Ownership and Immutability

Dr Alex Potanin

Aliasing

"The big lie of object-oriented programming is that objects provide encapsulation."

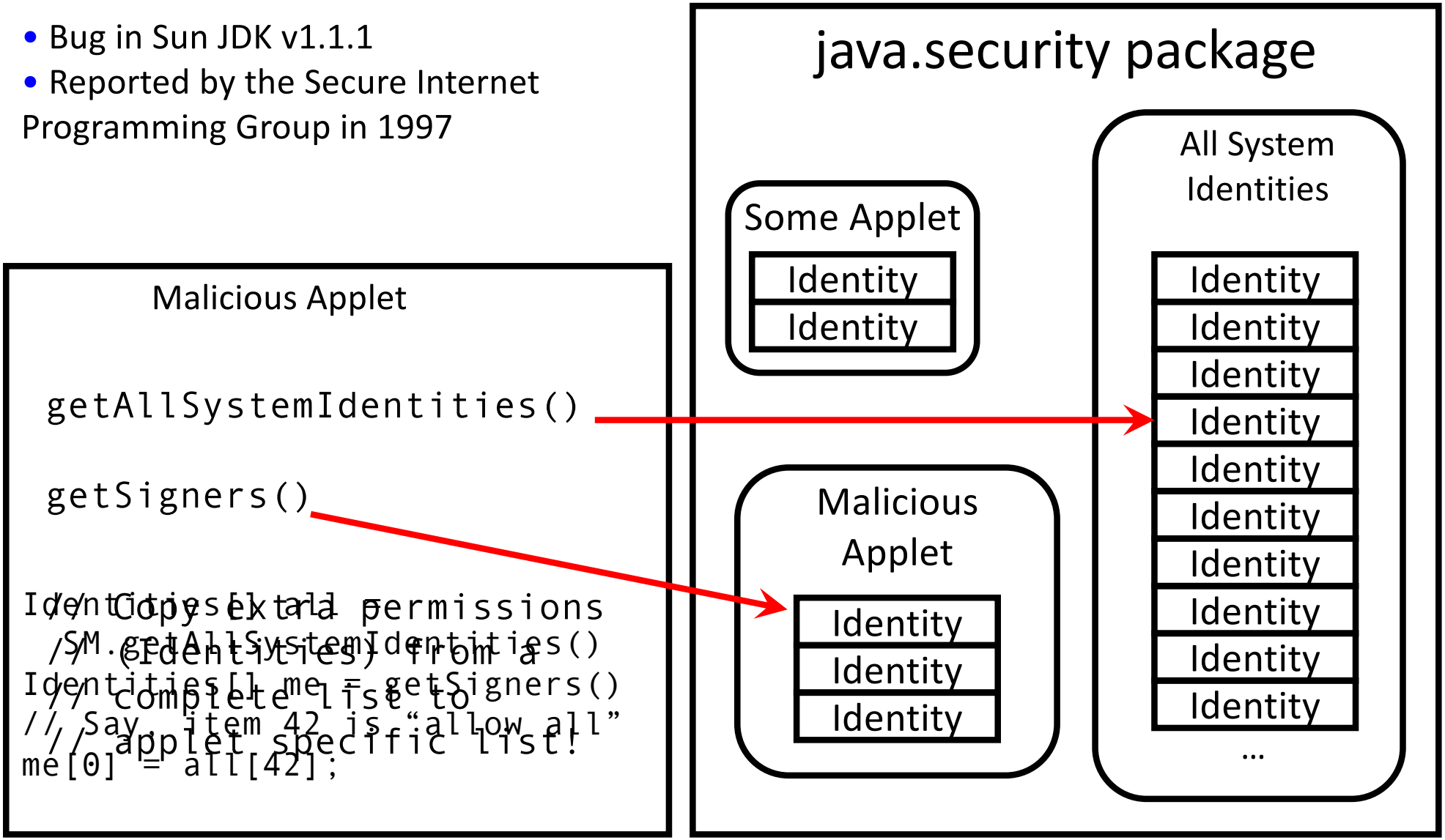
John Hogg

Islands: Aliasing Protection in Object-Oriented Languages

OOPSLA 1991

Aliasing (The Ugly)

