The Left Hand of Equals

Portland State
UNIVERSITY

The Left Hand of Equals

James Noble
Victoria University of Wellington
New Zealand
kjx@ecs.vuw.ac.nz

Andrew P. Black
Portland State University
U.S.A.
black@cs.pdx.edu

Kim B. Bruce
Pomona College
U.S.A
kim@cs.pomona.edu

Michael Homer
Victoria University of Wellington
New Zealand
mwh@ecs.vuw.ac.nz

Mark S. Miller
Google Inc.
U.S.A.
erights@google.com

Abstract

When is one object equal to another object? While object *identity* is fundamental to object-oriented systems, object *equality*, although tightly intertwined with identity, is harder to pin down. The distinction between identity and equality is reflected in object-oriented languages, almost all of which provide two variants of "equality", while some provide many more. Programmers can usually override at least one of these forms of equality, and can always define their own methods to distinguish their own objects.

This essay takes a reflexive journey through fifty years of identity and equality in object-oriented languages, and ends somewhere we did not expect: a "left-handed" equality relying on trust and grace.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructors and Features—Classes and objects.

Keywords equality, identity, abstraction, object-orientation

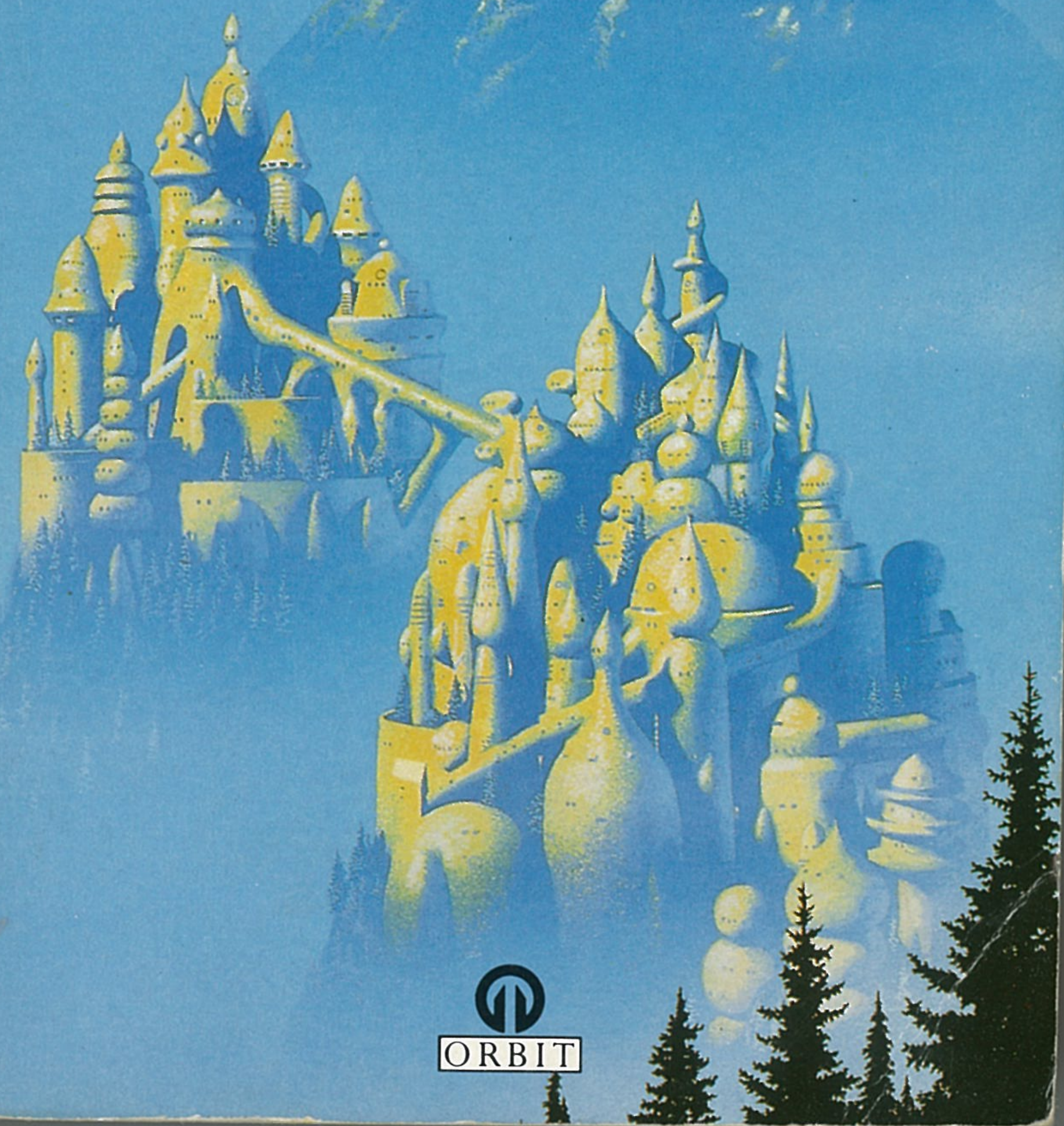
We began with Simula. This is hard to say now, for all of us who came of age in the golden years of programming language design feel in our bones that the world began with Smalltalk. Even though we know it's not so, we cherish the memories of the dusty underground shelf where the library hid the Smalltalk books, of the Tektronix 4404 Smalltalk machine, equipped with a "cat" as well as a "mouse", and of loading Smalltalk-80 off the Apple-branded floppy disks onto a Lisa. So much romance! Meanwhile, down in the basement machine room, Simula had been chugging along happily on the DECSYSTEM-10 since 1975. That Simula system lacked the sexy graphics of the Lisa and the 4404, but did offer an online debugging facility with breakpoints that has evolved but slightly into the debuggers of today.

We finish with Grace. Or perhaps: we hope to finish with grace, to finish gracefully. Much of our recent professional lives have been occupied with the design of a new object-oriented language — Grace — intended be useful in education (Black et al. 2012). Grace follows in the tradition of Simula, Smalltalk, Self, Basic, and Pascal, mixing in Java, Ruby, Python, Newspeak and many other languages. If this

URSULA LeGUIN

The Left Hand of Darkness

A CLASSIC OF SCIENCE FICTION



URSULA LeGUIN The Left Hand of Darkness

1: A PARADE IN ERHENRANG

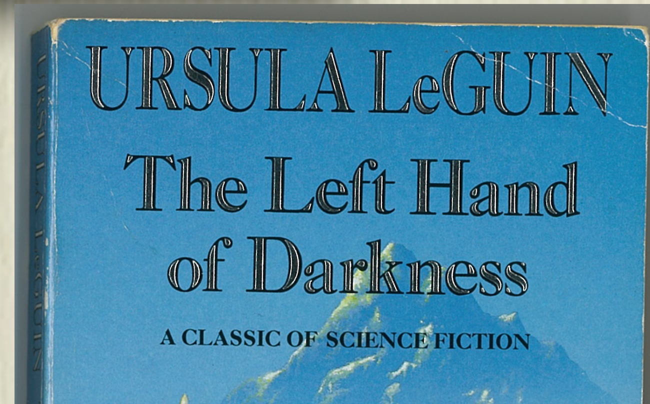
From the Archives of Hain. Transcript of Ansible Document 01-01101-934-2-Gethen: To the Stabile on Ollul: Report from Genly Ai, First Mobile on Gethen/Winter, Hainish Cycle 93 Ekumenical Year 1490–97.

I'll make my report as if I told a story, for I was taught as a child on my homeworld that Truth is a matter of the imagination. The soundest fact may fail or prevail in the style of its telling: like that singular organic jewel of our seas, which grows brighter as one woman wears it and, worn by another, dulls and goes to dust. Facts are no more solid, coherent, round, and real than pearls are. But both are sensitive.

1: A PARADE IN ERHENRANG

From the Archives of Hain. Transcript of Ansible Document 01-01101-934-2-Gethen: To the Stabile on Ollul: Report from Genly Ai, First Mobile on Gethen/Winter, Hainish Cycle 93 Ekumenical Year 1490–97.

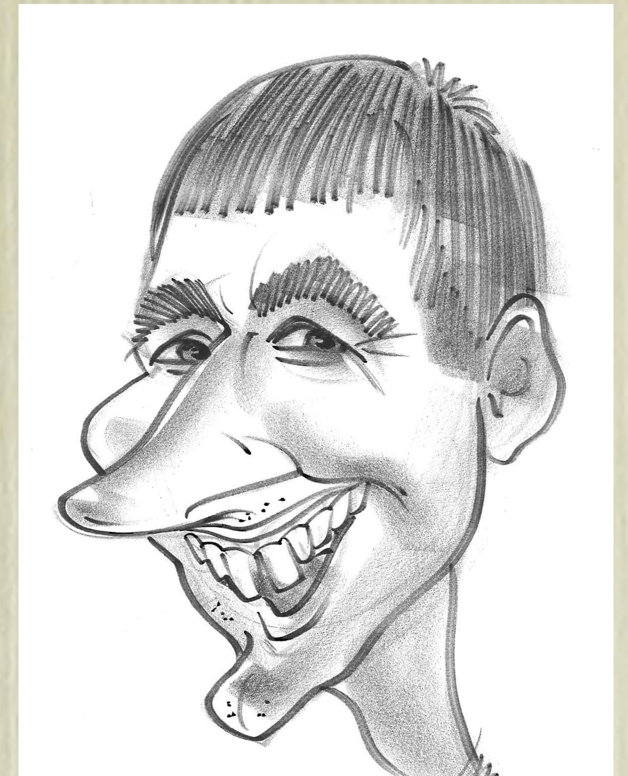
I'll make my report as if I told a story, for I was taught as a child on my homeworld that Truth is a matter of the imagination. The soundest fact may fail or prevail in the style of its telling: like that singular organic jewel of our seas, which grows brighter as one woman wears it and, worn by another, dulls and goes to dust. Facts are no more solid, coherent, round, and real than pearls are. But both are sensitive.



