Programming Languagages (Langages Evolués)

Roel Wuyts

Prototype-based languages and Self

Forget classes

- Used to share methods
- Very static
  - or need meta-programming or reflection to change things at runtime
    - complicated, error-prone
- Today: prototype-based systems
- only objects, no classes

A Lot of Languages

- Simpler descriptions of objects
  - Examples first vs. abstractions
- Simpler Programming Models with fewer Concepts
  - User Interface direct manipulation (Garnet)
- New capabilities to represent knowledge
  - Classes are too rigid

Some languages

- Self, Cecil, Omega, Io, Kevo, Agora, Obliq, NewtonScript, Lua, Garnet, Moostrap, ACT-1, JavaScript, Object-Lisp, Examplars

Key Features

- One kind of object
  - with attributes and methods
- Three primitive ways to create objects:
  - ex-nihilo
  - cloning
  - extension (differential copy)
- One control structure
  - message sending (+ delegation)

Variation Points

- Semantics of basic mechanisms:
  - cloning, differential copy, delegation
  - Semantics of object organization, object creation, object encapsulation, object activation, object inheritance
- Organization of programs: implicit sharing (prototypical instance, traits, maps)

Object Representation

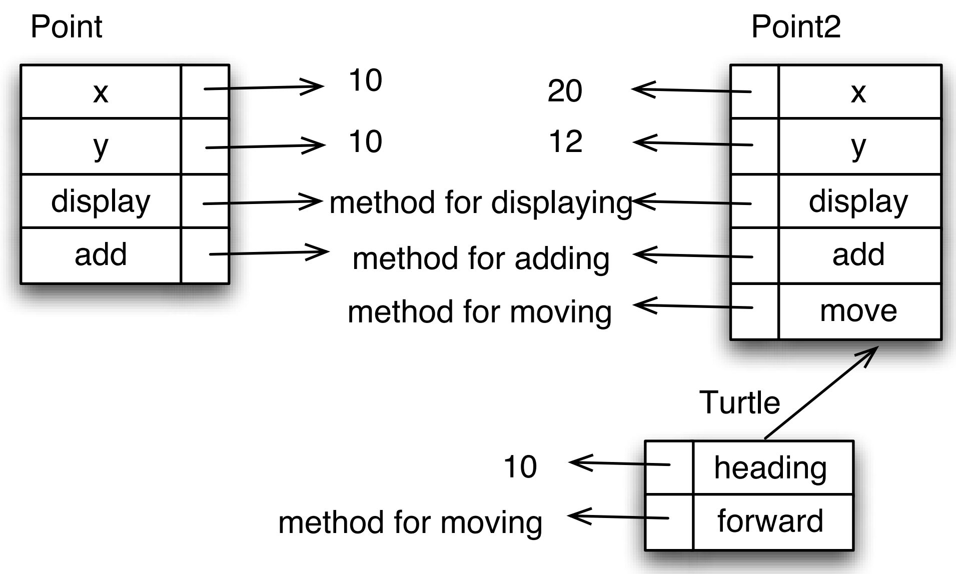
- Objects have name-value pairs
- Conceptually: attributes (state) and behavior
- Representation:
  - *Attributes* and *Methods* (Agora, Kevo, ...)
    - can have public methods, private attributes, etc.
  - *Slots* (SELF, Moostrap, NewtonScript, JavaScript, ...)
    - blur difference between state and behaviour
    - overriding of both attributes and behaviour

Object Creation

- 3 ways: ex-nihilo, cloning, extending
- ex-nihilo
  - empty object (require structure modification)
  - with initial structure
- cloning: creation-time sharing
  - copy another object
  - prototype and child are unrelated
- extending: life-time sharing
  - prototype changes -> child is impacted

Cloning

- After cloning both objects point to the same
- Objects can be modified independently



Group-Wide

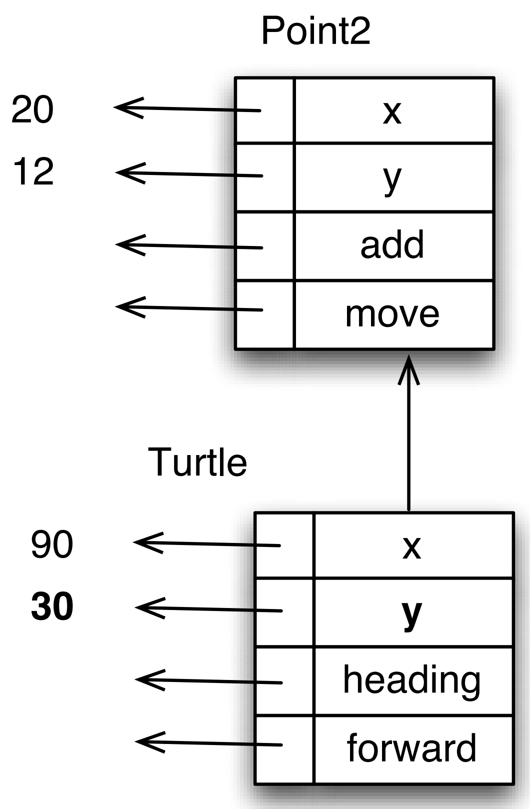
- When no extension mechanism is provided as in Kevo (i.e., only cloning), *life-time* sharing requires group-wide mechanism.
- Kevo supports cloning and addition of slot but also propagation
    - allows one to add a slot to all the objects issued of the same clone.

Sharing

- Want mechanisms to share data and to share behaviour
- For data: two mechanisms:
  - Value sharing
  - Properties sharing
- For behaviour: delegation
  - different from class based delegation (proxies)

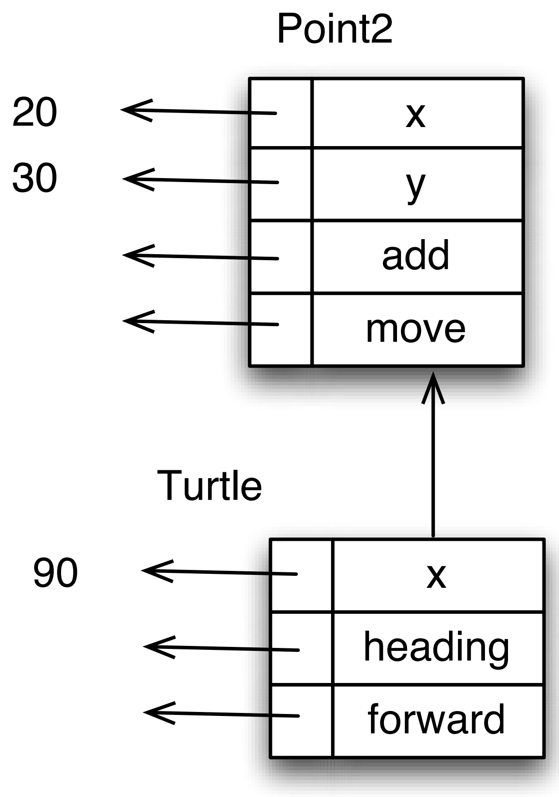
Value Sharing

- Express attribute sharing
  - Point2's properties are not Turtle's properties
  - Point2 provides default values for turtle
  - An object should not be modified when one of its descendants receives a message
    - Turtle y: 30 should not modify Point2
    - In fact parent should be a read-only access



Property Sharing

- Different parts of the same domain entity
  - Support split object and propagation between parent-children
  - It is coherent to modify a parent when sending a message to one of its children



Delegation

- Can an object have multiple parents?
    - JavaScript: only one prototype property
    - Self: multiple parent slots
- Can the parent-link be modified dynamically?

