interesting abstractions are almost always provided by sets of cooperating objects
- Definition:
An object structure is a set of objects that are connected via references

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 )  
        { ... }  
  
    ...  
}
```

Example 2: Doubly-Linked Lists



5. Object Structures and Aliasing

5.1 Aliasing

5.2 Problems of Aliasing

5.3 Unique References

5.4 Readonly References

Aliasing

- Definition:

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

Aliasing

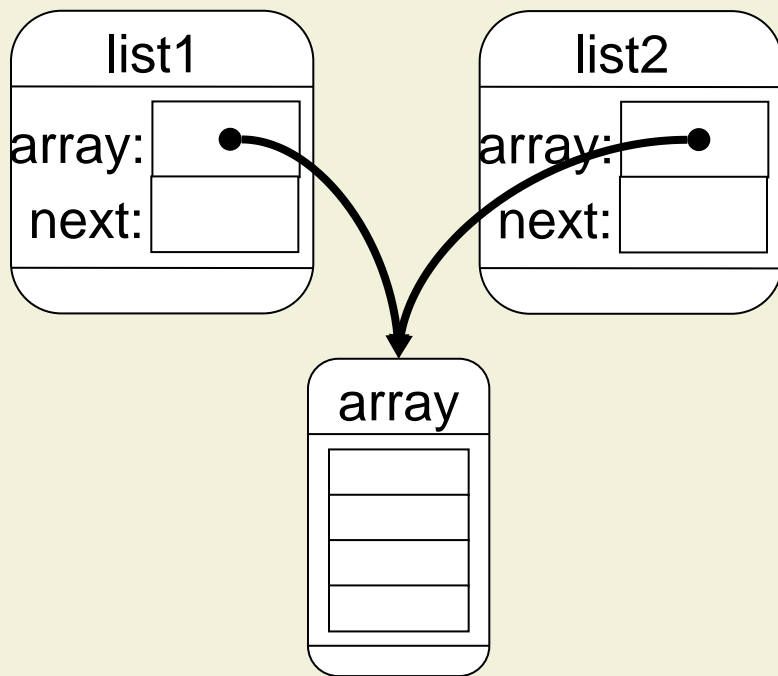
- Definition:

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

- Variables can be

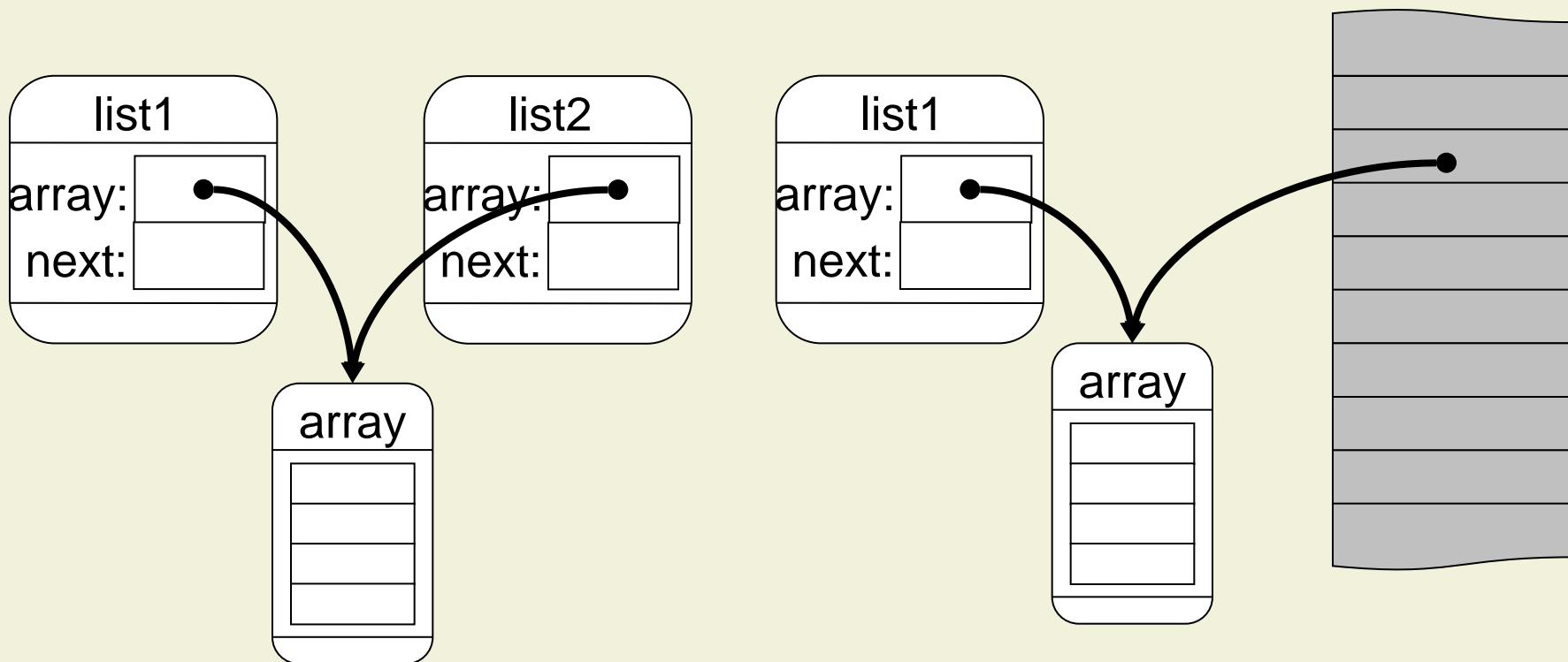
- Fields of objects (instance variables)
- Static fields (global variables)
- Local variables of method executions, including **this**
- Formal parameters of method executions
- Results of method invocations or other expressions

Aliasing from Heap and Stack Variables



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

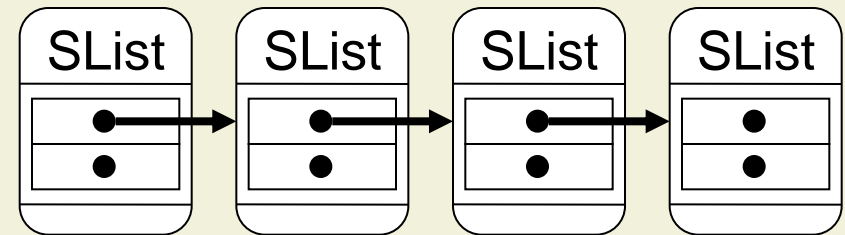
Aliasing from Heap and Stack Variables



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

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

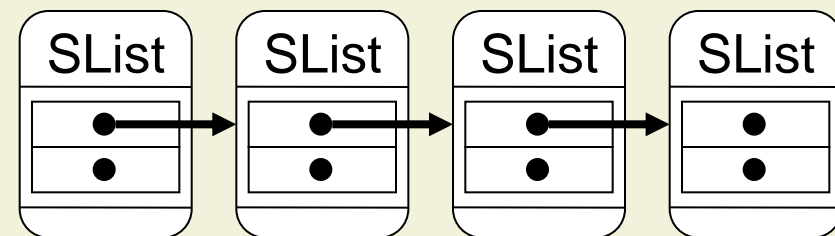
Intended Aliasing: Efficiency



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

Intended Aliasing: Efficiency

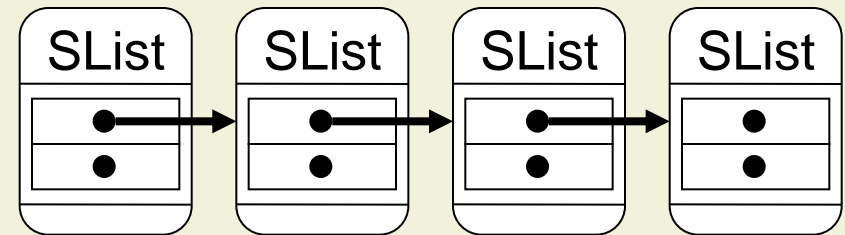
- In imperative programming, data structures are usually **not copied** when passed or modified
- Aliasing and **destructive updates** make imperative programming efficient



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

Intended Aliasing: Efficiency

- In imperative programming, data structures are usually **not copied** when passed or modified
- Aliasing and **destructive updates** make imperative programming efficient

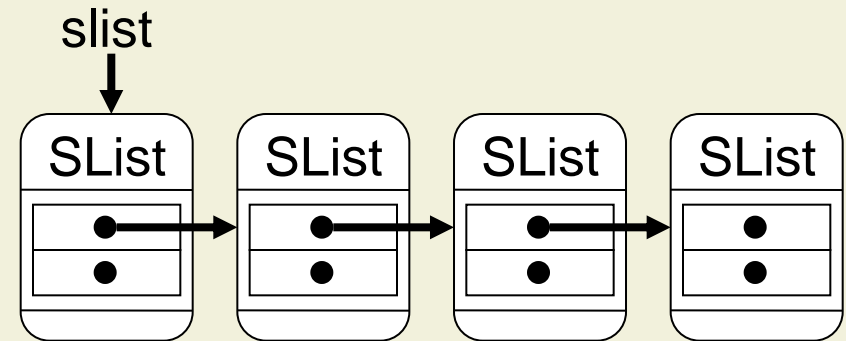


```
class SList {  
    SList next;  
    Object elem;  
    SList rest( ) { return next; }  
    void set( Object e ) { elem = e; }  
}
```

```
void foo( SList slist ) {  
    SList rest = slist.rest( );  
    rest.set( "Hello" ); }  
}
```

Intended Aliasing: Efficiency

- In imperative programming, data structures are usually **not copied** when passed or modified
- Aliasing and **destructive updates** make imperative programming efficient

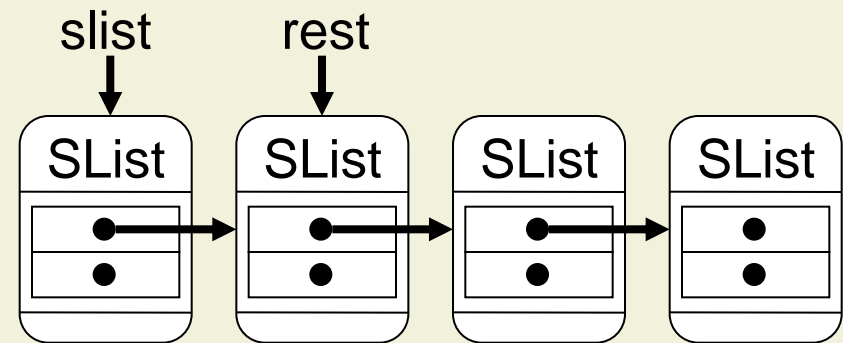


```
class SList {  
    SList next;  
    Object elem;  
    SList rest( ) { return next; }  
    void set( Object e ) { elem = e; }  
}
```

```
void foo( SList slist ) {  
    SList rest = slist.rest( );  
    rest.set( "Hello" );  
}
```

Intended Aliasing: Efficiency

- In imperative programming, data structures are usually **not copied** when passed or modified
- Aliasing and **destructive updates** make imperative programming efficient

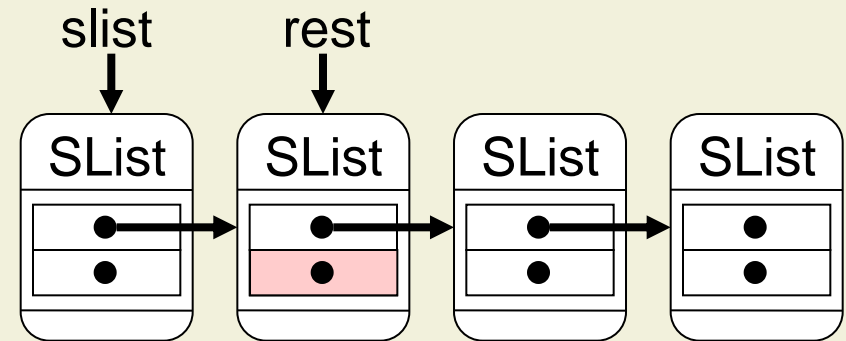


```
class SList {  
    SList next;  
    Object elem;  
    SList rest( ) { return next; }  
    void set( Object e ) { elem = e; }  
}
```

```
void foo( SList slist ) {  
    SList rest = slist.rest( );  
    rest.set( "Hello" ); }  
}
```

Intended Aliasing: Efficiency

- In imperative programming, data structures are usually **not copied** when passed or modified
- Aliasing and **destructive updates** make imperative programming efficient

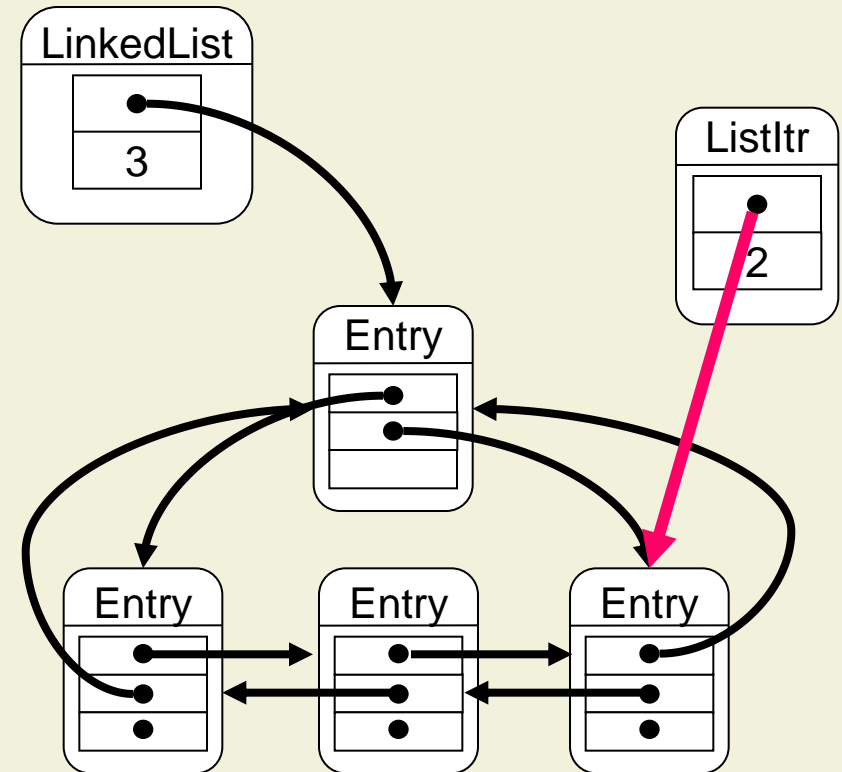


```
class SList {  
    SList next;  
    Object elem;  
    SList rest( ) { return next; }  
    void set( Object e ) { elem = e; }  
}
```

```
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

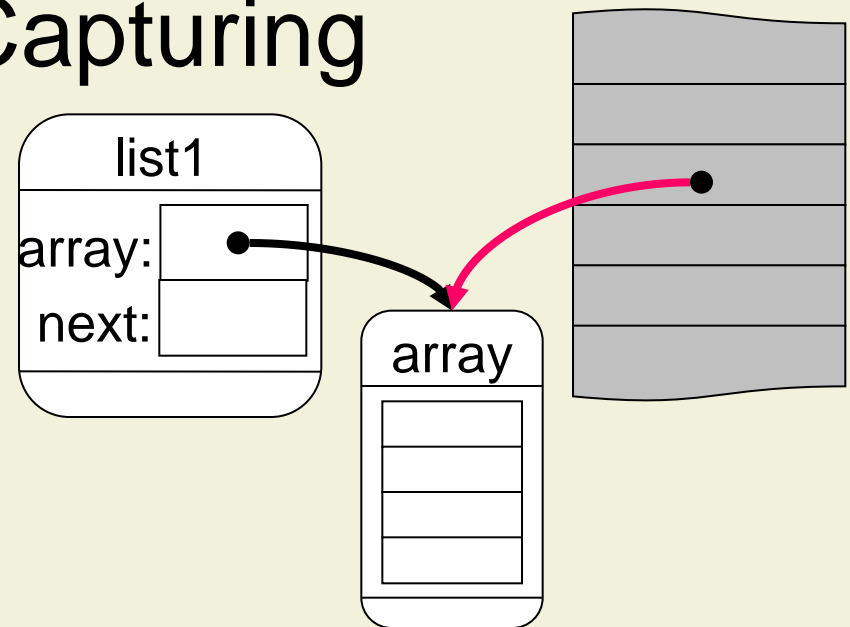
Unintended Aliasing: Capturing

- 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

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

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



```
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 structures **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

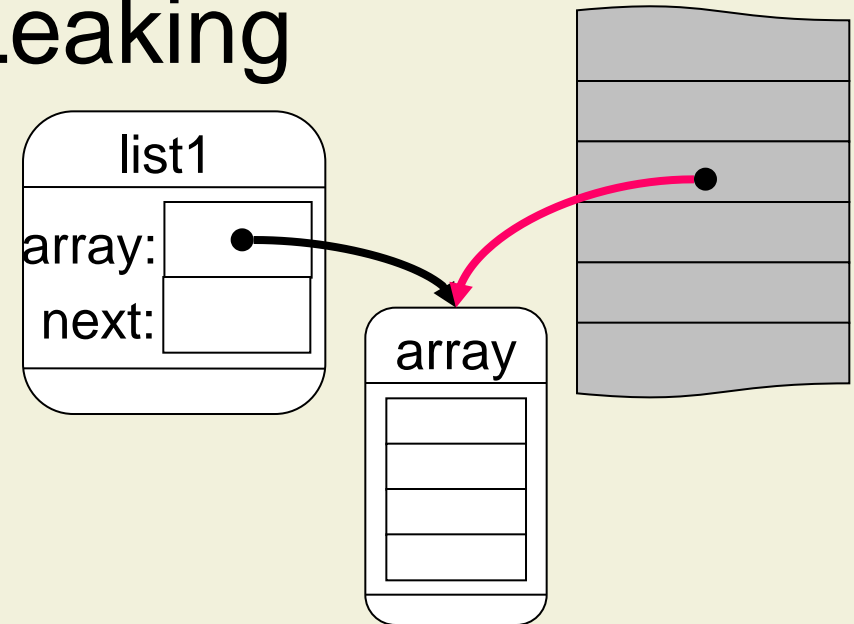
Unintended Aliasing: Leaking

- Leaking occurs when data structures **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

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

Unintended Aliasing: Leaking

- Leaking occurs when data structures **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



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

5. Object Structures and Aliasing

5.1 Aliasing

5.2 Problems of Aliasing

5.3 Unique References

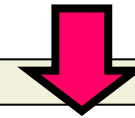
5.4 Readonly References

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]

Exchanging Implementations