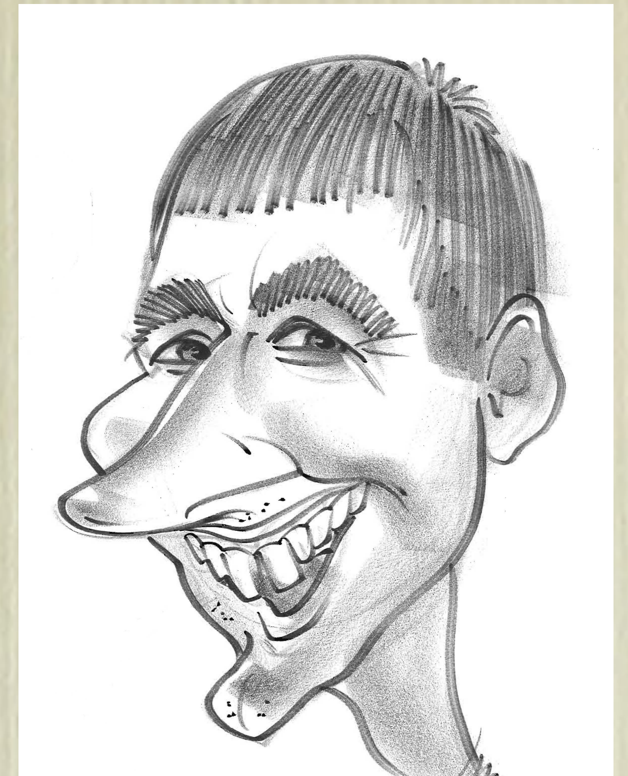
Portland State
UNIVERSITY

Prof Stéph:

Prof Andrew:



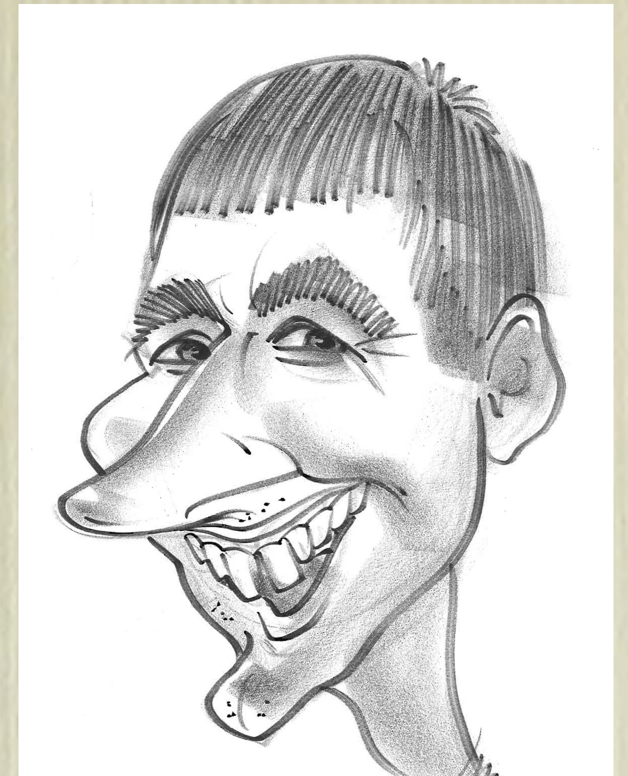
Prof Stéph: Prof Andrew:



Tell us about equality

Prof Stéph:

Prof Andrew:

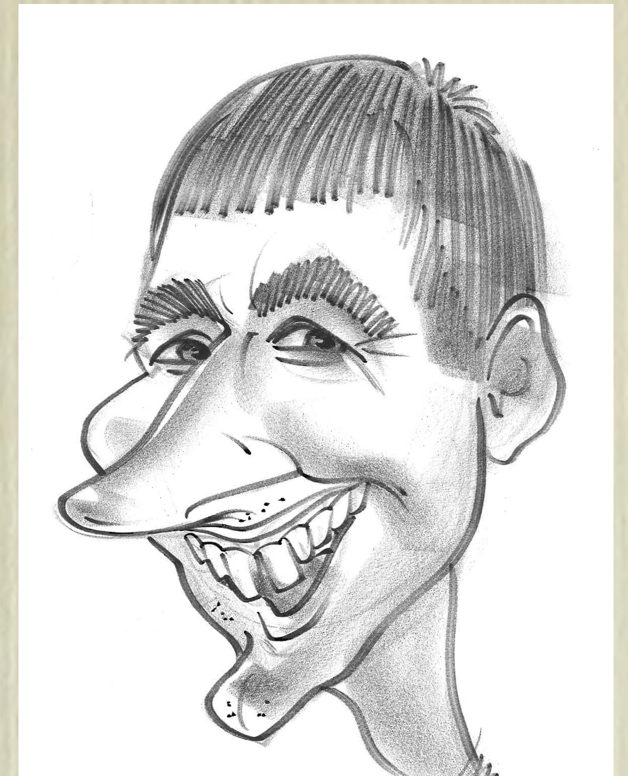


Tell us about equality

It's an essay; read it

Prof Stéph:

Prof Andrew:



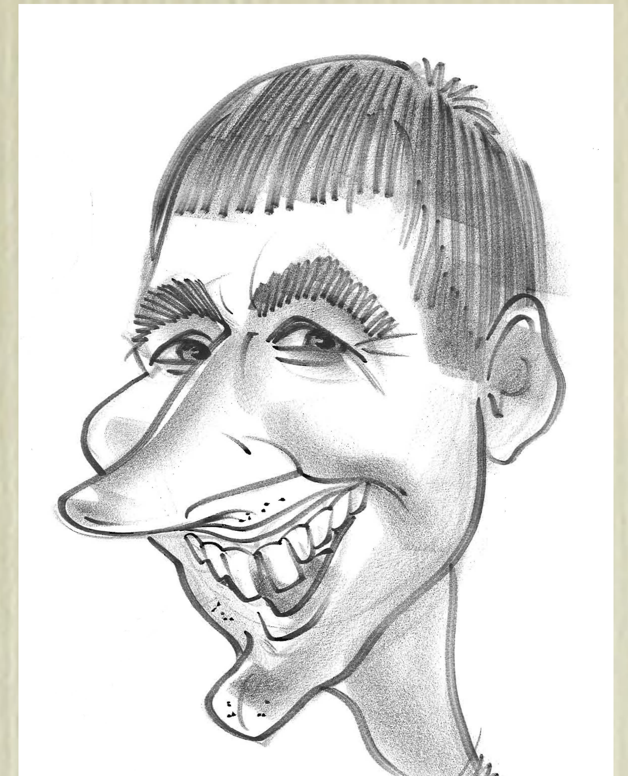
Tell us about equality

It's an essay; read it

TL; DR

Prof Stéph:

Prof Andrew:



Tell us about equality

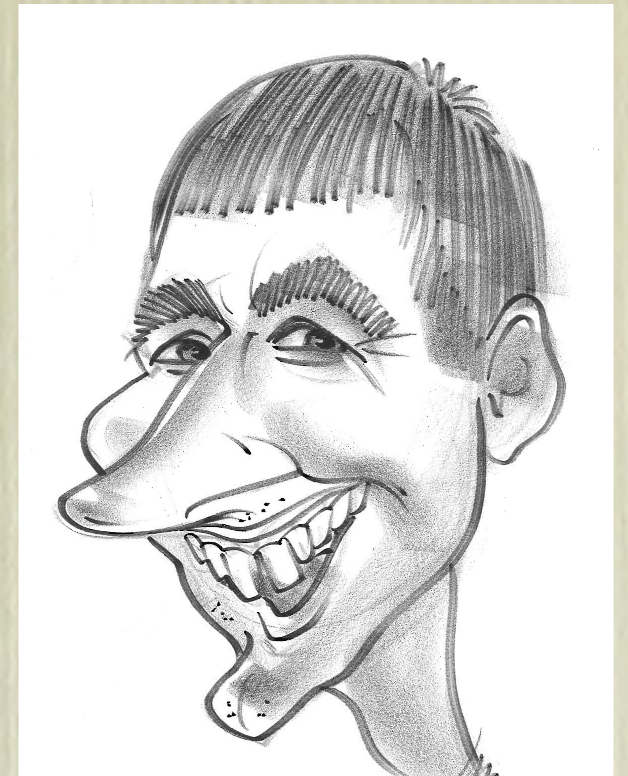
It's an essay; read it

TL; DR

I read it aloud at SPLASH

Prof Stéph:

Prof Andrew:



Tell us about equality

It's an essay; read it

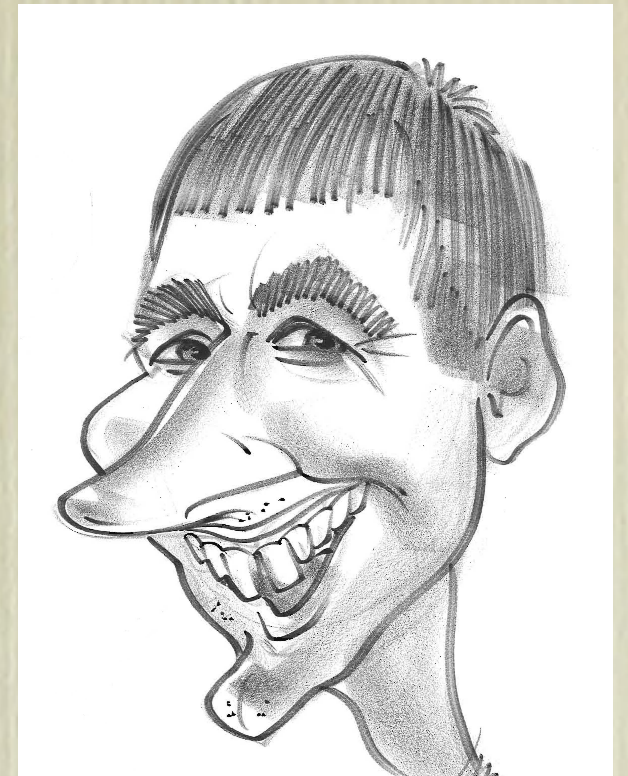
TL; DR

I read it aloud at SPLASH

Give us just the essence

Prof Stéph:

Prof Andrew:



Tell us about equality

It's an essay; read it

TL; DR

I read it aloud at SPLASH

Give us just the essence

OK. I'll try to do that

We begin with Simula ...

SIMULA BEGIN

Graham M Birtwistle
Ole-Johan Dahl
Bjørn Myhrhaug
Kristen Nygaard

Studentlitteratur
Box 1719, S-221 01 Lund, Sweden

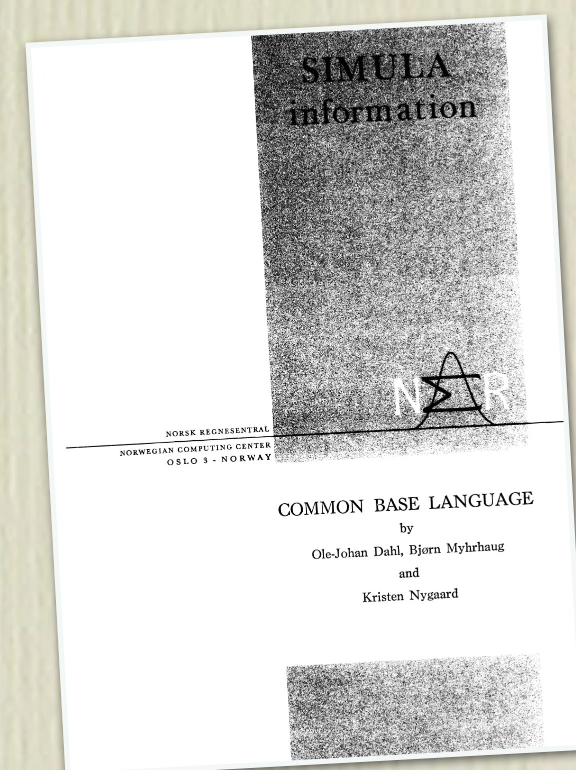
Bratt Institut für Neues Lernen

Chartwell-Bratt Ltd
Old Orchard, Bickley Road, Bromley, Kent BR1 2NE,

Simula

Associated with an object there is a unique "object reference" which identifies the object. … Two object references X and Y are said to be "identical" if they refer to the same object.

SIMULA-67 Common Base Standard (1970) (Dahl et al. 1970).



Simula

Simula

==   =/=

Reference Equality

Do two references refer to the same object?

Simula

==   =/=

Reference Equality

Do two references refer to the same object?

=   /=

Value Equality

Do two objects contain equal values?