Delegation

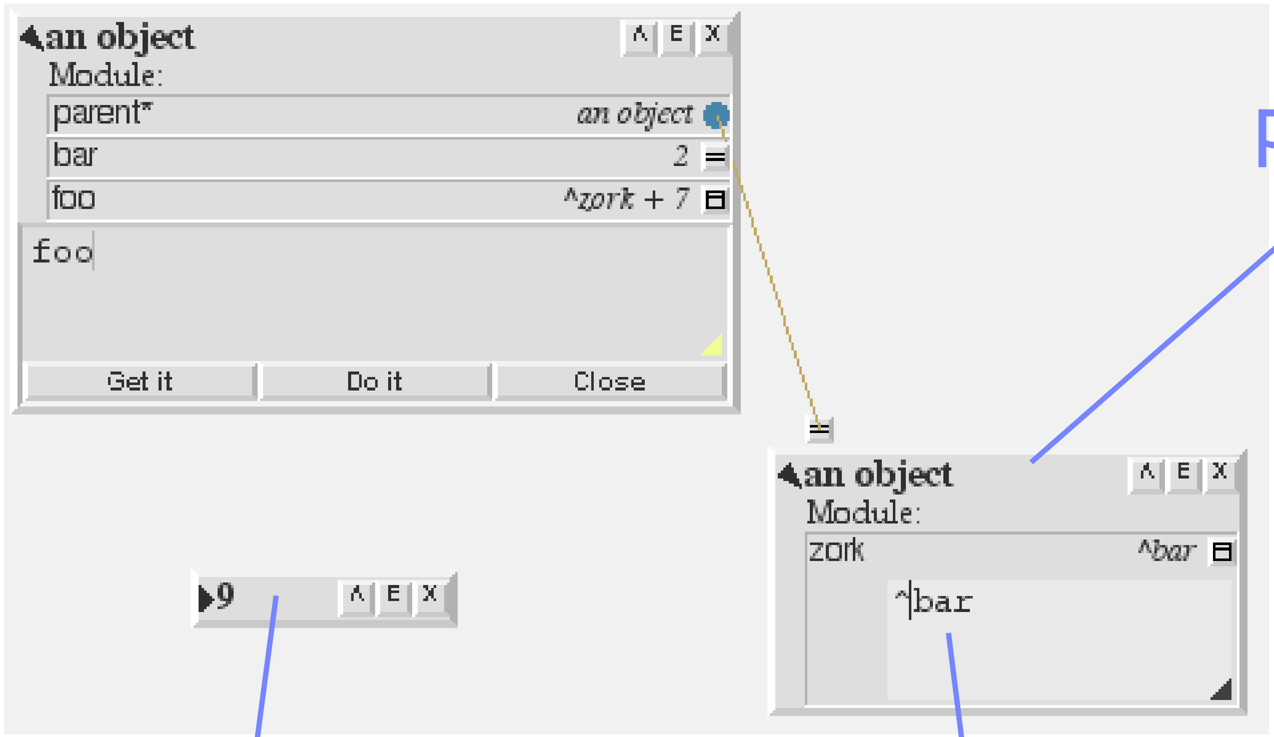
- Most prototype-based languages provide a sharing mechanism called delegation
- Binding not found in object, then **lookup** in parent
  - recursively traverse the (possibly cyclic) delegation graph
- Delegation can be *implicit* or *explicit*
  - ACT-1 and Obliq have *explicit delegation*
  - *Implicit delegation* is used with an extension mechanism

Binding parent's self

- The key aspect of prototype delegation is that the self in parent method is bound to the receiver of the original message.
  - Execute your code on me: your self = me
- "delegation" in class-based system does not bind self of the delegated method to the original receiver
  - cfr. proxy implementation

Delegation example

- in prototype system (Self)



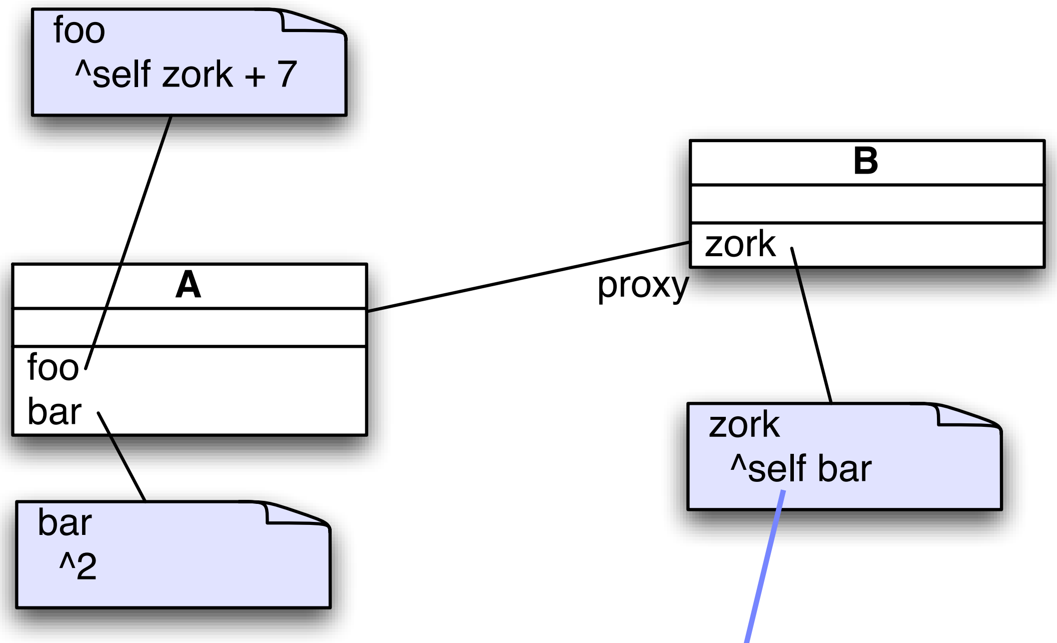
parent object

lookup bar slot in self

result of sending *foo*: 9

Delegation example

- in class-based system (proxy thing)



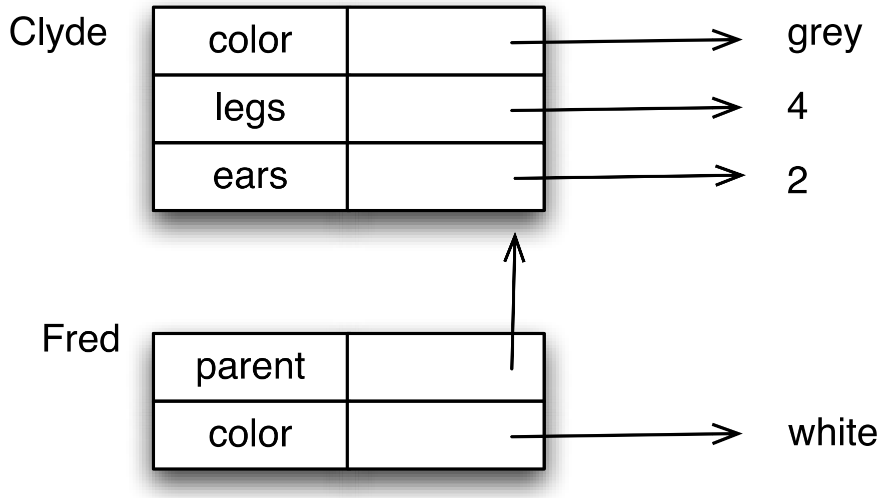
self is instance of B, not A

Program Organization

- Experience with prototype languages shows that some form of 'blueprint' objects are useful
  - cfr classes!
  - acknowledge organization of objects is needed
- Mechanisms were added to support this
  - Prototypical instances (Lieberman)
  - Traits
  - Maps (Self)/clone families (Kevo)

Prototypical instances

- Use prototypical instances and delegation to share properties
- In Lieberman
  - no specific status
  - prototypical modifiable
  - unintended propagation
    - clyde loses a leg -> all elephants have 3 legs
  - JavaScript