```
class Coordinate {  
    private double x,y;  
    ...  
    public double distOrigin( )  
        { return Math.sqrt( x*x + y*y ); }  
}
```

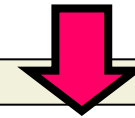


```
class Coordinate {  
    private double radius, angle;  
    ...  
    public double distOrigin( )  
        { return radius; }  
}
```

Exchanging Implementations

- Textbooks: information hiding enables safe changes of (hidden) implementation

```
class Coordinate {  
    private double x,y;  
    ...  
    public double distOrigin( )  
        { return Math.sqrt( x*x + y*y ); }  
}
```

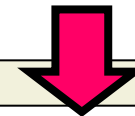


```
class Coordinate {  
    private double radius, angle;  
    ...  
    public double distOrigin( )  
        { return radius; }  
}
```

Exchanging Implementations

- Textbooks: information hiding enables safe changes of (hidden) implementation
- Observable behavior must be preserved

```
class Coordinate {  
    private double x,y;  
    ...  
    public double distOrigin( )  
        { return Math.sqrt( x*x + y*y ); }  
}
```

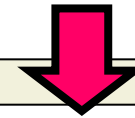


```
class Coordinate {  
    private double radius, angle;  
    ...  
    public double distOrigin( )  
        { return radius; }  
}
```

Exchanging Implementations

- Textbooks: information hiding enables safe changes of (hidden) implementation
- Observable behavior must be preserved
- But: beware of fragile baseclass problem

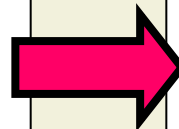
```
class Coordinate {  
    private double x,y;  
    ...  
    public double distOrigin( )  
        { return Math.sqrt( x*x + y*y ); }  
}
```



```
class Coordinate {  
    private double radius, angle;  
    ...  
    public double distOrigin( )  
        { return radius; }  
}
```

Exchanging Implementations (cont'd)

```
class List {  
  private int[ ] array;  
  private int next;  
  
  // requires ia != null  
  // ensures  $\forall i. 0 \leq i < \text{ia.length}:$   
  //           isElem( old( ia[ i ] ) )  
  public void setElems( int[ ] ia )  
    { array = ia; next = ia.length; }  
  
  ...  
}
```



```
class List {  
  private Entry head;  
  
  // requires ia != null  
  // ensures  $\forall i. 0 \leq i < \text{ia.length}:$   
  //           isElem( old( ia[ i ] ) )  
  public void setElems( int[ ] ia )  
    { ... /* create Entry for each  
          element */ }  
  
  ...  
}
```

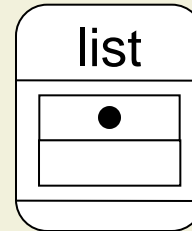
- Interface including contract remains unchanged

Exchanging Implementations (cont'd)

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

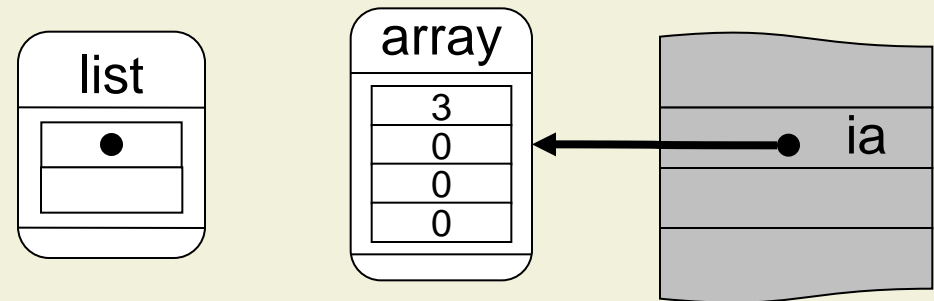
Exchanging Implementations (cont'd)

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



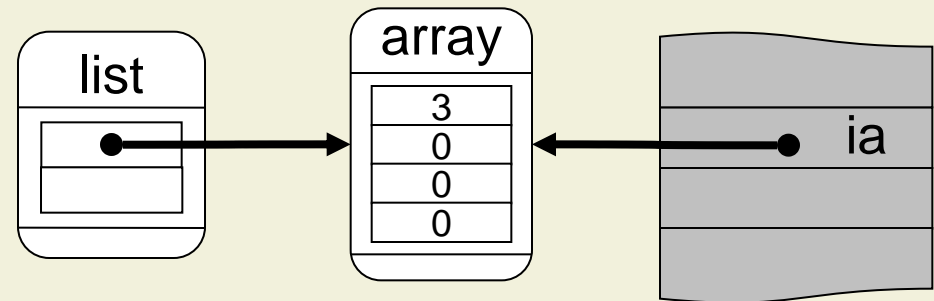
Exchanging Implementations (cont'd)

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



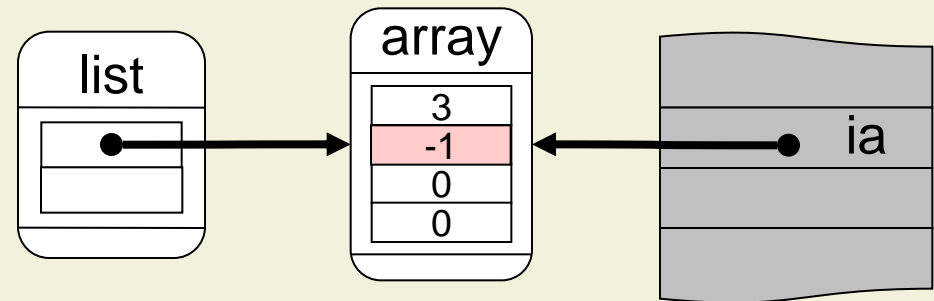
Exchanging Implementations (cont'd)

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



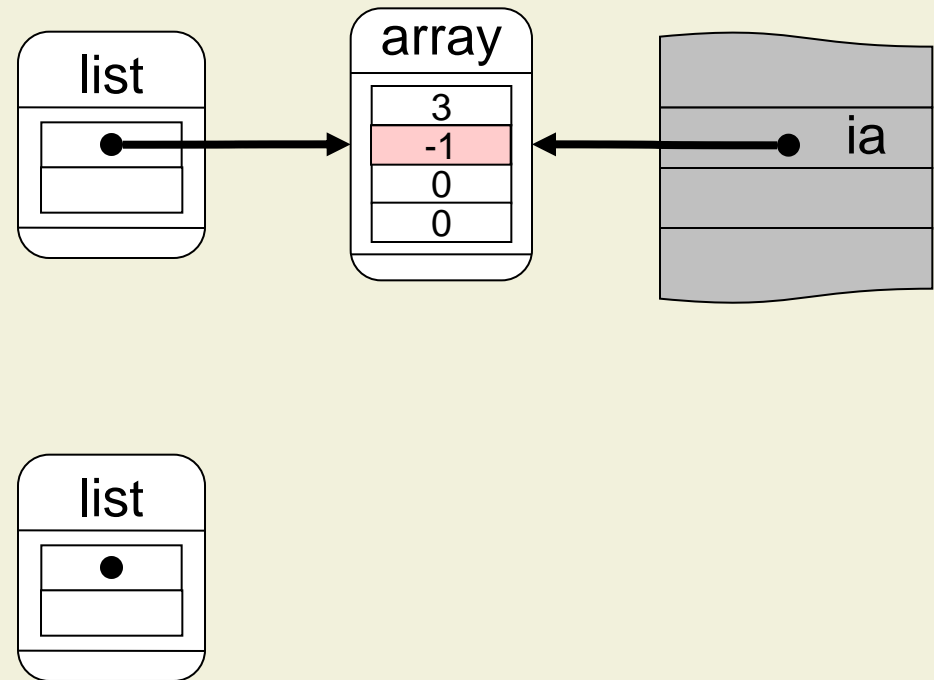
Exchanging Implementations (cont'd)

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



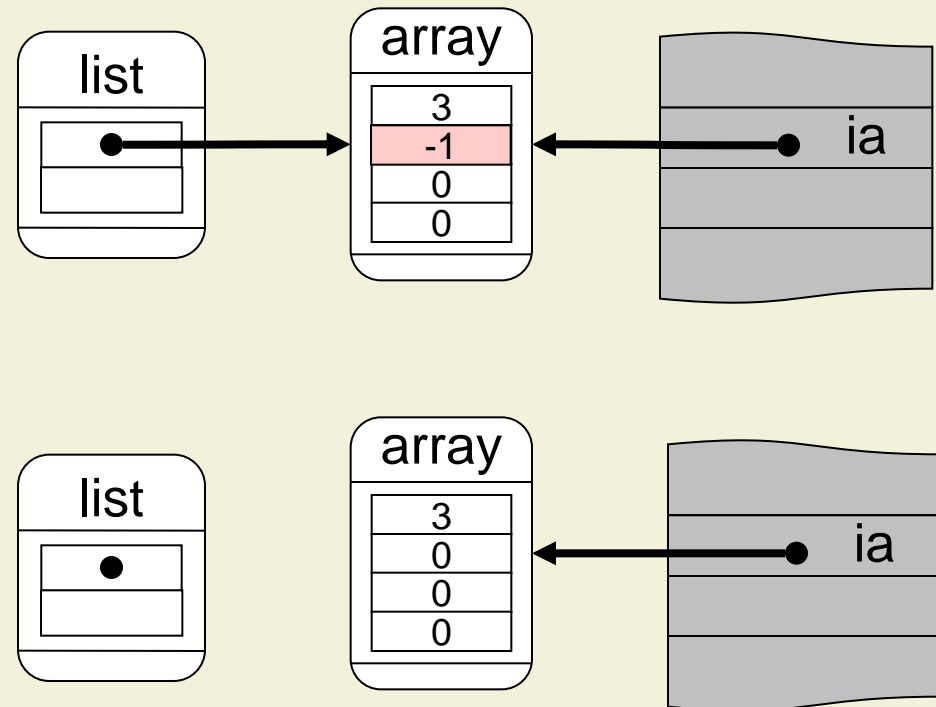
Exchanging Implementations (cont'd)

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



Exchanging Implementations (cont'd)

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

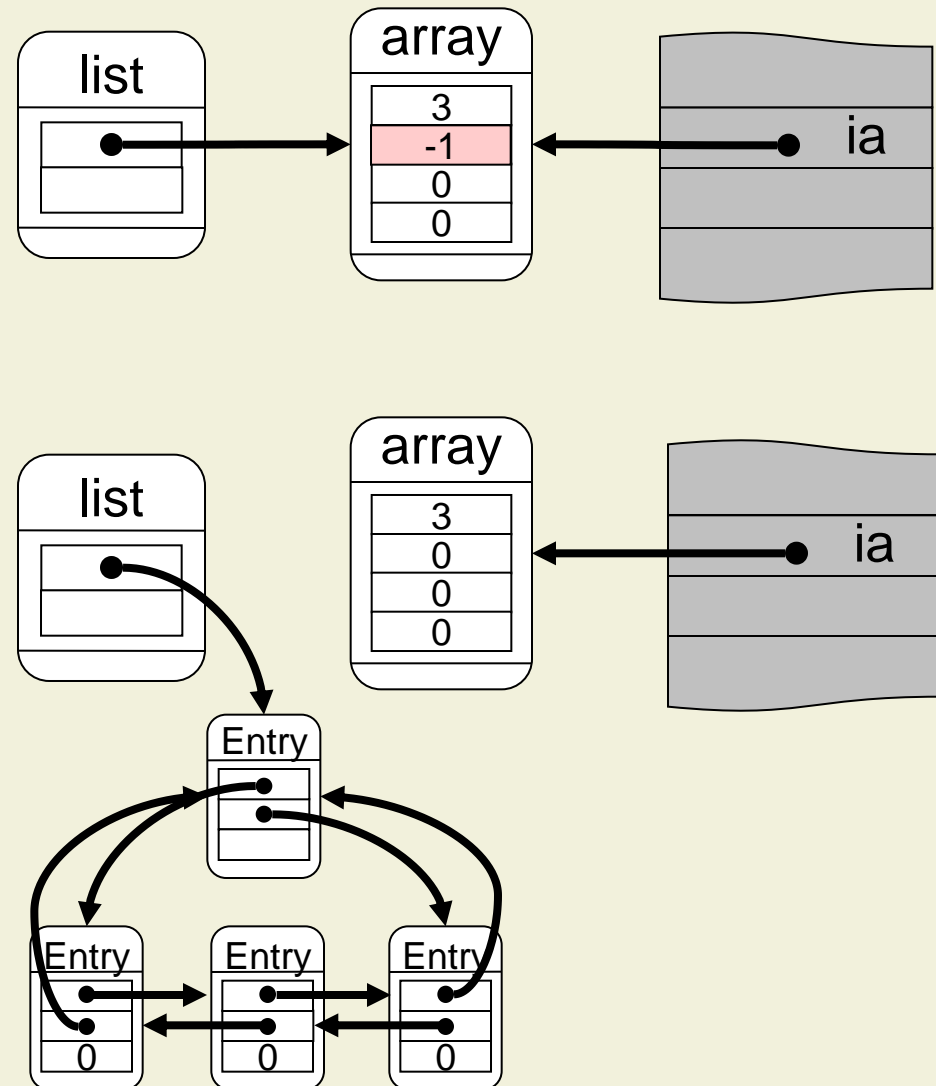


Exchanging Implementations (cont'd)

```

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

```

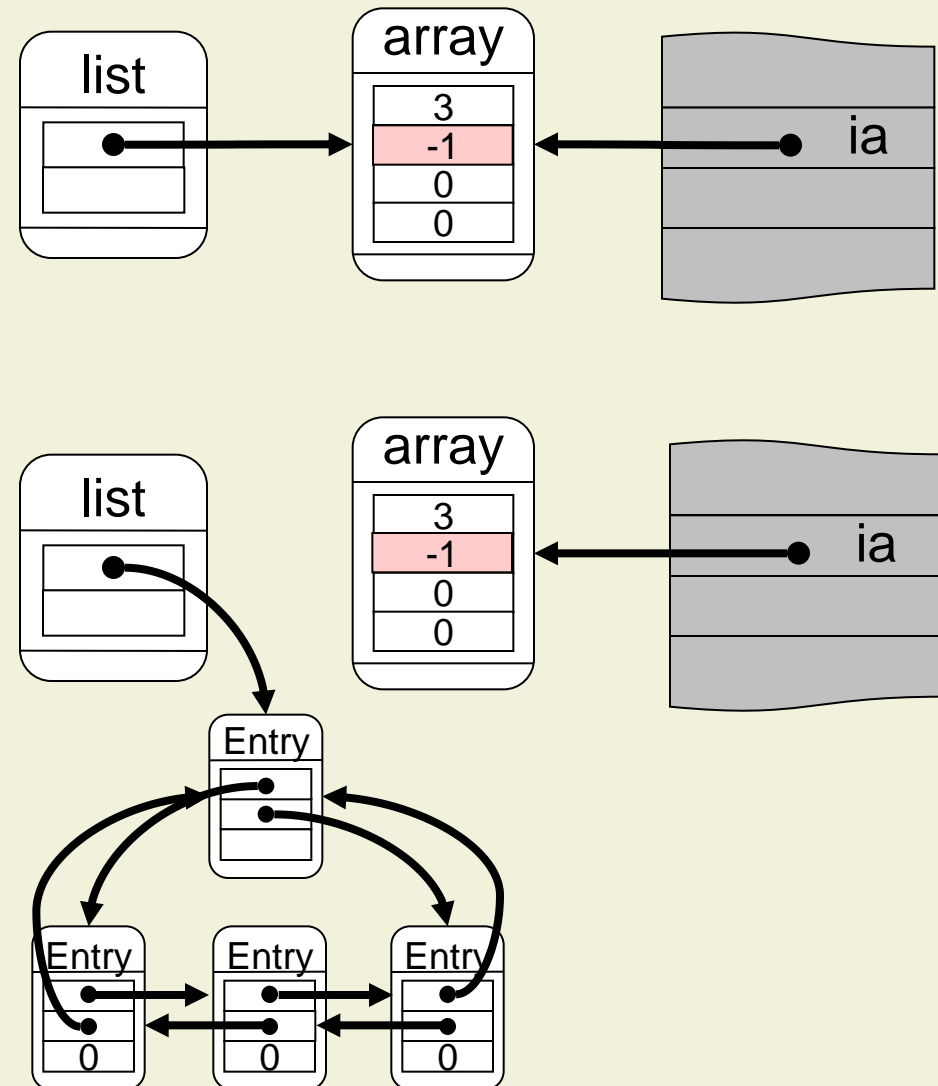


Exchanging Implementations (cont'd)

```

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

```



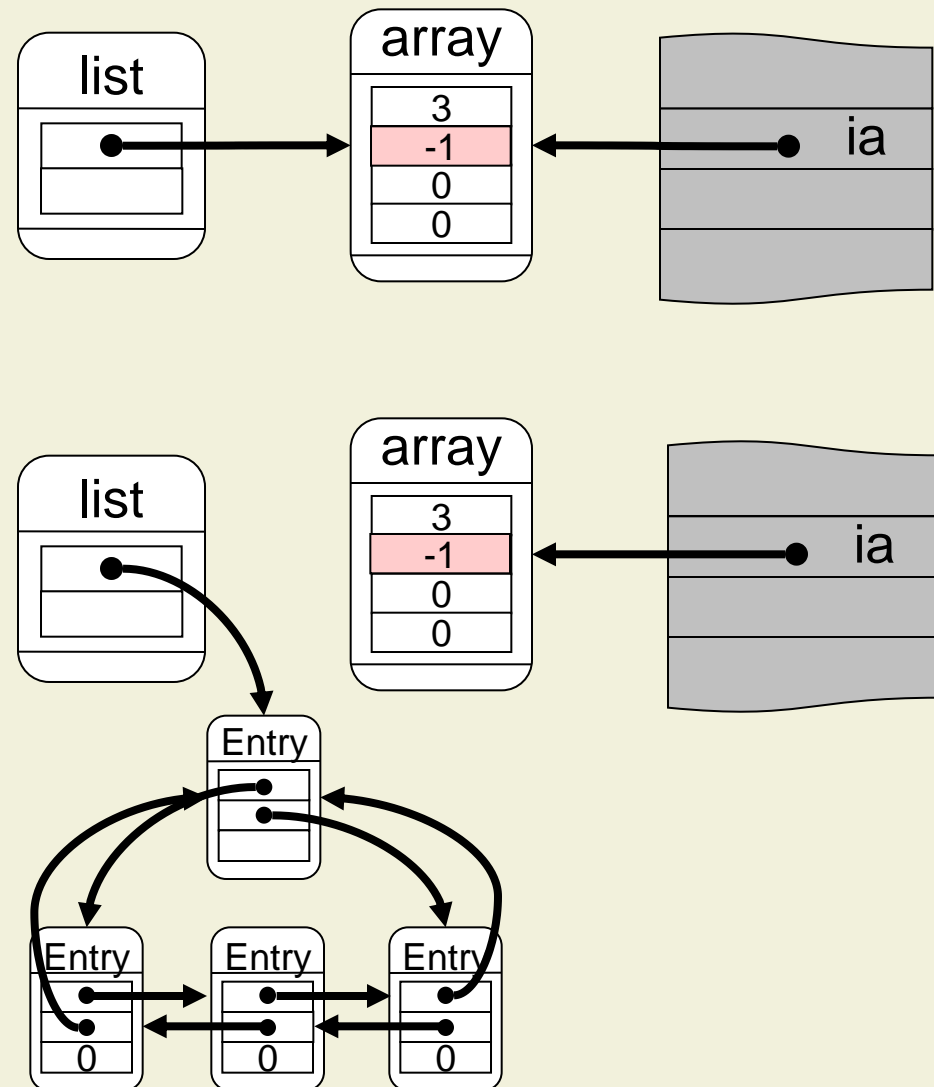
Exchanging Implementations (cont'd)

