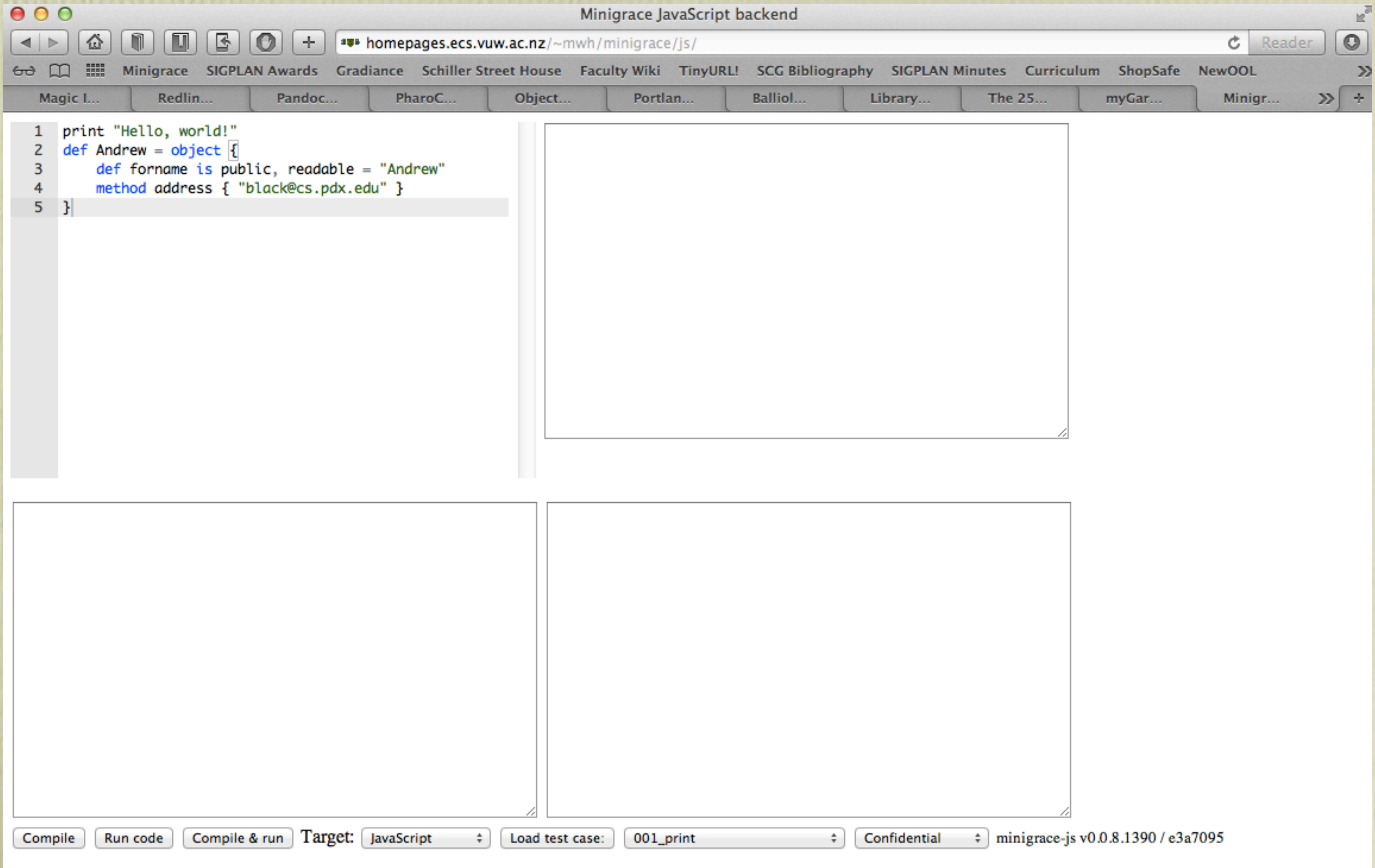
Traits as Objects in Grace

Andrew P. Black
Portland State University

Portland State
UNIVERSITY

minigrace



Minigrace JavaScript backend
homepages.ecs.vuw.ac.nz/~mwh/minigrace/js/
Reader
Minigrace
SIGPLAN Awards
Gradiance
Schiller Street House
Faculty Wiki
TinyURL!
SCG Bibliography
SIGPLAN Minutes
Curriculum
ShopSafe
NewOOL
Magic I...
Redlin...
Pandoc...
PharoC...
Object...
Portlan...
Balliol...
Library...
The 25...
myGar...
Minigr...
print "Hello, world!"
def Andrew = object {
def forname is public, readable = "Andrew"
method address { "black@cs.pdx.edu" }
}
Compile
Run code
Compile & run
Target:
JavaScript
Load test case:
001_print
Confidential
minigrace-js v0.0.8.1390 / e3a7095

Disclaimer

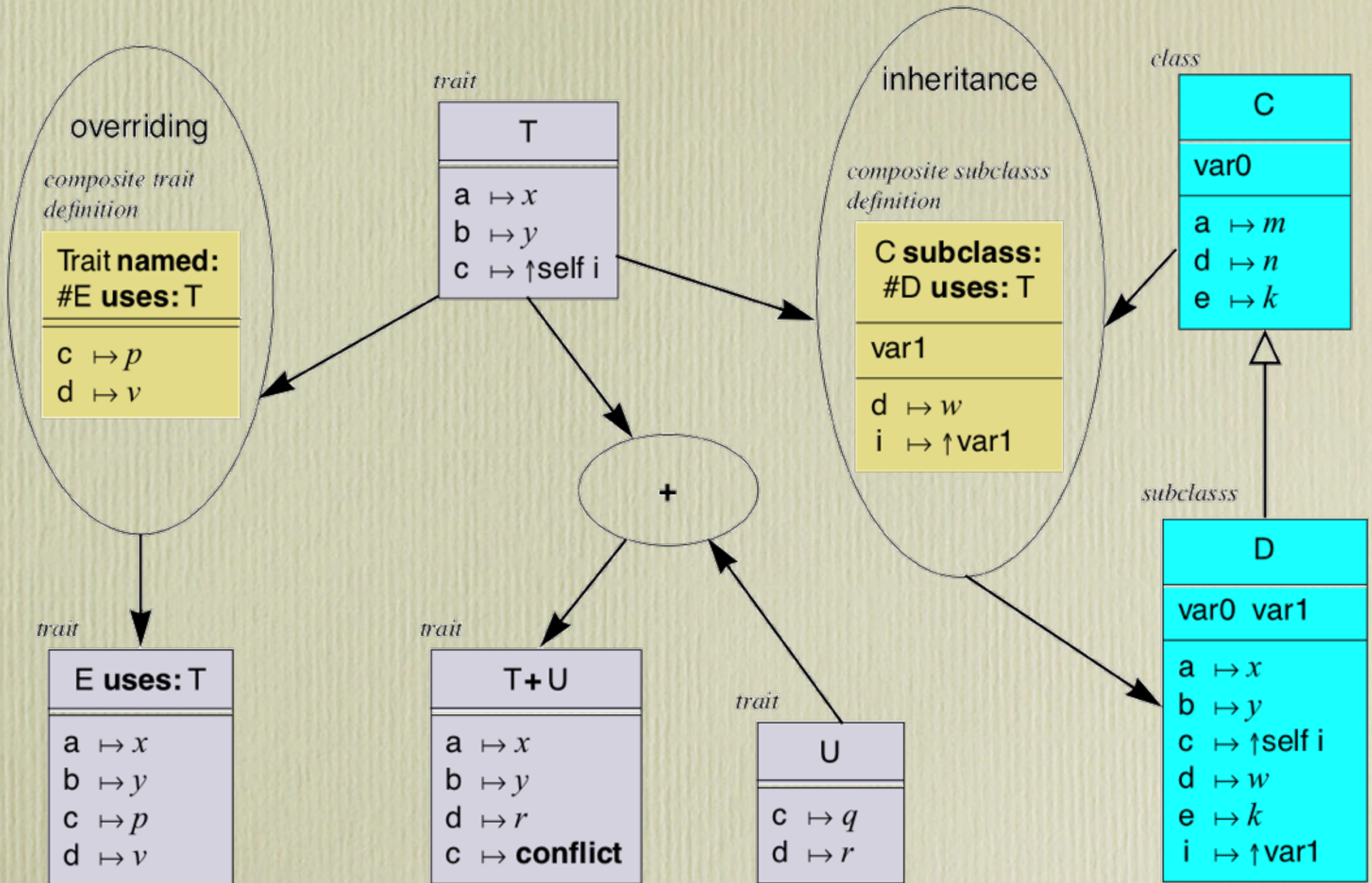
The views expressed here are my own,
and may not be representative of those of
the Grace design team

Disagreement has had a powerful beneficial
effect on the design of Grace

Background on Traits

- "Traits" = Smalltalk traits as described by Schärli *et al.* [ECOOP 2004, TOPLAS 2006]:
  - algebra of method combination
  - a trait is a set of named methods with operations +, -, @ and uses
  - traits have no state; just *pure* methods and bindings to self
- There are other definitions of trait, *e.g.*,
  - Curry *et al.* [SIGOA 1982]
  - Reppy & Turon [ECOOP 2007]

Portland State UNIVERSITY



overriding
composite trait definition
Trait named: #E uses: T
c ↦ p
d ↦ v
trait
T
a ↦ x
b ↦ y
c ↦ ↑self i
inheritance
composite subclasss definition
C subclass: #D uses: T
var1
d ↦ w
i ↦ ↑var1
class
C
var0
a ↦ m
d ↦ n
e ↦ k
+
subclasss
D
var0 var1
a ↦ x
b ↦ y
c ↦ ↑self i
d ↦ w
e ↦ k
i ↦ ↑var1
trait
E uses: T
a ↦ x
b ↦ y
c ↦ p
d ↦ v
trait
T+U
a ↦ x
b ↦ y
d ↦ r
c ↦ conflict
trait
U
c ↦ q
d ↦ r

What's Good about Traits

- "Despite their relative simplicity, traits offer a surprisingly rich calculus." [Reppy & Turon 2007]
- "more nimble and lighter-weight than either multiple inheritance or mixins" [ibid.]
  - but you can do MI and mixin-like stuff with traits
- Separates the unit of reuse (the trait) from the generator of objects (the class)
  - Classes struggle to fill both roles:

Fine-grained,
often incomplete

Complete,
monolithic

Grace doesn't have Traits

and I agreed to this!

- Why?
  - Designed for teaching, not large-scale software engineering
    - So code reuse is not so important (?)
  - Traits are not "mainstream"
    - We need to teach what is in common use
  - Inheritance is "mainstream" object-orientation
    - So Grace *must* contain inheritance
  - Grace is small and simple
    - So it should not have *two* reuse mechanisms

Andrew’s working hypothesis

- It's possible, in Grace to:
  - provide something very like traits using objects
  - build something very like inheritance out of traits
  - build more than one variety of inheritance
- This is a *Good Idea* because:
  - Core Grace would have one reuse mechanism, but
  - Grace could be used to teach a variety of reuse mechanisms

Portland State University

Objects in Grace

- Everything is an Object
  - but every object is not an instance of a class
- Instead: objects are self-contained
- Objects are created by executing an object constructor

```
object {
   def x:Number is public, readable = 2
   def y:Number is public, readable = 3
   method distanceTo (other:Point) → Number {
      ((x - other.x)^2 + (y - other.y)^2) } }
```

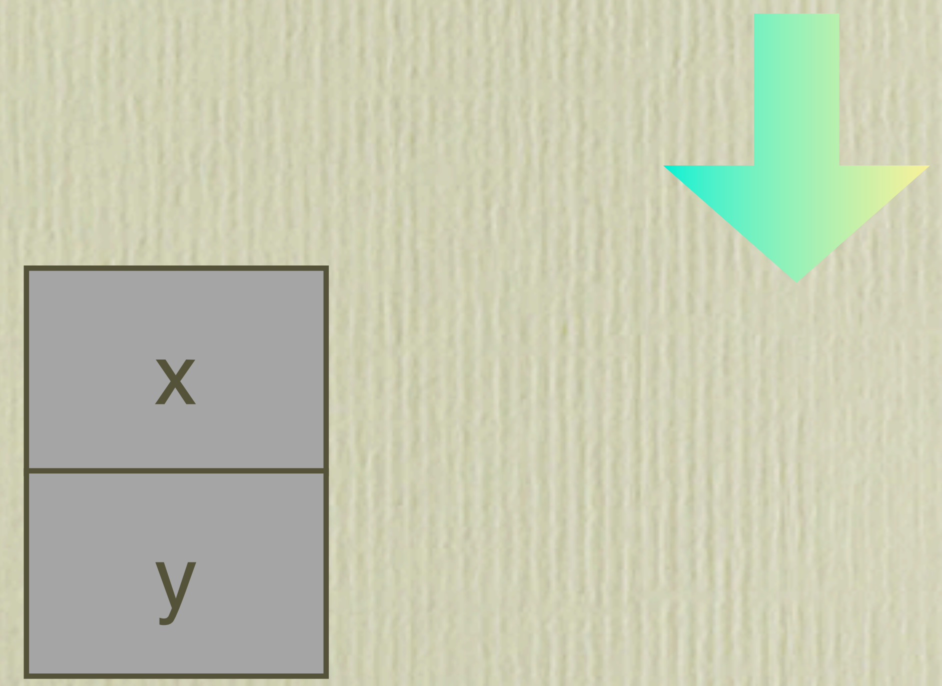
```
object {
  def x:Number is public, readable = 2
  def y:Number is public, readable = 3
  method distanceTo (other:Point) → Number {
    ((x - other.x)^2 + (y - other.y)^2) } }
```