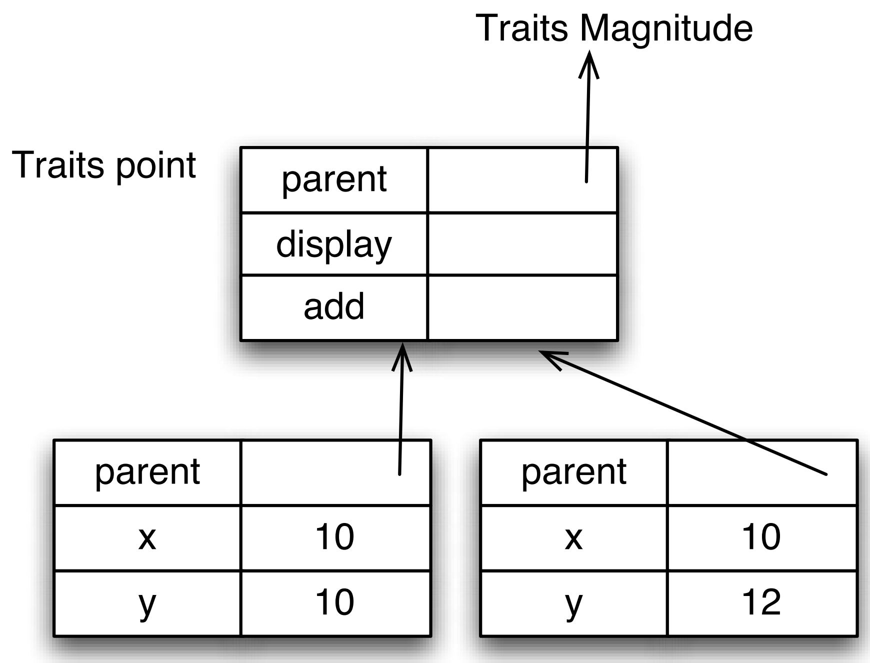


Traits

- Traits in Self: repositories of methods and shared variables
- Traits reach lobby (root object)
- Sharing objects all delegate to the same traits
- Objects can be decomposed into:
  - data or specific objects information
  - protocol or shared information

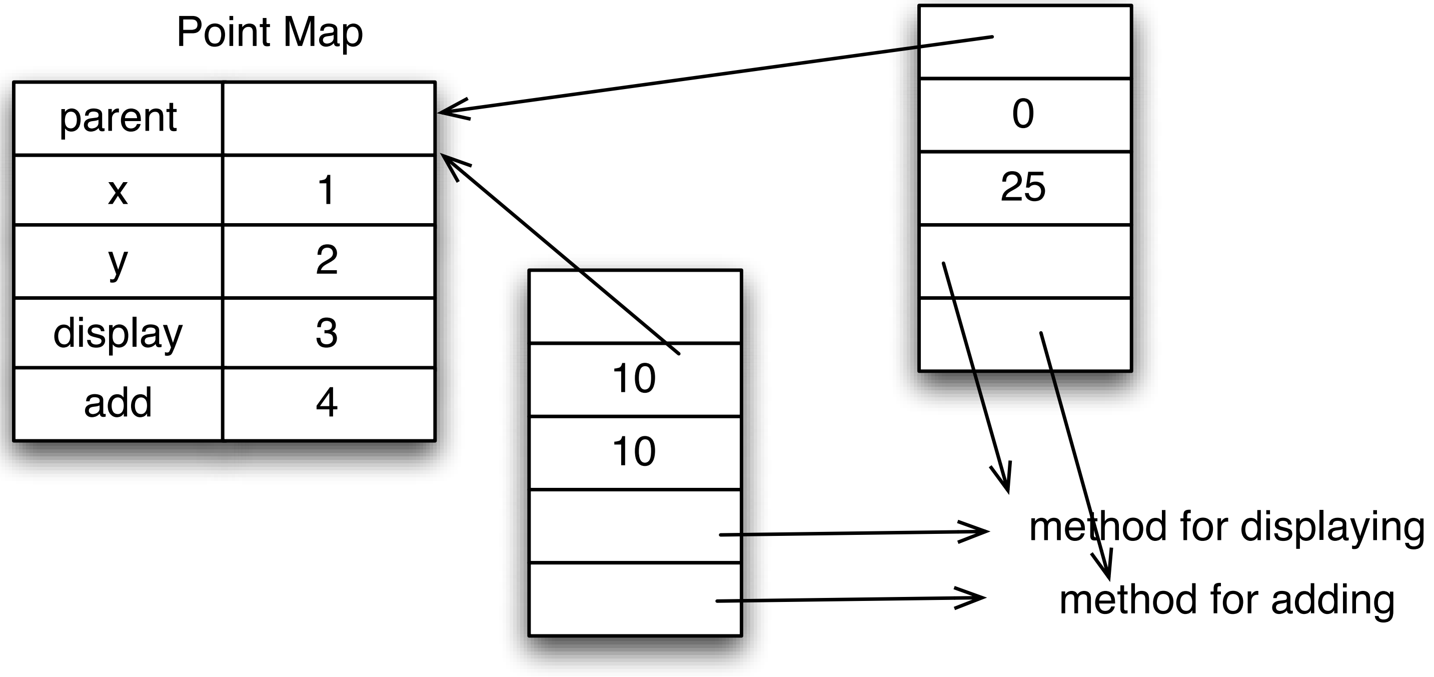
Example

- Traits promote
  - delegation hierarchies
  - high: traits
  - leaves: concrete objects
- Traits
  - emphasize similar behavior among objects
  - are more flexible and less abstract than classes



Maps

- Factors out structure in a map/clone family
- Maps are similar to prototypes but store indexes
    - Objects are then represented as vectors



Maps

- In Self: maps are not first class entities
  - Created automatically: when an object is cloned, the two clones share the same map.
- In Kevo: clone families are first class

Self

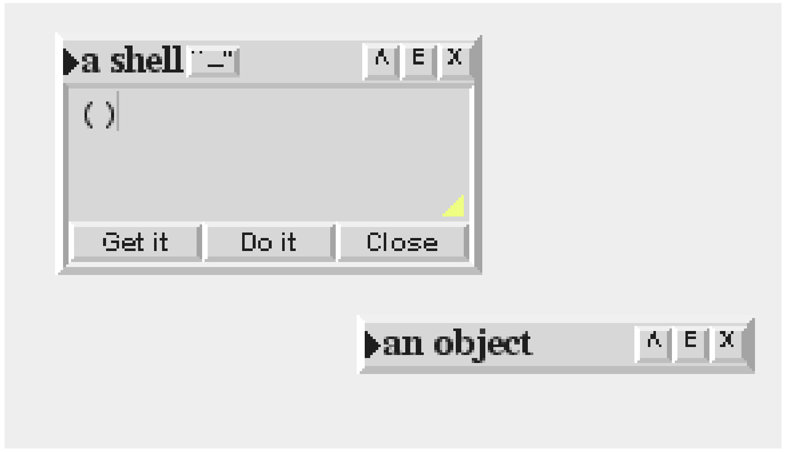
- Developed at Sun
- Introduced traits/mirrors
- Morphic: direct manipulation interfaces
- Uniform: environment written in itself
- Pay attention
  - several versions exist, with various semantics
  - research vehicle for prototypes

Everything is an object

- Unification and simplification of Smalltalk
- No class
- Objects have slots (data or method)
- Sharing behavior via traits
- Parents are shared parts of children