Portland State University

Simula

Simula

:=

Value Assignment

Portland State UNIVERSITY

Simula

: -

Reference Assignment

:=

Value Assignment

Families of Operators

- For each equality operator **=** , we assume:
  - an inequality operator **≠**
  - an operation **hash**
- with the usual properties:
  - $a \neq b \triangleq \neg (a = b)$
  - a = b ⇒ a **hash** = b **hash**
- I won't mention this again

Simula

Simula keeps things *mostly* straightforward:

- no *value* equality (or assignment) for objects
- no *reference* equality (or assignment) for values

So:

- objects use reference equality
- numbers and characters use value equality

Simula Strings

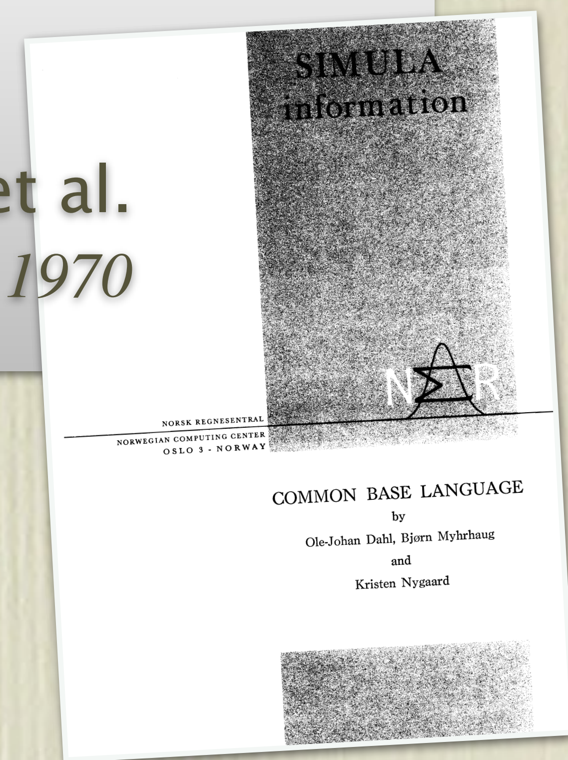
Strings (`texts`) have both reference equality and value equality

Simula Strings

Strings (`texts`) have both reference equality and value equality

If T and U are texts, then "the relations T=U and T=/=U may both have the value true"

Dahl et al.
SIMULA Common Base Language, 1970

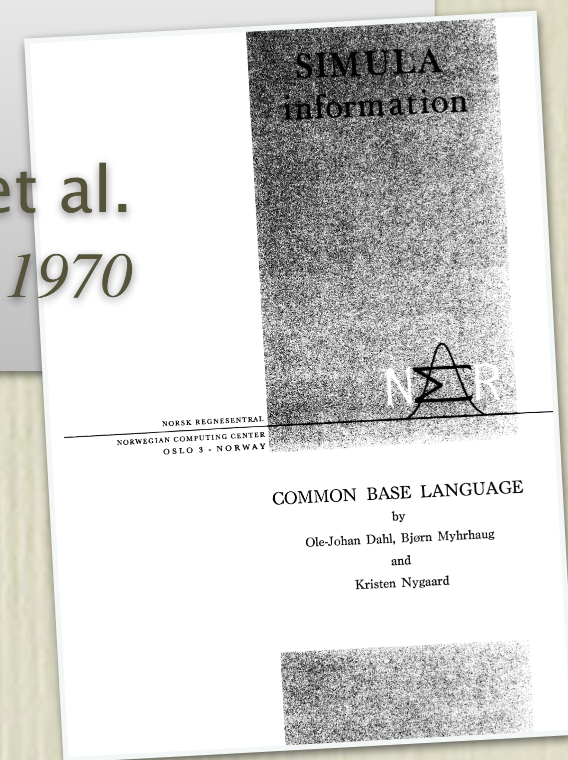


Simula Strings

Strings (`texts`) have both reference equality and value equality

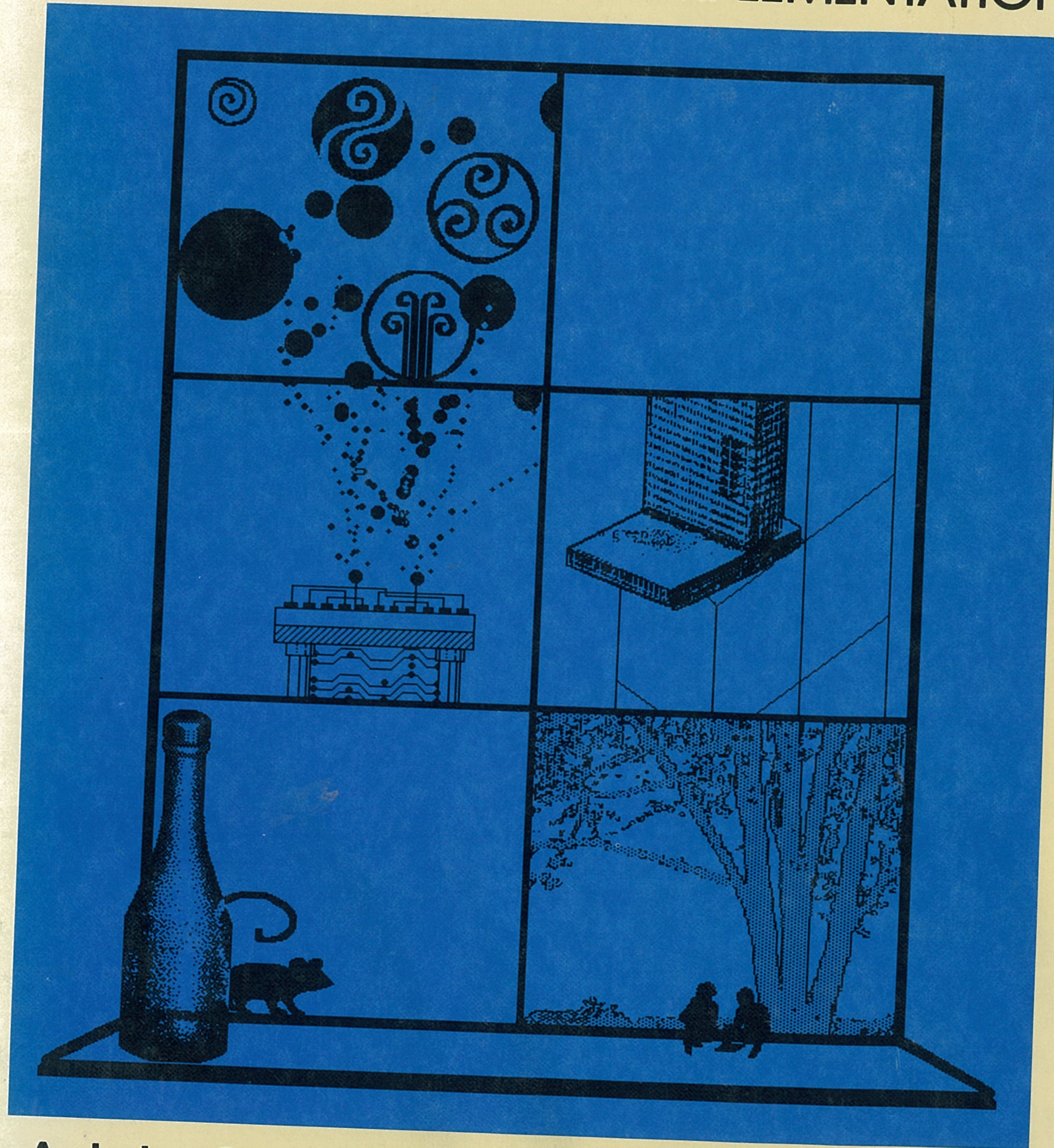
If T and U are texts, then "the relations T=U and T=/=U may both have the value true"

Dahl et al.
SIMULA Common Base Language, 1970



"Here is the worm in our garden of Eden"

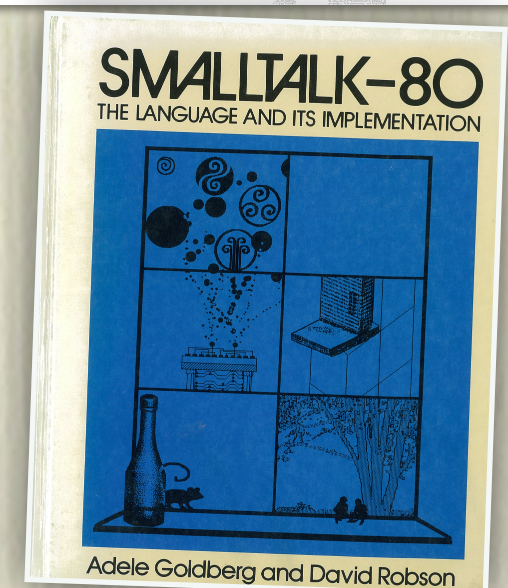
On to Smalltalk


SMALLTALK-80
THE LANGUAGE AND ITS IMPLEMENTATION
Adele Goldberg and David Robson

Smalltalk

Comparing Objects

Since all information in the system is represented as objects, there is a basic protocol provided for testing the identity of an object and for copying objects. The important comparisons specified in class Object are equivalence and equality testing. Equivalence (==) is the test of whether two objects are the same object. Equality (=) is the test of whether two objects represent the same component. The decision as to what it means to be "represent the same component" is made by the receiver of the message; each new kind of object that adds new instance variables typically must reimplement the = message in order to specify which of its instance variables should enter into the test of equality. For example, equality of two arrays is determined by checking the size of the arrays and then the equality of each of the elements of the arrays; equality of two numbers is determined by testing whether the two numbers represent the same value; and equality of two bank accounts might rest solely on the equality of each account identification number.



Adele Goldberg and David Robson
Smalltalk: The Language and its Implementation

Portland State UNIVERSITY

Smalltalk

Smalltalk

== ~~

Reference Equality

Do two references refer to the same object?

Smalltalk

== ~~

Reference Equality

Do two references refer to the same object?

= ~=

Value Equality

Do two objects contain equal values?

Smalltalk

==   ~~

Reference Equality

Do two references refer to the same object?

=   ~=

Value Equality

Do two objects contain equal values?

Smalltalk

== ~~

Reference Equality

Do two references refer to the same object?

= ~=

Abstract Equality

Do two objects represent the same abstract value?

Lisp and EGAL

Lisp and EGAL

6.3. Equality Predicates

Common Lisp provides a spectrum of predicates for testing for equality of two objects: `eq` (the most specific), `eql`, `equal`, and `equalp` (the most general). `eq` and `equal` have the meanings traditional in Lisp. `eql` was added because it is frequently needed, and `equalp` was added primarily in order to have a version of `equal` that would ignore type differences when comparing numbers and case differences when comparing characters. If two objects satisfy any one of these equality predicates, then they also satisfy all those that are more general.

GUY L. STEELE JR.
COMMON LISP
THE LANGUAGE
SECOND EDITION

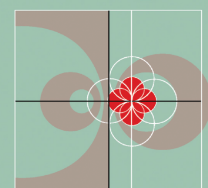
`eq` *x* *y* [*Function*]

`(eq` *x* *y*`)` is true if and only if *x* and *y* are the same identical object. (Implementationally, *x* and *y* are usually `eq` if and only if they address the same identical memory location.)