Portland State University

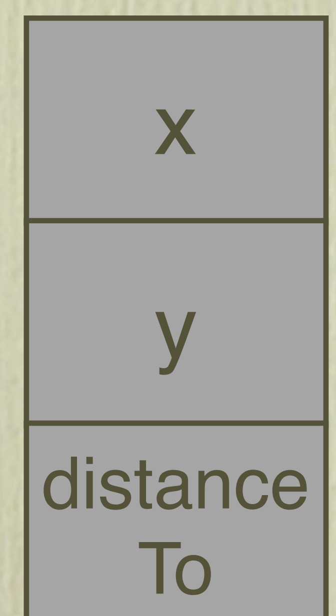
```
object {
  def x:Number is public, readable = 2
  def y:Number is public, readable = 3
  method distanceTo (other:Point) → Number {
    ((x - other.x)^2 + (y - other.y)^2) } }
```

```
object {
  def x:Number is public, readable = 2
  def y:Number is public, readable = 3
  method distanceTo (other:Point) → Number {
    ((x - other.x)^2 + (y - other.y)^2) } }
```

```
object {
  def x:Number is public, readable = 2
  def y:Number is public, readable = 3
  method distanceTo (other:Point) → Number {
    ((x - other.x)^2 + (y - other.y)^2) } }
```

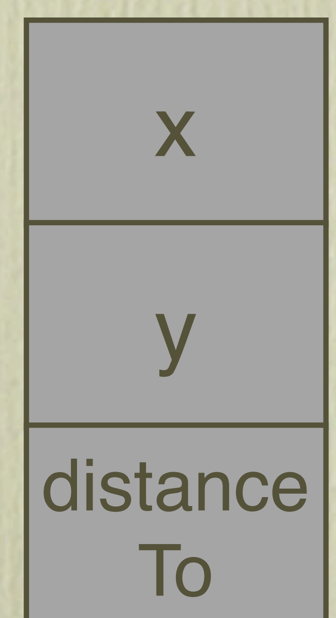


```
object {
  def x:Number is public, readable = 2
  def y:Number is public, readable = 3
  method distanceTo (other:Point) → Number {
    ((x - other.x)^2 + (y - other.y)^2) } }
```

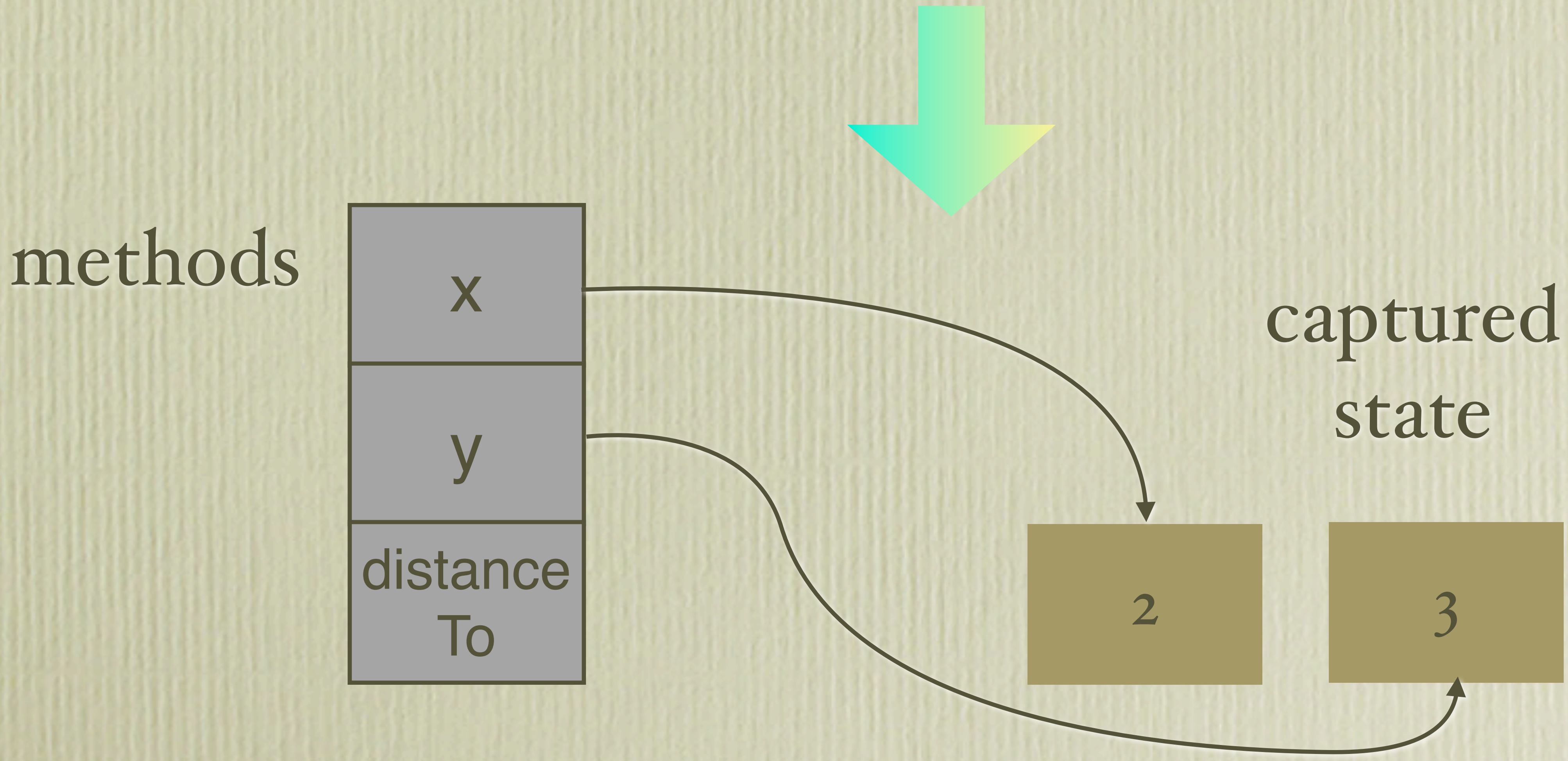


```
object {
   def x:Number is public, readable = 2
   def y:Number is public, readable = 3
   method distanceTo (other:Point) → Number {
      ((x - other.x)^2 + (y - other.y)^2) } }
```

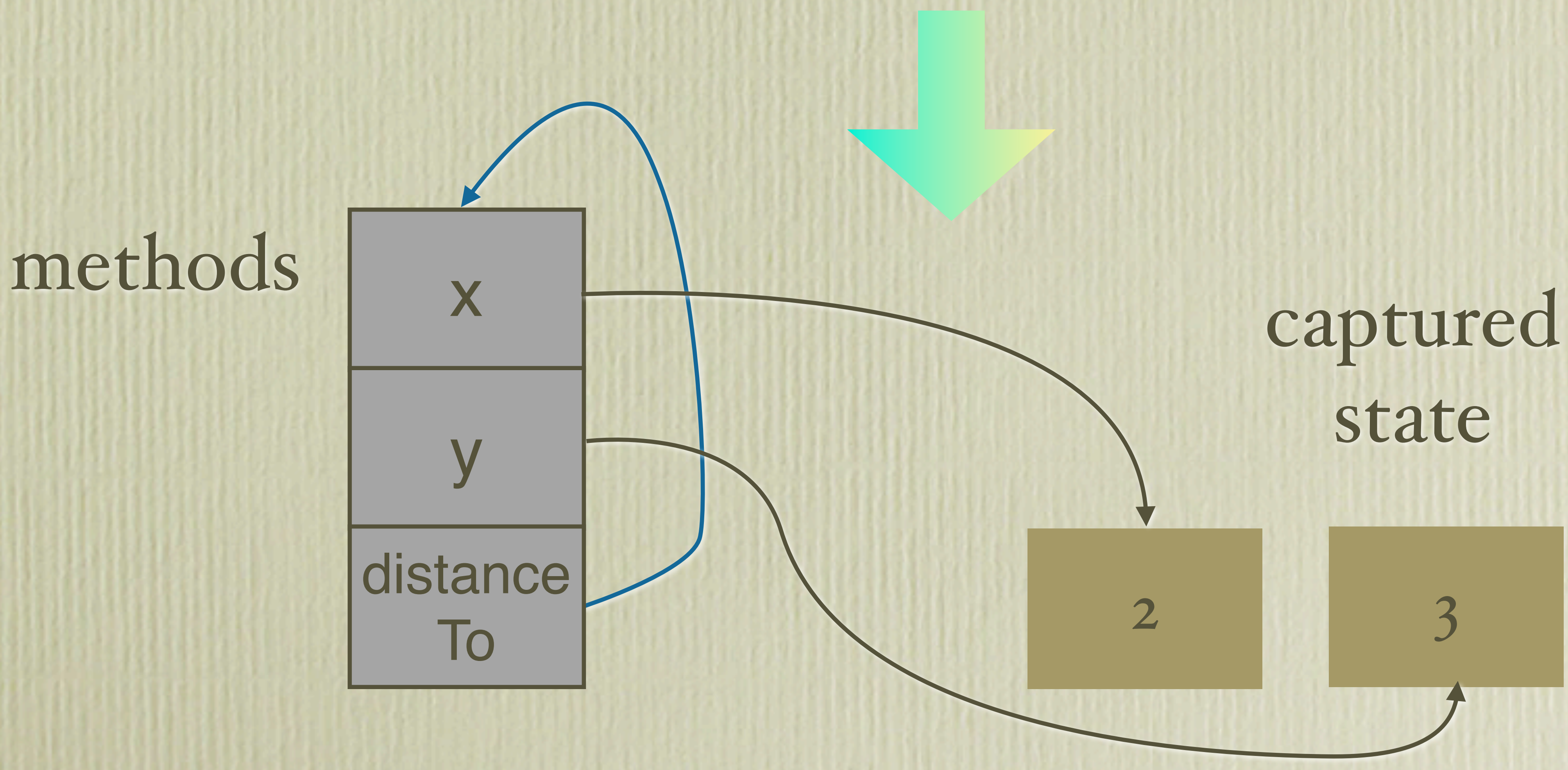
methods



```
object {
  def x:Number is public, readable = 2
  def y:Number is public, readable = 3
  method distanceTo (other:Point) → Number {
    ((x - other.x)^2 + (y - other.y)^2) } }
```

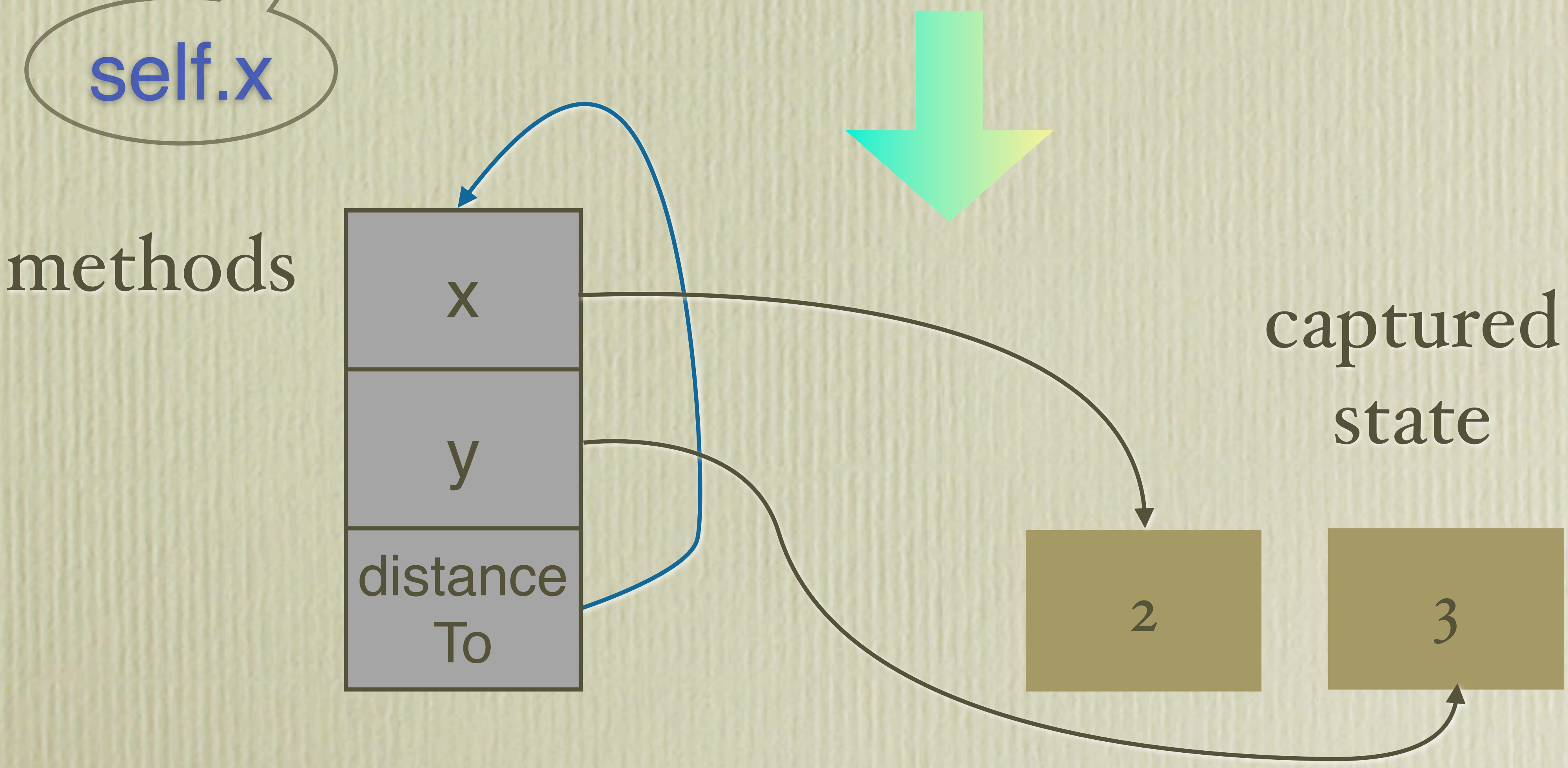


```
object {
   def x:Number is public, readable = 2
   def y:Number is public, readable = 3
   method distanceTo (other:Point) → Number {
      ((x - other.x)^2 + (y - other.y)^2) } }
```



```
object {
  def x:Number is public, readable = 2
  def y:Number is public, readable = 3
  method distanceTo (other:Point) → Number {
    ((x - other.x)^2 + (y - other.y)^2) } }
```