Empty Object

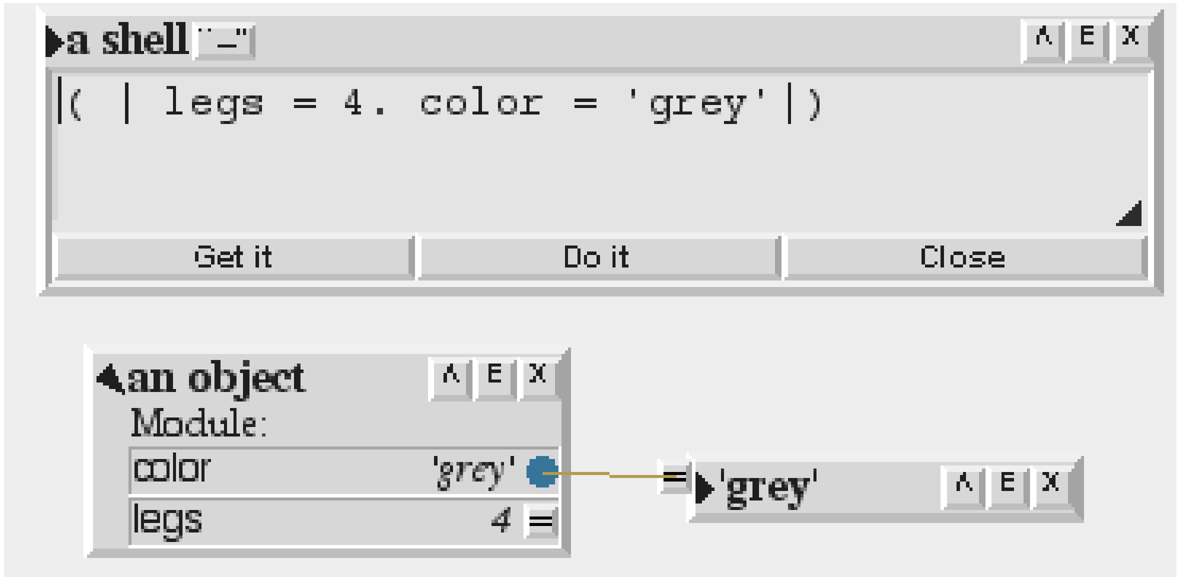
- ( ) creates an empty object
  - ex-nihilo creation without default values



```
( )
```

An Object with Slots

- (| slot1 . slot2 = 3|)
- Creates an object with two slots (slot1 and slot2)
- Data objects are objects without code
- Returns itself when evaluated



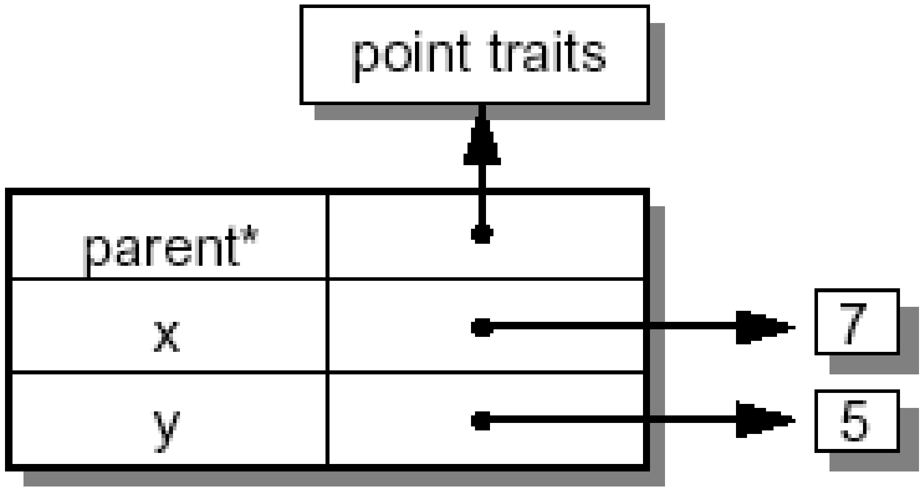
a shell

```
( | legs = 4. color = 'grey' |)
```

Ready-only Slots

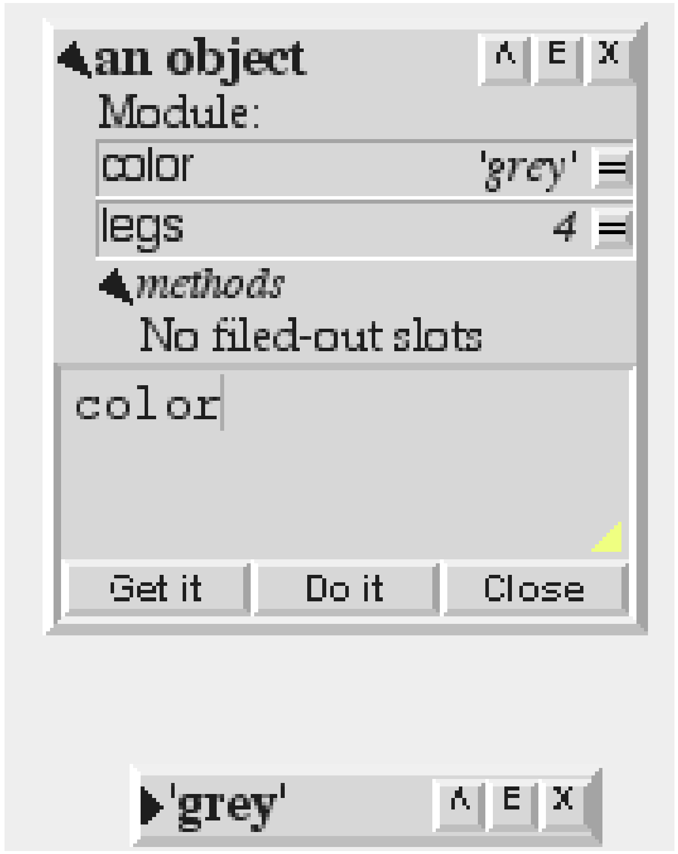
- *slot-name = expression*
- slot initialized to the result of evaluating the expression in the Lobby (the root context).
- Example: A read-only point:

```
(| parent* = traits point.
   x = 3.
   y = 5|)
```



Message

- As in Smalltalk: unary > binary > keyword-based
  - result of last expression is returned
  - or use ^


an object
Module:

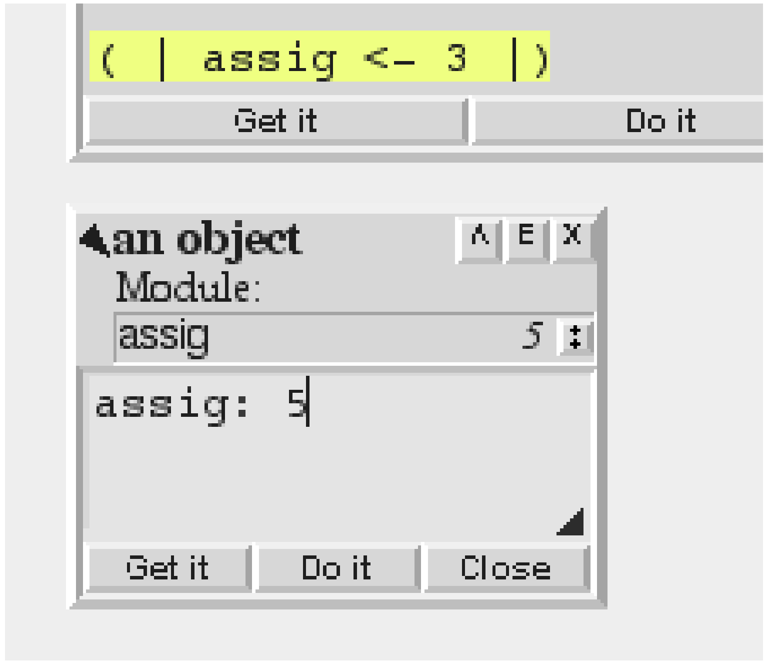
color	'grey'
legs	4

methods
No filed-out slots

```
color
```

Assignment Slots

- *slot-name <- expression*
  - store expression in data slot with name *slot-name*



```
( | assig <- 3 |)
```

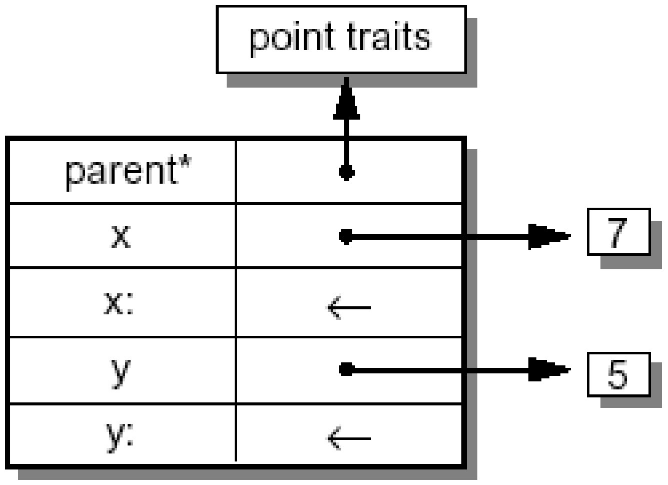
```
assig: 5
```

Read/Write Slots

- *slot-name: expression*
  - only assignment construct
  - creates 2 slots: slot-name and slot-name:
  - evaluates expression in the root context
  - stores result at parse time -> initialization
- Slot x can be assigned iff slot x: exist on the same object
- methods and attributes are unified!