```

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

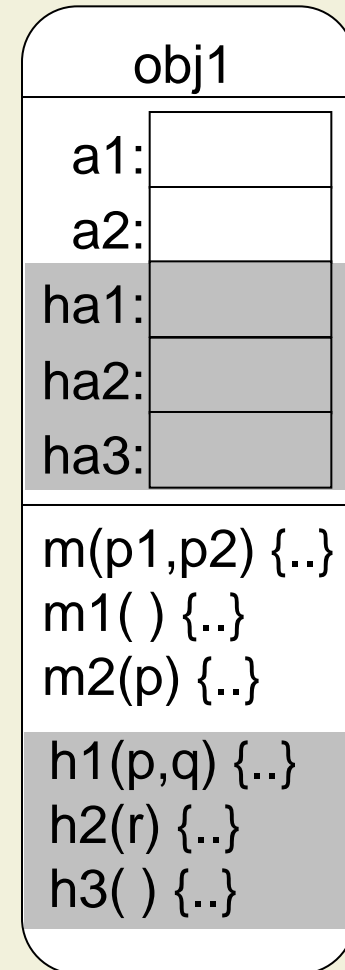
```

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



Consistency of Objects

- Consistency can include
 - Properties of one execution state (invariants)
 - Relations between execution states (history constraints)
- The internal representation of an object is encapsulated if it can be manipulated only by using the object's interfaces



Checks for Invariants: Textbook Solution

- Assume that all objects *o* are capsules
 - Only methods executed on *o* can modify *o*'s state
 - The invariant of object *o* refers only to the encapsulated fields of *o*

- For each invariant, we have to show
 - That all exported methods preserve the invariants
of the receiver object
 - That all constructors establish the invariants
of the new object

Object Consistency in Java

```
class Redundant {  
    private int a, b;  
    private Redundant next;  
    // invariant a == b  
  
    ...  
    public void set( int v ) { ... }  
  
    public void violate( ) {  
        // all invariants hold  
        next.a = next.b + 1;  
        // invariant of next does not hold  
    }  
}
```

Object Consistency in Java

- Declaring all fields **private** does not guarantee encapsulation on the level of individual objects

```
class Redundant {  
    private int a, b;  
    private Redundant next;  
    // invariant a == b  
    ...  
    public void set( int v ) { ... }  
  
    public void violate( ) {  
        // all invariants hold  
        next.a = next.b + 1;  
        // invariant of next does not hold  
    }  
}
```

Object Consistency in Java

- Declaring all fields **private** does not guarantee encapsulation on the level of individual objects
- Objects of same class can break the invariant

```
class Redundant {  
    private int a, b;  
    private Redundant next;  
    // invariant a == b  
    ...  
    public void set( int v ) { ... }  
  
    public void violate( ) {  
        // all invariants hold  
        next.a = next.b + 1;  
        // invariant of next does not hold  
    }  
}
```

Object Consistency in Java

- Declaring all fields **private** does not guarantee encapsulation on the level of individual objects
- Objects of same class can break the invariant
- Eiffel supports encapsulation on the object level
 - **feature { NONE }**

```
class Redundant {  
    private int a, b;  
    private Redundant next;  
    // invariant a == b  
    ...  
    public void set( int v ) { ... }  
  
    public void violate( ) {  
        // all invariants hold  
        next.a = next.b + 1;  
        // invariant of next does not hold  
    }  
}
```

Textbook Invariants for Java

- Assumption: The invariants of object *o* may refer only to **private fields** of *o*
- For each invariant, we have to show
 - That all exported methods **and constructors of class *T*** preserve the invariants **of all objects of *T***
 - That all constructors **in addition** establish the invariants of the new object

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

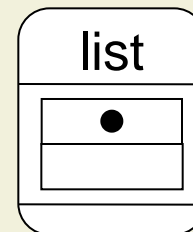
```
class ArrayList {  
    private int[ ] array;  
    private int next;  
  
    // invariant array != null    &&  
    //  0<=next<=array.length  &&  
    //   $\forall i. 0 \leq i < \text{next}: \text{array}[i] \geq 0$   
  
    public void add( int i )    { ... }  
    public void setElems( int[ ] ia )  
        { ... }  
  
    ...  
}
```

Consistency of Object Structures (cont'd)

```
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
}
```

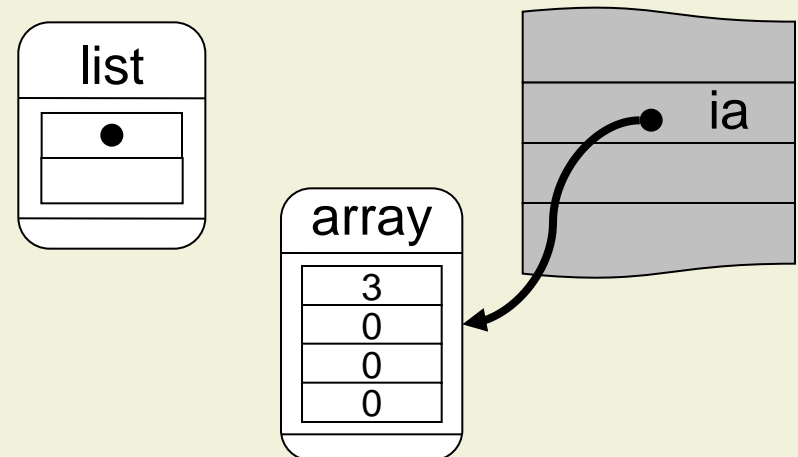
Consistency of Object Structures (cont'd)

```
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
}
```



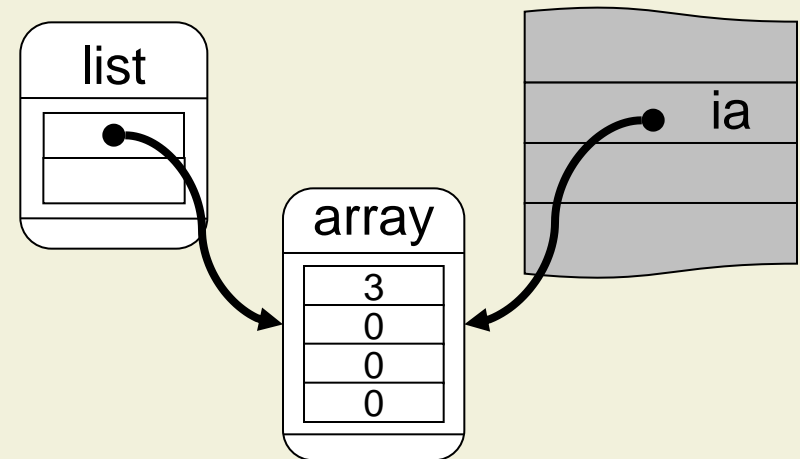
Consistency of Object Structures (cont'd)

```
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
}
```



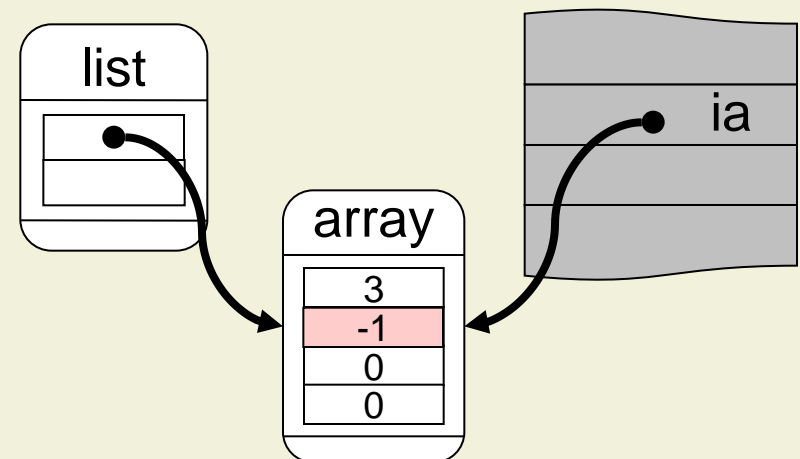
Consistency of Object Structures (cont'd)

```
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
}
```



Consistency of Object Structures (cont'd)

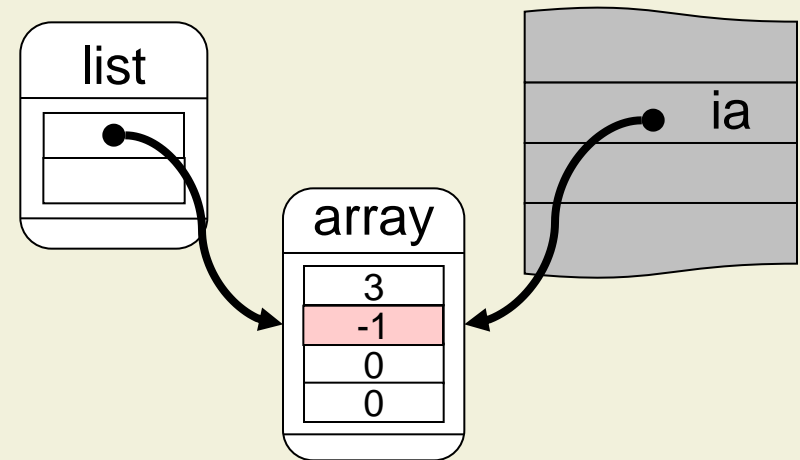
```
int foo( ArrayList list ) {    // invariant of list holds
    int[ ] ia = new int[ 3 ];
    list.setElems( ia );      // invariant of list holds
    ia[ 0 ] = -1;             // invariant of list violated
}
```



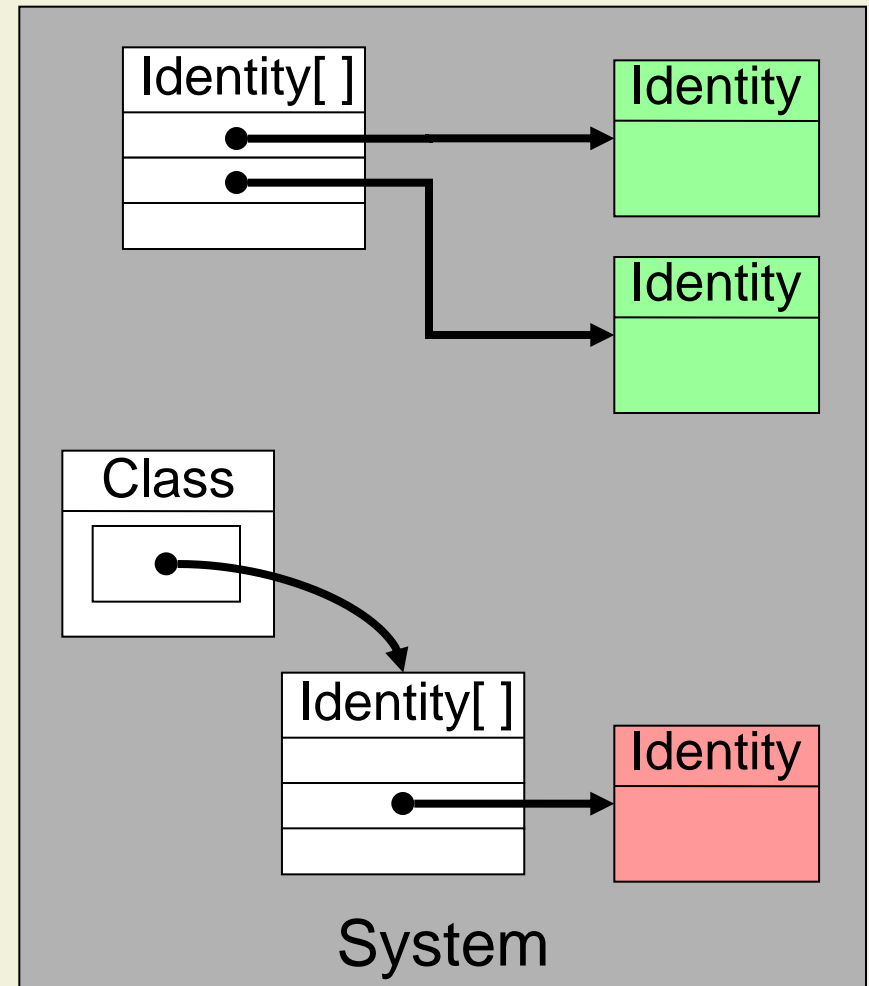
Consistency of Object Structures (cont'd)

```
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
}
```

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

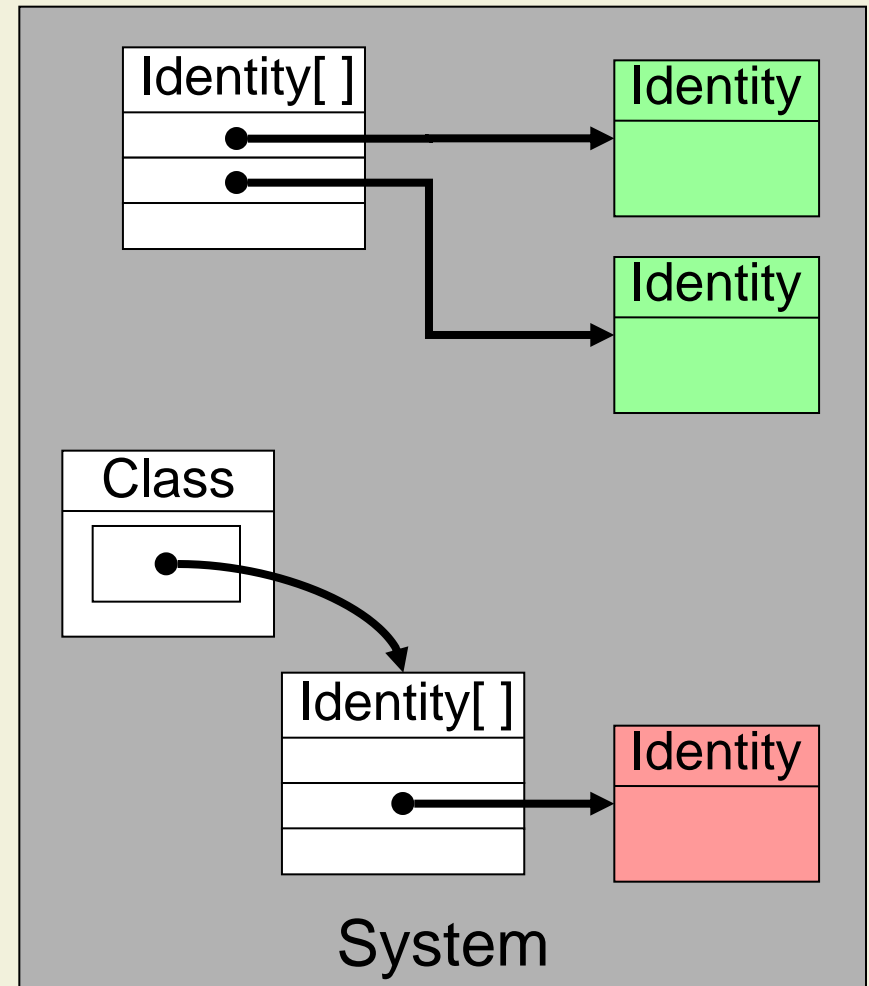


Security Breach in Java 1.1.1



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 */  
    }  
}
```



Security Breach in Java 1.1.1

```
class Malicious {
```

```
  void bad( ) {
```

```
    Identity[ ] s;
```

```
    Identity trusted = java.Security...;
```

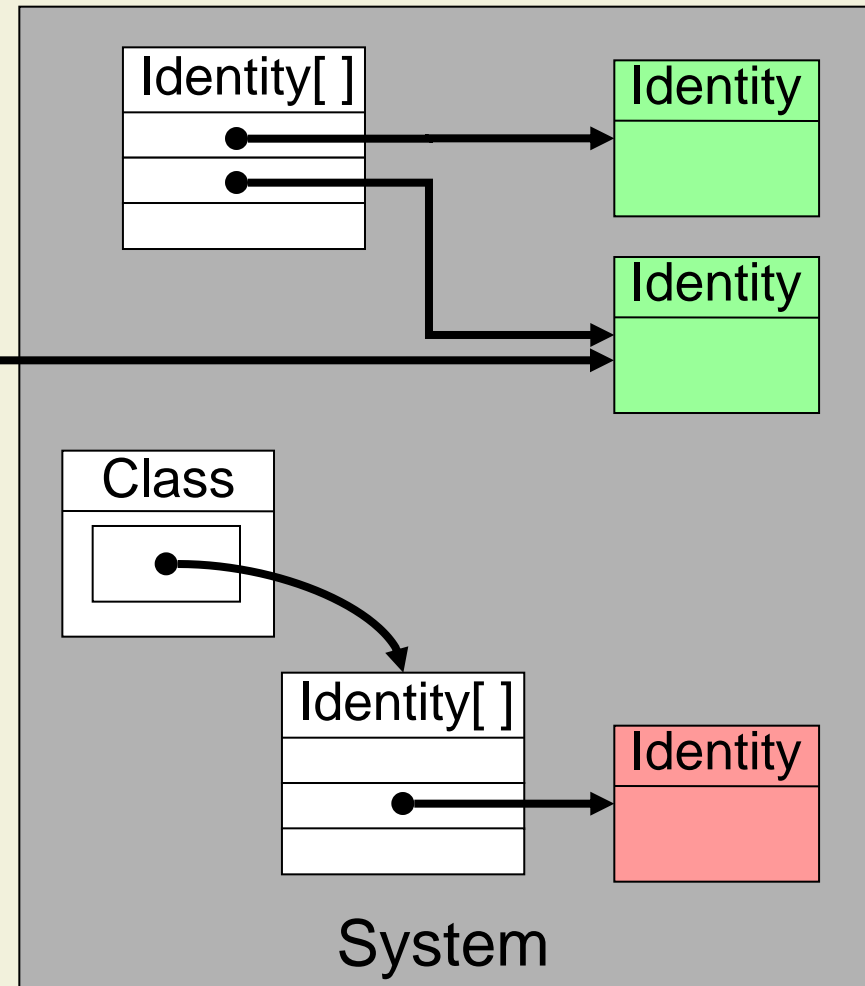
```
    s = Malicious.class.getSigners( );
```

```
    s[ 0 ] = trusted;
```

```
    /* abuse privilege */
```

```
  }
```

```
}
```