self.x



Notice:

- Nothing in an object but methods
- No “instance variables” *per se*
  - methods can capture any def or var in scope
- Objects can be *created* with interesting fields
  - makes it simple to define simple objects
  - the *only* way to create objects with def fields

An aside on self

- In the previous figure, self is treated just like any other bound variable
- Alternative: self means “the receiver”
- What's the difference?

An aside on self

- In the previous figure, self is treated just like any other bound variable
- Alternative: self means “the receiver”
- What's the difference?

```
object {
  def mouseAction is public, readable = { self.click }
  method click is public { self.highlight; self.doIt }
  method doIt = { … }
}
```

from one to many

from one to many

```
object {
  def x:Number is public, readable = 2
  def y:Number is public, readable = 3
  method distanceTo (other:Point) → Number {
    ((x - other.x)^2 + (y - other.y)^2) } }
```

from one to many

```
object {
   def x:Number is public, readable = 2
   def y:Number is public, readable = 3
   method distanceTo (other:Point) → Number {
      ((x - other.x)^2 + (y - other.y)^2) } }
```

```
def aPoint = object {
     method x(xcoord)y(ycoord) {
       object {
         def x:Number is public, readable = xcoord
         def y:Number is public, readable = ycoord
         method distanceTo (other:Point) → Number {
            ((x - other.x)^2 + (y - other.y)^2) } } } }
```

```
def aPoint = object {
    method x(xcoord)y(ycoord) {
      object {
        def x:Number is public, readable = xcoord
        def y:Number is public, readable = ycoord
        method distanceTo (other:Point) → Number {
            ((x - other.x)^2 + (y - other.y)^2) } } } }
```

Portland State UNIVERSITY

```
def aPoint = object {
    method x(xcoord)y(ycoord) {
      object {
        def x:Number is public, readable = xcoord
        def y:Number is public, readable = ycoord
        method distanceTo (other:Point) → Number {
            ((x - other.x)^2 + (y - other.y)^2) } } } }
```

aPoint:

Portland State UNIVERSITY

```
def aPoint = object {
    method x(xcoord)y(ycoord) {
       object {
          def x:Number is public, readable = xcoord
          def y:Number is public, readable = ycoord
          method distanceTo (other:Point) → Number {
                ((x - other.x)^2 + (y - other.y)^2) } } } }
```

aPoint:

methods x()y()

```
def aPoint = object {
    method x(xcoord)y(ycoord) {
      object {
        def x:Number is public, readable = xcoord
        def y:Number is public, readable = ycoord
        method distanceTo (other:Point) → Number {
          ((x - other.x)^2 + (y - other.y)^2) } } } }
```

aPoint:

methods x()y()

aPoint is a class object

```
def aPoint = object {
    method x(xcoord)y(ycoord) {
      object {
        def x:Number is public, readable = xcoord
        def y:Number is public, readable = ycoord
        method distanceTo (other:Point) → Number {
            ((x - other.x)^2 + (y - other.y)^2) } } } }
```

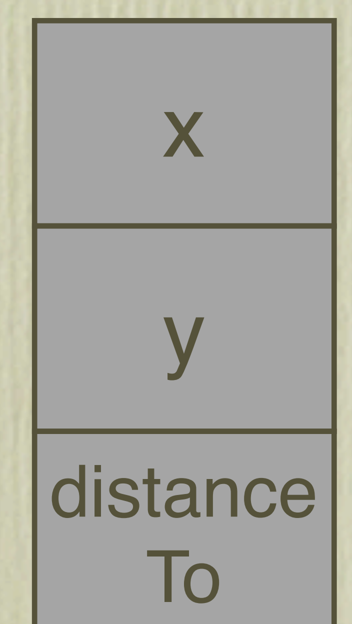
aPoint.x(2)y(3):

Portland State UNIVERSITY

```
def aPoint = object {
    method x(xcoord)y(ycoord) {
      object {
        def x:Number is public, readable = xcoord
        def y:Number is public, readable = ycoord
        method distanceTo (other:Point) → Number {
            ((x - other.x)^2 + (y - other.y)^2) } } } }
```

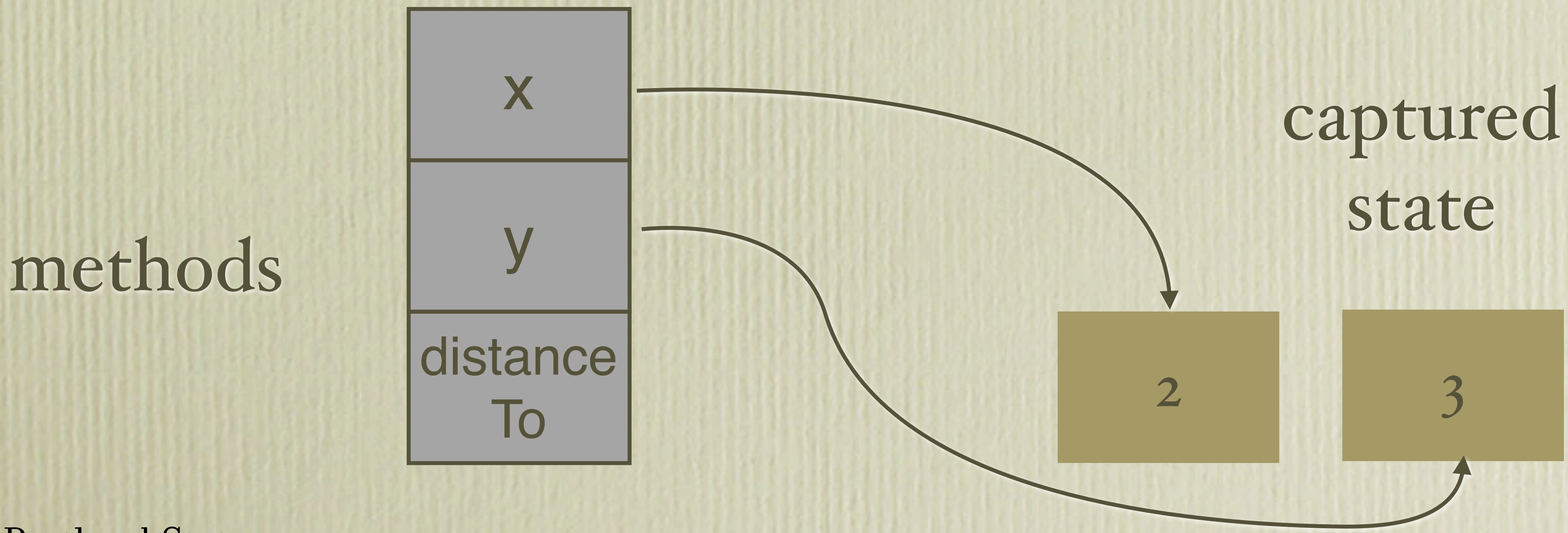
aPoint.x(2)y(3):

methods



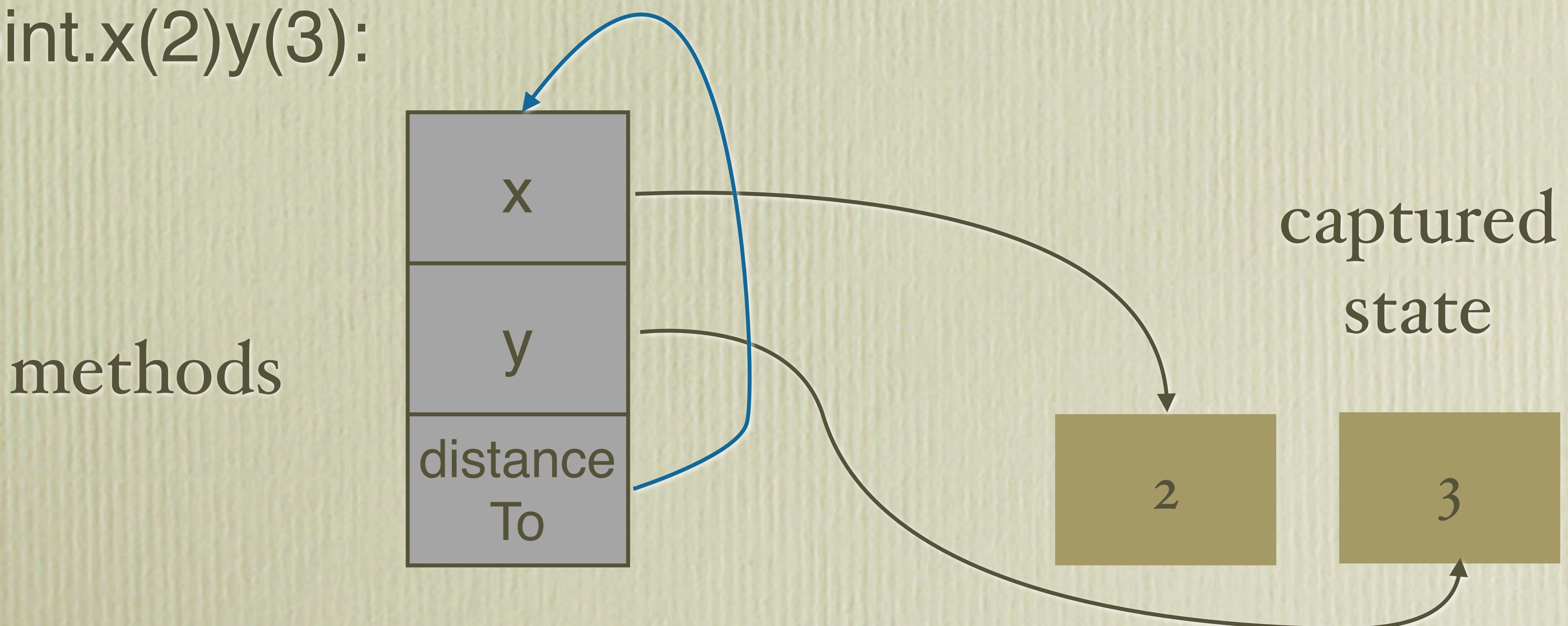
```
def aPoint = object {
    method x(xcoord)y(ycoord) {
      object {
        def x:Number is public, readable = xcoord
        def y:Number is public, readable = ycoord
        method distanceTo (other:Point) → Number {
            ((x - other.x)^2 + (y - other.y)^2) } } } }
```

aPoint.x(2)y(3):



```
def aPoint = object {
    method x(xcoord)y(ycoord) {
      object {
        def x:Number is public, readable = xcoord
        def y:Number is public, readable = ycoord
        method distanceTo (other:Point) → Number {
            ((x - other.x)^2 + (y - other.y)^2) } } } }
```

aPoint.x(2)y(3):



Inheritance, Version I

- Inheritance from objects
  - restricted to “definitively static” objects, to make the job of the static type-checker easier

```
object {
  inherits aPoint.x(2)y(3)
  def color is public, readable = aColor.black
}
```