Example: A Point object

```
(| parent* = traits point.
   x <- 7.
   y <- 5.
   |)
```



Method Definition

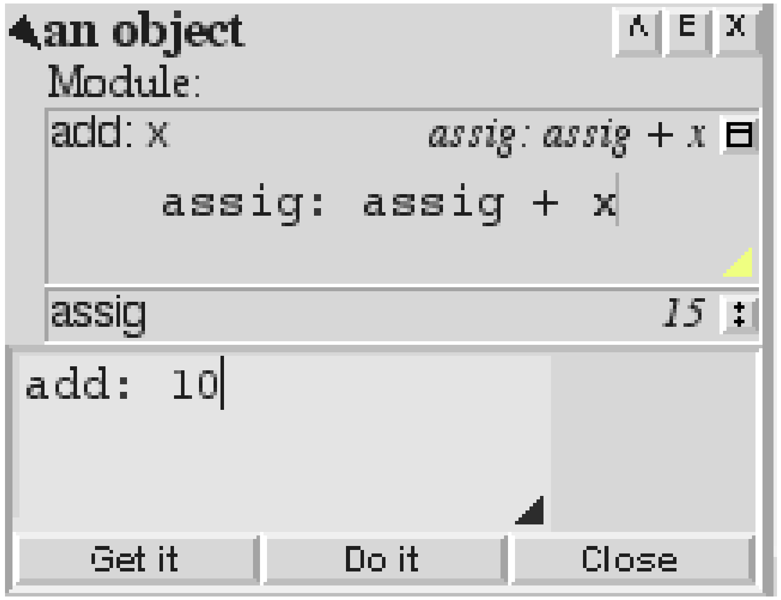
- name1: arg1 name2:arg2 = (body)
- Example:

  add: x = (assig: assig + x)

  equivalent to (in Smalltalk):

  add: x

  self assig: self assig + x



an object

Module:

```
add: x                assig: assig + x
      assig: assig + x
assig                 15
add: 10
```

- Note that = expects an object

Syntactic sugar

```
( | ifTrue: False: = ( | :trueBlock. :falseBlock |
                        trueBlock value ).
  | )
```

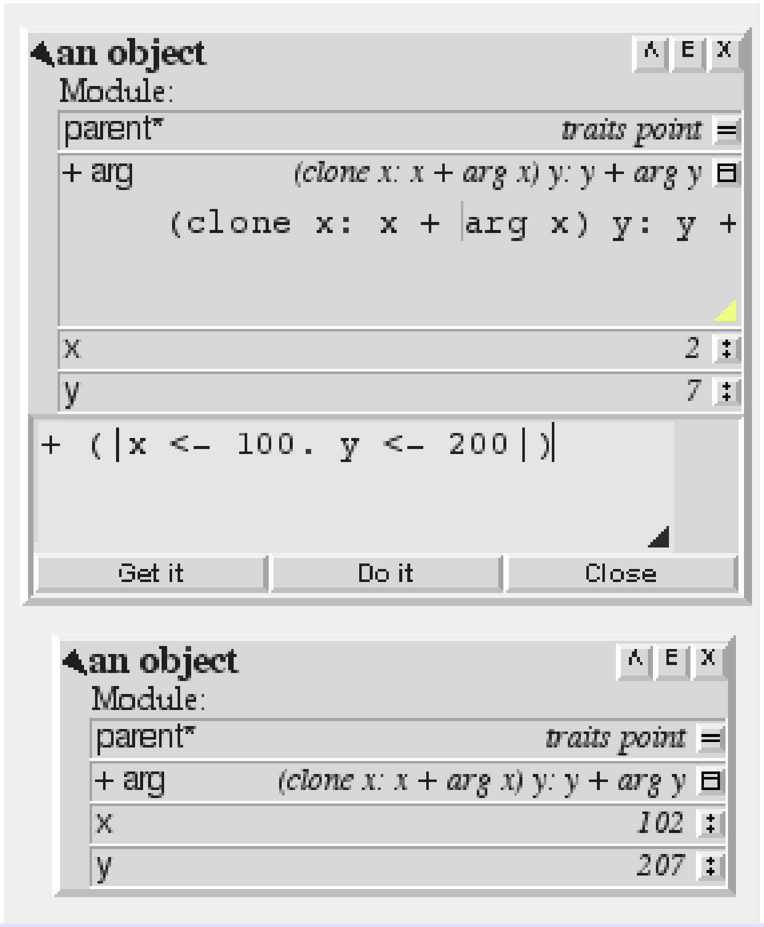
Is the same as

```
( | ifTrue: trueBlock False: falseBlock =
      ( trueBlock value ).
  | )
```

Method with Argument

(| + = (| :arg | (clone x: x + arg x) y: y + arg y|)

(| + arg = ((clone x: x + arg x) y: y + arg y|)



an object
Module:

parent*	traits point
+ arg	(clone x: x + arg x) y: y + arg y

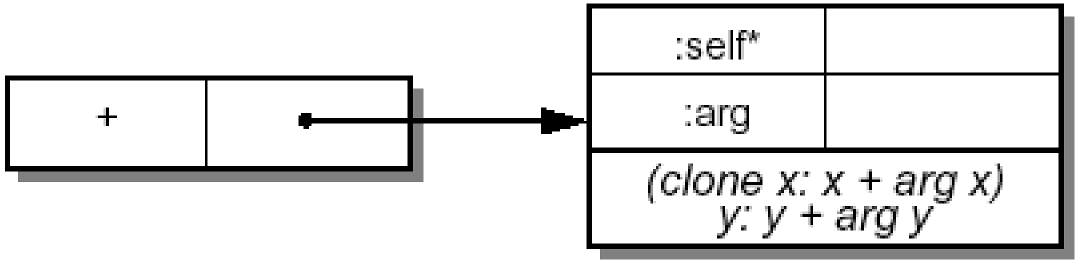
```
(clone x: x + arg x) y: y +
```

x	2
y	7

```
+ (|x <- 100. y <- 200|)
```

an object
Module:

parent*	traits point
+ arg	(clone x: x + arg x) y: y + arg y
x	102
y	207

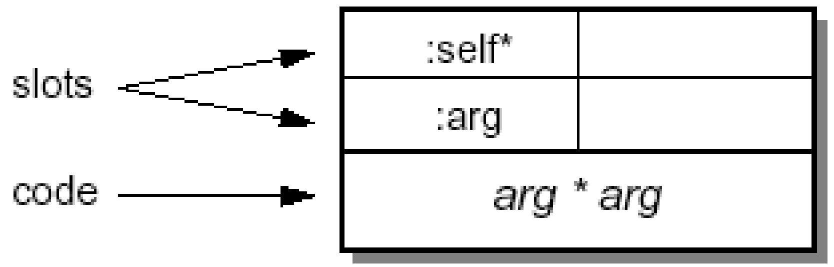


Unification in Self

- Self unifies local variable and slot access.
- What is the impact of having methods as objects?