`eql` *x* *y* [*Function*]

The `eql` predicate is true if its arguments are `eq`, or if they are numbers of the same type with the same value, or if they are character objects that represent the same character. For example:

GUY L. STEELE JR.
COMMON
LISP



THE LANGUAGE
SECOND EDITION

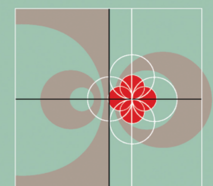
Portland State
UNIVERSITY

It should be noted that things that print the same are not necessarily `eq` to each other. Symbols with the same print name usually are `eq` to each other because of the use of the `intern` function. However, numbers with the same value need not be `eq`, and two similar lists are usually not `eq`. For example:

In Common Lisp, unlike some other Lisp dialects, the implementation is permitted to make "copies" of characters and numbers at any time. (This permission is granted because it allows tremendous performance improvements in many common situations.) The net effect is that Common Lisp makes no guarantee that `eq` will be true even when both its arguments are "the same thing" if that thing is a character or number. For example:

`(let ((x 5)) (eq x x))` might be true or false.

GUY L. STEELE JR.
COMMON
LISP

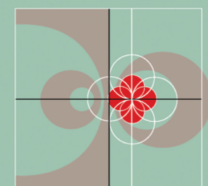


THE LANGUAGE
SECOND EDITION

It should be noted that things that print the same are not necessarily `eq` to each other. Symbols with the same print name usually are `eq` to each other because of the use of the `intern` function. However, numbers with the same value need not be `eq`, and two similar lists are usually not `eq`. For example:

In Common Lisp, unlike some other Lisp dialects, the implementation is permitted to make "copies" of characters and numbers at any time. (This permission is granted because it allows tremendous performance improvements in many common situations.) The net effect is that Common Lisp makes no guarantee that `eq` will be true even when both its arguments are "the same thing" if that thing is a character or number. For example:

`(let ((x 5)) (eq x x))` might be true or false.

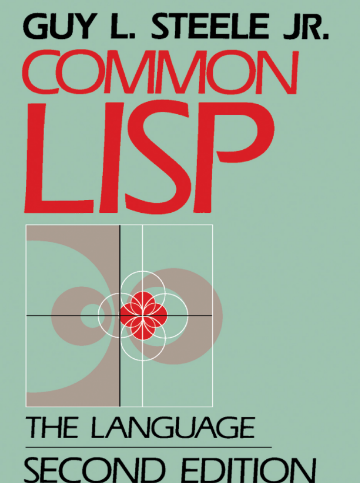


It should be noted that things that print the same are not necessarily `eq` to each other. Symbols with the same print name usually are `eq` to each other because of the use of the `intern` function. However, numbers with the same value need not be `eq`, and two similar lists are usually not `eq`. For example:

In Common Lisp, unlike some other Lisp dialects, the implementation is permitted to make "copies" of characters and numbers at any time. (This permission is granted because it allows tremendous performance improvements in many common situations.) The net effect is that Common Lisp makes no guarantee that `eq` will be true even when both its arguments are "the same thing" if that thing is a character or number. For example:

`(let ((x 5)) (eq x x))` might be true or false.

They should have read ...



Efficient Message Handling

It should be obvious that to [implement the system the way that it is specified] … will result in a slow system … The implementor must cheat, but not get caught. In other words, the user of the system should not perceive any non-uniformity.

Efficient Message Handling

It should be obvious that to [implement the system the way that it is specified] … will result in a slow system … The implementor must cheat, but not get caught. In other words, the user of the system should not perceive any non-uniformity.

Dan Ingalls,
The Smalltalk-76 Programming System
POPL V (1978)

Baker’s EGAL: one equality operator to rule them all

OOPS MESSENGER

A Quarterly Publication of the Special Interest Group on Programming Languages

VOLUME 4 NUMBER 4 OCTOBER 1993

CONTENTS:

A Short Note from the Editor 1

TECHNICAL CONTRIBUTIONS:

Henry G. Baker: : Equal Rights for Functional Objects or, The More Things Change, The More They are the Same 2

June Power : The Object in Perspective 28

Pei-Chi Wu, Feng-Jian Wang : Applying Classification and Inheritance into Compiling 33

W. Craig Scratchley: : Using Smalltalk for Wait-Free Implementation of Highly-Concurrent Objects 44

VOLUME 4 NUMBER 4 OCTOBER 1993

A Short Note from the Editor 1

TECHNICAL CONTRIBUTIONS:

Henry G. Baker: : Equal Rights for Functional Objects or, The More Things Change, The More They are the Same 2

June Power : The Object in Perspective 28

Pei-Chi Wu, Feng-Jian Wang : Applying Classification and Inheritance into Compiling 33

W. Craig Scratchley: : Using Smalltalk for Wait-Free Implementation of Highly-Concurrent Objects 44

VOLUME 4 NUMBER 4 OCTOBER 1993

A Short Note from the Editor 1

TECHNICAL CONTRIBUTIONS:

Henry G. Baker: : Equal Rights for Functional Objects or, The More Things Change, The More They are the Same 2

June Power : The Object in Perspective 28

Pei-Chi Wu, Feng-Jian Wang : Applying Classification and Inheritance into Compiling 33

W. Craig Scratchley: : Using Smalltalk for Wait-Free Implementation of Highly-Concurrent Objects 44

Equal Rights for Functional Objects[1] or, The More Things Change, The More They Are the Same[2]

Henry G. Baker

Nimble Computer Corporation
16231 Meadow Ridge Way, Encino, CA 91436 (818) 501-4956 (818) 986-1360 FAX

August, October, 1990, October, 1991, and April, 1992

This work was supported in part by the U.S. Department of Energy Contract No. DE-AC03-88ER80663

We argue that intensional *object identity* in object-oriented programming languages and databases is best defined operationally by side-effect semantics. A corollary is that "functional" objects have extensional semantics. This model of object identity, which is analogous to the normal forms of relational algebra, provides cleaner semantics for the value-transmission operations and built-in primitive equality predicate of a programming language, and eliminates the confusion surrounding "call-by-value" and "call-by-reference" as well as the confusion of multiple equality predicates.

Implementation issues are discussed, and this model is shown to have significant performance advantages in persistent, parallel, distributed and multilingual processing environments. This model also provides insight into the "type equivalence" problem of Algol-68, Pascal and Ada.

1. INTRODUCTION

Parallel, distributed and persistent programming languages are leaving the laboratories for more wide-spread use. Due to the substantial differences between the semantics and implementations of these languages and traditional serial programming languages, however, some of the most basic notions of programming languages must be refined to allow efficient, *portable* implementations. In this paper, we are concerned with defining *object identity* in parallel, distributed and persistent systems in such a way that the intuitive semantics of serial implementations are transparently preserved. Great hardware effort and expense—e.g., cache coherency protocols for shared memory multiprocessors—are the result of this desire for transparency. Yet much of the synchronization cost of these protocols is wasted on *functional/immutable* objects, which do not have a coherency problem. If programming languages distinguish functional/immutable objects from non-functional/mutable objects, and if programs utilize a "mostly functional" style, then such programs will be efficient even in a non-shared-memory ("message-passing") implementation. Since it is likely that a cache-coherent shared-memory paradigm will not apply to a large fraction of distributed and persistent applications, our treatment of object identity provides increased cleanliness and efficiency even for non-shared-memory applications.

The most intuitive notion of object identity is offered by simple Smalltalk implementations in which "everything is a pointer". In these systems, an "object" is a sequence of locations in memory, and all "values" are homogeneously implemented as addresses (pointers) of such "objects". There are several serious problems with this model. First, "objects" in two different locations may have the same bit pattern both representing the integer "9"; an implementation must either make sure that copies like this cannot happen, or fix the equality comparison to dereference the pointers in this case. Second, the "everything is a pointer" model often entails an "everything is heap-allocated" policy, with its attendant overheads; an efficient implementation might wish to manage small fixed-size "things" like complex floating point numbers directly, rather than through pointers. Third, read access to the bits of an object may become a bottleneck in a multiprocessor environment due to locking and memory contention, even when the object is functional/immutable and could be transparently copied. In light of these problems, we seek a more efficient and less implementation-dependent notion of object identity than that of an address in a random-access computer memory.

A more efficient, but also more confusing, notion of object identity is offered by languages such as Pascal, Ada and C. These languages can be more efficient because they directly manipulate values other than pointers. This efficiency is gained, however, at the cost of an implementation-dependent notion of object identity. To a first

---

[1]"Functional objects" is triply overloaded, meaning immutable objects, function closures or objects with functional dependencies.

[2]*Plus ça change, plus c'est la même chose*—Alphonse Karr, as translated in [Cohen60,p.214].

3. OUR MODEL OF OBJECT IDENTITY

A. Mutability Definition of Object Identity

Our model for object identity is similar to Scheme's concept of "operational identity" [Rees86], in which objects which *behave* the same should *be* the same. However, since "behave the same" is undecidable for functions and function-closures, we back down from "operational identity" to "operational identity of data structure representations". Operational identity for data structures is much easier than operational identity for function-closures, because there are only a few well-defined operations on data structures, but function-closures can do anything. We define a single, computable, primitive equality predicate called EGAL which we show is consistent with the notion of "operational identity of data structures". *Egal* is the obsolete Norman term for *equal*, and *Égalité* is the French word for social equality. "During the seventeenth century *two parallel vertical lines* were frequently used [to denote equality], especially in France, instead of =" [Young11]; we will later find that || is a remarkably satisfying infix symbol for *egal*.

Our model for object identity distinguishes mutable objects from immutable objects, and mutable components of aggregate objects from immutable components. We consider an immutable component of an object to be an integral part of the object's identity, since it cannot be separated from the object. Unlike a normalized (factored) relational database, which attempts to *minimize* the size of a "key" which holds the essence of an entity [Ullman80,s.5.4], we *maximize* the size of the object "key" to include all of its static components. Because these components are static, we cannot create any "update anomalies" with this policy. In particular, this object identity can be used as a key to a Common Lisp hash table [Steele90,p.435], and no hash entries will become inaccessible as a result of a key element being modified.

3. OUR MODEL OF OBJECT IDENTITY

A. Mutability Definition of Object Identity

Our model for object identity is similar to Scheme's concept of "operational identity" [Rees86], in which objects which *behave* the same should *be* the same. However, since "behave the same" is undecidable for functions and function-closures, we back down from "operational identity" to "operational identity of data structure representations". Operational identity for data structures is much easier than operational identity for function-closures, because there are only a few well-defined operations on data structures, but function-closures can do anything. We define a single, computable, primitive equality predicate called `EGAL` which we show is consistent with the notion of "operational identity of data structures". *Egal* is the obsolete Norman term for *equal*, and *Égalité* is the French word for social equality. "During the seventeenth century *two parallel vertical lines* were frequently used [to denote equality], especially in France, instead of =" [Young11]; we will later find that || is a remarkably satisfying infix symbol for *egal*.

Our model for object identity distinguishes mutable objects from immutable objects, and mutable components of aggregate objects from immutable components. We consider an immutable component of an object to be an integral part of the object's identity, since it cannot be separated from the object. Unlike a normalized (factored) relational database, which attempts to *minimize* the size of a "key" which holds the essence of an entity [Ullman80,s.5.4], we *maximize* the size of the object "key" to include all of its static components. Because these components are static, we cannot create any "update anomalies" with this policy. In particular, this object identity can be used as a key to a Common Lisp hash table [Steele90,p.435], and no hash entries will become inaccessible as a result of a key element being modified.

EGAL

Compare mutable objects with reference equality, and immutable objects with value equality.

EGAL

Compare mutable objects with reference equality, and immutable objects with value equality.

EGAL is *stable*:

- it does not depend on mutable state
- it's an *always equal* operator

Clojure

Clojure

EGAL
`=`

Reference Equality
`identical`

Clojure

EGAL
`=`

Reference Equality
`identical`

Pyret

Reference Equality
`<=>` `identical`

Value Equality
`=~` `equals-now`

EGAL
`=` `equals-always`

The *intention* of equality

- Should some (or all) equality operators be equivalence relations?
- What does it mean for two objects to be equal?