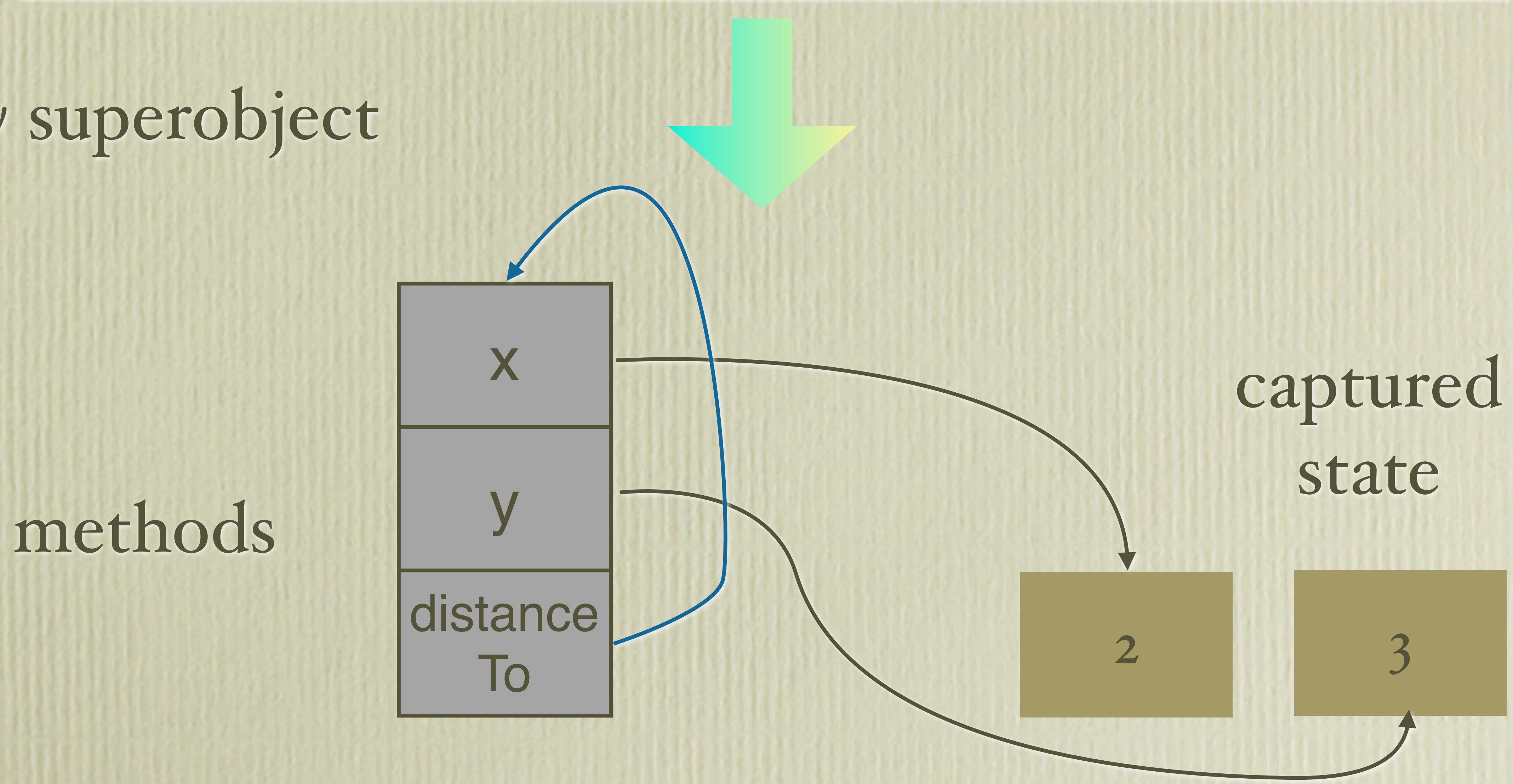
```
object {
  inherits aPoint.x(2)y(3)
  def color is public, readable = aColor.black
}
```

```
object {
  inherits aPoint.x(2)y(3)
  def color is public, readable = aColor.black
}
```

1. *Copy* superobject

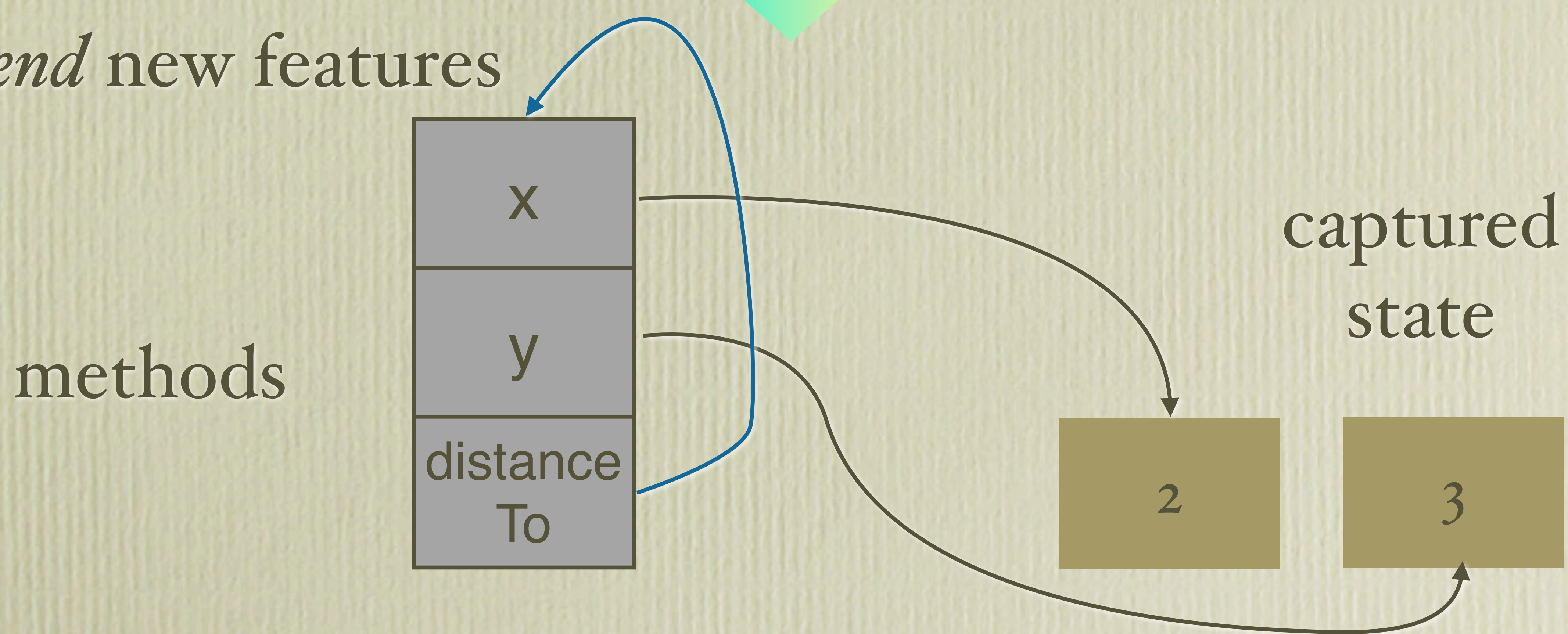
```
object {
  inherits aPoint.x(2)y(3)
  def color is public, readable = aColor.black
}
```

1. *Copy* superobject



```
object {
  inherits aPoint.x(2)y(3)
  def color is public, readable = aColor.black
}
```

1. *Copy* superobject
2. *Append* new features

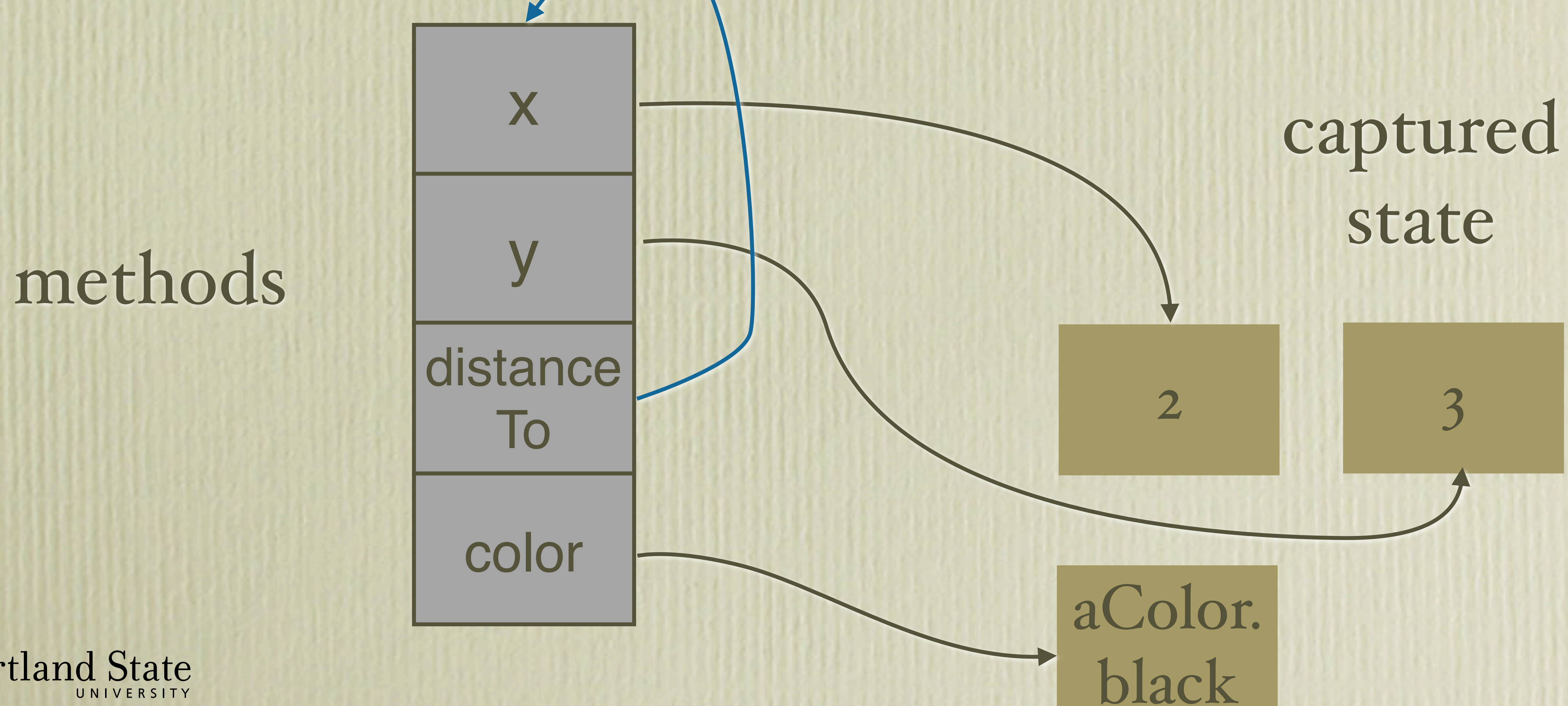


Portland State University

```
object {
  inherits aPoint.x(2)y(3)
  def color is public, readable = aColor.black
}
```

1. *Copy* superobject
2. *Append* new features

Optimization: *Copy* can be elided if safe



Conceptual Problem

- Every object from which one might wish to inherit must have a copy method

Practical Problem

- Referential Transparency: creation of super-object is oblivious to its context

```
def aPoint = object {
    method x(xcoord)y(ycoord) {
      object {
        def x:Number is public, readable = xcoord
        def y:Number is public, readable = ycoord
        method distanceTo (other:Point) → Number {
            ((x - other.x)^2 + (y - other.y)^2) }
        registry.add(self)
      } } }
```

Practical Problem

- Referential Transparency: creation of super-object is oblivious to its context

```
def aPoint = object {
    method x(xcoord)y(ycoord) {
      object {
        def x:Number is public, readable = xcoord
        def y:Number is public, readable = ycoord
        method distanceTo (other:Point) → Number {
            ((x - other.x)^2 + (y - other.y)^2) }
        registry.add(self)
} } }
```

registration
part of object *creation*

Problem: Referential Transparency

- registry.add(self) registers the *super-object*
  - this object is copied, then dropped

```
object {
  inherits aPoint.x(2)y(3)
  def color is public, readable = aColor.black
}
```

Inheritance, Version II

- Inheritance via object mutation
  - restricted to "definitively static" objects
  - restricted to "fresh" objects, to *hide* mutation

```
object {
  inherits aPoint.x(2)y(3)
  def color is public, readable = aColor.black
}
```