- Bug in Sun JDK v1.1.1
- Reported by the Secure Internet Programming Group in 1997



Based on *Confined Types* by Bokowski and Vitek (OOPSLA 1998)

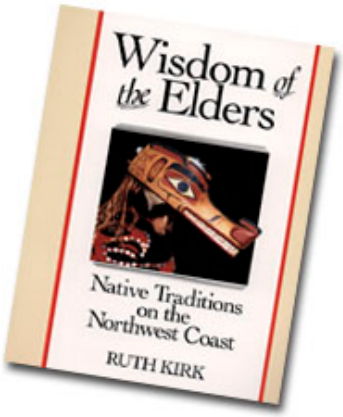
THE IDEA: GENERIC OWNERSHIP

Motivation (simple Map in Java <5)

```
class Node { . . . }
public class Map {
  private Vector nodes;
  public Vector expose() { return this.nodes; }
  void put(Comparable key, Object value) {
    nodes.add(new Node(key, value));
  }
  Object get(Comparable k) {
    Iterator i = nodes.iterator();
    while (i.hasNext()) {
      Node mn = (Node) i.next();
      if (((Comparable) mn.key).equals(k)) return mn.value;
    }
    return null;
  }
}
```

Book

Box



Book

```
Map books = new Map();
books.put("Wisdom", new Book());
Object b = books.get("Wisdom"); // Don't know what get returns!
Vector aliasedNodes = books.expose(); // Private field exposed!
```

Motivation (generic Map in Java 5+)

```
class Node<Key extends Comparable, Value> { . . . }
public class Map<Key extends Comparable, Value> {
  private Vector<Node<Key, Value>> nodes;
  public Vector<Node<Key, Value>> expose() {
    return this.nodes;
  }
  void put(Key key, Value value) {
    nodes.add(new Node<Key, Value>(key, value));
  }
  Value get(Key k) {
    Iterator<Node<Key, Value>> i = nodes.iterator();
    while (i.hasNext()) {
      Node<Key, Value> mn = i.next();
      if (mn.key.equals(k)) return mn.value;
    }
    return null;
  }
}
```

Box (Bird)

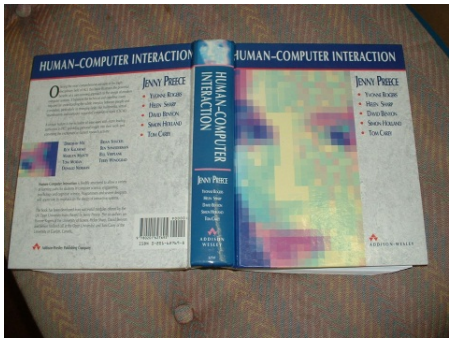
Box (Book)


```
Map<String, Book> books = new Map<String, Book>();
books.put("Wisdom", new Book());
Book b = books.get("Wisdom"); // Type of Map knows what get returns
Object b = books.get("Wisdom"); // Type of Map knows what get returns
Vector<Node<String, Book>> exposed = books.expose(); // Exposed!
Vector<Node<String, Book>> exposed = books.expose(); // Exposed!
```

Motivation (ownership Map in Safe Java)

```
class Node<nodeOwner, kOwner, vOwner> { . . . }
public class Map<mOwner, kOwner, vOwner> {
  private Vector<this, this> nodes;
  public Vector<this, this> expose() { return this.nodes; }
  void put(Comparable<kOwner> key, Object<vOwner> value) {
    nodes.add(new Node<this, kOwner, vOwner>(key, value));
  }
  Object<vOwner> get(Comparable<kOwner> key) {
    Iterator<this, this> i = nodes.iterator();
    while (i.hasNext()) {
      Node<this, kOwner, vOwner> mn =
        (Node<this, kOwner, vOwner>) i.next();
      if (mn.key.equals(key)) return mn.value;
    }
    return null;
  }
}
```

Book (Me)



Book (Robert)

```
Map<String, Book> books = new Map<this, world, world>();
books.put("Wisdom", new Book());
Book b = books.get("Wisdom"); // Type of Map knows what get returns!
Vector<Node<String, Book>> aliasedNodes = books.expose(); // books.nodes is not this.
```