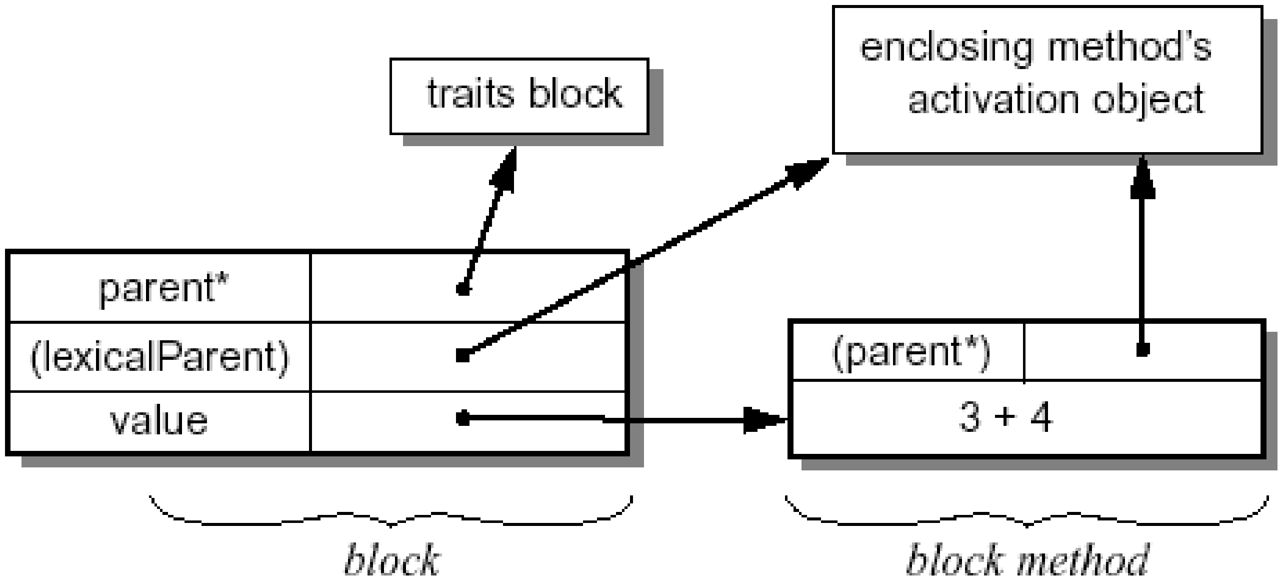
Methods are Objects

- A method object has code, whereas a data object has not.
- Evaluating a method object does not simply return the object itself, as with simple data objects; its code is executed and the resulting value is returned.
- An ordinary method always has an **implicit** parent argument slot named **self**. Ordinary methods are SELF's equivalent of Smalltalk's methods.



Blocks are Objects

- [3 + 4] defines two objects:
  - block method object (code's block)
  - enclosing block data object (with parent to block behavior)
- No self slot



Local Variable, Slot Access

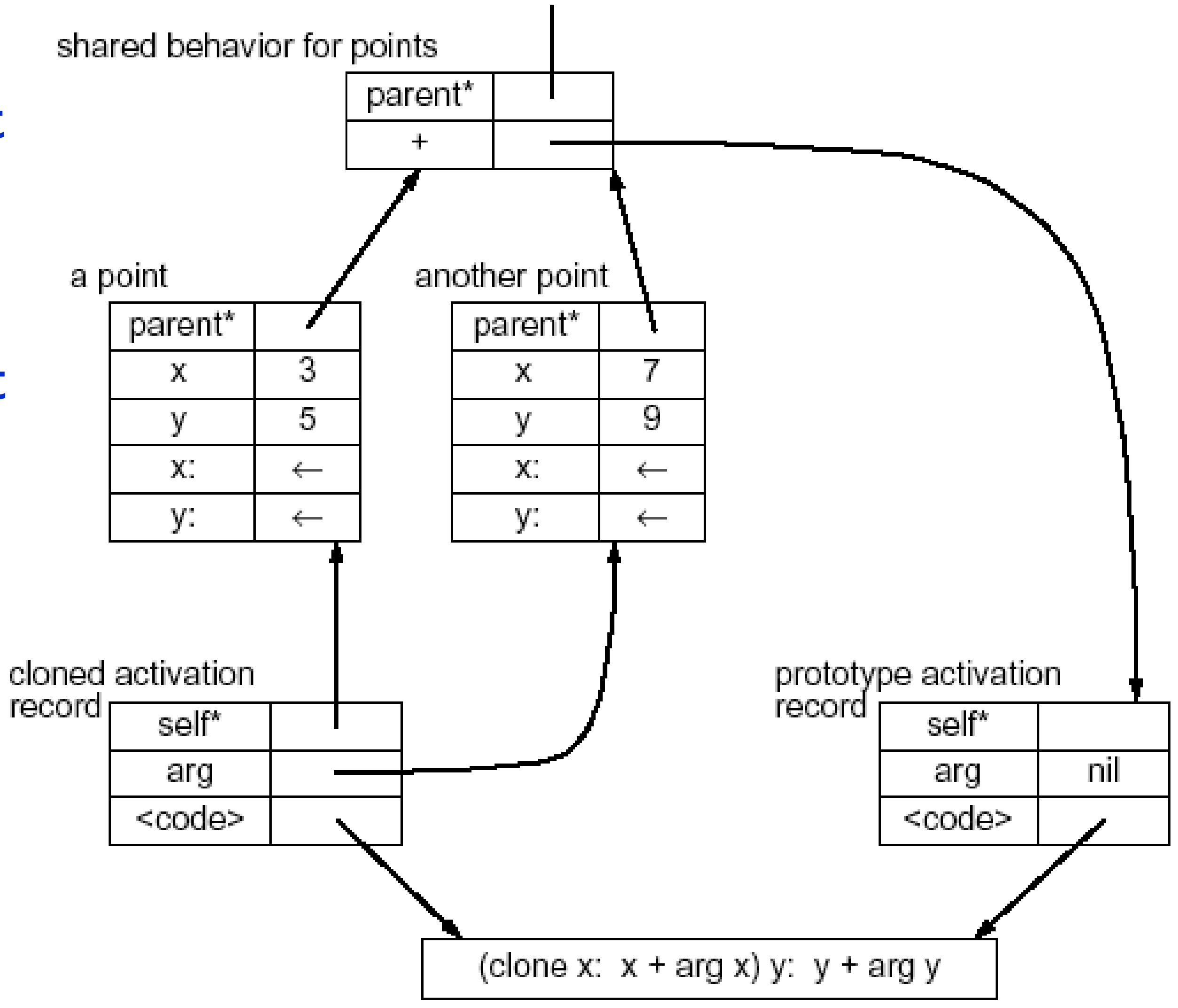
- In Self, the method lookup begins at the current *activation context* rather than the current *receiver*
  - *local variables* of methods are slots of the *activation context.*
  - Activation context inherits from the *receiver*
- Accesses to slots of the receiver (local or inherited) are achieved by implicit message sends to self.
- Having the lookup for the **implicit** self receiver starts at the current activation allows local variables, instance variables, and method lookup to be unified in SELF.

Method Execution

- Steps performed when a method slot is evaluated as result of a message send:
  - The method object is cloned, creating a new method activation object containing slots for the method's arguments and locals.
  - The clone's self parent slot is initialized to the receiver of the message.
  - The clone's argument slots are initialized to the values of the corresponding actual arguments.
  - The code of the method is executed in the context of this new activation object.

Example 3@5+ 7@9

- Lookup #+ in (3@5)
- finds a matching slot. slot is a method object, it is cloned,
- the clone’s argument slot is set to the argument of the message, and its parent is set to the receiver.
- the lookup for x will find the receiver’s x slot by following the inheritance chain from the current activation record.