Security Breach in Java 1.1.1

```
class Malicious {
```

```
void bad( ) {
```

```
    Identity[ ] s;
```

```
    Identity trusted = java.Security...;
```

```
    s = Malicious.class.getSigners( );
```

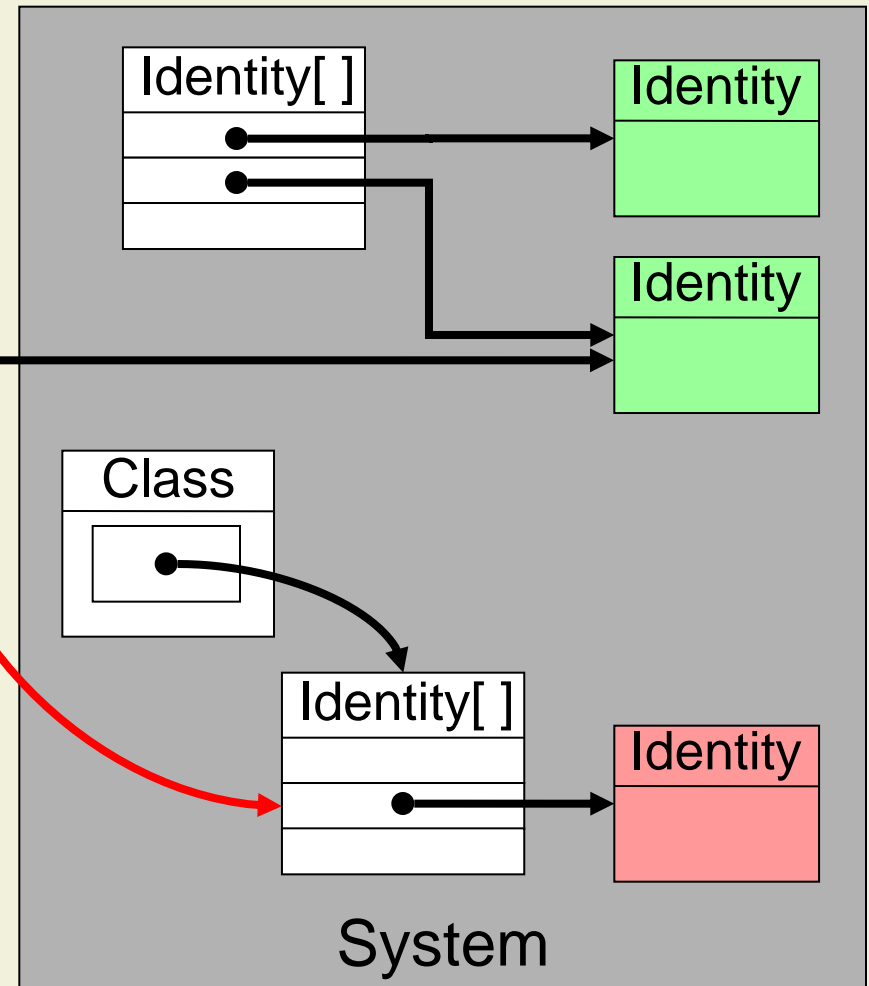
```
    s[ 0 ] = trusted;
```

```
    /* abuse privilege */
```

```
}
```

```
}
```

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



Security Breach in Java 1.1.1

```
class Malicious {
```

```
  void bad( ) {
```

```
    Identity[ ] s;
```

```
    Identity trusted = java.Security...;
```

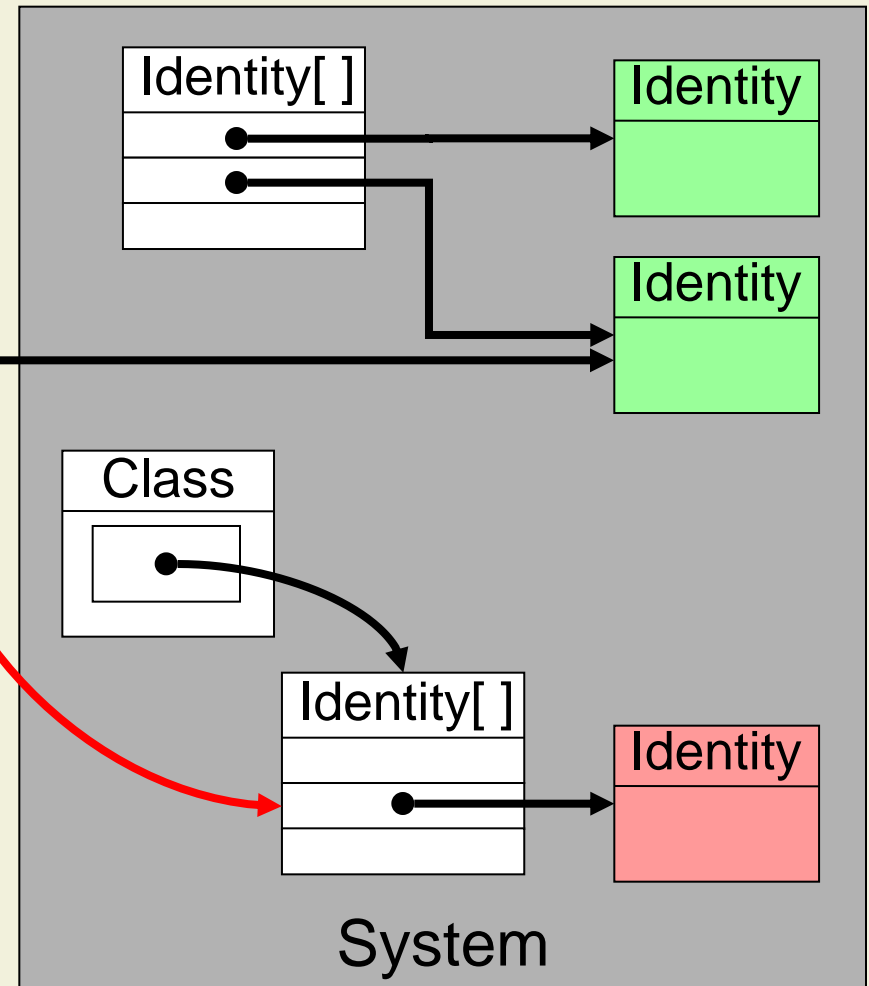
```
    s = Malicious.class.getSigners( );
```

```
    s[ 0 ] = trusted;
```

```
    /* abuse privilege */
```

```
  }
```

```
}
```



Security Breach in Java 1.1.1

```
class Malicious {
```

```
void bad( ) {
```

```
    Identity[ ] s;
```

```
    Identity trusted = java.Security...;
```

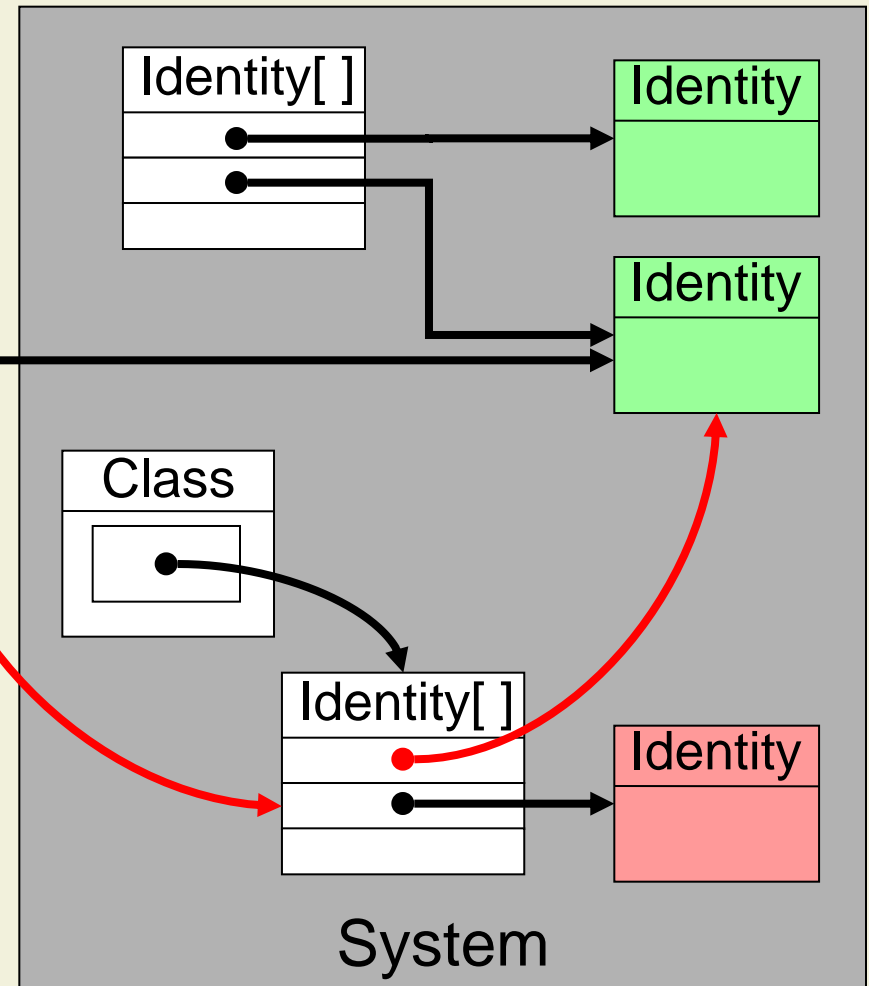
```
    s = Malicious.class.getSigners( );
```

```
    s[ 0 ] = trusted;
```

```
    /* abuse privilege */
```

```
}
```

```
}
```



Security Breach in Java 1.1.1

```
class Malicious {
```

```
void bad( ) {
```

```
    Identity[ ] s;
```

```
    Identity trusted = java.Security...;
```

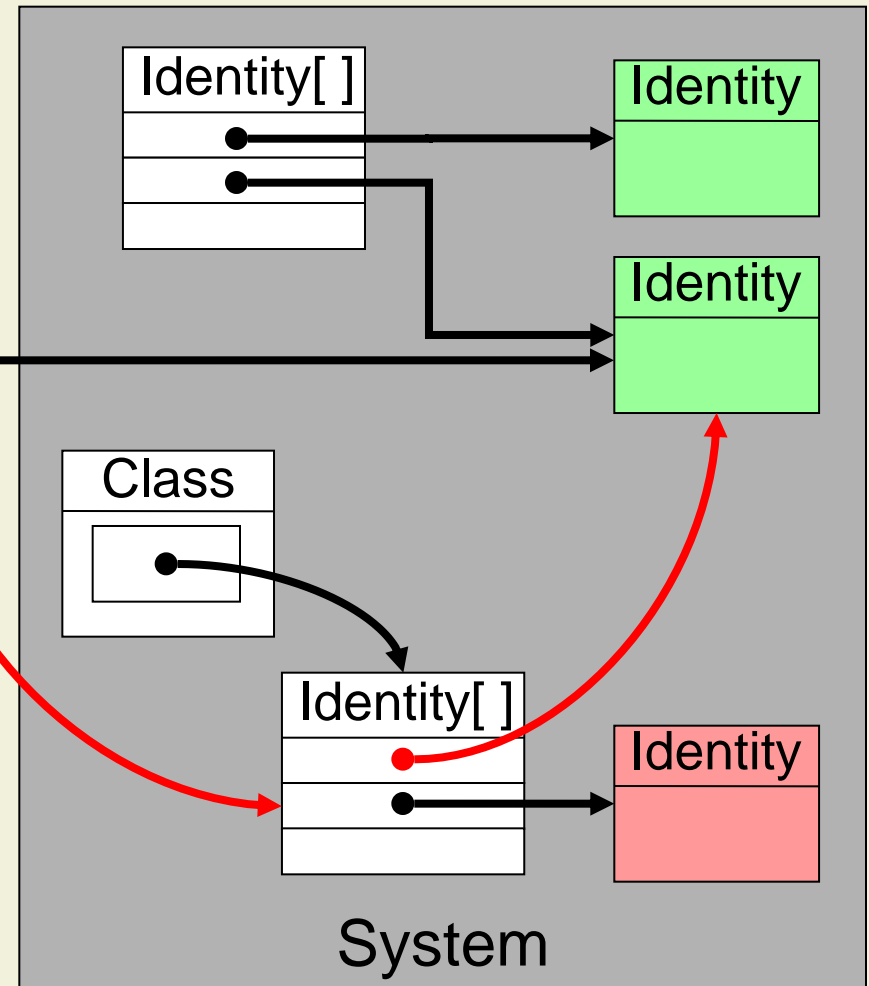
```
    s = Malicious.class.getSigners( );
```

```
    s[ 0 ] = trusted;
```

```
    /* abuse privilege */
```

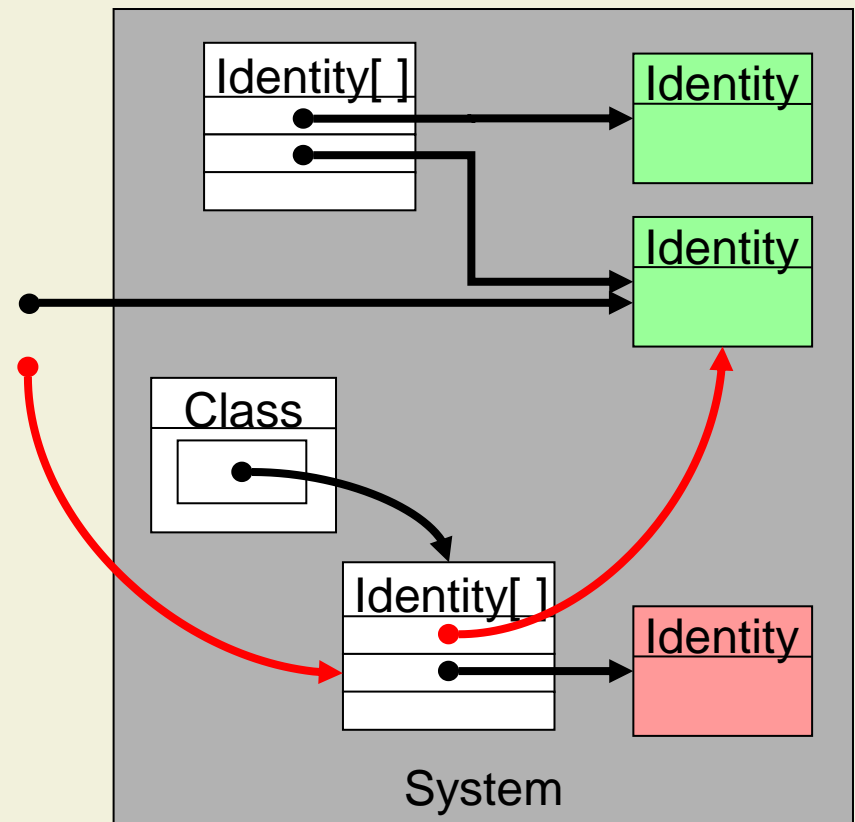
```
}
```

```
}
```



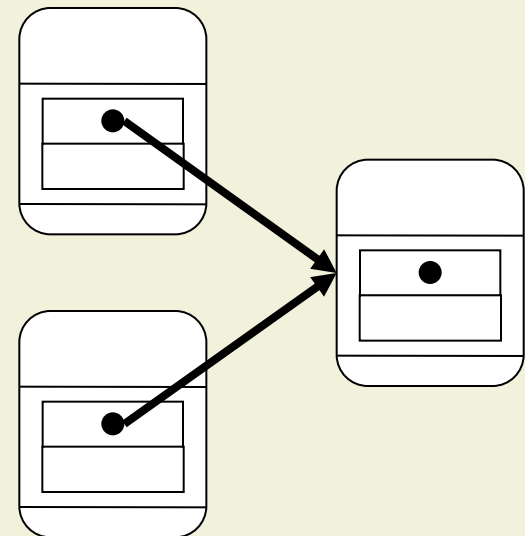
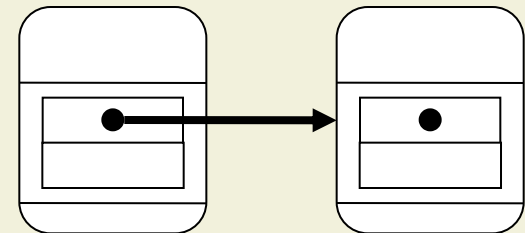
Problem Analysis

- 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
- Explicit memory management
 - Object may be freed only when the last reference disappears
- Optimizations
 - For instance, object inlining is not possible for aliased objects