Motivation (ownership & generic Map?)

```
class Node<nodeOwner>
  [Key<kOwner> extends Comparable<kOwner>, Value<vOwner>] { . . . }
public class Map<mOwner>
  [Key<kOwner> extends Comparable<kOwner>, Value<vOwner>] {
  private Vector<this>[Node<this>[Key<kOwner>, Value<vOwner>]] nodes;
  public Vector<this>[Node<this>[Key<kOwner>, Value<vOwner>]] expose() {
    return this.nodes;
  }
  void put(Key<kOwner> key, Value<vOwner> value) {
    nodes.add(new Node<this>[Key<kOwner>, Value<vOwner>](key, value));
  }
  Value<vOwner> get(Key<kOwner> key) {
    Iterator<this>[Nodes<this>[Key<kOwner>, Value<vOwner>]] = nodes.iterator();
    while(i.hasNext()) {
      Node<this>[Key<kOwner>, Value<vOwner>] mn = i.next();
      if (mn.key.equals(k)) return mn.value;
    }
    return null;
  }
}
```

```
Map<this>[String<world>, Book<world>] books =
   new Map<this>[String<world>, Book<world>] ();
books.put("Wisdom", new Book<world>());
Book<world> b = books.get("Wisdom"); // Map type knows what get returns
Vector<this>[Node<this>[String<world>, Book<world>]] aliasedNodes =
  books.expose(); // books. is not this.
```

See page 29 of *Safe Java: A Unified Type System for Safe Programming (PhD Thesis by Chandra Boyapati)* for the origins of this syntax.

```
Map<this, world, world> books = new Map<this, world, world>();
books.put("Wisdom", new Book());
Book<world> b = books.get("Wisdom"); // Don't know what get returns!
Vector<this, this> aliasedNodes = books.expose(); // books. is not this.
```

Motivation (Generic Ownership ™ Map)

```
class Node<Key extends Comparable, Value, Owner extends World> { . . . }
public class Map<Key extends Comparable, Value, Owner extends World> {
  private Vector<Node<Key, Value, This>, This> nodes;
  public Vector<Node<Key, Value, This>, This> expose() { return this.nodes; }
  public void put(Key key, Value value) {
    nodes.add(new Node<Key, Value, This>(key, value));
  }
  public Value get(Key key) {
    Iterator<Node<Key, Value, This>, This> i = nodes.iterator();
    while (i.hasNext()) {
      Node<Key, Value, This> mn = i.next();
        if (mn.key.equals(key)) return mn.value;
      }
    return null;
  }
}
```

Box of *My Computer Books*

```
Map<this>[String<world>, Book<world>] books =
   new Map<this>[String<world>, Book<world>] ();
books.put("Wisdom", new Book<world>());
Book<world> b = books.get("Wisdom"); // Map type knows what get returns
```

```
Map<String, Book, This> books = new Map<String, Book, This>();
books.put("Wisdom", new Book());
  books.expose(); // books. is not this.
Book b = books.get("Wisdom"); // Type of Map knows what get returns
Vector<this, this> aliasedNodes = books.expose(); // books. is not this.
```

```
Vector<this>[Node<this>[String<world>, Book<world>]] aliasedNodes =
```

OIGJ: OWNERSHIP AND IMMUTABILITY GENERIC JAVA

Ownership + Immutability

- Our previous work
  - OGJ: added Ownership to Generic Java
  - IGJ: added Immutability to Generic Java
- This work
  - OIGJ: combine Ownership + Immutability
  - The sum is greater than its parts
    - Immutable cyclic data structures (e.g. doubly linked list)
    - IGJ could not type check *Java Collections*
    - OIGJ can, without any code changes

Problem 1: Representation exposure

- Internal representation leaks to the outside
  - `private` doesn't offer real protection!