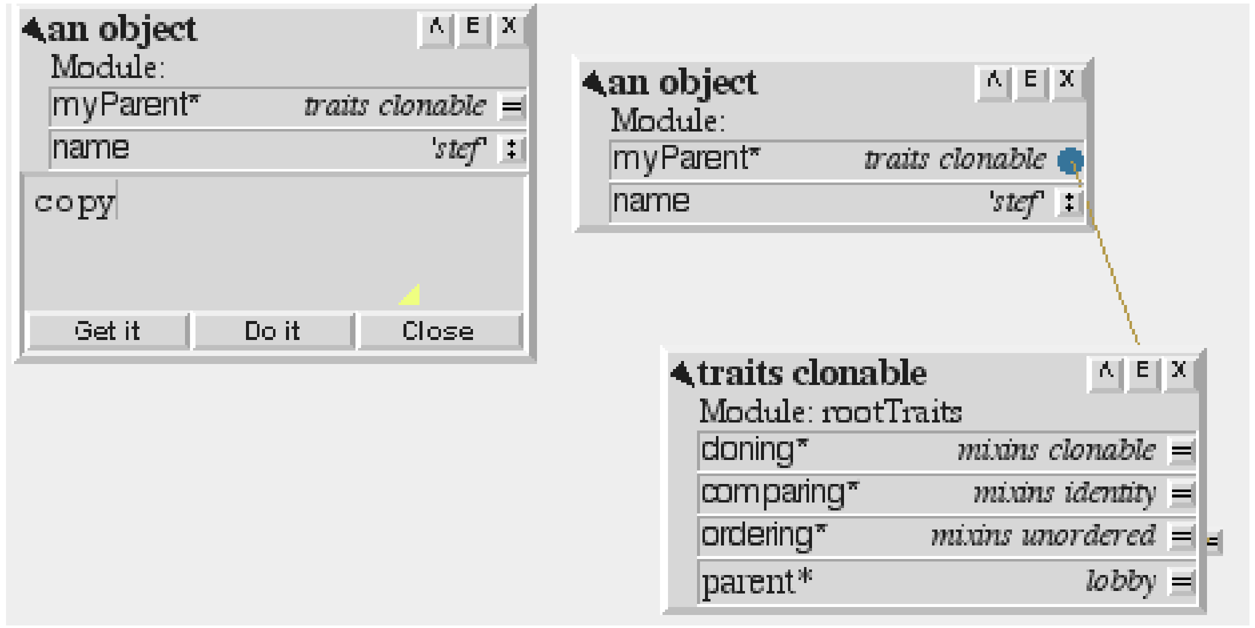
Implicit vs Explicit send

- A message sent explicitly to self is not equivalent to an implicit-receiver send because the former won't search *local* slots before searching the receiver.
- Example:

```
(foo = ( 42)
bar = ( 23)
baz = (
   |foo|
   foo: 0.
   foo.          "returns 0"
   self foo: 0."error- no #foo: method on self"
   self foo.     "returns 42".
   bar.          "returns 23 - thisContext delegates to self"
   self bar.     "returns 23" )
```

(Dynamic) Inheritance

- Multiple inheritance (was prioritized)
- Assignable: *dynamic* inheritance

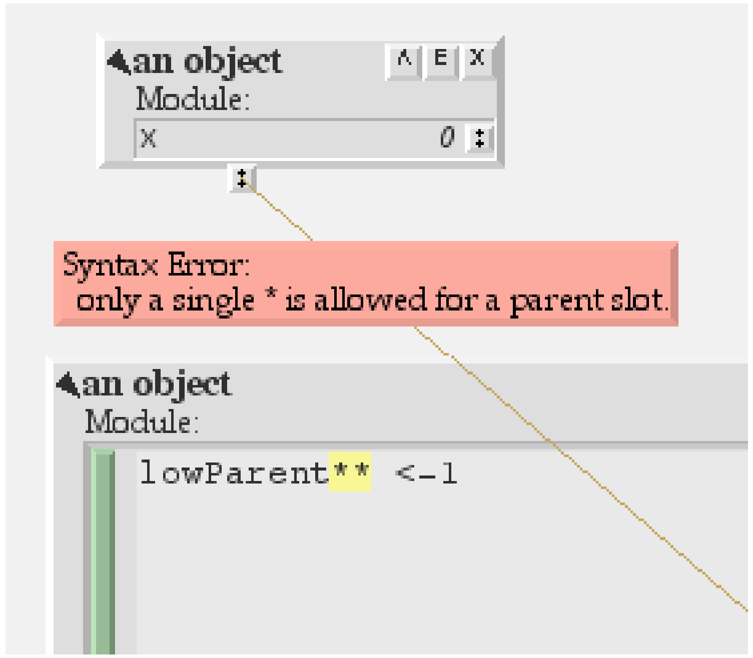


an object
Module:
myParent* traits clonable
name 'stef'

```
copy
```

an object
Module:
myParent* traits clonable
name 'stef'

traits clonable
Module: rootTraits
cloning* mixins clonable
comparing* mixins identity
ordering* mixins unordered
parent* lobby



an object
Module:
x 0

Syntax Error:
only a single * is allowed for a parent slot.

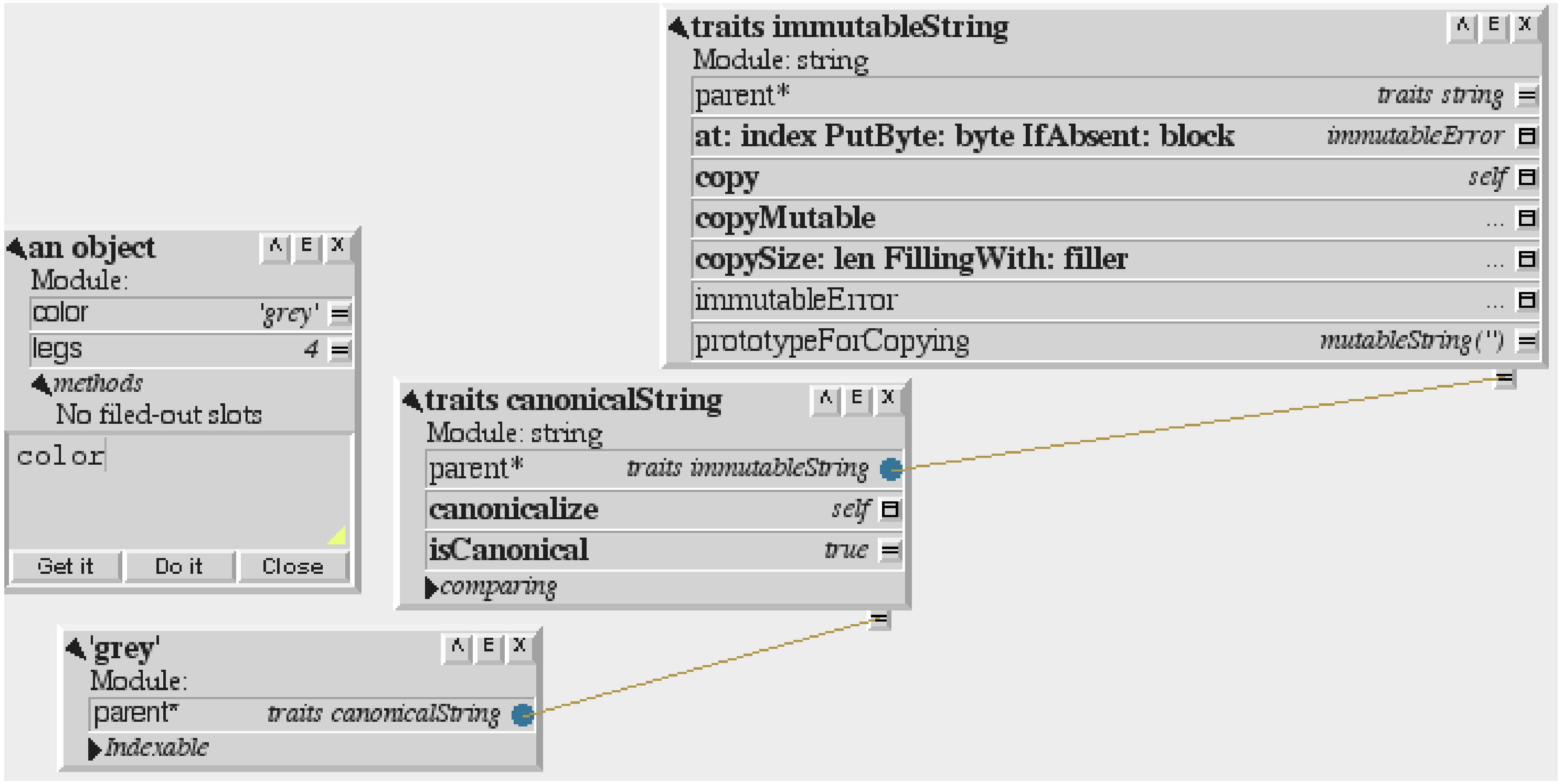
an object
Module:

```
lowParent** <-1
```

Parents

- Parent slot = Slot with *
    - Method lookup follows *
- An object can have several parents
- Can be
    - any object
    - traits (object with parent going to Lobby)
    - Mixin traits without parents to be plugged in object with traits

Parents



traits immutableString
Module: string

parent*	*traits string*
at: index PutByte: byte IfAbsent: block	*immutableError*
copy	*self*
copyMutable	...
copySize: len FillingWith: filler	...
immutableError	...
prototypeForCopying	*mutableString('')*

an object
Module:

color	'grey'
legs	4

methods
No filed-out slots

```
color
```

traits canonicalString
Module: string

parent*	*traits immutableString*
canonicalize	*self*
isCanonical	*true*

comparing

'grey'
Module:

parent*	*traits canonicalString*

Indexable

Diamond?