5. Object Structures and Aliasing

5.1 Aliasing

5.2 Problems of Aliasing

5.3 Unique References

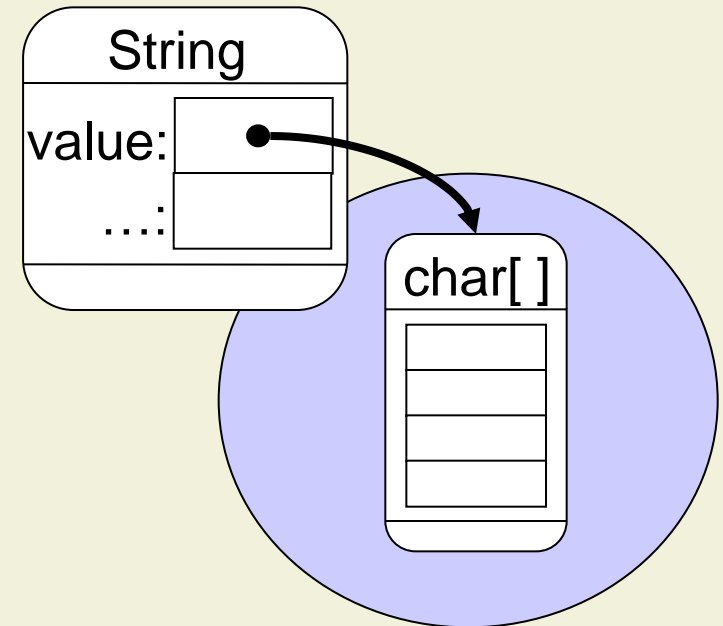
5.4 Readonly References

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
- Listltr is a **private inner class** of LinkedList
 - Interface Listliterator provides controlled access to Listltr-objects
 - Listltr-objects are passed out, but in a controlled fashion
 - Subclasses cannot manipulate or leak Listltr-objects
- **Subclassing is severely restricted**

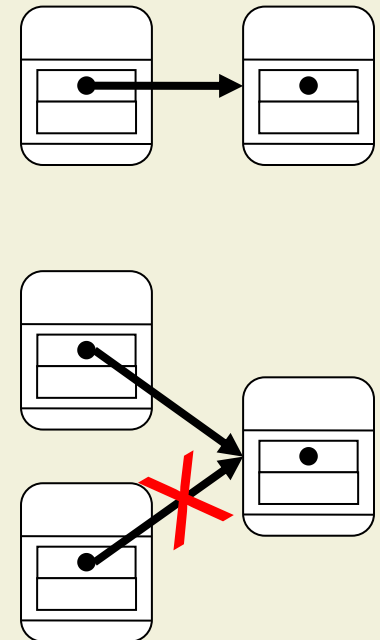
Alias Control in Java: String

- All **fields** are **private**
- References to internal character-array are not passed out
- **Subclassing is prohibited** (final)



Language Support: Unique References

- Preventing aliasing through a coding discipline is restrictive and error prone
- Language support for **unique references** facilitates alias control and can leverage non-aliasing guarantees to simplify memory management, thread synchronization, etc.
- A reference to an object is **unique** if it is the only reference to the object (the object is not aliased)



Unique Pointers in C++

- C++ offers a **library solution** (as templates)

```
class Address {  
    Address( string c ) { city = c; }  
  
    ...  
}
```

C++

```
class Person {  
    unique_ptr<Address> addr;  
  
public:  
    Person(string c) {  
        addr = make_unique<Address>( c );  
    }  
}
```

C++

Unique Pointers in C++

- C++ offers a **library solution** (as templates)

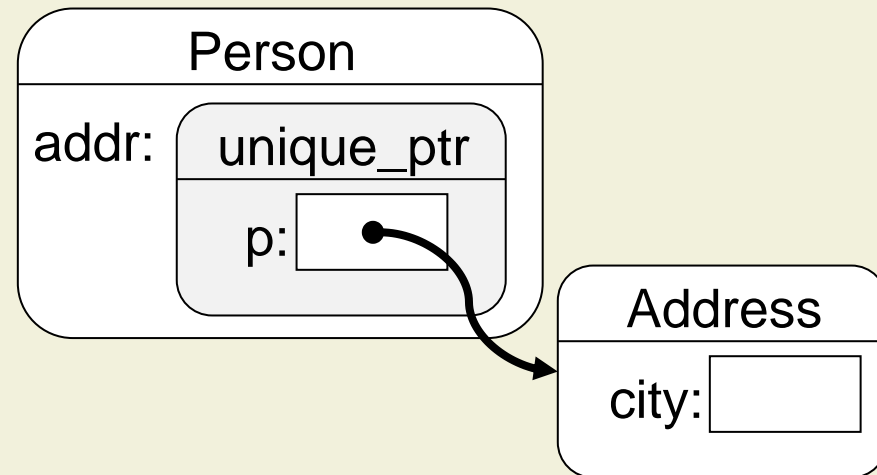
```
class Address {  
    Address( string c ) { city = c; }  
  
    ...  
}
```

C++

```
class Person {  
    unique_ptr<Address> addr;  
  
public:  
    Person(string c) {  
        addr = make_unique<Address>( c );  
    }  
}
```

C++

- Unique pointers are wrappers around the underlying pointer



Unique Pointers in C++

- C++ offers a **library solution** (as templates)

```
class Address {  
    Address( string c ) { city = c; }  
    ...  
}
```

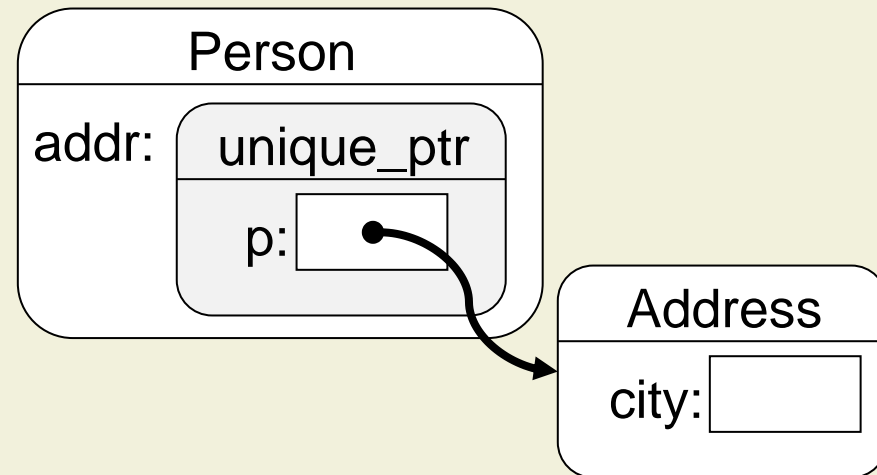
C++

```
class Person {  
    unique_ptr<Address> addr;  
  
public:  
    Person(string c) {  
        addr = make_unique<Address>( c );  
    }  
}
```

C++

Create Address object and wrap it in unique pointer

- Unique pointers are wrappers around the underlying pointer



Unique Pointers in C++

- C++ offers a **library solution** (as templates)

```
class Address {  
    Address( string c ) { city = c; }  
    ...  
}
```

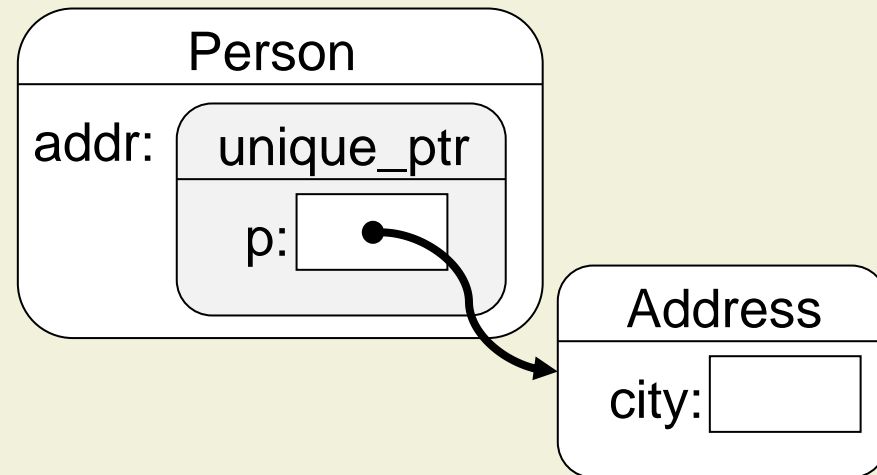
C++

```
class Person {  
    unique_ptr<Address> addr;  
  
public:  
    Person(string c) {  
        addr = make_unique<Address>( c );  
    }  
}
```

C++

Create Address object and wrap it in unique pointer

- Unique pointers are wrappers around the underlying pointer



- They express **object ownership**

Member Access Via Unique Pointers

- Method and field accesses on unique pointers use the familiar syntax
- `unique_ptr` redefines the access operator `->` to yield a pointer to the owned object

```
class Person {  
    unique_ptr<Address> addr;  
public:  
    string getCity() {  
        return addr->getCity( );  
    }  
}
```

C++

Protecting Uniqueness

- The pointer wrapped inside a `unique_ptr` struct is **intended** to be unique
- Direct **leaking or capturing is prevented** statically
- This is achieved by deleting the copy constructor and the assignment operator

```
class Person {  
    unique_ptr<Address> addr;  
}
```

C++

Protecting Uniqueness

- The pointer wrapped inside a `unique_ptr` struct is **intended** to be unique
- Direct **leaking or capturing is prevented** statically
- This is achieved by deleting the copy constructor and the assignment operator

```
class Person {  
    unique_ptr<Address> addr;  
}
```

C++

```
unique_ptr<Address> getAddr( ) {  
    return addr;  
}
```

C++

Protecting Uniqueness

- The pointer wrapped inside a `unique_ptr` struct is **intended** to be unique
- Direct **leaking or capturing is prevented** statically
- This is achieved by deleting the copy constructor and the assignment operator

```
class Person {  
    unique_ptr<Address> addr;  
}
```

C++

```
unique_ptr<Address> getAddr( ) {  
    return addr;  
}
```

C++

```
void setAddr( unique_ptr<Address> a ) {  
    addr = a;  
}
```

C++

Memory Management

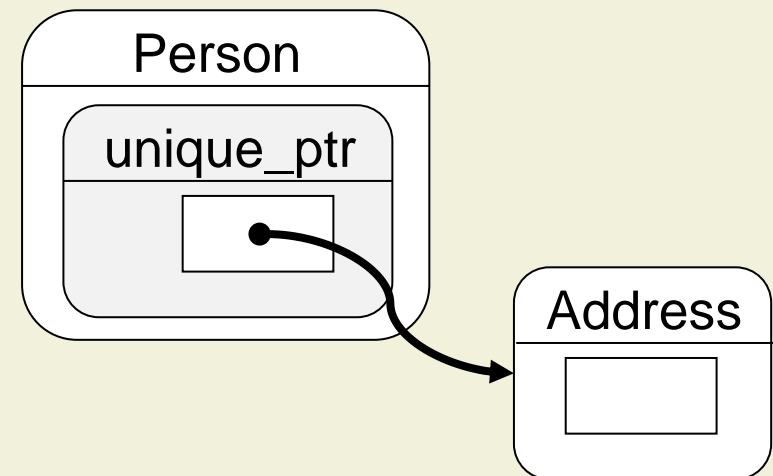
- C++ leverages uniqueness to **simplify memory management**
- When a unique pointer gets destroyed, it automatically de-allocates the underlying object

```
class Person {  
    unique_ptr<Address> addr;  
}
```

C++

```
Person* p = new Person( );  
delete p;
```

C++



Memory Management

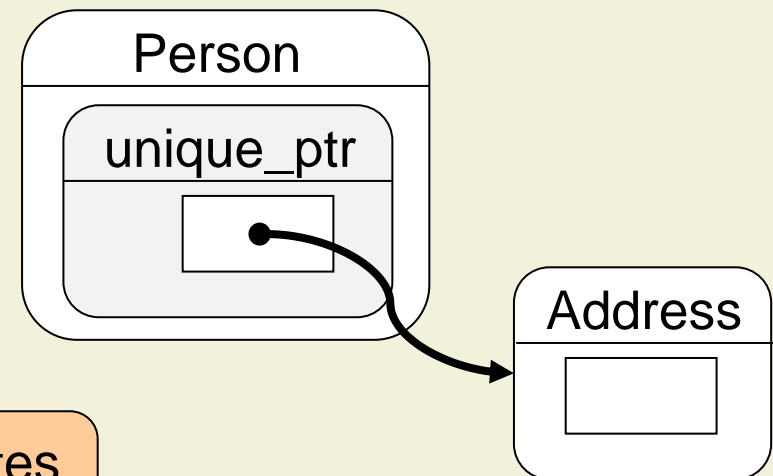
- C++ leverages uniqueness to **simplify memory management**
- When a unique pointer gets destroyed, it automatically de-allocates the underlying object

```
class Person {  
    unique_ptr<Address> addr;  
}
```

C++

```
Person* p = new Person( );  
delete p;
```

Automatically de-allocates
owned Address object



Ownership Transfer

- Object ownership can be transferred, for instance, to intentionally capture an object
- A **move assignment** transfers ownership

```
void setAddr( unique_ptr<Address> a ) {  
    addr = std::move( a );  
}
```

C++

Ownership Transfer

- Object ownership can be transferred, for instance, to intentionally capture an object
- A **move assignment** transfers ownership

```
void setAddr( unique_ptr<Address> a ) {  
    addr = std::move( a );  
}
```

C++

- To preserve uniqueness, a move assignment sets the pointer of the origin to null (**destructive read**)

Temporary Ownership Transfer

- Performing operations on owned objects requires a temporary ownership transfer

```
int compare( unique_ptr<Address> a, unique_ptr<Address> b ) {  
    return a->getCity( ).compare( b->getCity( ) );  
}
```

C++

```
unique_ptr<Address> a = make_unique<Address>( "Zurich" );  
unique_ptr<Address> b = make_unique<Address>( "Berne" );  
  
int x = compare( std::move(a), std::move(b) );
```

C++

Temporary Ownership Transfer

- Performing operations on owned objects requires a temporary ownership transfer

```
int compare( unique_ptr<Address> a, unique_ptr<Address> b ) {  
    return a->getCity( ).compare( b->getCity( ) );  
}
```

C++

```
unique_ptr<Address> a = make_unique<Address>( "Zurich" );  
unique_ptr<Address> b = make_unique<Address>( "Berne" );  
  
int x = compare( std::move(a), std::move(b) );  
int y = compare( std::move(a), std::move(b) );
```

C++

Temporary Ownership Transfer

- Performing operations on owned objects requires a temporary ownership transfer

```
int compare( unique_ptr<Address> a, unique_ptr<Address> b ) {  
    return a->getCity( ).compare( b->getCity( ) );  
}
```

C++

```
unique_ptr<Address> a = make_unique<Address>( "Zurich" );  
unique_ptr<Address> b = make_unique<Address>( "Berne" );
```

```
int x = compare( std::move(a), std::move(b) );  
int y = compare( std::move(a), std::move(b) );
```

After the first call,
a and b contain
null-pointers
(destructive read)

Temporary Ownership Transfer

- Performing operations on owned objects requires a temporary ownership transfer

```
int compare( unique_ptr<Address> a, unique_ptr<Address> b ) {  
    return a->getCity( ).compare( b->getCity( ) );  
}
```

C++

```
unique_ptr<Address> a = make_unique<Address>( "Zurich" );  
unique_ptr<Address> b = make_unique<Address>( "Berne" );
```

```
int x = compare( std::move(a), std::move(b) );  
int y = compare( std::move(a), std::move(b) );
```

Null-pointer
dereferencing

After the first call,
a and b contain
null-pointers
(destructive read)

Temporary Ownership Transfer (cont'd)

- To use parameter objects after call, ownership has to be transferred back to the caller

```
Triple compare( unique_ptr<Address> a, unique_ptr<Address> b ) {  
    Triple r; // int res; unique_ptr<Address> a, b;  
    r.res = a->getCity().compare( b->getCity() );  
    r.a = std::move( a ); r.b = std::move( b );  
    return r;  
}
```

C++

```
unique_ptr<Address> a = make_unique<Address>( "Zurich" );  
unique_ptr<Address> b = make_unique<Address>( "Berne" );  
  
Triple r = compare( std::move(a), std::move(b) );  
r = compare( std::move(r.a), std::move(r.b) );
```

C++

Parameter Passing Without Transfer

- To avoid transferring ownership back and forth, one can pass a pointer (or reference) to the unique pointer

```
int compare( unique_ptr<Address>* a, unique_ptr<Address>* b ) {  
    return (*a)->getCity( ).compare( (*b)->getCity( ) );  
}
```

C++

```
unique_ptr<Address> a = make_unique<Address>( "Zurich" );  
unique_ptr<Address> b = make_unique<Address>( "Berne" );  
  
int x = compare( &a, &b );  
x = compare( &a, &b );
```

C++

Passing the Owned Object

- (Library) methods typically expect regular pointers

```
int compare( Address* a, Address* b ) {  
    return a->getCity( ).compare( b->getCity( ) );  
}
```

C++

- To use such methods, one has to call the get method of a unique pointer to obtain a regular pointer to the owned object

```
unique_ptr<Address> a = make_unique<Address>( "Zurich" );  
unique_ptr<Address> b = make_unique<Address>( "Berne" );  
  
int x = compare( a.get(), b.get() );
```

C++

No Uniqueness Guarantee

- `get()` creates alias to owned object
- (Library) methods are not aware that their receiver is supposed to be unique and might create an alias
- Using `get()` and `->` is unavoidable

```
unique_ptr<Address> a;  
a = make_unique<Address>("Zurich");  
Address* alias = a.get();  
alias->setCity("Hagen");  
cout << a->getCity();
```

C++

```
static Address* cache;  
class Address {  
    string getCity() { cache = this; return city; }  
}
```

C++

```
class Person {  
    unique_ptr<Address> addr;  
    string getCity() { return addr->getCity( ); }  
}
```

C++

Memory Management Revisited

- Destroying a unique pointer de-allocates the underlying object, **even if aliases exist**
- Subsequent accesses lead to run-time errors

```
Address* returnAlias() {  
    unique_ptr<Address> a;  
    a = make_unique<Address>("Zurich");  
    return a.get();  
}
```

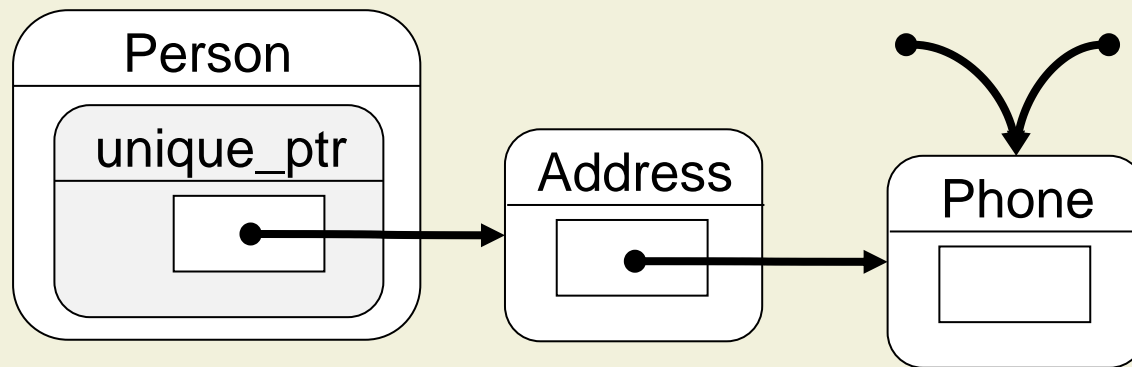
C++

```
Address* a = returnAlias();  
cout << a->getCity();
```