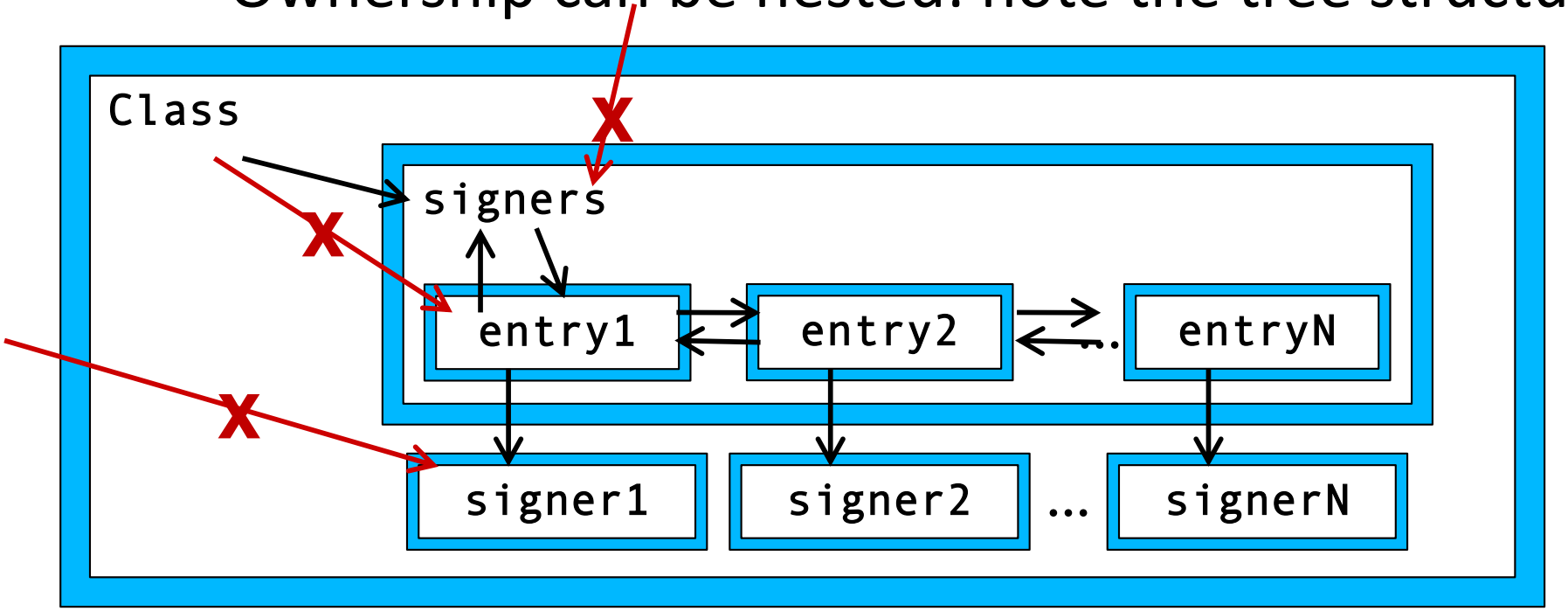
```
class Class {
  private List signers;
  public List getSigners() {
    return this.signers;
  }
}
```

Forgot to copy signers!

  - http://java.sun.com/security/getSigners.html
  - **Bug:** the system thinks that code signed by one identity is signed by a different identity

Solution for Representation Exposure

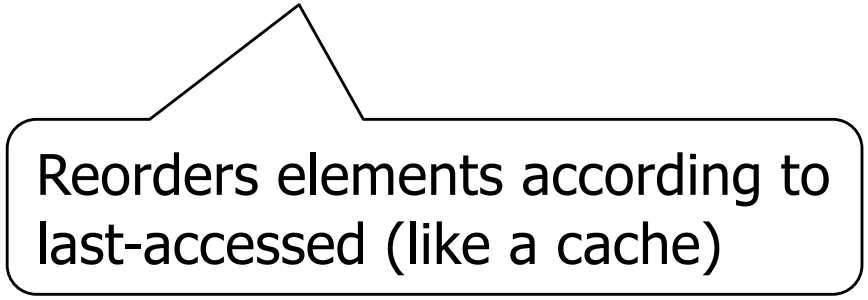
- Ownership: owner-as-dominator
  - `Class` should own the list `signers`
  - No **outside** alias can exist
  - Ownership can be nested: note the tree structure