Java Specifies an Intention:

`public boolean equals(Object obj)`

Indicates whether some other object is "equal to" this one.

The equals method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return true.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return true if and only if `y.equals(x)` returns true.
- It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` should return true.
- It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return false.

...

Returns:

true if this object is the same as the obj argument; false otherwise.

The Java Platform (Gosling et al. 2005)
(our underlining)

Portland State UNIVERSITY

Java's specification allows objects to be equal when they are *not* EGAL

e.g., two distinct mutable objects can never be EGAL, but they can be `equals`

The First Lesson

Specifying an intention

conditioned on no object changing its state

is not very useful in a world of

stateful objects

Pragmatics of Equality

- Once a programmer knows that two objects are equal, what can they assume?
- Cook, and Ungar, argue that *equal always* should mean bi-simulation

equal-always

equal-always

- Given two references, A and B, $A == B$ implies that for any message m, you could send m to A or send m to B and there would be no observable change in the future response to messages of the system.

equal-always

- Given two references, *A* and *B*, *A* == *B* implies that for any message *m*, you could send *m* to *A* or send *m* to *B* and there would be no observable change in the future response to messages of the system.
  - This is *equal-always,* but not necessarily reference equality

equal-always

- Given two references, *A* and *B*, *A* == *B* implies that for any message *m*, you could send *m* to *A* or send *m* to *B* and there would be no observable change in the future response to messages of the system.
  - This is *equal-always,* but not necessarily reference equality

A

equal-always

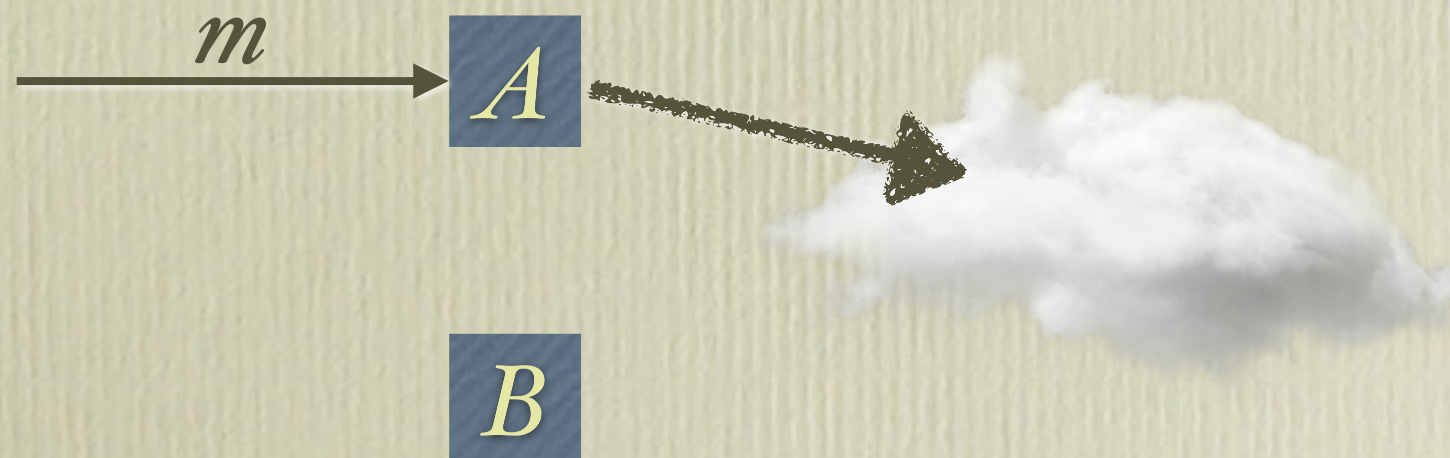
- Given two references, *A* and *B*, *A* == *B* implies that for any message *m*, you could send *m* to *A* or send *m* to *B* and there would be no observable change in the future response to messages of the system.
  - This is *equal-always,* but not necessarily reference equality

equal-always

- Given two references, *A* and *B*, *A* == *B* implies that for any message *m*, you could send *m* to *A* or send *m* to *B* and there would be no observable change in the future response to messages of the system.
  - This is *equal-always,* but not necessarily reference equality

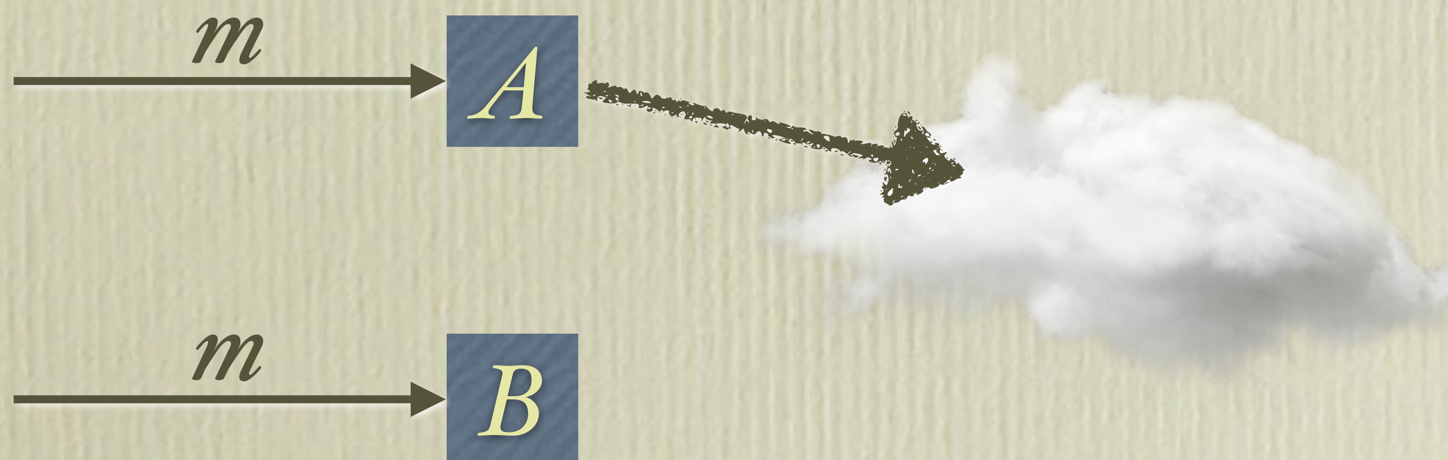
equal-always

- Given two references, *A* and *B*, *A* == *B* implies that for any message *m*, you could send *m* to *A* or send *m* to *B* and there would be no observable change in the future response to messages of the system.
  - This is *equal-always,* but not necessarily reference equality



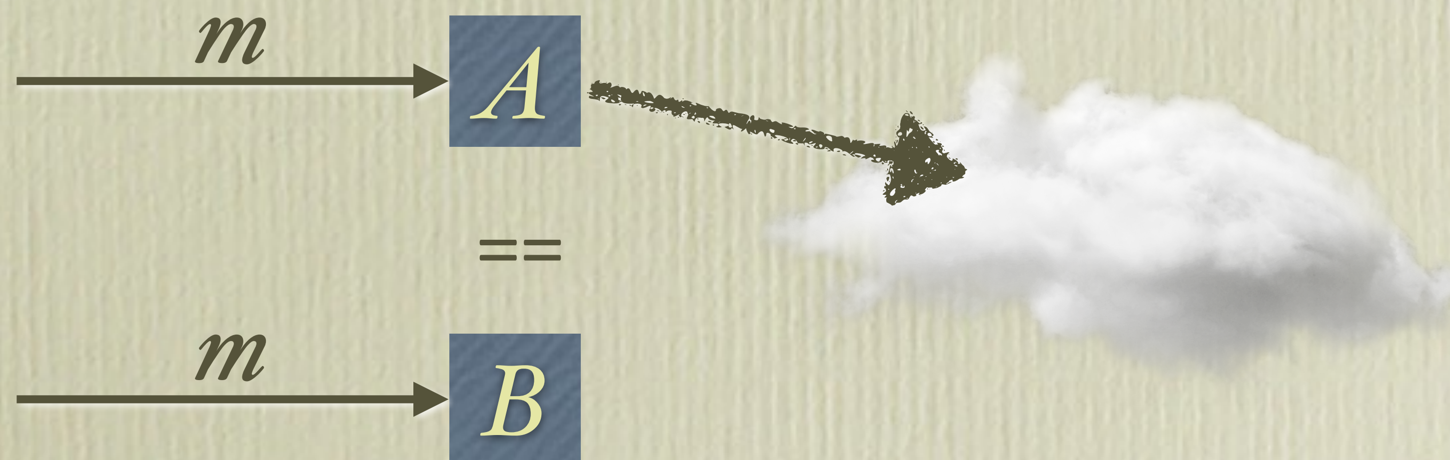
equal-always

- Given two references, *A* and *B*, *A* == *B* implies that for any message *m*, you could send *m* to *A* or send *m* to *B* and there would be no observable change in the future response to messages of the system.
  - This is *equal-always,* but not necessarily reference equality



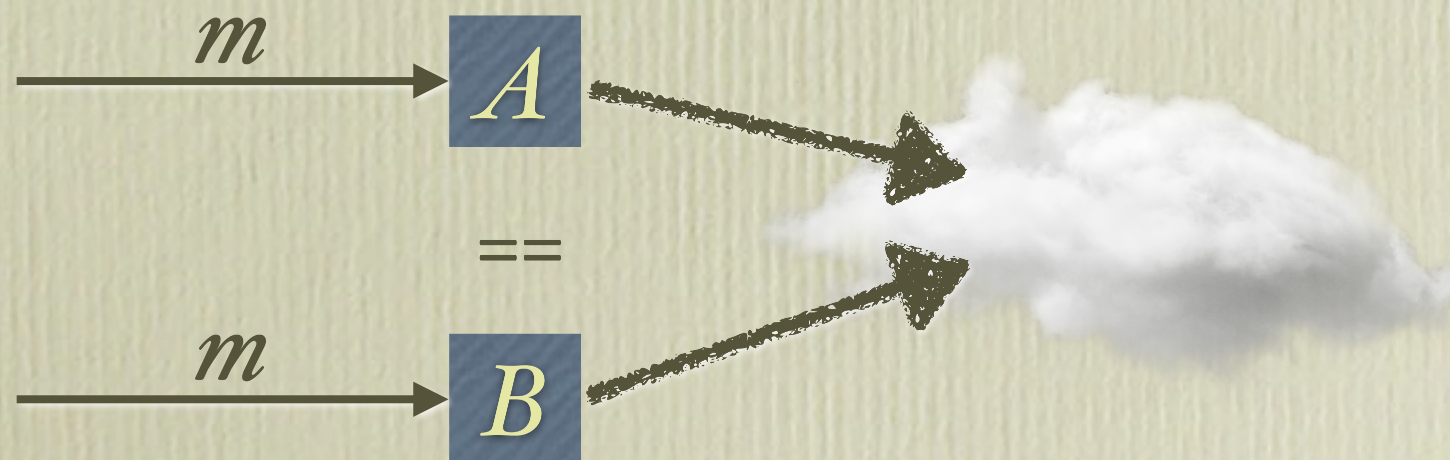
equal-always

- Given two references, *A* and *B*, *A* == *B* implies that for any message *m*, you could send *m* to *A* or send *m* to *B* and there would be no observable change in the future response to messages of the system.
  - This is *equal-always,* but not necessarily reference equality



equal-always

- Given two references, *A* and *B*, *A* == *B* implies that for any message *m*, you could send *m* to *A* or send *m* to *B* and there would be no observable change in the future response to messages of the system.
  - This is *equal-always,* but not necessarily reference equality



equal-now

- Weaker: equal-always ⇒ equal-now
  - Given two references, A and B, A is equal-now to B implies that if you sent m to A and received R as the result, or if you sent m to B and received S as the result, R and S would be equal now.

equal-now

- Weaker: equal-always ⇒ equal-now
  - Given two references, A and B, A is equal-now to B implies that if you sent m to A and received R as the result, or if you sent m to B and received S as the result, R and S would be equal now.