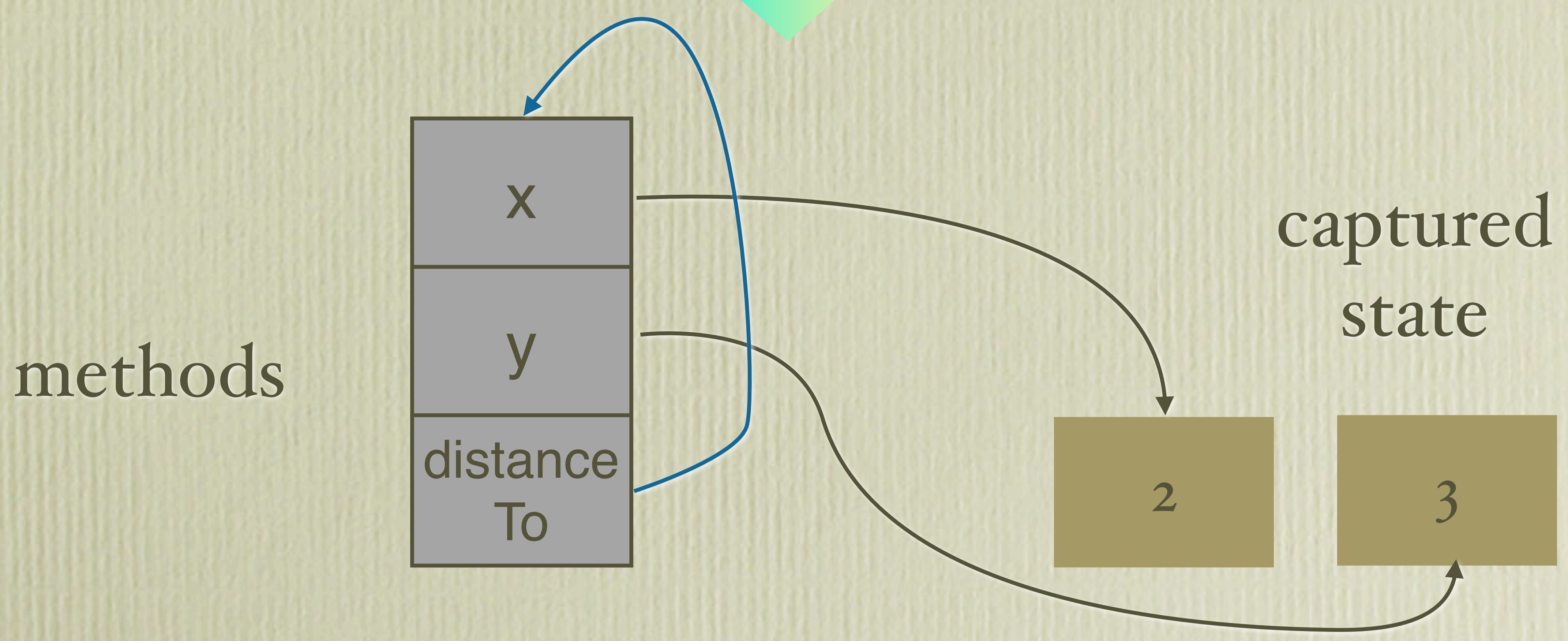
```
object {
  inherits aPoint.x(2)y(3)
  def color is public, readable = aColor.black
}
```

```
object {
  inherits aPoint.x(2)y(3)
  def color is public, readable = aColor.black
}
```

1. Start with actual superobject

```
object {
  inherits aPoint.x(2)y(3)
  def color is public, readable = aColor.black
}
```

1. Start with actual superobject

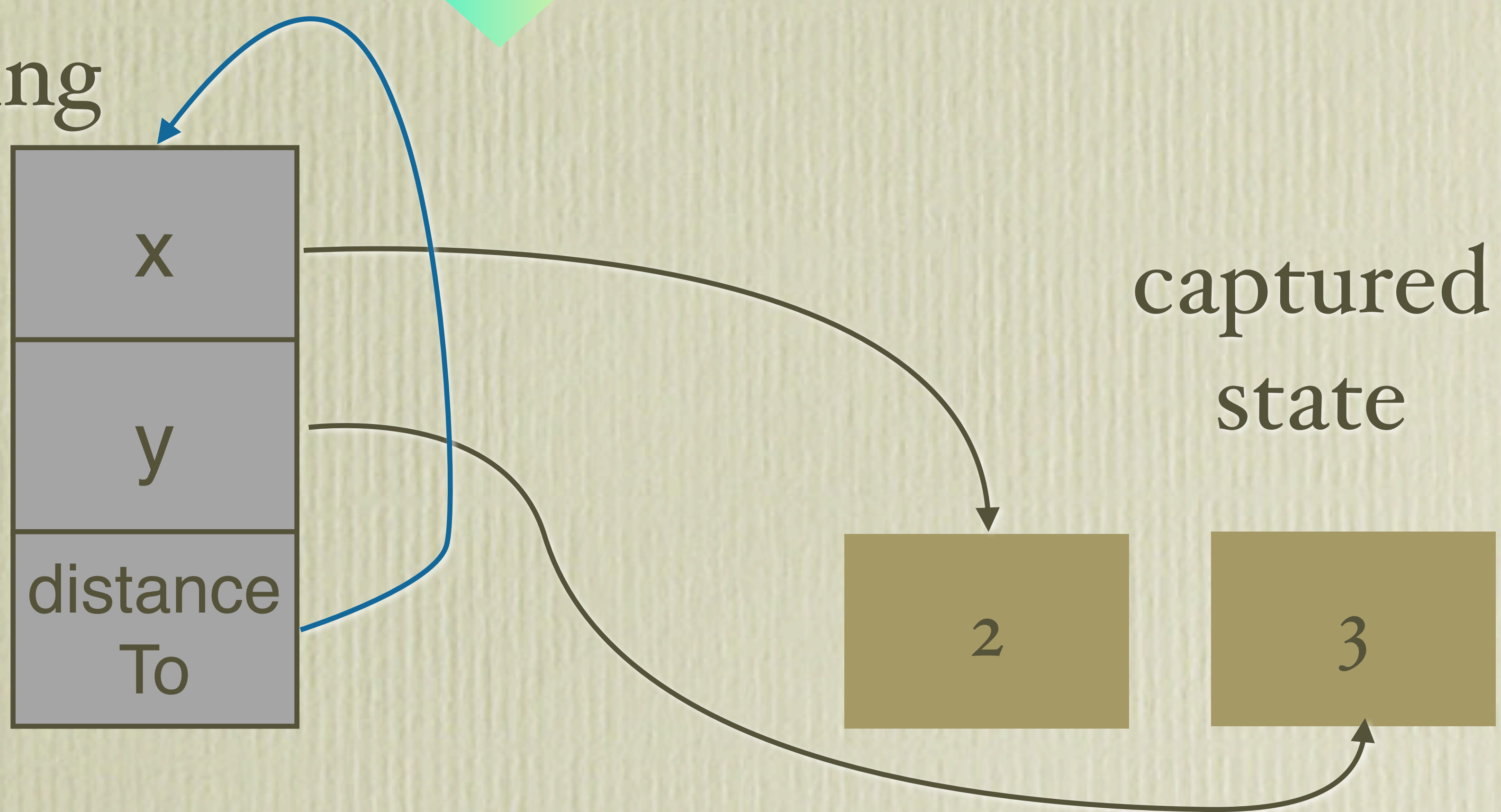


```
object {
  inherits aPoint.x(2)y(3)
  def color is public, readable = aColor.black
}
```

1. Start with actual superobject

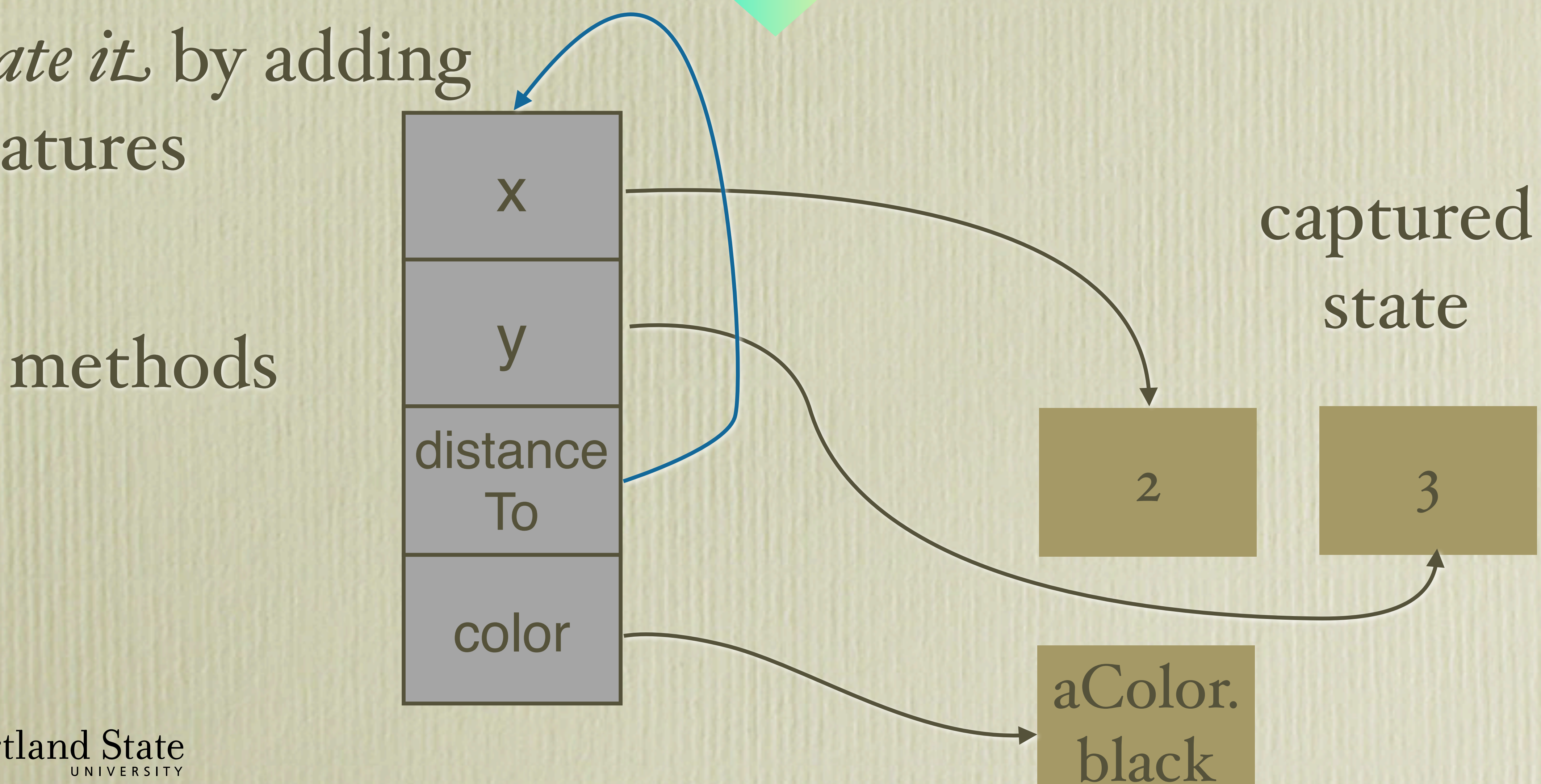
2. *Mutate it* by adding new features

methods



```
object {
  inherits aPoint.x(2)y(3)
  def color is public, readable = aColor.black
}
```

1. Start with actual superobject
2. *Mutate it* by adding new features



Problem: can't inherit from objects

- "Freshness" requirement means that you must inherit from object *constructors*, or *copies*

Problem: can't inherit from objects

- "Freshness" requirement means that you must inherit from object *constructors*, or *copies*

```
class SuccessfulMatch.new(result', bindings') {
    inherits true
    def result is public, readable = result'
    def bindings is public, readable = bindings'
    method asString {
        "SuccessfulMatch(result = {result}, bindings = {bindings})"
    }
}
```

Problem: can't inherit from objects

- "Freshness" requirement means that you must inherit from object *constructors*, or *copies*

```
class SuccessfulMatch.new(result', bindings') {
  inherits true
    def result is public, readable = result'
    def bindings is public, readable = bindings'
    method asString {
        "SuccessfulMatch(result = {result}, bindings = {bindings})"
    }
}
```

Problem: can't inherit from objects

- "Freshness" requirement means that you must inherit from object *constructors*, or *copies*

```
class SuccessfulMatch.new(result', bindings') {
  inherits true
    def result is public, readable = result'
    def bindings is public, readable = bindings'
    method asString {
      "SuccessfulMatch(result = {result}, bindings = {bindings})"
    }
}
```

Portland State UNIVERSITY

Problem: can't inherit from objects

- "Freshness" requirement means that you must inherit from object *constructors*, or *copies*

Problem: can't inherit from objects

- "Freshness" requirement means that you must inherit from object *constructors*, or *copies*
- If we eliminate the freshness requirement, we are visibly mutating "immutable" objects

Problem: can’t inherit from objects

- “Freshness” requirement means that you must inherit from object *constructors*

Problem: can’t inherit from objects

- “Freshness” requirement means that you must inherit from object *constructors*

```
def AssertionTrait = object {
    method assert(bb: Boolean)description(str) is public {
      if (! bb) …
    }
    method deny(bb: Boolean)description(str) is public {
      assert (! bb) description (str)
    }
    …
}
```