Problem 2: Unintended Modification

- Modification is not explicit in Java
  - Can `Map.get()` modify the map?

```
for (Object key : map.keySet())
        map.get(key);
```



  - Throws `ConcurrentModificationException` for the following map:

```
new LinkedHashMap(100, 1, true)
```

Solution: Immutability

- Object immutability
  - mutable / immutable
- Readonly references
  - mutable / immutable / **readonly**

```
class Student {
  @Immutable Date dateOfBirth; ...
  void setTutor(@ReadOnly Student tutor) @Mutable { ... }
}
```

Method may modify the `this` object

Raw vs Cooked

- Creation of an immutable object
  - **Raw** state: Fields can be assigned
  - **Cooked** state: Fields cannot be assigned
  - When does an object become cooked?
- Traditionally
  - An object is cooked when **its** constructor finishes

OIGJ's novel idea

- Connect ownership & immutability
  - An object is cooked when **its owner's** constructor finishes
  - The outside world will not see this cooking phase
  - The complex object and its representation become immutable simultaneously
  - Enables creation of **immutable** cyclic structures

Cooking LinkedList (1 of 2)

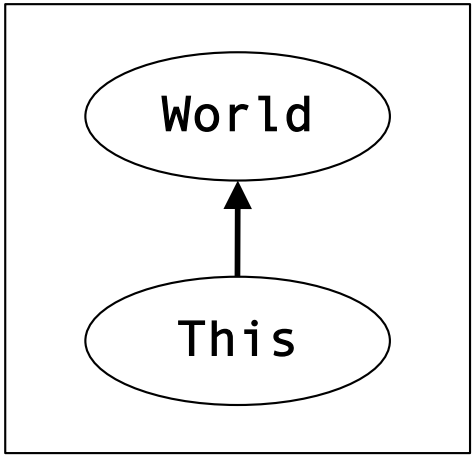
```
1 : LinkedList(Collection<E> c) {
2 :   this(); // Initializes this.header
3 :   Entry<E> succ = this.header, pred = succ.prev;
4 :   for (E e : c) {
5 :     Entry<E> entry =
6 :       new Entry<E>(e,succ,pred);
7 :     // An entry is modified after its constructor finished
8 :     pred.next = entry; pred = entry;
9 :   }
10:   succ.prev = pred;
11: }
```