A

equal-now

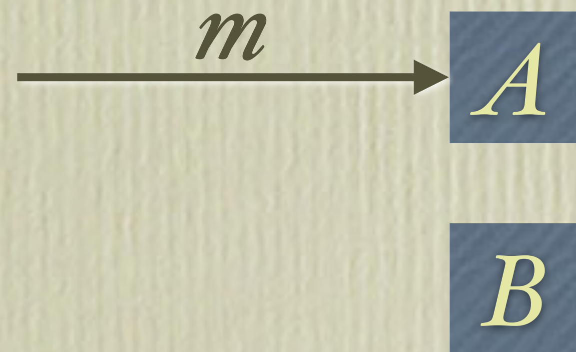
- Weaker: equal-always ⇒ equal-now
  - Given two references, A and B, A is equal-now to B implies that if you sent m to A and received R as the result, or if you sent m to B and received S as the result, R and S would be equal now.

A

B

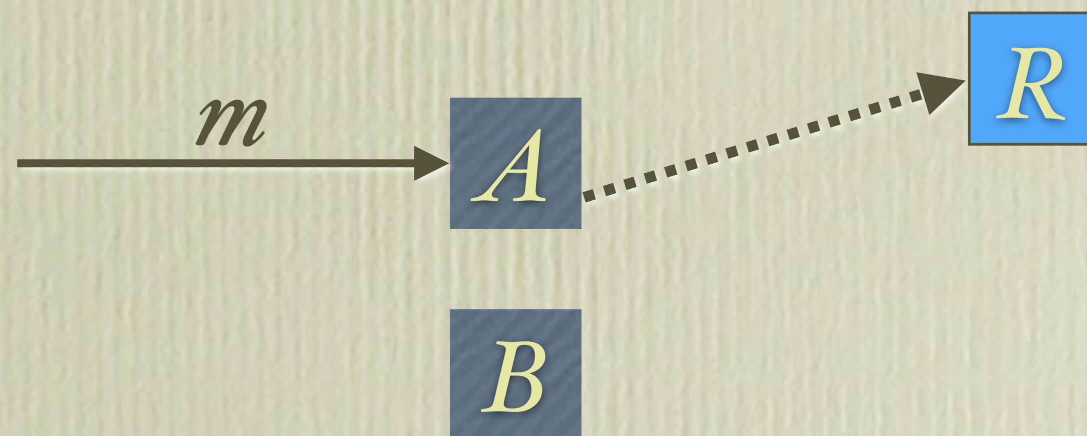
equal-now

- Weaker: equal-always ⇒ equal-now
  - Given two references, A and B, A is equal-now to B implies that if you sent m to A and received R as the result, or if you sent m to B and received S as the result, R and S would be equal now.



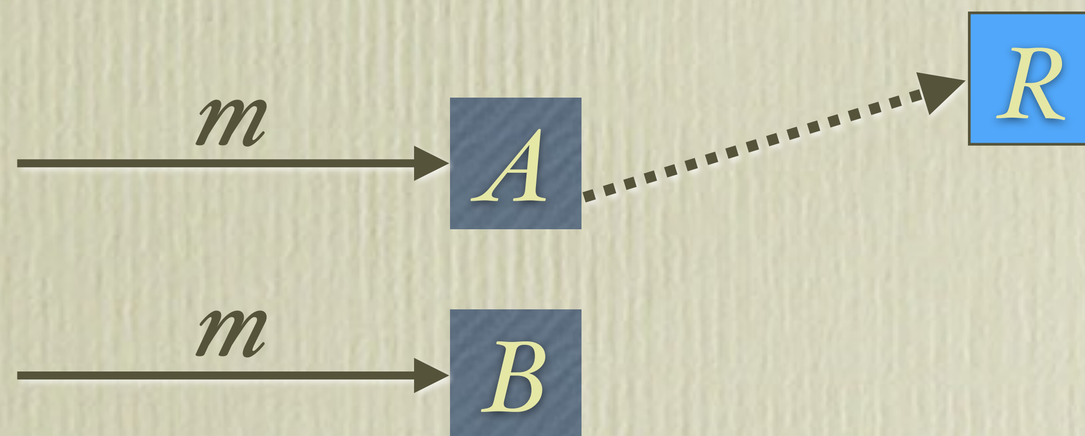
equal-now

- Weaker: equal-always ⇒ equal-now
  - Given two references, A and B, A is equal-now to B implies that if you sent m to A and received R as the result, or if you sent m to B and received S as the result, R and S would be equal now.



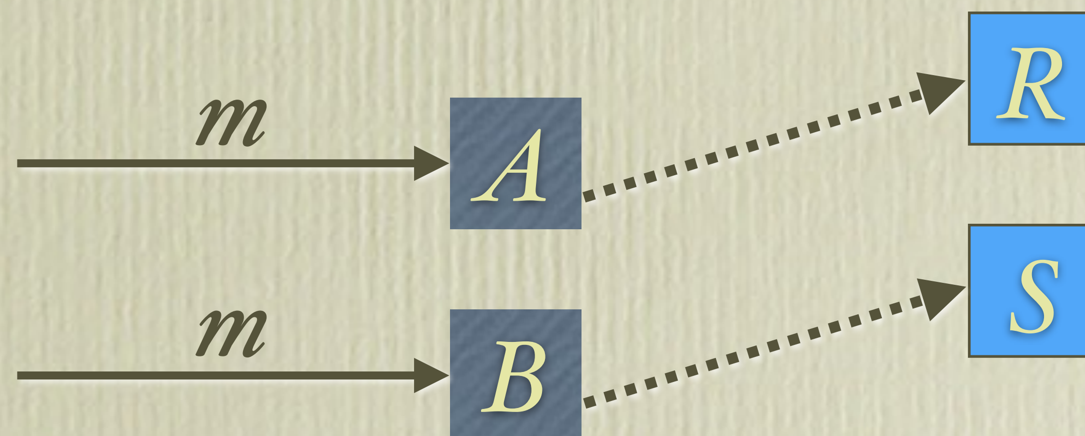
equal-now

- Weaker: equal-always ⇒ equal-now
  - Given two references, A and B, A is equal-now to B implies that if you sent m to A and received R as the result, or if you sent m to B and received S as the result, R and S would be equal now.



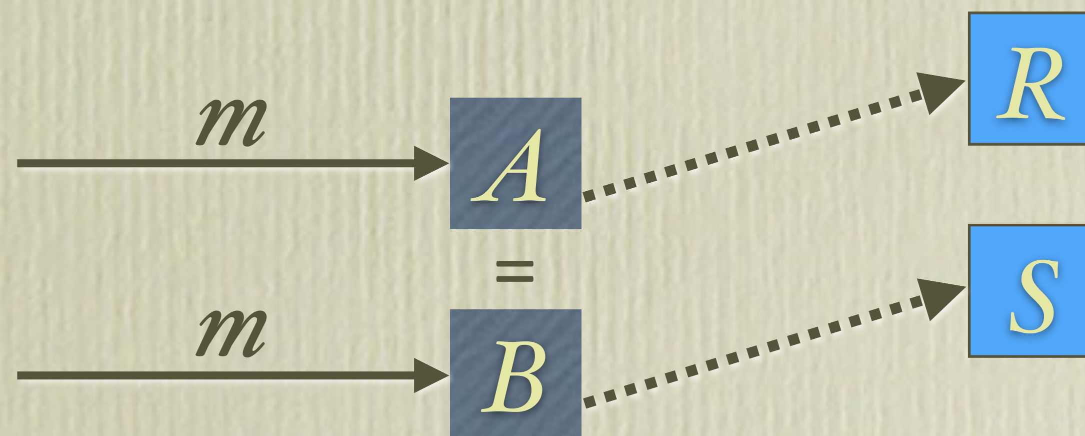
equal-now

- Weaker: equal-always ⇒ equal-now
  - Given two references, A and B, A is equal-now to B implies that if you sent m to A and received R as the result, or if you sent m to B and received S as the result, R and S would be equal now.



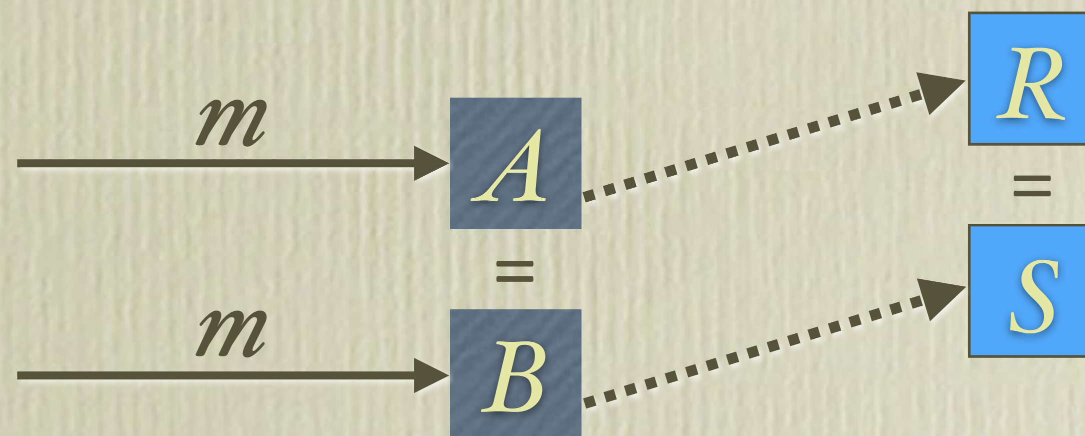
equal-now

- Weaker: equal-always ⇒ equal-now
  - Given two references, A and B, A is equal-now to B implies that if you sent m to A and received R as the result, or if you sent m to B and received S as the result, R and S would be equal now.



equal-now

- Weaker: equal-always ⇒ equal-now
  - Given two references, A and B, A is equal-now to B implies that if you sent m to A and received R as the result, or if you sent m to B and received S as the result, R and S would be equal now.



Object-Orientation

O is for Object,
which is the granddaddy of all soap bubbles.

Brian Alexander, ABC's for object-gifted children,
(Alexander 1992)

objects have identity, methods and state

objects have identity, methods and state

objects have identity, methods and state

Portland State UNIVERSITY

- The Scandinavian view:
  - an OO system is one whose creators realise that programming is modelling.

- The Scandinavian view:
  - an OO system is one whose creators realise that programming is modelling.
- The mystical view:
  - an OO system is one that is built out of objects that communicate by sending messages to each other, and computation is the messages flying from object to object.