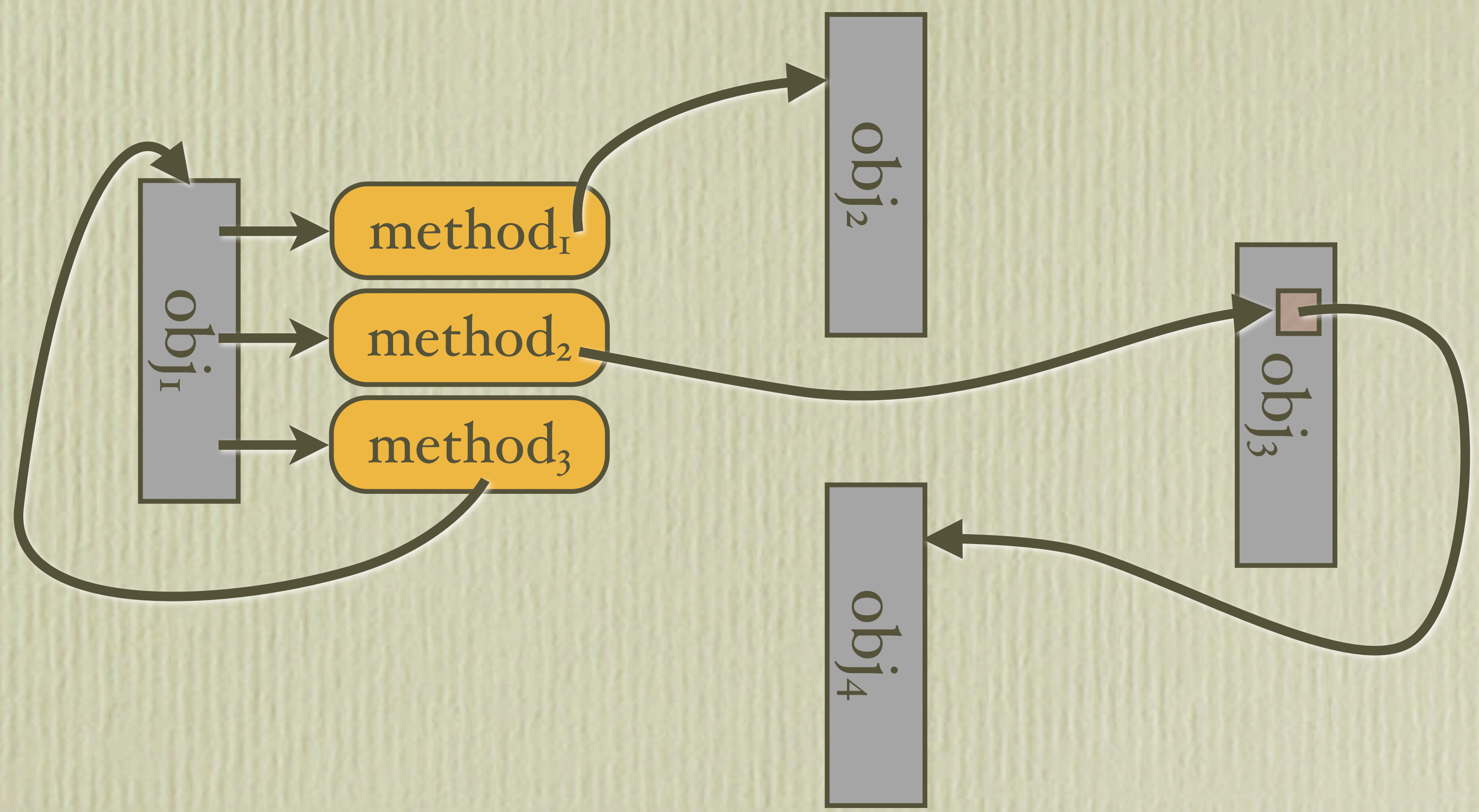
Problem: can't inherit from objects

- "Freshness" requirement means that you must inherit from object *constructors*

```
def AssertionTrait = object {
    method assert(bb: Boolean)description(str) is public {
      if (! bb) …
    }
    method deny(bb: Boolean)description(str) is public {
      assert (! bb) description (str)
    }
    …
}
```

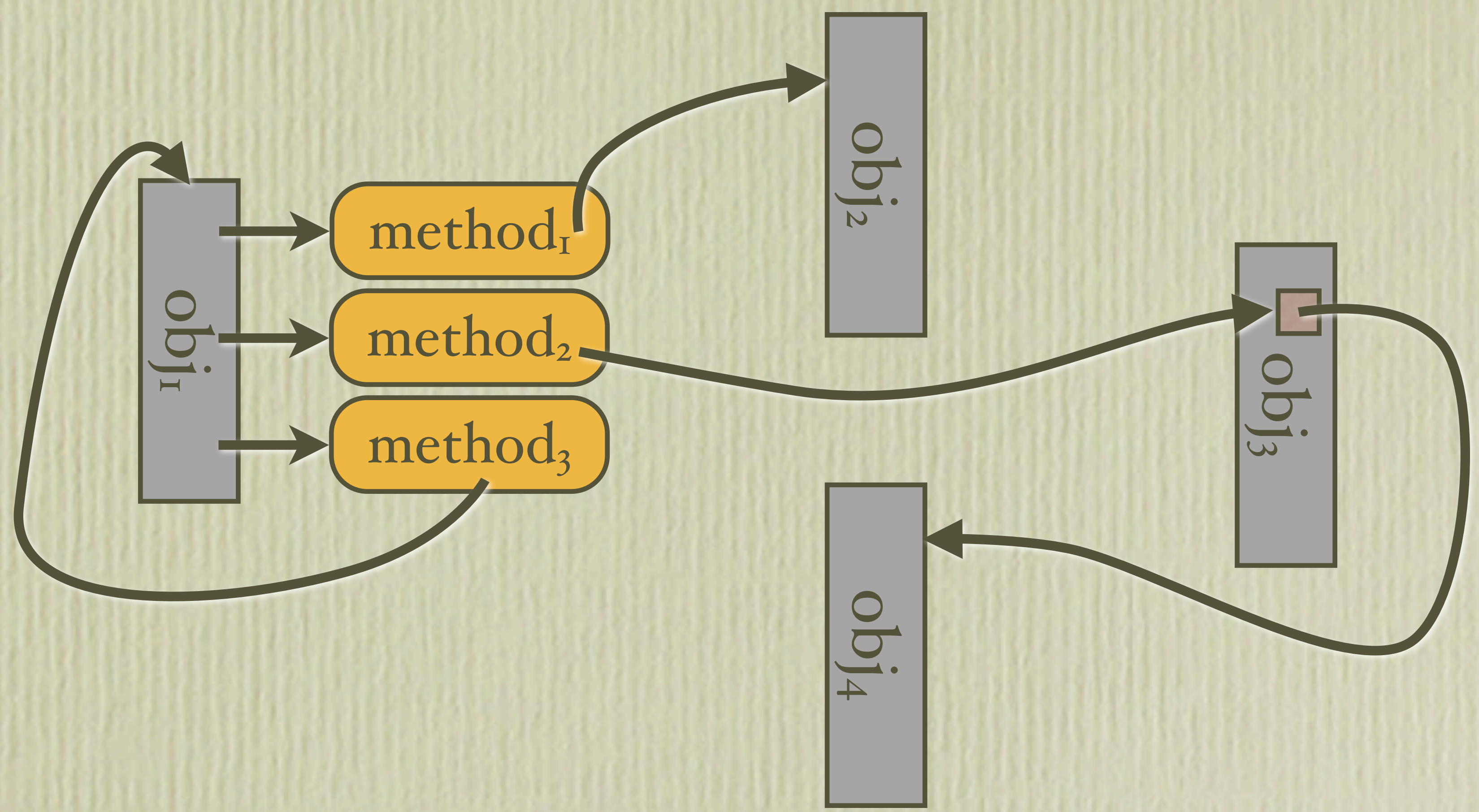
Conclusion: copy is essential

- Both versions of inheritance need copy as a primitive (built-in) method
- What does copy mean?

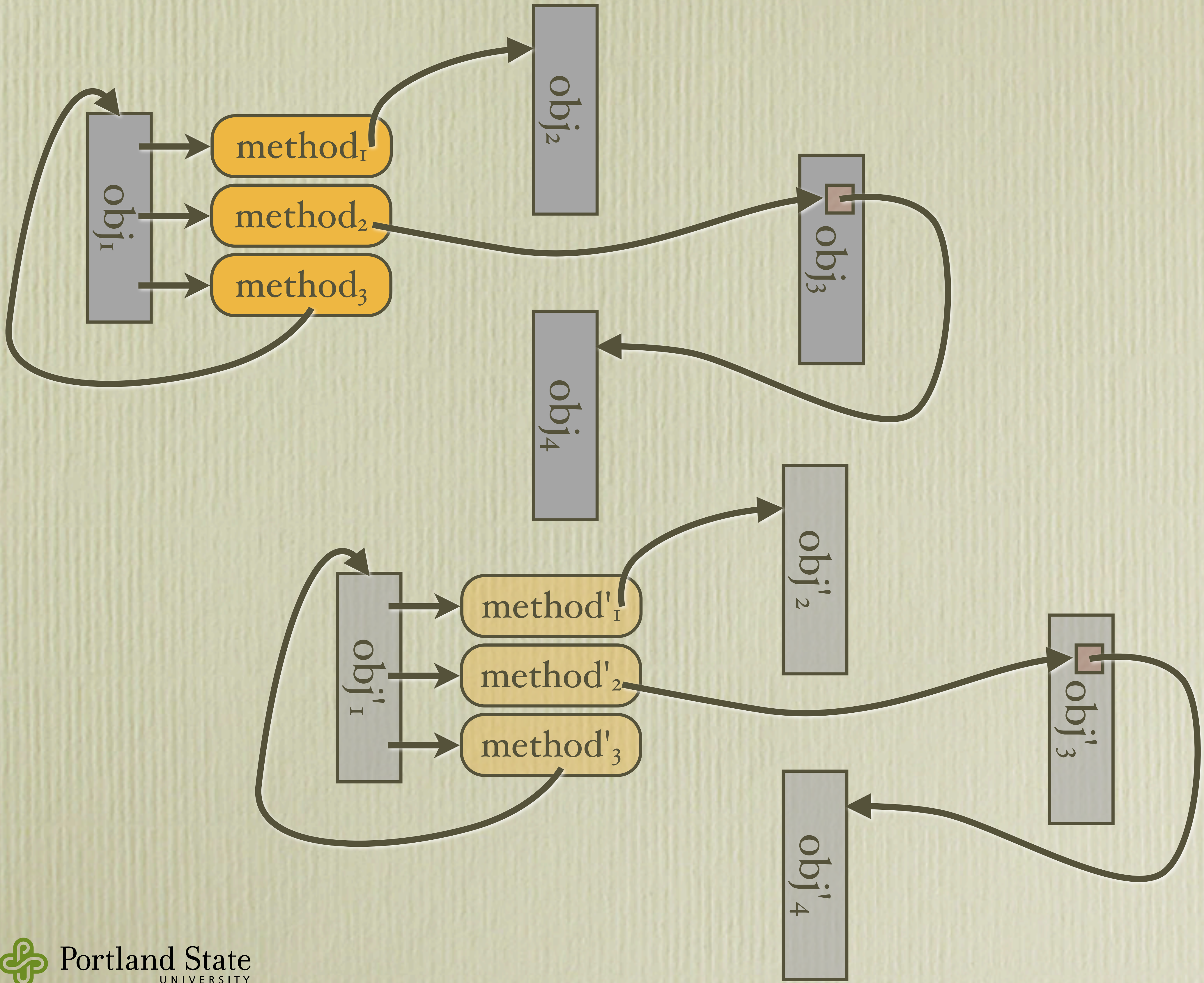


obj1
method1
method2
method3
obj2
obj3
obj4

obj_1.copy ?