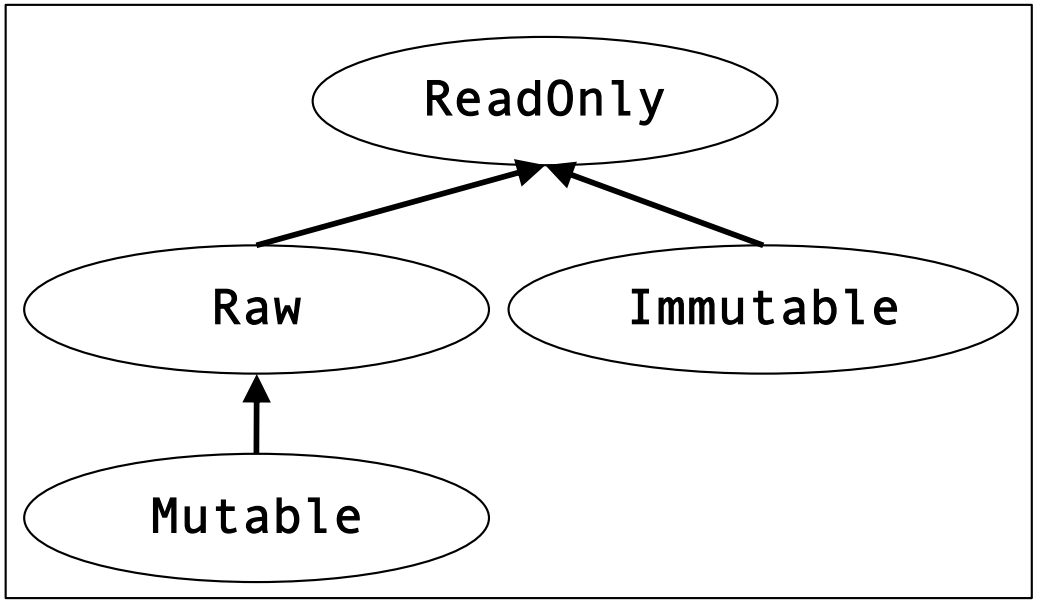
- Goal: No refactoring

Cooking LinkedList (2 of 2)

```
1 : LinkedList(@ReadOnly Collection<E> c) @Raw {
2 :   this(); // Initializes this.header
3 :   @This @I Entry<E> succ = this.header, pred = succ.prev;
4 :   for (E e : c) {
5 :     @This @I Entry<E> entry =
6 :       new @This @I Entry<E>(e,succ,pred);
7 :     // An entry is modified after its constructor finished
8 :     pred.next = entry; pred = entry;
9 :   }
10:   succ.prev = pred;
11: }
```

OIGJ Annotations





Ownership hierarchy

`World` – anyone can access

`This` – `this` owns the object

Immutability hierarchy

`ReadOnly` – no modification

`Raw` – object under construction

Immutability

```
1:class Foo {
2:  // An immutable reference to an immutable date.
    @Immutable Date imD;
3:  // A mutable reference to a mutable date.
    @Mutable Date mutD;
4:  // A readonly reference to any date.
    @ReadOnly Date roD = ... ? imD : mutD;
4:  // A reference to a date with the same immutability as this.
    @I Date sameI;
5:  // Can be called on any receiver; cannot mutate this.
    int readonlyMethod() @ReadOnly {...}
6:  // Can be called only on mutable receivers; can mutate this.
    void mutatingMethod() @Mutable {...}
7:}
```

Ownership

```
1: // A date with the same owner and immutability as this
   @O @I Date sameD;
2: // A date owned by this; it cannot leak.
   @This @I Date ownedD;
3: // Anyone can access this date.
   @World @I Date publicD;
```

```
class LinkedList<E> {
   @This @I Entry<E> header;
   ...
}
class Entry<E> {
   @O @I Entry<E> next, prev;
   ...
}
```

Formalisation: Featherweight OIGJ

- Novel idea: **Cookers**
  - Every object in the heap is of the form:

`x→Foo<y,Mutable>` or `x→Foo<y,Immutable`$_z$`>`

  - y is the owner of x
  - z is the **cooker** of x, i.e., x becomes **cooked** when the constructor of z finishes
  - Formalism tracks the ongoing constructors
  - Subtyping rules connect cookers and owners
- Proved soundness and type preservation

Case Study

- Implementation uses the Checkers Framework
  - Only 1600 lines of code (but still a prototype)
  - Requires type annotations available in JSR308
- Java Collections case study
  - 77 classes, 33K lines of code
  - Only 85 ownership-related annotations
  - Only 46 immutability-related annotations

Case Study Conclusions

- Verified that collections own their representation
- Method `clone` does not type check
  - It is broken: shallow copy breaks ownership
- Suggestion: *sheep* clone
  - Ownership-aware copy: between shallow and deep copy
  - Compiler-generated `clone` that nullifies fields, and then calls a user-defined copy method

Related Work

- Universes
  - Relaxed owner-as-**dominator** to owner-as-**modifier**
- Reference immutability
  - C++'s `const`
  - Javari
- Initialisation & Immutability
  - X10's proto (gone?), Frozen object, Flexible Initialisation
- Ownership & Immutability
  - JOE_3, MOJO, Delayed Types

Conclusions

- Ownership Immutability Generic Java (OIGJ)
  - Simple, intuitive, static, backward compatible
- Connected ownership & immutability
  - Cyclic immutable structures, factory and visitor design patterns
- Case study showing usefulness
  - No syntax changes or runtime overhead
  - Verified encapsulation of Java collections
  - Few annotations due to *smart defaults*
- Formal proof of soundness (see TR for FOIGJ:
  - https://dspace.mit.edu/handle/1721.1/36850