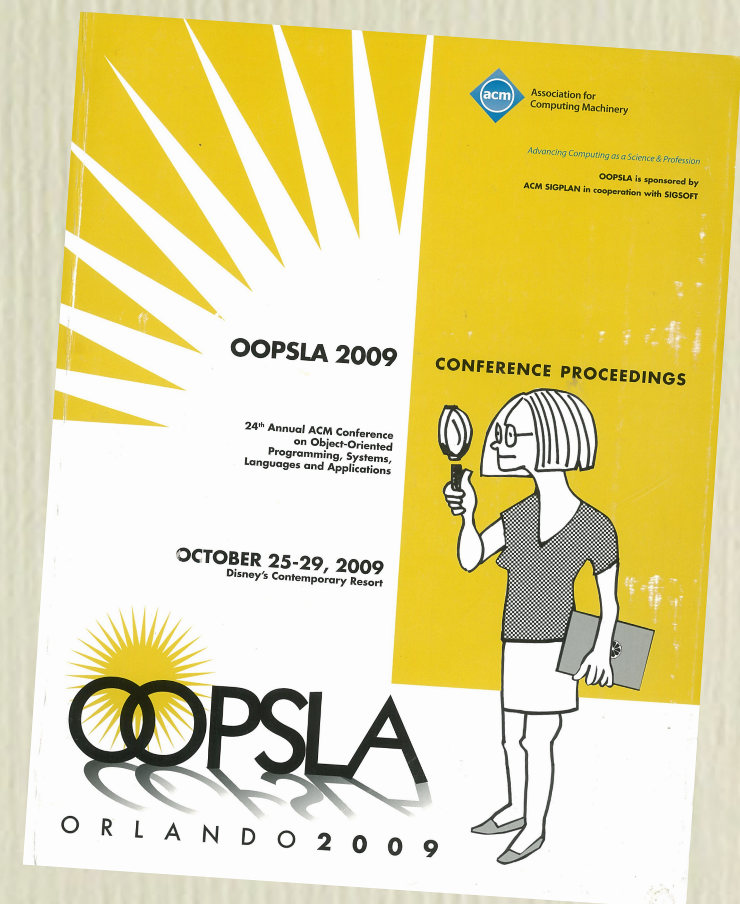
- The Scandinavian view:
  - an OO system is one whose creators realise that programming is modelling.
- The mystical view:
  - an OO system is one that is built out of objects that communicate by sending messages to each other, and computation is the messages flying from object to object.
- The software engineering view:
  - an OO system is one that supports data abstraction, polymorphism by late-binding of function calls, and inheritance.

Portland State UNIVERSITY

Ralph Johnson

Cook's Autognostic Principle

William Cook
On Understanding Data Abstraction, Revisited

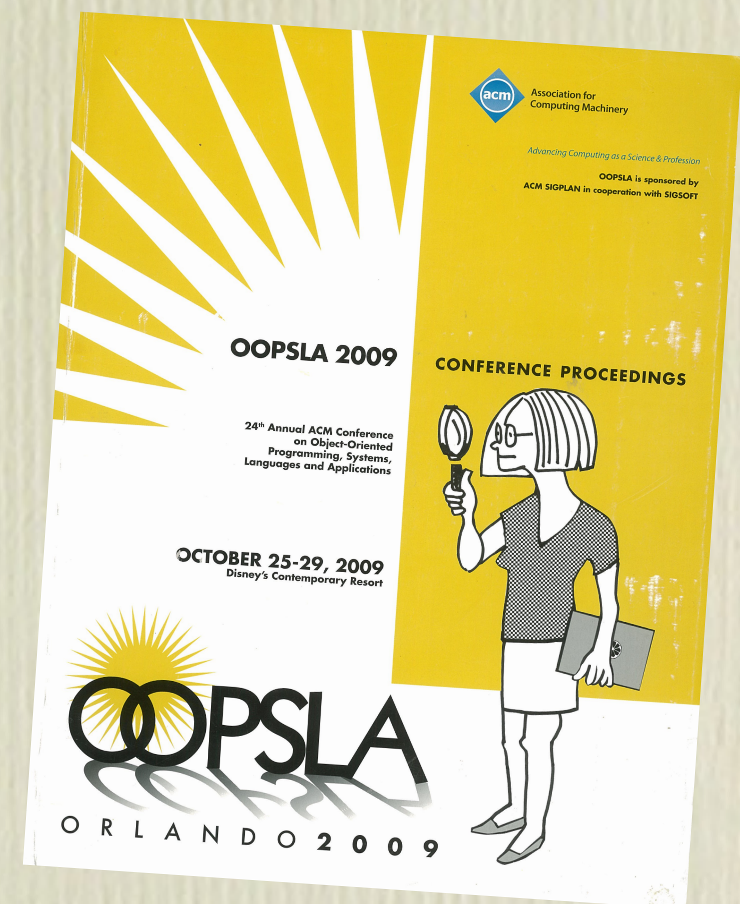


Portland State
UNIVERSITY

Cook's Autognostic Principle

William Cook
On Understanding Data Abstraction, Revisited

An object can access other objects only through their public interfaces.



Portland State UNIVERSITY

William Cook
On Understanding Data Abstraction, Revisited
OOPSLA 2009

- Autognosis means “self knowledge”
- An autognostic object can have detailed knowledge *only* of itself. All other objects are abstract.
- The converse is useful: any programming model that allows inspection of the representation of more than one abstraction at a time is *not* object-oriented.

William Cook
On Understanding Data Abstraction, Revisited
OOPSLA 2009

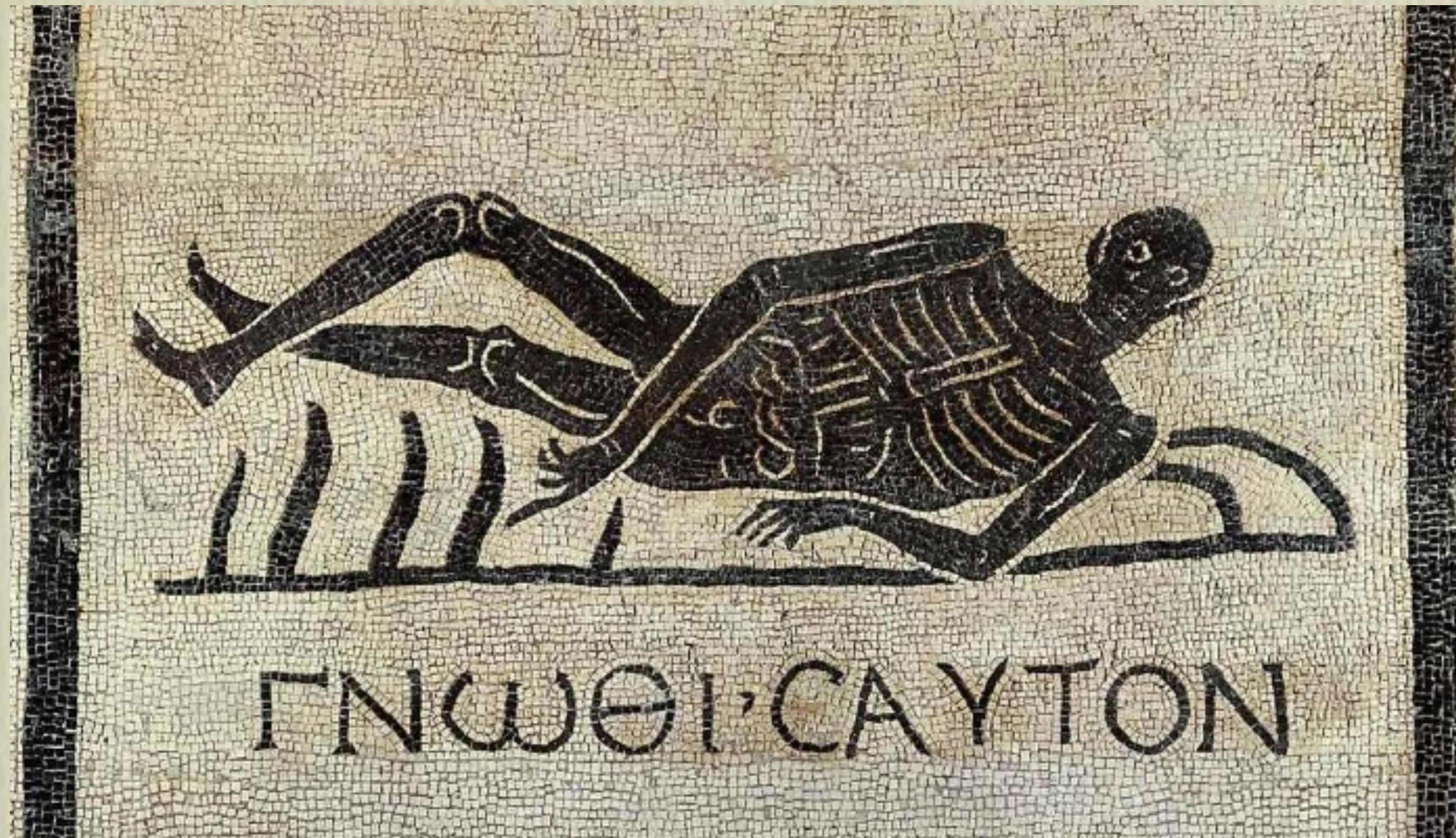
Portland State UNIVERSITY

Equality operators for autognostic languages

- What’s the best we can do?
- What’s the least we can get away with?

Left-handed Equality

Left-handed Equality



Know Thyself: Inscription on Apollo's Temple at Delphi

Consequences of Autognosticism

Consequences of Autognosticism

- No "third party" identity test
  - object *y* can't ask if *a* `==` *b*

Consequences of Autognosticism

- No "third party" identity test
  - object *y* can't ask if *a* `==` *b*
- Example: *strings*. Whether they are intern'ed or not is an implementation secret.
  - abstract equality keeps that secret
  - a third-party `==` test (as in Java) exposes it

Object Identity

A Position Paper for I-WOOOS '93

Andrew P. Black
Cambridge Research Laboratory
Digital Equipment Corporation

This position paper discusses the rôle of Object Identity in object-oriented systems. A distinction is drawn between object identity and object identifier; the former is an intrinsic part of an object-oriented system, while the latter is not. An equality test on object identifiers breaches encapsulation; such a test should therefore only be enabled at the specific request of the implementor of an abstraction.

1 Object identity is fundamental

I believe that object identity is fundamental to object-oriented systems: the existence of some concept of object identity is one of the attributes that makes a system object-oriented.

Consider the semantic equations that give meaning to the constructs of an object-oriented language. Clearly, the meaning of **self** depends on the identity of the containing object. The meaning of any identifier i also depends on the identity of the containing object. Thus, the semantic function that defines the meaning of an expression like i must take amongst its parameters some indication of the identity of the current object.

An application built on an object-oriented operating system may have its own notion of identity that differs from that provided by the underlying system. For example, clients of a replicated directory service may wish to regard all the replicas of a particular directory as identical. Because these replicas are (by design) distinct system-level objects, they have different identities as far as the underlying system is concerned. Indeed, understanding that they are distinct objects is vital to the correct implementation of the replication protocol. The fact that different notions of identity exist at different levels of abstraction is not an argument against the centrality of the concept of identity to object-orientation.

2 "Identity" should be distinguished from "Identifier"

Just because identity is essential, it does not follow that it must be possible to reify the identity of an object into a form that can be manipulated in the language itself. For example, it is possible to give a perfectly satisfactory semantics for an object-oriented language by representing each object as a separate state function: rather than having a single global state from which the right part must be extracted by indexing with some sort of object identifier, each object can instead be represented as a separate state. These states, being functions from locations to values, might not themselves be expressible in the language. Even if they are expressible, they cannot be compared for equality.