C++

Shallow Ownership

- Unique pointers express ownership of the referenced object, **but not its sub-objects**



- Deep ownership of an entire object structure needs to be implemented explicitly by making all relevant pointers unique
 - Not possible for library classes

Unique Pointers in C++: Discussion

Pros

Cons

Unique Pointers in C++: Discussion

Pros

- Unique pointers express design intent

Cons

Unique Pointers in C++: Discussion

Pros

- Unique pointers express design intent
- Direct leaking or capturing is prevented statically

Cons

Unique Pointers in C++: Discussion

Pros

- Unique pointers express design intent
- Direct leaking or capturing is prevented statically
- Memory management is simplified (but requires care)

Cons

Unique Pointers in C++: Discussion

Pros

- Unique pointers express design intent
- Direct leaking or capturing is prevented statically
- Memory management is simplified (but requires care)

Cons

- No static guarantees

Unique Pointers in C++: Discussion

Pros

- Unique pointers express design intent
- Direct leaking or capturing is prevented statically
- Memory management is simplified (but requires care)

Cons

- No static guarantees
- Not effective in solving the problems caused by aliasing

Unique Pointers in C++: Discussion

Pros

- Unique pointers express design intent
- Direct leaking or capturing is prevented statically
- Memory management is simplified (but requires care)

Cons

- No static guarantees
- Not effective in solving the problems caused by aliasing
- Parameter passing is awkward

Unique Pointers in C++: Discussion

Pros

- Unique pointers express design intent
- Direct leaking or capturing is prevented statically
- Memory management is simplified (but requires care)

Cons

- No static guarantees
- Not effective in solving the problems caused by aliasing
- Parameter passing is awkward
- Destructive reads are error prone

Ownership in Rust

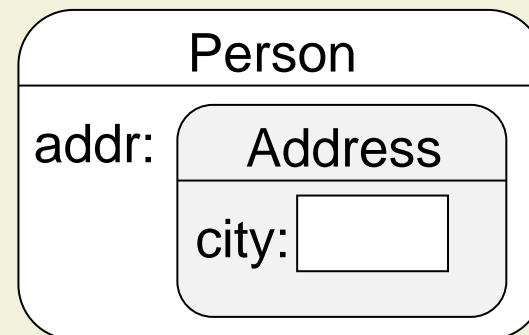
```
struct Address {  
  city: String,  
}
```

Rust

```
struct Person {  
  addr: Address,  
}  
impl Person {  
  fn new(c: String) -> Self {  
    let a = Address { city: c };  
    Self{ addr: a }  
  }  
}
```

Rust

- Rust supports ownership via its type system
- An expression representing a memory location (a place) of a type T owns the value in that location
- Ownership is deep



Ownership in Rust (With Pointers)

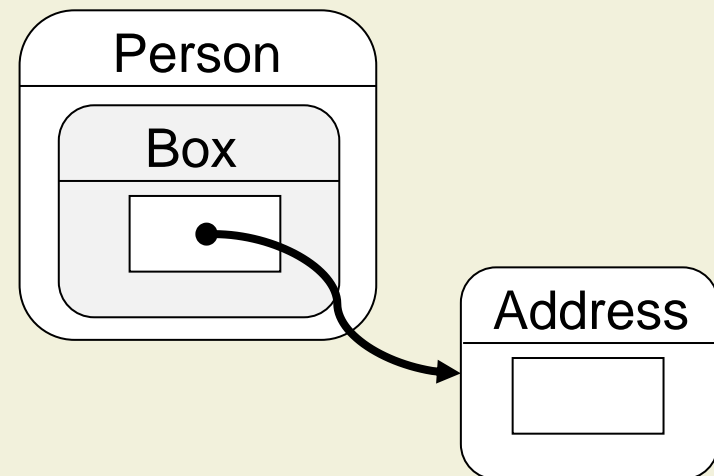
```
struct Address {  
  city: String,  
}
```

Rust

```
struct Person {  
  addr: Box<Address>,  
}  
impl Person {  
  fn new(c: String) -> Self {  
    let a = Address { city: c };  
    Self{ addr: Box::new( a ) }  
  }  
}
```

Rust

- Type Box provides **owning pointers** to heap-allocated values



Ownership Transfer

- When reading from a place, ownership of the owned value is transferred
- Every assignment is a **move assignment**

```
fn createPerson(a: Address) -> Person {  
    Person { addr: a }  
}
```

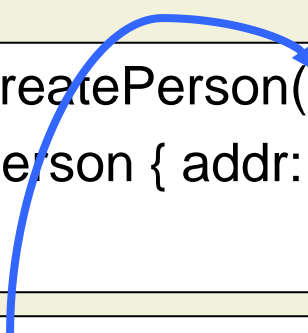
Rust

```
let a = Address { city: String::from("Zurich") };  
let p = createPerson( a );  
let c = p.getCity();
```

Rust

Ownership Transfer

- When reading from a place, ownership of the owned value is transferred
- Every assignment is a **move assignment**



```
fn createPerson(a: Address) -> Person {  
    Person { addr: a }  
}
```

Rust

```
let a = Address { city: String::from("Zurich") };  
let p = createPerson( a );  
let c = p.getCity();
```

Rust

Ownership Transfer

- When reading from a place, ownership of the owned value is transferred
- Every assignment is a **move assignment**

```
fn createPerson(a: Address) -> Person {  
    Person { addr: a }  
}
```

Rust

```
let a = Address { city: String::from("Zurich") };  
let p = createPerson( a );  
let c = p.getCity();
```

Rust

Ownership Transfer (cont'd)

- When a value is moved out of a place, the place becomes **unusable** until a new value is assigned

```
fn createPerson(a: Address) -> Person {  
    Person { addr: a }  
}
```

Rust

```
let a = Address { city: String::from("Zurich") };  
let p = createPerson( a );
```

Rust

Ownership Transfer (cont'd)

- When a value is moved out of a place, the place becomes **unusable** until a new value is assigned

```
fn createPerson(a: Address) -> Person {  
    Person { addr: a }  
}
```

Rust

```
let a = Address { city: String::from("Zurich") };  
let p = createPerson( a );  
let q = createPerson( a );
```

Rust

Ownership Transfer (cont'd)

- When a value is moved out of a place, the place becomes **unusable** until a new value is assigned

```
fn createPerson(a: Address) -> Person {  
    Person { addr: a }  
}
```

Rust

```
let a = Address { city: String::from("Zurich") };  
let p = createPerson( a );  
let q = createPerson( a );
```

Compile time error:
use of a moved value

Ownership Transfer (cont'd)

- When a value is moved out of a place, the place becomes **unusable** until a new value is assigned

```
fn createPerson(a: Address) -> Person {  
    Person { addr: a }  
}
```

Rust

```
let a = Address { city: String::from("Zurich") };  
let p = createPerson( a );  
let q = createPerson( a );
```

Compile time error:
use of a moved value

- Statically safe (in contrast to C++'s destructive read)
 - Whenever a place can be used, it contains a proper (non-null) value

Borrowing

- Like in C++, operations could be invoked by transferring ownership to the operation and back
- Borrowing provides temporary access to a value through a reference, without transferring ownership

Borrowing

- Like in C++, operations could be invoked by transferring ownership to the operation and back
- Borrowing provides temporary access to a value through a reference, without transferring ownership

```
fn compare( a: &mut Address, b: &mut Address ) -> Ordering {  
    a.getCity().cmp( b.getCity() )  
}
```

Rust

Borrowing

- Like in C++, operations could be invoked by transferring ownership to the operation and back
- Borrowing provides temporary access to a value through a reference, without transferring ownership

```
fn compare( a: &mut Address, b: &mut Address ) -> Ordering {  
    a.getCity().cmp( b.getCity() )  
}
```

Reference type

Rust

Borrowing

- Like in C++, operations could be invoked by transferring ownership to the operation and back
- Borrowing provides temporary access to a value through a reference, without transferring ownership

```
fn compare( a: &mut Address, b: &mut Address ) -> Ordering {  
    a.getCity().cmp( b.getCity() )  
}
```

Reference type

Rust

```
let mut a1 = Address { city: String::from("Zurich") };  
let mut a2 = Address { city: String::from("Hagen") };  
let o = compare( &mut a1, &mut a2 );  
let c = a1.getCity();
```

Rust

Borrowing

- Like in C++, operations could be invoked by transferring ownership to the operation and back
- Borrowing provides temporary access to a value through a reference, without transferring ownership

```
fn compare( a: &mut Address, b: &mut Address ) -> Ordering {  
    a.getCity().cmp( b.getCity() )  
}
```

Reference type

Rust

```
let mut a1 = Address { city: String::from("Zürich") };  
let mut a2 = Address { city: String::from("Bern") };  
let o = compare( &mut a1, &mut a2 );  
let c = a1.getCity();
```

Create reference to the owned value

Rust

Borrowing

- Like in C++, operations could be invoked by transferring ownership to the operation and back
- Borrowing provides temporary access to a value through a reference, without transferring ownership

```
fn compare( a: &mut Address, b: &mut Address ) -> Ordering {  
    a.getCity().cmp( b.getCity() )  
}
```

Reference type

Rust

```
let mut a1 = Address { city: String::from("Zürich") };  
let mut a2 = Address { city: String::from("Bern") };  
let o = compare( &mut a1, &mut a2 );  
let c = a1.getCity();
```

Create reference to the owned value

Value can be used after the borrow expires

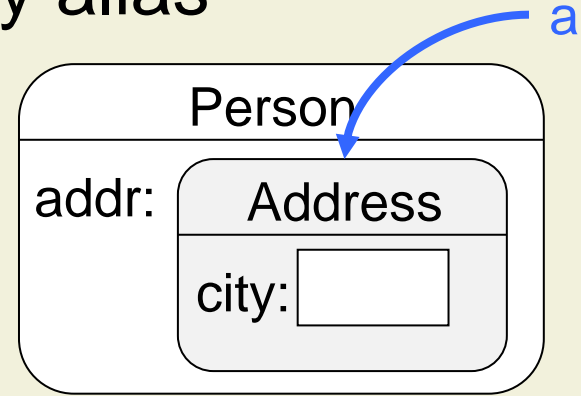
Rust

More on Borrowing

- Borrowing creates a temporary alias

```
let mut p = Person::new( ... );  
let a = &mut p.addr;
```

Rust

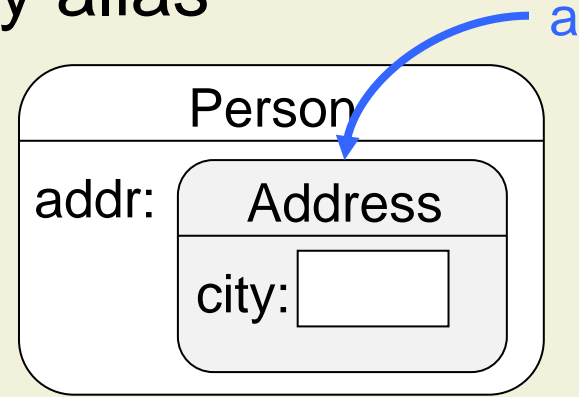


More on Borrowing

- Borrowing creates a temporary alias

```
let mut p = Person::new( ... );  
let a = &mut p.addr;
```

Rust



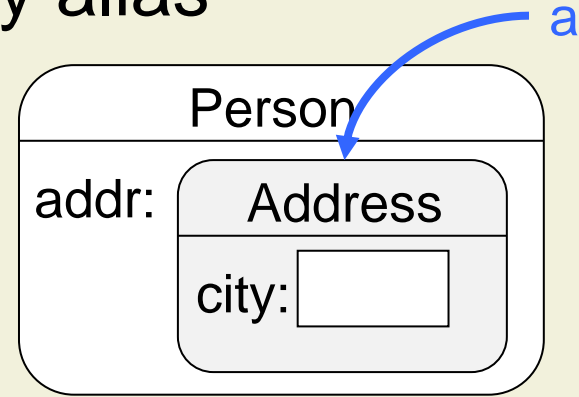
- The borrowed-from place is **unusable** as long as the borrow exists

More on Borrowing

- Borrowing creates a temporary alias

```
let mut p = Person::new( ... );  
let a = &mut p.addr;
```

Rust



- The borrowed-from place is **unusable** as long as the borrow exists

```
let mut p = Person::new( ... );  
let a = &mut p.addr;  
  
println!( "{ }", a.city );
```

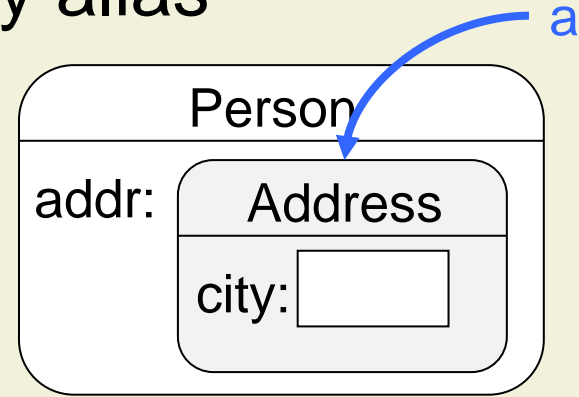
Rust

More on Borrowing

- Borrowing creates a temporary alias

```
let mut p = Person::new( ... );  
let a = &mut p.addr;
```

Rust



- The borrowed-from place is **unusable** as long as the borrow exists

```
let mut p = Person::new( ... );  
let a = &mut p.addr;  
  
println!( "{ }", a.city );
```

Rust

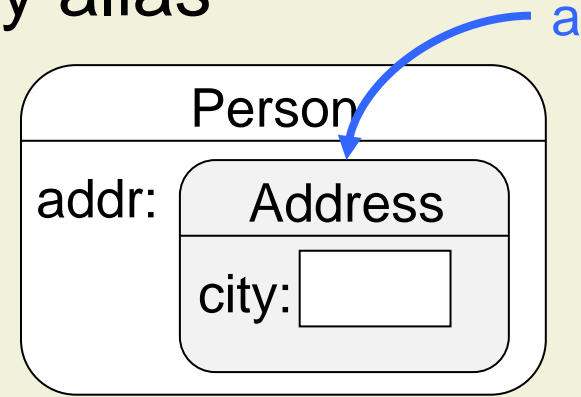
Borrow is
created here

More on Borrowing

- Borrowing creates a temporary alias

```
let mut p = Person::new( ... );  
let a = &mut p.addr;
```

Rust



- The borrowed-from place is **unusable** as long as the borrow exists

```
let mut p = Person::new( ... );  
let a = &mut p.addr;  
  
println!( "{ }", a.city );
```

Borrow is
created here

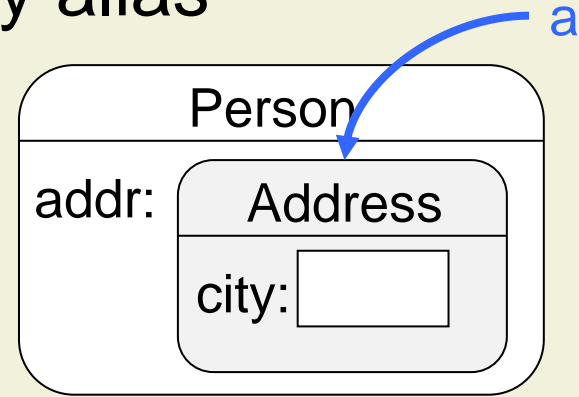
Borrow expires
here

More on Borrowing

- Borrowing creates a temporary alias

```
let mut p = Person::new( ... );  
let a = &mut p.addr;
```

Rust



- The borrowed-from place is **unusable** as long as the borrow exists

```
let mut p = Person::new( ... );  
let a = &mut p.addr;  
println!( "{ }", p.name );  
println!( "{ }", a.city );
```

Borrow is
created here

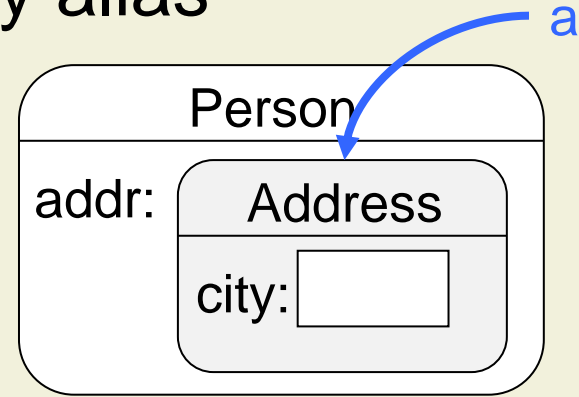
Borrow expires
here

More on Borrowing

- Borrowing creates a temporary alias

```
let mut p = Person::new( ... );  
let a = &mut p.addr;
```

Rust



- The borrowed-from place is **unusable** as long as the borrow exists

Compile time error: cannot use p.addr while it is borrowed

```
let mut p = Person::new( ... );  
let a = &mut p.addr;  
println!( "{ }", p.addr.city );  
println!( "{ }", a.city );
```

Rust

Borrow is created here

Borrow expires here

Borrow Checker

- The Rust compiler uses a static analysis (called borrow checker) to determine when borrows expire

```
let mut p = Person::new( ... );  
let a = &mut p.addr;  
let c = &mut a.city;  
  
println!( "{ }", c );
```

Rust

Borrow Checker

- The Rust compiler uses a static analysis (called borrow checker) to determine when borrows expire

Compile time
error: cannot
use p.addr while
it is borrowed

```
let mut p = Person::new( ... );  
let a = &mut p.addr;  
let c = &mut a.city;  
println!( "{ }", p.addr.city );  
println!( "{ }", c );
```

Rust

Borrow Checker

- The Rust compiler uses a static analysis (called borrow checker) to determine when borrows expire