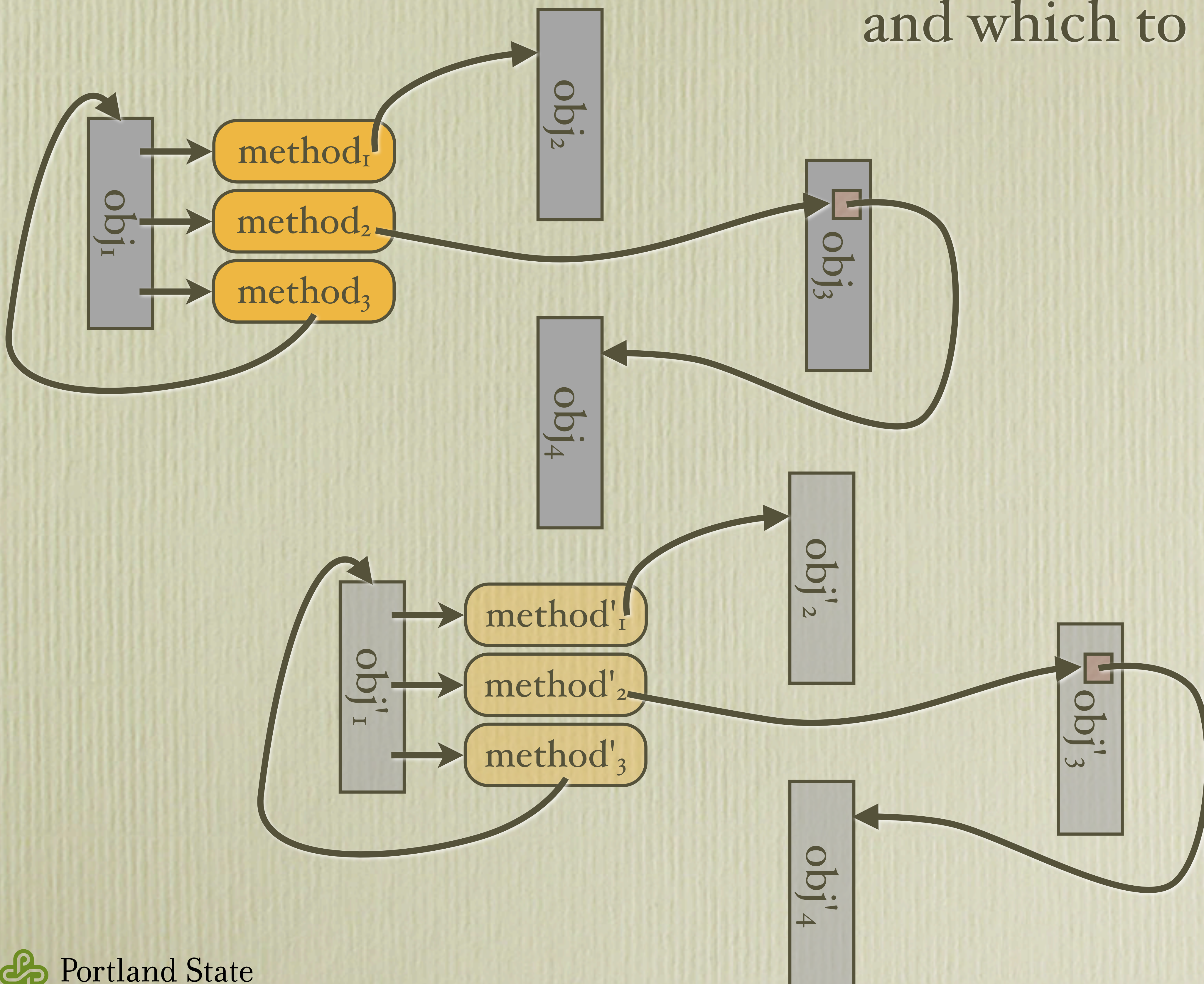
obj_1.copy ?



obj₁
method₁
method₂
method₃
obj₂
obj₃
obj₄
obj'₁
method'₁
method'₂
method'₃
obj'₂
obj'₃
obj'₄

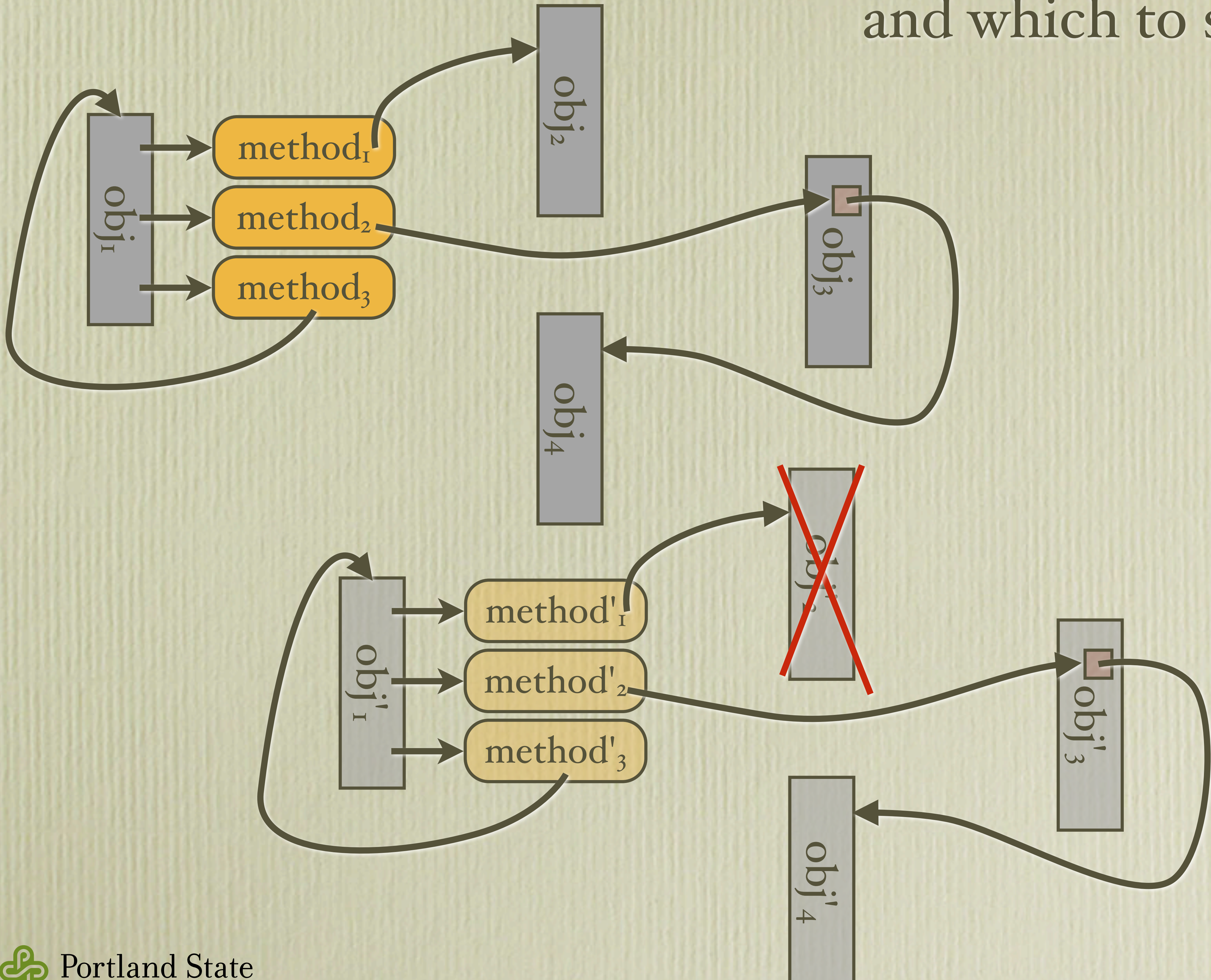
obj₁.copy ?

Which objects to copy and which to share?



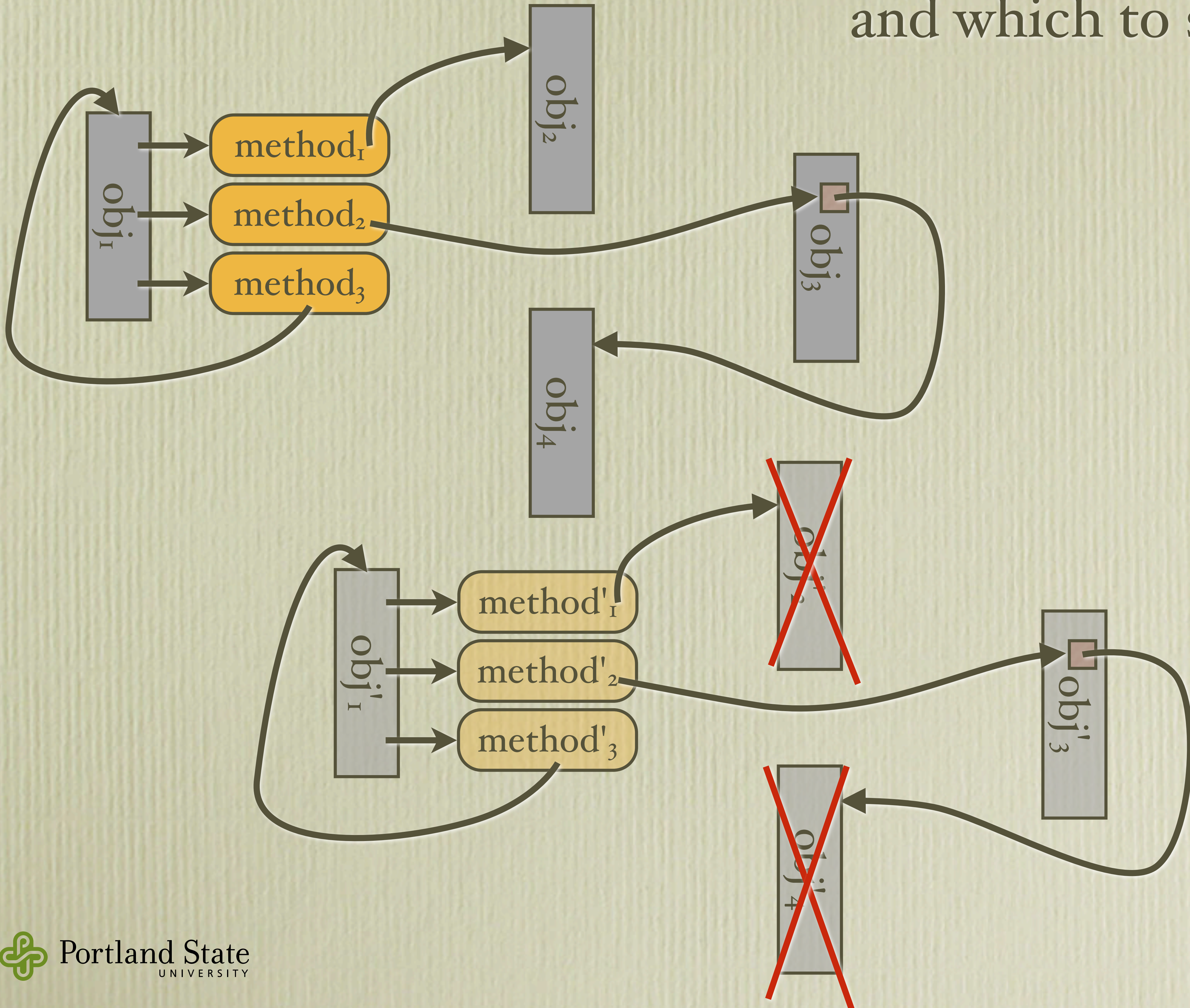
obj₁.copy ?

Which objects to copy and which to share?



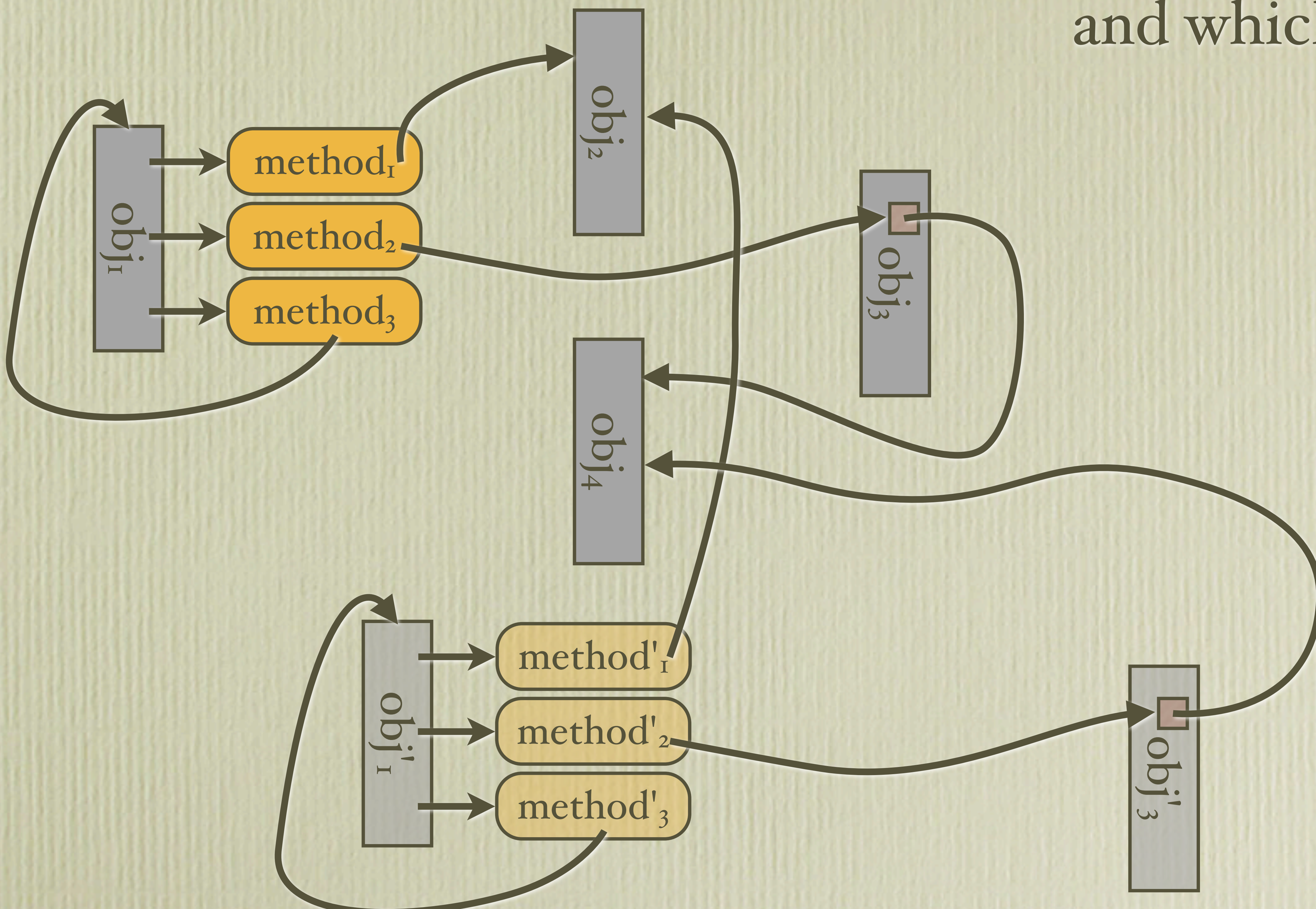
obj_1.copy ?