Follow Up Work

ECOOP 2013

```
class List{
  final int e; final List next;
  List(int e, List' next){this.e=e; this.next=next;}
  }
class ListProducer{
  List' mkAll(int e, int n, List' head){
    if(n==1) return new List(e, head);
    return new List(e, this.mkAll(e+1, n-1, head));
    }
  List make(int e,int n){
    List x = this.mkAll(e, n, x);
    return x;
    }
  }
...
  new ListProducer().make(100,10)
```

The Billion-Dollar Fix

Safe Modular Circular Initialisation with Placeholders and Placeholder Types

Marco Servetto, Julian Mackay, Alex Potanin, and James Noble

Victoria University of Wellington
School of Engineering and Computer Science
{servetto,mackayjuli,alex,kjx}@ecs.vuw.ac.nz

Abstract. Programmers often need to initialise circular structures of objects. Initialisation should be safe (so that programs can never suffer null pointer exceptions or otherwise observe uninitialised values) and modular (so that each part of the circular structure can be written and compiled separately). Unfortunately, existing languages do not support modular circular initialisation: programmers in practical languages resort to Tony Hoare's "Billion Dollar Mistake": initialising variables with nulls, and then hoping to fix them up afterward. While recent research languages have offered some solutions, none fully support safe modular circular initialisation.

We present *placeholders*, a straightforward extension to object-oriented languages that describes circular structures simply, directly, and modularly. In typed languages, placeholders can be described by *placeholder types* that ensure placeholders are used safely. We define an operational semantics for placeholders, a type system for placeholder types, and prove soundness. Incorporating placeholders into object-oriented languages should make programs simultaneously simpler to write, and easier to write correctly.

Aliasing in Object-Oriented Programming. Types, Analysis and Verification pp 233-269 | Cite as

Immutability

Authors | Authors and affiliations

Alex Potanin, Johan Östlund, Yoav Zibin, Michael D. Ernst

Part of the Lecture Notes in Computer Science book series (LNCS, volume 7850)

Abstract

One of the main reasons aliasing has to be controlled, as highlighted in another chapter [1] of this book [2], is the possibility that a variable can unexpectedly change its value without the referrer's knowledge. This book will not be complete without a discussion of the impact of *immutability* on reference-abundant imperative object-oriented languages. In this chapter we briefly survey possible definitions of immutability and present recent work by the authors on adding immutability to object-oriented languages and how it impacts aliasing.

Keywords

Type System | Ownership Type | Subtype Relation | Pure Method | Mutable Reference

These keywords were added by machine and not by the authors. This process is experimental and the keywords may be updated as the learning algorithm improves.

APPENDIX: OIGJ TYPE RULES

OIGJ Typing Rules

- 14 typing rules (see paper and TR)
- Our formalisation uses *generic types* following our previous work
- We will highlight the following:
  - Ownership Nesting
  - Field Access
  - Field Assignment
  - Method Invocation
  - Method Guards

(F)OIGJ Syntax: Fields

```
1:class Foo<O extends World, I extends ReadOnly> {
2:  // An immutable reference to an immutable date.
    Date<O,Immutable> imD = new Date<O,Immutable>();
3:  // A mutable reference to a mutable date.
    Date<O,Mutable> mutD = new Date<O,Mutable>();
4:  // A readonly reference to any date. Both roD and imD cannot mutate
    // their referent, however the referent of roD might be mutated by an
    // alias, whereas the referent of imD is immutable.
    Date<O,ReadOnly> roD = ... ? imD : mutD;
5:  // A date with the same owner and immutability as this
    Date<O,I> sameD;
6:  // A date owned by this; it cannot leak.
    Date<This,I> ownedD;
7:  // Anyone can access this date.
    Date<World,I> publicD;
```

- Every type contains two extra generic parameters

(F)OIGJ Syntax: Methods

```
8 : // Can be called on any receiver; cannot mutate this.
    <I extends ReadOnly>? int readonlyMethod(){...}
9 : // Can be called only on mutable receivers; can mutate this.
    <I extends Mutable>? void mutatingMethod(){...}
10: // Constructor that can create (im)mutable objects.
    <I extends Raw>? Foo(Date<O,I> d) {
11:   this.sameD = d;
12:   this.ownedD = new Date<This,I>();
13:   // Illegal, because sameD came from the outside.
      // this.sameD.setTime(...);
14:   // OK, because Raw is transitive for owned fields.
      this.ownedD.setTime(...);
15: }
```