Compile time error: cannot use p.addr.city while it is borrowed

```
let mut p = Person::new( ... );  
let a = &mut p.addr;  
let c = &mut a.city;  
println!( "{ }", p.addr.city );  
println!( "{ }", c );
```

Rust

Borrow for a is created here

Borrow Checker

- The Rust compiler uses a static analysis (called borrow checker) to determine when borrows expire

Compile time error: cannot use p.addr.city while it is borrowed

```
let mut p = Person::new( ... );  
let a = &mut p.addr;  
let c = &mut a.city;  
println!( "{ }", p.addr.city );  
println!( "{ }", c );
```

Rust

Borrow for a is created here

Borrow for c is created here

Borrow Checker

- The Rust compiler uses a static analysis (called borrow checker) to determine when borrows expire

Compile time error: cannot use p.addr while it is borrowed

```
let mut p = Person::new( ... );  
let a = &mut p.addr;  
let c = &mut a.city;  
println!( "{ }", p.addr.city );  
println!( "{ }", c );
```

Borrow for a is created here

Borrow for c is created here

Both borrows expire here because c re-borrows from a

Borrow Checker

- The Rust compiler uses a static analysis (called borrow checker) to determine when borrows expire

Compile time error: cannot use p.addr while it is borrowed

```
let mut p = Person::new( ... );  
let a = &mut p.addr;  
let c = &mut a.city;  
println!( "{ }", p.addr.city );  
println!( "{ }", c );
```

Borrow for a is created here

Borrow for c is created here

Both borrows expire here because c re-borrows from a

- Borrow checking across functions may require annotations (lifetimes, not discussed here)

Leaking and Capturing

- Rust **does not prevent (temporary) leaking** through a reference

```
fn getAddr( &mut self ) -> &mut Address {  
    &mut self.addr  
}
```

Rust

Result is a **temporary alias** that cannot be captured in a value that outlives the borrow

- **Capturing is safe** due to ownership transfer

```
fn setAddr( &mut self, a: Address ) {  
    self.addr = a  
}
```

Rust

Caller can no longer use the transferred value

(Simplified) Uniqueness Guarantee

- Rust guarantees that, in each execution state, there is **at most one usable place** that can access a value
 - No usable aliases
 - We will make this guarantee more precise later
- Rust leverages this guarantee:
 - To ensure data race freedom
 - To perform automatic memory management (without a garbage collector)

Memory Management

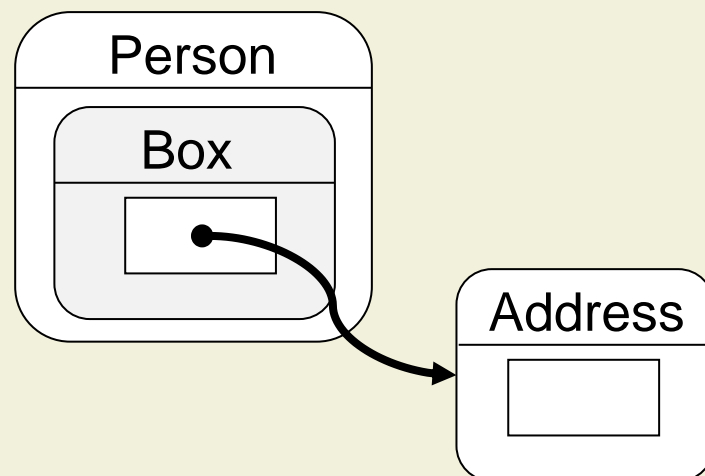
- Automatic memory management in Rust is safe
- When a place goes out of scope, all values it (transitively) owns are automatically de-allocated

```
struct Person {  
  addr: Box<Address>,  
}
```

Rust

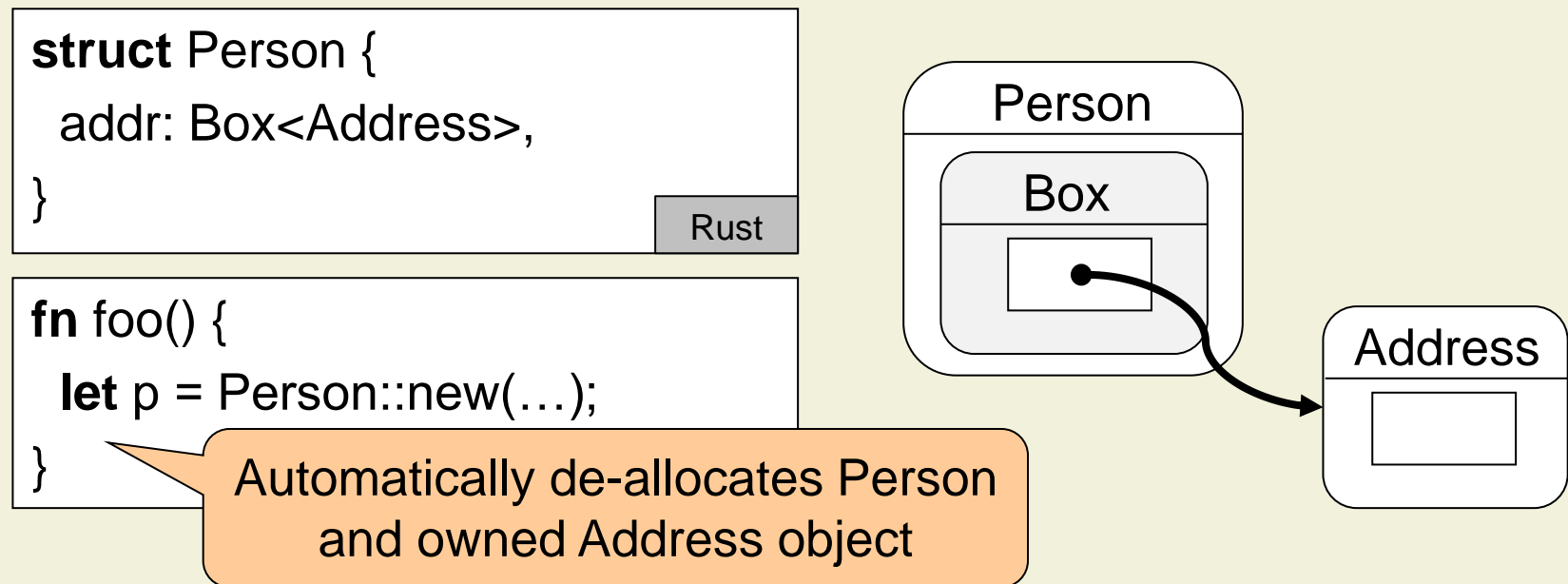
```
fn foo() {  
  let p = Person::new(...);  
}
```

Rust



Memory Management

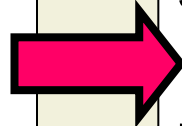
- Automatic memory management in Rust is safe
- When a place goes out of scope, all values it (transitively) owns are automatically de-allocated



Exchanging Implementations

```
struct List {  
  array: Vec<i32>,  
}
```

Rust



```
struct List {  
  head: Box<Node>,  
}
```

Rust

- Creating an alias via capturing is prevented by ownership

```
fn setElems(&mut self, ia: Vec<i32>) {  
  self.array = ia;  
}
```

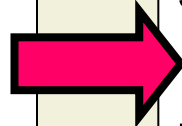
Rust

- Callers cannot use array after call to setElems

Exchanging Implementations

```
struct List {  
  array: Vec<i32>,  
}
```

Rust



```
struct List {  
  head: Box<Node>,  
}
```

Rust

- Creating an alias via capturing is prevented by ownership

```
fn setElems(&mut self, ia: Vec<i32>) {  
  self.array = ia;  
}
```

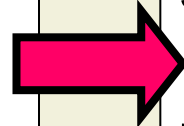
Ownership transfer
from caller

- Callers cannot use array after call to setElems

Exchanging Implementations (cont'd)

```
struct List {  
    array: Vec<i32>,  
}
```

Rust



```
struct List {  
    head: Box<Node>,  
}
```

Rust

- Creating an alias via leaking is still possible

```
fn getElems( &mut self )  
    -> &mut Vec<i32> {  
    &mut self.array  
}
```

Rust

```
fn getElems( &mut self )  
    -> Vec<i32> {  
    let res = Vec::new( );  
    ... }
```

Rust

- Leaking and non-leaking methods typically have **different result types**
 - Effects on clients are unlikely (but possible)

Consistency of Object Structures

```
struct List {  
    array: Vec<i32>,  
  
    // invariant  
    //   $\forall i. 0 \leq i < \text{array.len}():$   
    //      array[ i ] >= 0  
}
```

Rust

```
fn getElems( &mut self )  
    -> &mut Vec<i32> {  
    &mut self.array  
}
```

Rust

- Consistency of object structures depends on fields of several objects
- Since leaking is still possible, clients can violate invariant

```
fn violate( l: &mut List ) {  
    let a = l.getElems( );  
    a[ 0 ] = -1;  
}
```

Rust

Security Breach in Java 1.1.1

```
class Malicious {
```

```
  void bad( ) {
```

```
    Identity[ ] s;
```

```
    Identity trusted = java.Security...;
```

```
    s = Malicious.class.getSigners( );
```

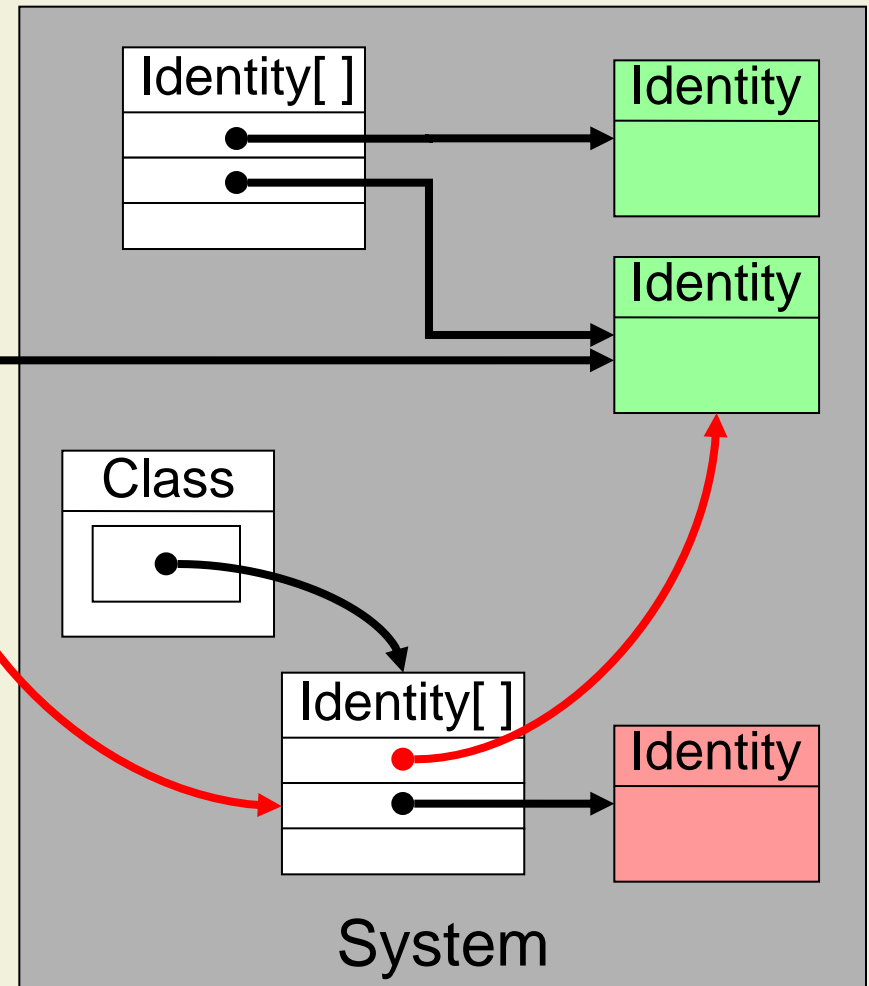
```
    s[ 0 ] = trusted;
```

```
    /* abuse privilege */
```

```
  }
```

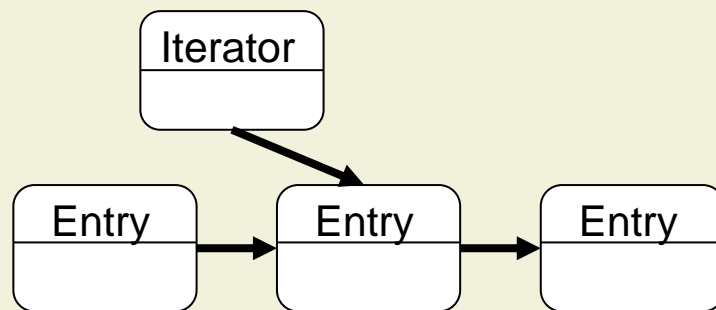
```
}
```

Leaking the identity array would also be possible in Rust

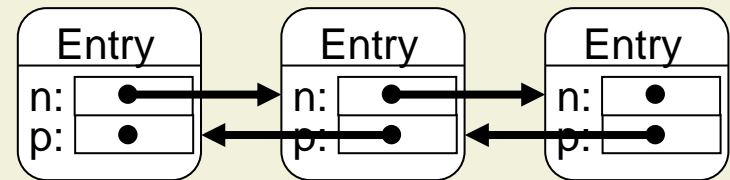


Limitations

- Rust's deep ownership and uniqueness imply that **all data structures are tree-shaped**



No sharing



No cyclic data structures

- **Unsafe Rust** provides **raw pointers** to work around the strict ownership and borrowing rules
 - Unsafe code should be encapsulated inside libraries
 - We assume here that only safe Rust is used

Ownership in Rust: Discussion

Pros

Cons

Ownership in Rust: Discussion

Pros

- Ownership guarantees effective uniqueness

Cons

Ownership in Rust: Discussion

Pros

- Ownership guarantees effective uniqueness
- Creating aliases via capturing is prevented statically

Cons

Ownership in Rust: Discussion

Pros

- Ownership guarantees effective uniqueness
- Creating aliases via capturing is prevented statically
- Uniqueness ensures safe automatic memory management and data race freedom

Cons

Ownership in Rust: Discussion

Pros

- Ownership guarantees effective uniqueness
- Creating aliases via capturing is prevented statically
- Uniqueness ensures safe automatic memory management and data race freedom
- No run time overhead

Cons

Ownership in Rust: Discussion

Pros

- Ownership guarantees effective uniqueness
- Creating aliases via capturing is prevented statically
- Uniqueness ensures safe automatic memory management and data race freedom
- No run time overhead

Cons

- Leaking is still possible

Ownership in Rust: Discussion

Pros

- Ownership guarantees effective uniqueness
- Creating aliases via capturing is prevented statically
- Uniqueness ensures safe automatic memory management and data race freedom
- No run time overhead

Cons

- Leaking is still possible
- Not effective in solving some of the problems caused by aliasing

Ownership in Rust: Discussion

Pros

- Ownership guarantees effective uniqueness
- Creating aliases via capturing is prevented statically
- Uniqueness ensures safe automatic memory management and data race freedom
- No run time overhead

Cons

- Leaking is still possible
- Not effective in solving some of the problems caused by aliasing
- Ownership regime is sometimes too restrictive

Ownership in Rust: Discussion

Pros

- Ownership guarantees effective uniqueness
- Creating aliases via capturing is prevented statically
- Uniqueness ensures safe automatic memory management and data race freedom
- No run time overhead

Cons

- Leaking is still possible
- Not effective in solving some of the problems caused by aliasing
- Ownership regime is sometimes too restrictive
- Guarantees may be compromised by using unsafe Rust

5. Object Structures and Aliasing

5.1 Aliasing

5.2 Problems of Aliasing

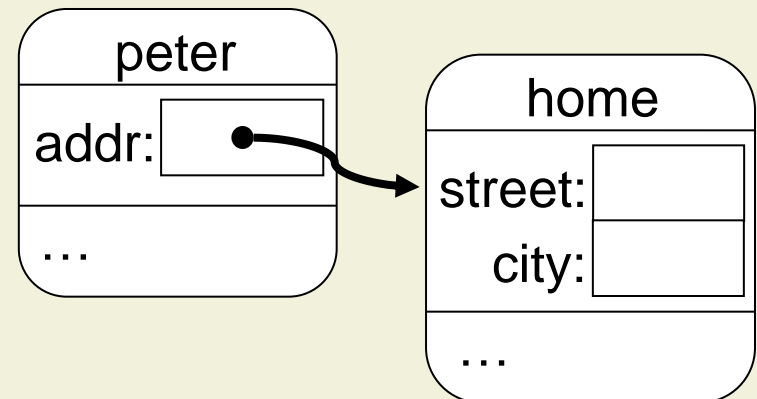
5.3 Unique References

5.4 Readonly References

Object Structures Revisited

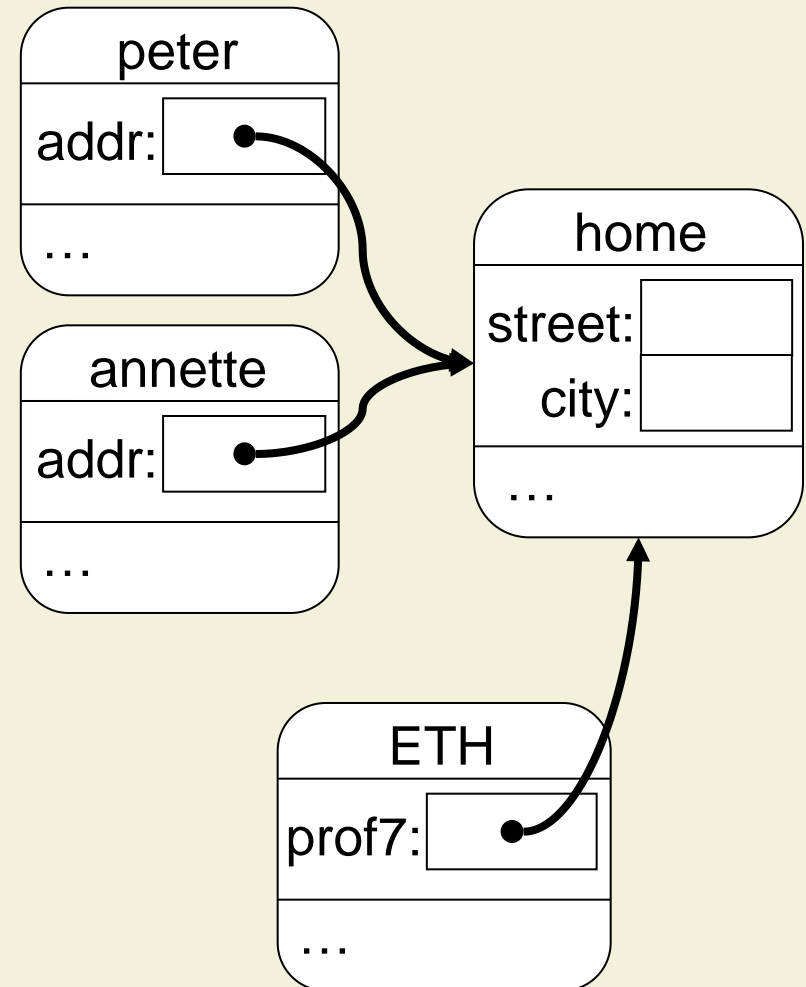
```
class Address ... {  
    private String street;  
    private String city;  
  
    public String getStreet( ) { ... }  
    public void setStreet( String s )  
        { ... }  
  
    public String getCity( ){ ... }  
    public void setCity( String s )  
        { ... }  
    ...  
}
```