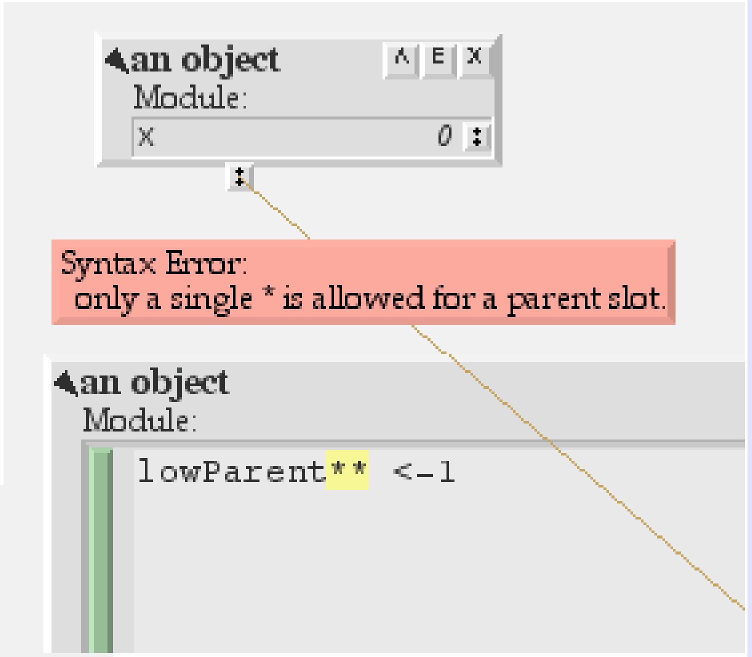
- Standard rule that an object may override its ancestors' slots: If an object and one of its ancestors define slots with the same name, then the object's slot takes precedence over the ancestor's slot:

- An ancestor's slots are only included <u>once</u>, no matter how many paths lead from the receiver to the ancestor, so an object won't generate ambiguities with its own slots if it is inherited along several paths.

Lookup

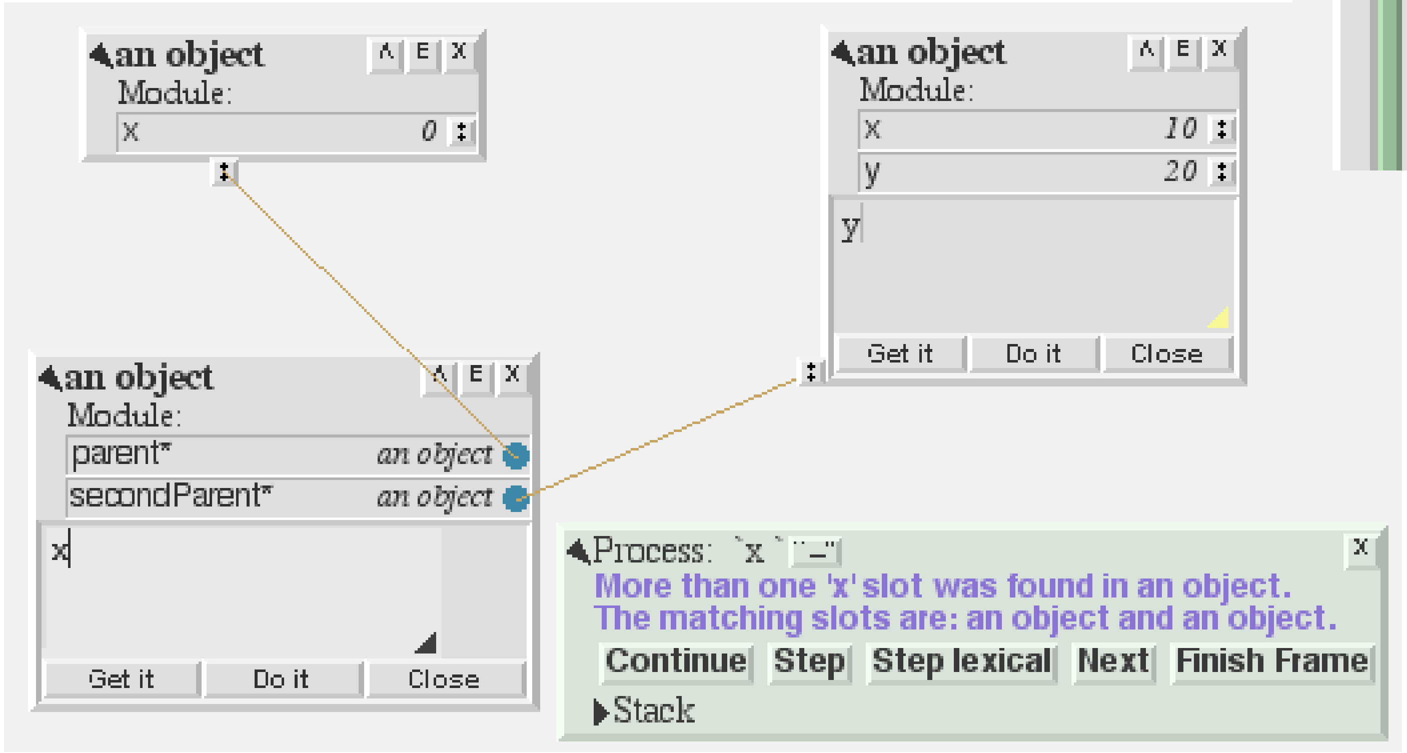
- The lookup algorithm recursively traverses the (possibly cyclic) inheritance graph
- The search starts in the object itself and then continues to search every parent.
    - No object is searched twice along any single path.
    - Parent slots are not evaluated during the lookup. if a parent slot contains an object with code, the code will not be executed; the object will merely be searched for matching slots.

Conflicts...



Syntax Error:
only a single * is allowed for a parent slot.

```
lowParent** <-1
```



More than one 'x' slot was found in an object.
The matching slots are: an object and an object.

Resending a Msg

- A resend allows an overridding method to invoke the overridden method.
  - Directed resends constrain the lookup to search a single parent slot.
- Resends and directed resends may change the name of the message being sent from the name of the current method, and may pass different arguments than the arguments passed to the current method.
- The receiver of a resend or a directed resend must be the implicit receiver.

Resend

- *resend.message*
  - no whitespace between . and other parts
- Intuitively, resend is similar to Java, Smalltalk's super send and CLOS' call-next-method.
- Examples:

```
resend.display
resend.+ 5
resend.min: 17 Max: 23
```

Directed Resend

- *parent-slot.message*
- Constrains the resend to the specified parent.
- Dangerous because it fixes the parent.
- Examples:

```
listParent.height
intParent.min: 17 Max: 23
```

Wrap-up

- Prototype-based language concepts:
  - objects + message sending + delegation
  - object creation: ex-nihilo, cloning, extension
- Traits and maps provide ‘blueprints’
- Self: prototype language
  - unifies state and behaviour
    - through activation contexts
  - maps created implicitly when cloning

References

- http://www.ulb.ac.be/di/rwuyts/INFO020_2003/