- Method guard `<T extends U>?` does two things:
  - The method can be called only if `T` extends `U`
  - Inside the method, the bound of `T` is assumed to be `U`

Ownership Nesting

The main owner parameter must be inside any other owner parameter.

```
List<This, I,Date<World,I>> l1; // Legal nesting
List<World,I,Date<This, I>> l2; // Illegal!
```

- It is illegal because we can store l2 in this variable:

```
public static Object<World,ReadOnly> alias_l2;
```

Field Access/Assignment

this-owned fields can be accessed/assigned only via **this**

```
1: class Foo<O extends World, I extends ReadOnly> {
2:   Date<This,I> ownedD; // this-owned field
3:   Date<O,I> sameD;
4:   <I extends Mutable>? void bar(Foo<This,I> other) {
5:     this.ownedD = …; // Legal: assign via this
6:     other.ownedD = …; // Illegal: not via this
7:     other.sameD = …; // Legal: not this-owned
8:   }
```

Field Assignment

1) A field can be assigned only if the object is **Raw** or **Mutable**.
2) If it is **Raw**, then the object must be **this** or **this**-owned.

```
1 : class Foo<O extends World, I extends ReadOnly> {
2 :  Date<O,I> sameD;
3 :  <I extends Raw>? void bar(
4 :        Foo<?,Mutable> mutableFoo,
5 :        Foo<?,ReadOnly> readonlyFoo,
6 :        Foo<?,I> rawFoo1,
7 :        Foo<This,I> rawFoo2) {
8 :    mutableFoo.sameD = …; // Legal: object is Mutable
9 :    readonlyFoo.sameD = …; // Illegal: object is not Raw nor Mutable
10:    rawFoo1.sameD = …; // Illegal: object is not this nor this-owned
11:    rawFoo2.sameD = …; // Legal: object is Raw and this-owned
12:    this.sameD = …; // Legal: object is Raw and this
13:  }
```

Method Invocation

Method invocation is the same as field access/assignment:

1) If any parameter is **`this`**-owned, then the receiver must be **`this`**.
2) If the guard is **`Raw`** and the object is **`Raw`**, then the receiver must be **`this`** or **`this`**-owned.

```
1 : class Foo<O extends World, I extends ReadOnly> {
2 :  Date<This,I> m1() { … } // Parameter is this-owned
3 :  <I extends Raw>? void m2() { … }
4 :  <I extends Raw>? void bar(
5 :        Foo<?,I> rawFoo1,
6 :        Foo<This,I> rawFoo2) {
7 :    this.m1(); // Legal: object is this
8 :    rawFoo2.m1(); // Illegal: object is not this
9 :    rawFoo1.m2(); // Illegal: object is not this nor this-owned
10:    rawFoo2.m2(); // Legal: both Raw and object is this-owned
11:    this.m2(); // Legal: both Raw and object is this
12:  }
```

Method Guards

Guard "**<T extends U>?**" has a dual purpose:

1) The receiver's **T** must be a subtype of **U**.
2) Inside the method, the bound of **T** is **U**.

```
1: class Foo<O extends World, I extends ReadOnly> {
2:   <I extends Raw>? void rawM() { … }
3:   <I extends Mutable>? void bar(
4:           Foo<?,ReadOnly> readonlyFoo,
5:           Foo<?,I> mutableFoo) {
6:     readonlyFoo.rawM(); // Illegal: ReadOnly is not a subtype of Raw
       // The bound of I in this method is Mutable
7:     mutableFoo.rawM(); // Legal: Mutable is a subtype of Raw
8:     this.rawM(); // Legal: Mutable is a subtype of Raw
9:   }
```

Conditional Java (cJ) proposed method guards for Java

Final Points

- We use *smart defaults*:
  - Typically `@O` and `@I`
  - Statics are `@World` and `@Mutable`
- Ownership assumes:
  - Inner classes have access to the outer class (a la Boyapati)
  - Local variables can point at anything as long as the heap invariant is maintained (a la Clarke and Drossopoulou)