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



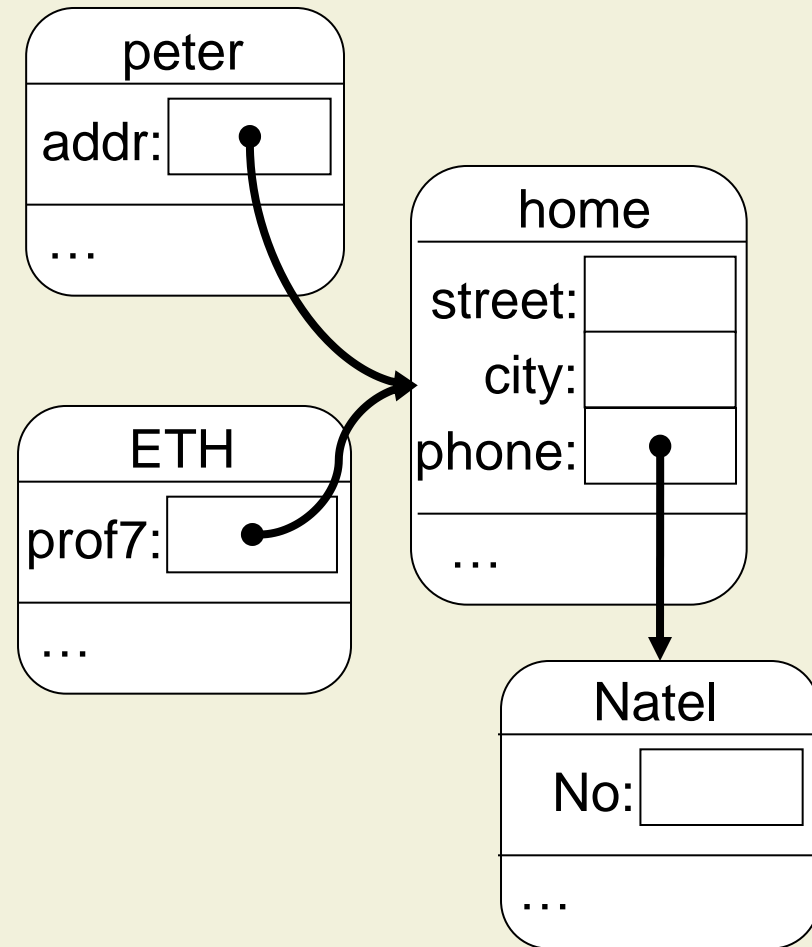
Drawbacks of Uniqueness

- 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
- **Effectiveness**
 - Prevent unwanted modifications
- **Transitivity**
 - Access restrictions extend to sub-objects



Readonly Access via Supertypes

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

```
class Address  
    implements ReadonlyAddress ... {  
    ... /* as before */ }
```

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( ); }  
    ... }
```

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

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

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

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

```
interface ReadonlyAddress {
    ...
    public PhoneNo getPhone( );
}
```

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

```
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**

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

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

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 C++: const Pointers

```
class Address {  
    string city;  
public:  
    string getCity( )  
        { 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

Readonly Access in C++: const Pointers

```
class Address {  
    string city;  
public:  
    string getCity( )  
        { 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( );  
}
```

C++

Readonly Access in C++: const Pointers

```
class Address {  
    string city;  
public:  
    string getCity( )  
        { 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
error

Readonly Access in C++: const Pointers

```
class Address {  
    string city;  
public:  
    string getCity( )  
        { 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( ) 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

Readonly Access in C++: const Functions

```
class Address {  
    string city;  
public:  
    string getCity( ) 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( );  
}
```

C++

Readonly Access in C++: const Functions

```
class Address {  
    string city;  
public:  
    string getCity( ) 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( );  
}
```

Compile-time
error

Readonly Access in C++: const Functions

```
class Address {  
    string city;  
public:  
    string getCity( ) 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( ) 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

It wouldn't be C++ ...

```
class Address {  
    string city;  
public:  
    string getCity( ) 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( ) 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++

Readonly Access in C++: Transitivity

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

C++

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

C++

Readonly Access in C++: Transitivity

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

C++

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

C++

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

C++

- **const** pointers are not transitive
 - **const**-ness of sub-objects has to be indicated explicitly
 - **const** functions must not modify sub-objects

Readonly Access in C++: Discussion

Pros

Readonly Access in C++: Discussion

Pros

- const pointers provide readonly pointers to **mutable objects**
 - Prevent field updates
 - Prevent calls of non-const functions

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**

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 arrays, fields, and non-public methods

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 arrays, fields, and non-public methods

Cons

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 arrays, fields, and non-public methods

Cons

- const-ness is **not transitive**

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 arrays, fields, and non-public methods

Cons

- const-ness is **not transitive**
- const pointers are **unsafe**
 - Explicit casts

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 arrays, fields, and non-public methods

Cons

- const-ness is **not transitive**
- const pointers are **unsafe**
 - Explicit casts
- const pointers **do not prevent capturing**

Immutable References in Rust

- So far, we discussed the creation of **mutable references**

```
fn getAddr( &mut self ) -> &mut Address {  
    &mut self.addr  
}
```

Rust

- Rust also supports **immutable references**

```
fn getAddr( &self ) -> &Address {  
    &self.addr  
}
```

Rust

Shared References

- In contrast to mutable references, there can be **multiple immutable references** to a value

```
fn getAddr( &self ) -> &Address {  
    &self.addr  
}
```

Rust

```
let a1 = p.getAddr( );  
let a2 = p.getAddr( );  
println!( "{ }", a1.getCity() );  
println!( "{ }", a2.getCity() );
```

Rust

- The value is **statically guaranteed** to remain immutable as long as at least one immutable borrow exists

Uniqueness Guarantee

- Rust guarantees that, in each execution state, there is either **at most one usable place** that can **mutably** access a value, **or the value is immutable**
- Rust refers to this guarantee as **Aliasing XOR Mutability**
- Rust leverages this guarantee:
 - To ensure data race freedom
 - To perform automatic memory management (without a garbage collector)

Readonly Access in Rust: Effectiveness

```
struct Address { city: String, }  
  
impl Address {  
  fn getCity( &self ) -> &String  
    { &self.city }  
  fn setCity( &mut self, s: String )  
    { self.city = s }  
}
```

Rust

```
struct Person { addr: Address, }  
  
impl Person {  
  fn getAddr( &self ) -> &Address  
    { &self.addr }  
  fn setAddr( &mut self, a: Address )  
    { self.addr = a; /* move */ }  
}
```

Rust

- Immutable references do not allow:
 - field updates
 - mutator calls

Readonly Access in Rust: Effectiveness

```
struct Address { city: String, }  
  
impl Address {  
  fn getCity( &self ) -> &String  
    { &self.city }  
  fn setCity( &mut self, s: String )  
    { self.city = s }  
}
```

Rust

```
struct Person { addr: Address, }  
  
impl Person {  
  fn getAddr( &self ) -> &Address  
    { &self.addr }  
  fn setAddr( &mut self, a: Address )  
    { self.addr = a; /* move */ }  
}
```

Rust

- Immutable references do not allow:
 - field updates
 - mutator calls

```
fn m( p: Person ) {  
  let a = p.getAddr();  
  a.setCity(String::from("Hagen"));  
  a.city = String::from("Hagen");  
}
```

Rust

Readonly Access in Rust: Effectiveness

```
struct Address { city: String, }  
  
impl Address {  
  fn getCity( &self ) -> &String  
    { &self.city }  
  fn setCity( &mut self, s: String )  
    { self.city = s }  
}
```

Rust

```
struct Person { addr: Address, }  
  
impl Person {  
  fn getAddr( &self ) -> &Address  
    { &self.addr }  
  fn setAddr( &mut self, a: Address )  
    { self.addr = a; /* move */ }  
}
```

Rust

- Immutable references do not allow:
 - field updates
 - mutator calls

```
fn m( p: Person ) {  
  let a = p.getAddr();  
  a.setCity(String::from("Hagen"));  
  a.city = String::from("Hagen");  
}
```

Compile-time
error

Readonly Access in Rust: Effectiveness

```
struct Address { city: String, }  
  
impl Address {  
  fn getCity( &self ) -> &String  
    { &self.city }  
  fn setCity( &mut self, s: String )  
    { self.city = s }  
}
```

Rust

```
struct Person { addr: Address, }  
  
impl Person {  
  fn getAddr( &self ) -> &Address  
    { &self.addr }  
  fn setAddr( &mut self, a: Address )  
    { self.addr = a; /* move */ }  
}
```

Rust

- Immutable references do not allow:
 - field updates
 - mutator calls

```
fn m( p: Person ) {  
  let a = p.getAddr();  
  a.setCity(String::from("Hagen"));  
  a.city = String::from("Hagen");  
}
```

Compile-time
errors

Readonly Access in Rust: Reading

```
struct Address { city: String, }  
  
impl Address {  
  fn getCity( &self ) -> &String  
    { &self.city }  
  fn setCity( &mut self, s: String )  
    { self.city = s }  
}
```

Rust

```
struct Person { addr: Address, }  
  
impl Person {  
  fn getAddr( &self ) -> &Address  
    { &self.addr }  
  fn setAddr( &mut self, a: Address )  
    { self.addr = a; /* move */ }  
}
```

Rust

- Signature determines whether a function may be called on an immutable reference

Readonly Access in Rust: Reading

```
struct Address { city: String, }  
  
impl Address {  
  fn getCity( &self ) -> &String  
    { &self.city }  
  fn setCity( &mut self, s: String )  
    { self.city = s }  
}
```

Rust

```
struct Person { addr: Address, }  
  
impl Person {  
  fn getAddr( &self ) -> &Address  
    { &self.addr }  
  fn setAddr( &mut self, a: Address )  
    { self.addr = a; /* move */ }  
}
```

Rust

- Signature determines whether a function may be called on an immutable reference

```
fn m( p: Person ) {  
  let a = p.getAddr();  
  let c = a.getCity();  
  ...  
}
```

Rust

Readonly Access in Rust: Transitivity

```
struct Phone { number: u32, };
```

Rust

```
struct Address {  
  city: String,  
  phone: Phone,  
}  
  
impl Address {  
  fn getPhone( &self ) -> &Phone  
    { &self.phone }  
}
```

Rust

Readonly Access in Rust: Transitivity

```
struct Phone { number: u32, };
```

Rust

```
struct Address {  
  city: String,  
  phone: Phone,  
}  
  
impl Address {  
  fn getPhone( &self ) -> &Phone  
    { &self.phone }  
}
```

Rust

```
fn m( p: Person ) {  
  let a = p.getAddr();  
  let ph = &a.phone;  
  ph.number = 2331;  
}
```

Rust

Compile-time
error

Readonly Access in Rust: Transitivity

```
struct Phone { number: u32, };
```

Rust

```
struct Address {  
  city: String,  
  phone: Phone,  
}  
  
impl Address {  
  fn getPhone( &self ) -> &Phone  
    { &self.phone }  
}
```

Rust

```
fn m( p: Person ) {  
  let a = p.getAddr();  
  let ph = &a.phone;  
  ph.number = 2331;  
}
```

Rust

Compile-time
error

- Immutability guarantee is transitive
- It is not possible to obtain a mutable reference through an immutable reference

Readonly Access in Rust: Mutable Values

```
struct Address { city: String, }
```

Rust

```
struct University { prof7: &Address, }
```

Rust

```
fn m( p: Person ) {  
    let mut a = Address{ city: String::from("Zurich") };  
    let u = University{ prof7: &a };  
    a.city = String::from("Winterthur");  
    let c = &u.prof7.city;  
}
```

Rust

Readonly Access in Rust: Mutable Values

```
struct Address { city: String, }
```

Rust

```
struct University { prof7: &Address, }
```

Create
immutable
reference

```
fn m( p: Person ) {  
    let mut a = Address{ city: String::from("Zurich") };  
    let u = University{ prof7: &a };  
    a.city = String::from("Winterthur");  
    let c = &u.prof7.city;  
}
```

Rust

Readonly Access in Rust: Mutable Values

```
struct Address { city: String, }
```

Rust

```
struct University { prof7: &Address, }
```

```
fn m( p: Person ) {  
  let mut a = Address{ city: String::from("Zurich") };  
  let u = University{ prof7: &a };  
  a.city = String::from("Winterthur");  
  let c = &u.prof7.city;  
}
```

Create
immutable
reference

Immutable
reference is
used until here

Rust

Readonly Access in Rust: Mutable Values

```
struct Address { city: String, }
```

Rust

```
struct University { prof7: &Address, }
```

```
fn m( p: Person ) {  
  let mut a = Address{ city: String::from("Zurich") };  
  let u = University{ prof7: &a };  
  a.city = String::from("Winterthur");  
  let c = &u.prof7.city;  
}
```

Create
immutable
reference

Compile-time
error: value is
immutable

Immutable
reference is
used until here

Readonly Access in Rust: Mutable Values

```
struct Address { city: String, }
```

Rust

```
struct University { prof7: &Address, }
```

```
fn m( p: Person ) {  
  let mut a = Address{ city: String::from("Zurich") };  
  let u = University{ prof7: &a };  
  a.city = String::from("Winterthur");  
  let c = &u.prof7.city;  
}
```

Create
immutable
reference

Compile-time
error: value is
immutable

Immutable
reference is
used until here

- Value is
immutable
while
immutable
references
exist

Consistency of Object Structures

```
struct List {  
    array: Vec<i32>,  
  
    // invariant  
    //   $\forall i. 0 \leq i < \text{array.len}():$   
    //      array[ i ] >= 0  
}
```

Rust

```
fn getElems( &self ) -> &Vec<i32> {  
    &self.array  
}
```

Rust

- Immutable references enable **safe leaking**
- Clients **cannot** use an immutable reference to **violate invariant**
- But leaking mutable references is also possible (**mut** keyword increases awareness)

Security Breach in Java 1.1.1

```
class Malicious {
```

```
void bad( ) {
```

Identity[] s;

Identity trusted = java.Security...

```
s = Malicious.class.getSigners();
```

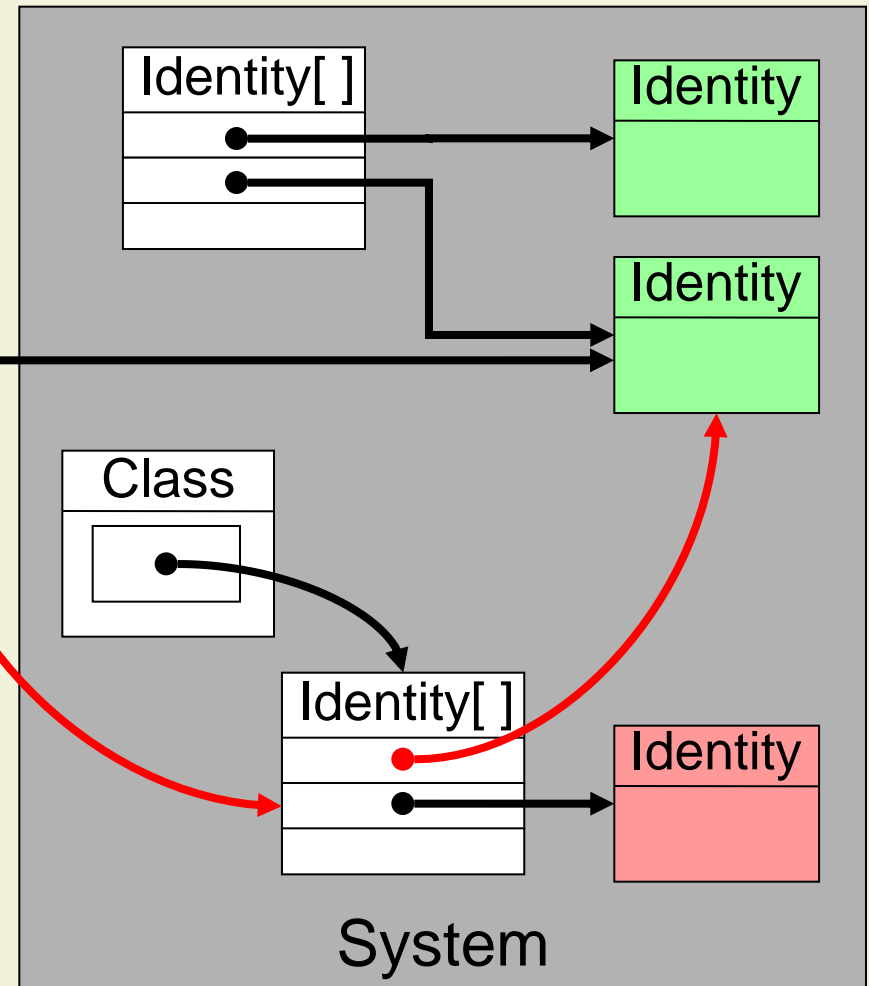
```
s[ 0 ] = trusted;
```

```
/* abuse privilege */
```

}

}

Leaking the identity array as **immutable reference** would be secure



Readonly Access in Rust: Discussion

- Immutable references provide **temporary** readonly access to **mutable values**
 - Work for **library classes**
 - Support arrays, fields, and non-public methods
- Readonly access is **transitive**
- Value is **immutable while immutable references exist**
 - Prevents, e.g., data races
 - Not suitable for all data structures

References

- Bjarne Stroustrup: *A Tour of C++*. Pearson, 2022
- Steve Klabnik and Carol Nichols: *The Rust Programming Language*. No Starch Press, 2023