3 Should Objects have Identifiers?

The question that we should ask, then, is not whether the concept of object identity should be manifest to programmers, but in what ways should programmers be allowed to manipulate object identity. Given a reference to an object, what can be done with it, beyond invoking the object to perform one of the operations in its protocol? At one extreme is the answer "nothing at all"; the other extreme is to give object references a full set of operations, like those available on integers.

The minimal set of operations that enable a programmer to implement more complex operations seems to be

$$\begin{aligned} \mathit{hash} &: \text{object} \to \text{integer} \\ \mathit{equal} &: \text{object} \times \text{object} \to \text{Boolean} \end{aligned}$$

In theory, *hash* is superfluous: it is possible to implement *hash* using *equal* and exhaustive search. Although the inefficiency that this would introduce makes *hash* desirable in practice, for the purpose of theoretical analysis we can confine discussion to the

of the containing object. The meaning of any identifier i also depends on the identity of the containing object. Thus, the semantic function that defines the meaning of an expression like i must take amongst its parameters some indication of the identity of the current object.

An application built on an object-oriented operating system may have its own notion of identity that differs from that provided by the underlying system. For example, clients of a replicated directory service may wish to regard all the replicas of a particular directory as identical. Because these replicas are (by design) distinct system-level objects, they have different identities as far as the underlying system is concerned. Indeed, understanding that they are distinct objects is vital to the correct implementation of the replication protocol. The fact that different notions of identity exist at different levels of abstraction is not an argument against the centrality of the concept of identity to object-orientation.

The ques
whether the
ifest to prog
grammers b
Given a refe
it, beyond i
operations i
swer "nothi
object refer
available on

The min
programmer
seems to be

h

eq

In theory
implement
Although th
makes *hash*
theoretical a

of the containing object. The meaning of any identifier i also depends on the identity of the containing object. Thus, the semantic function that defines the meaning of an expression like i must take amongst its parameters some indication of the identity of the current object.

An application built on an object-oriented operating system may have its own notion of identity that differs from that provided by the underlying system. For example, clients of a replicated directory service may wish to regard all the replicas of a particular directory as identical. Because these replicas are (by design) distinct system-level objects, they have different identities as far as the underlying system is concerned. Indeed, understanding that they are distinct objects is vital to the correct implementation of the replication protocol. The fact that different notions of identity exist at different levels of abstraction is not an argument against the centrality of the concept of identity to object-orientation.

The ques
whether the
ifest to prog
grammers b
Given a refer
it, beyond in
operations in
swer “nothin
object refe
available on

The min
programmer
seems to be

h
eq

In theory
implement
Although th
makes *hash*
theoretical a

Consequences of Autognosticism

Consequences of Autognosticism

- It's OK to ask

```
self == other
```

Consequences of Autognosticism

- It's OK to ask

self == other

Self Reference Equality

Does another reference refer to **self**?

- Reference equality is used — crucially — to define `equals` for *java.lang.Object*

```
public boolean equals(Object obj) {
        return (this == obj);
}
```

- Reference equality is used — crucially — to define **`equals`** for *java.lang.Object*

```
public boolean equals(Object obj) {
        return (this == obj);
}
```

- (Almost) the same code can be used to define a self-referential equality:

- Reference equality is used — crucially — to define **`equals`** for *java.lang.Object*

```
public boolean equals(Object obj) {
        return (this == obj);
}
```

- (Almost) the same code can be used to define a self-referential equality:

confidential

```
~~public~~ boolean refEqualsSelf(Object obj) {
        return (this == obj);
}
```

leftEquals

- Assume an autognostic reference quality test

```
refEqualSelf(other)
```

- An object that wishes to expose an identity test to its clients can define

```
method leftEquals(other:Object) {
   refEqualSelf(other)
}
```

- Big difference: `==` is trustworthy

Trust

the truth is not an obstacle for someone such as me,
she said, because you see
we all create our own reality
and if a problem should arise
the best thing you can say is
don't worry, be happy, and have a nice day

Truth is Out of Style, (MC 900 Ft. Jesus 1989)

Trust

the truth is not an obstacle for someone such as me,
she said, because you see
we all create our own reality
and if a problem should arise
the best thing you can say is
don't worry, be happy, and have a nice day

Truth is Out of Style, (MC 900 Ft. Jesus 1989)

Trust and the **obeys** predicate

Trust and the **obeys** predicate

- trust is an assumption, not an assertion

Trust and the **obeys** predicate

- trust is an assumption, not an assertion
- does an object obey its specification?

Trust and the **obeys** predicate

- trust is an assumption, not an assertion
- does an object obey its specification?
  - that's a question about its *implementation*
  - when we encapsulate implementations, *we also encapsulate their correctness*

Left-handed Equality

`a.leftEquals(b)`

means

$\texttt{a obeys } \mathcal{E} \Rightarrow \mathcal{E}(\texttt{a, b})$

where $\mathcal{E}$ is a specification of leftEquality

`_.leftEquals(_)` is *not* an equivalence relation

if even *one* object does not obey the specification

`_.leftEquals(_)` is *not* an equivalence relation

if even *one* object does not obey the specification

```
def perverse = object {
 method leftEquals(other : Object) → Boolean { true }
}
```

`_.leftEquals(_)` is *not* an equivalence relation

if even *one* object does not obey the specification

```
def perverse = object {
 method leftEquals(other : Object) → Boolean { true }
}
```

perverse.leftEquals(o) → true $\forall$o

o.leftEquals(perverse) → false $\forall o \in U \setminus perverse$

Practical Consequences

Practical Consequences

- If you are trying to compare objects for identity, be sure that you trust the receiver

Practical Consequences

- If you are trying to compare objects for identity, be sure that you trust the receiver
- Example:

```
someVariable == false
```

Practical Consequences

- If you are trying to compare objects for identity, be sure that you trust the receiver
- Example:

```
someVariable == false
```

Practical Consequences

- If you are trying to compare objects for identity, be sure that you trust the receiver

- Example:

```
someVariable == false
false == someVariable
```

The Second Lesson

Object-oriented equality

is not, *and should not be,*

an equivalence relation

Reflection

== and !== should usually be avoided; if you really care about object identities then you should probably be using mirrors, since object identity is a reflective concept.

Portland State University

Reflection

== and !== should usually be avoided; if you really care about object identities then you should probably be using mirrors, since object identity is a reflective concept.

Reflection

== and !== should usually be avoided; if you really care about object identities then you should probably be using mirrors, since object identity is a reflective concept.

Reflective Equality
Do two mirrors reflect on the same object?

Grace

LIST OF PRINCIPLES

Abstraction: Avoid requiring something to be stated more than once; factor out the recurring pattern.

Automation: Automate mechanical, tedious, or error-prone activities.

Defense in Depth: Have a series of defenses so that if an error isn't caught by one, it will probably be caught by another.

Information Hiding: The language should permit modules designed so that (1) the user has all of the information needed to use the module correctly, and nothing more; (2) the implementor has all of the information needed to implement the module correctly, and nothing more.

Labeling: Avoid arbitrary sequences more than a few items long; do not require the user to know the absolute position of an item in a list. Instead, associate a meaningful label with each item and allow the items to occur in any order.

Localized Cost: Users should only pay for what they use; avoid distributed costs.

Manifest Interface: All interfaces should be apparent (manifest) in the syntax.

Orthogonality: Independent functions should be controlled by independent mechanisms.

Portability: Avoid features or facilities that are dependent on a particular machine or a small class of machines.

Preservation of Information: The language should allow the representation of information that the user might know and that the compiler might need.

Regularity: Regular rules, without exceptions are easier to learn, use, describe, and implement.

Security: No program that violates the definition of the language, or its own intended structure, should escape detection.

Simplicity: A language should be as simple as possible. There should be a minimum number of concepts with simple rules for their combination.

Structure: The static structure of the program should correspond in a simple way with the dynamic structure of the corresponding computations.

Syntactic Consistency: Similar things should look similar; different things different.

Zero-One-Infinity: The only reasonable numbers are zero, one, and infinity.

Regularity: Regular rules, without exceptions are easier to learn, use, describe, and implement.

Security: No program that violates the definition of the language, or its own intended structure, should escape detection.

Simplicity: A language should be as simple as possible. There should be a minimum number of concepts with simple rules for their combination.

Structure: The static structure of the program should correspond in a simple way with the dynamic structure of the corresponding computations.

Syntactic Consistency: Similar things should look similar; different things different.

Zero-One-Infinity: The only reasonable numbers are zero, one, and infinity.

Regularity: Regular rules, without exceptions are easier to learn, use, describe, and implement.

Security: No program that violates the definition of the language, or its own intended structure, should escape detection.

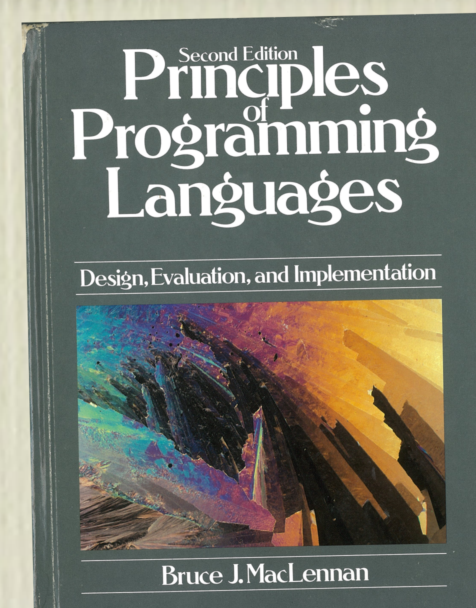
Simplicity: A language should be as simple as possible. There should be a minimum number of concepts with simple rules for their combination.

Structure: The static structure of the program should correspond in a simple way with the dynamic structure of the corresponding computations.

Syntactic Consistency: Similar things should look similar; different things different.

Zero-One-Infinity: The only reasonable numbers are zero, one, and infinity.

Portland State UNIVERSITY



J.R.R. Tolkien

One Ring to rule them all, One Ring to find them,
One Ring to bring them all, and in the darkness bind them...

J.R.R. Tolkien

Hence:

Grace should have one

- autognostic
- left-handed
- abstract equality

represented by the method `==`

Implementation

- a *confidential* method `isMe`, inherited by all objects from `graceObject`
- a trait that defines a *public* method `==` in terms of `isMe`, available to any object that chooses to `use` it
- an `==` method on an object's mirror that determines if anotherMirror images the same object as `self`

```
type Object = interface {
     ≠ (other : Object) → Boolean
     hash → Boolean
     …
}
trait graceObject {
     // root of the inheritance hierarchy
     method isMe(other:Object) → Boolean
          is primitive, confidential { }
     method ≠ (other : Object) { (self == other).not }
     …
}
trait identity {
     method ==(other:Object) → Boolean { self.isMe(other) }
}
type Reflection = interface {
     reflect(reflectee:Object) → ObjectMirror
     …
}
type ObjectMirror = Object & interface {
     == (other) → Boolean
     …
}
```

The story is not all mine, nor told by me alone. Indeed I am not sure whose story it is.

Ursula Le Guin, The Left Hand of Darkness, (LeGuin 1969)

Acknowledgements

William Cook

Sophia Drossopoulou

Shriram Krishnamurthi

Joe Gibs Politz

The reviewers

Acknowledgements

William Cook

Sophia Drossopoulou

Shriram Krishnamurthi

Joe Gibs Politz

The reviewers

James Noble