Which objects to copy and which to share?



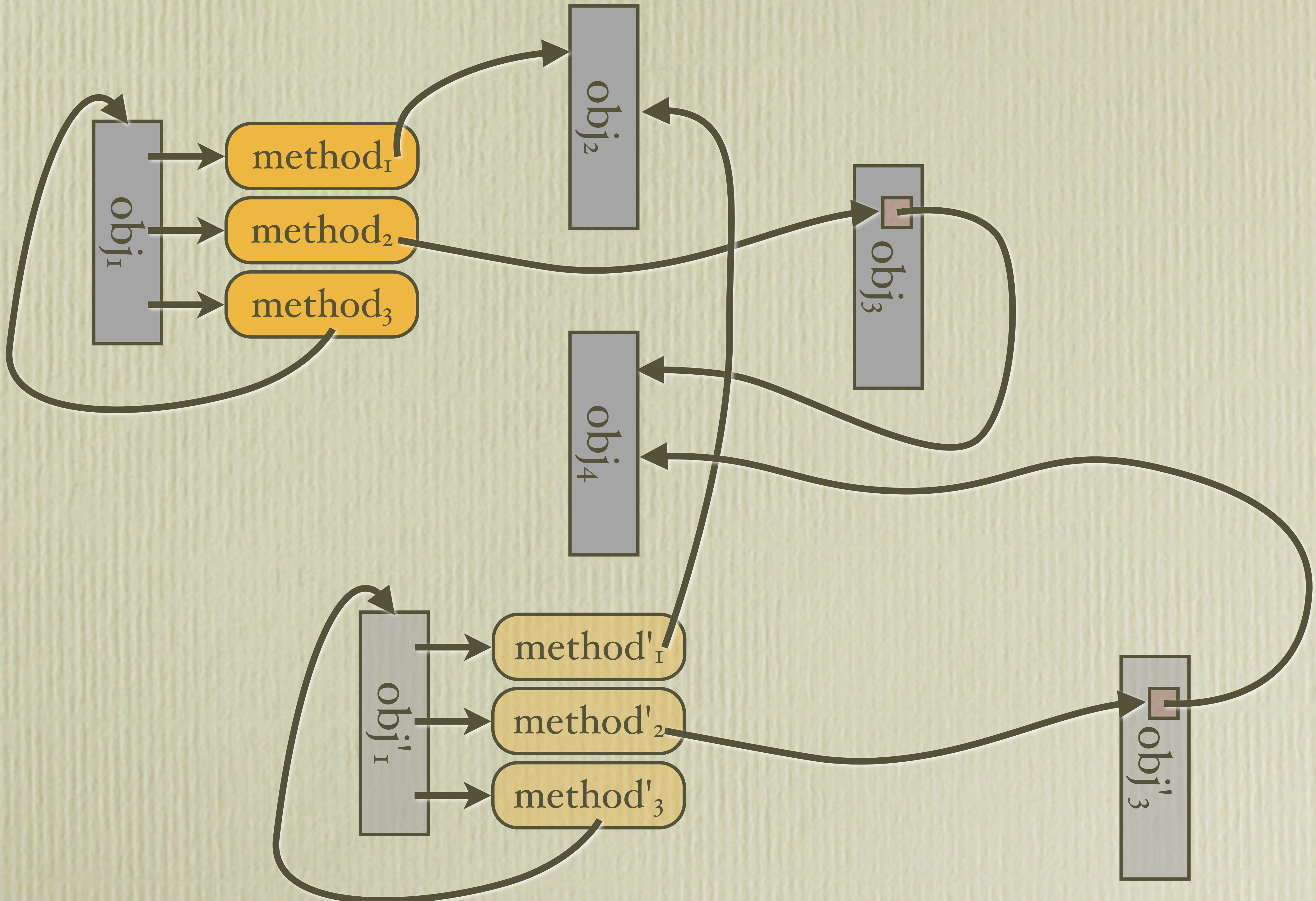
obj_1.copy ?

Which objects to copy and which to share?



$s(x) = x \in \{ obj_1, obj_3, method_1, method_2, method_3 \}$

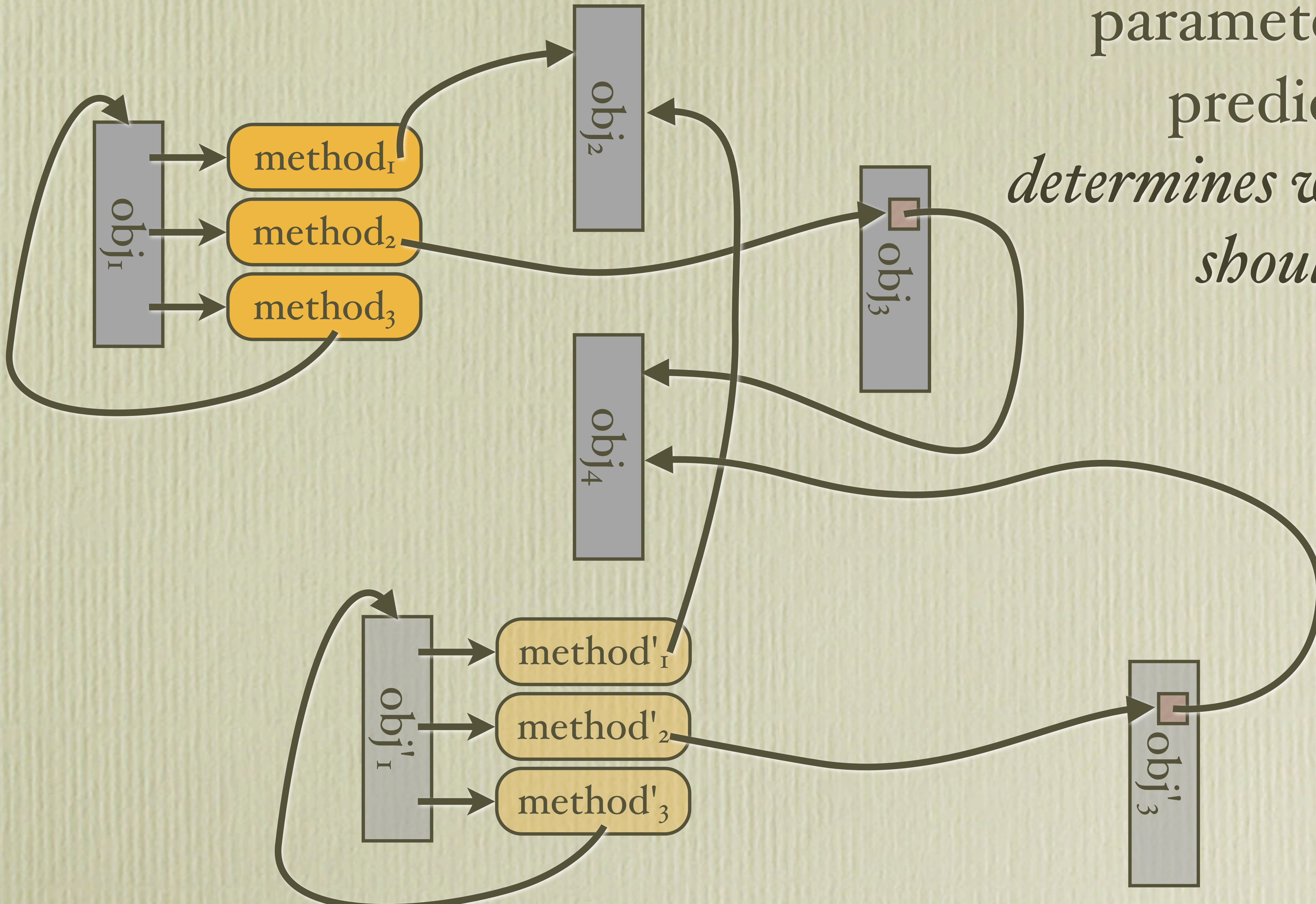
obj$_1$.copy ?



$s(x) = x \in \{ \text{obj}_1, \text{obj}_3, \text{method}_1, \text{method}_2, \text{method}_3 \}$

obj_1.copy ?

copy should be parameterized by a predicate *s that determines which nodes should be copied*



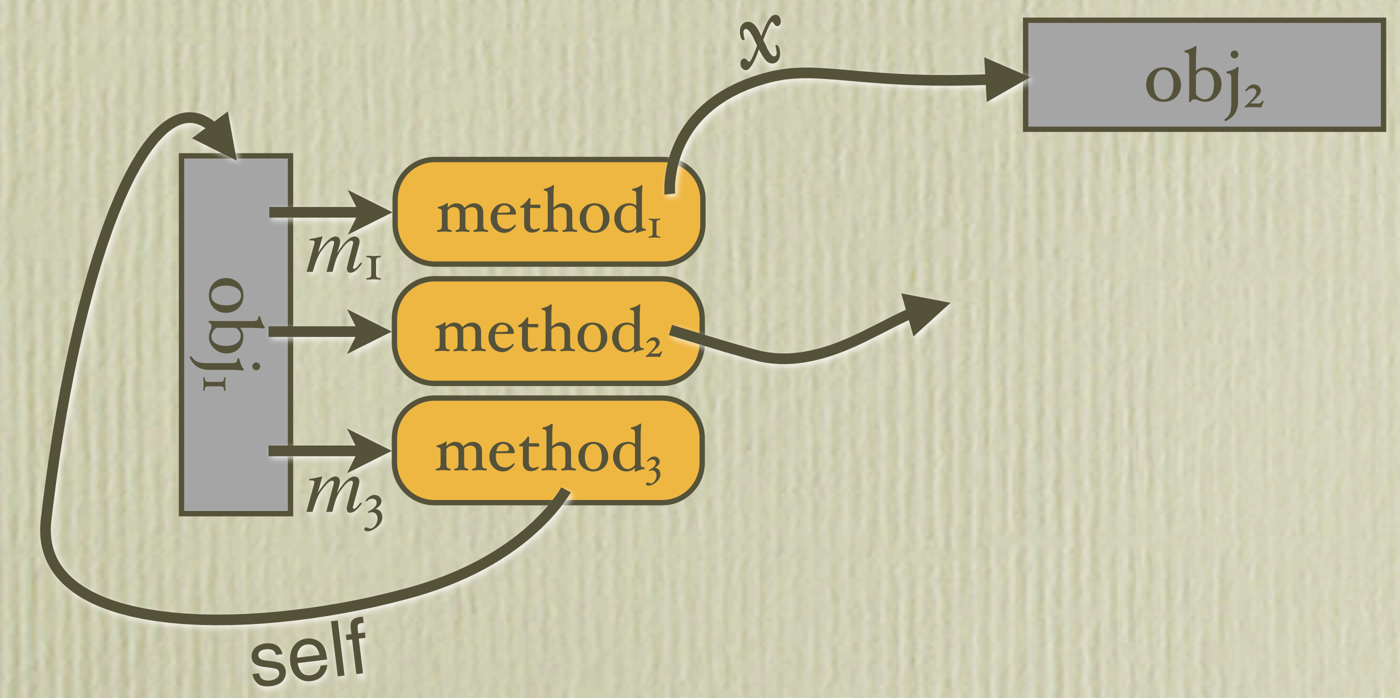
$$s(x) = x \in \{ obj_1, obj_3, method_1, method_2, method_3 \}$$

- $m, n \in N$ is a set of Nodes (objects, variables, methods)
- $x \in L$, a set of labels
- $p, q \in E \subset (N \times L \times N)$ is a set of Edges
(pointers, object references, variable references).
- If $\langle m, x, n\rangle \in E$, then node m has an edge labeled x leading to node n, and we write $m.x = n$
- A *path* (of length k) $\vec{x} = x_1 x_2 \cdots x_k \in L^k$ is *valid* from a root node n_0 exactly when $\exists n_1, n_2, \ldots, n_k \in N$ such that $\langle n_0, x_1, n_1\rangle \in E, \langle n_1, x_2, n_2\rangle \in E, \ldots, \langle n_{k-1}, x_k, n_k\rangle \in E$. We write $n_0.\vec{x} = n_k$
- F is a set of *fresh* nodes
- $s \in N \rightarrow \mathit{Boolean}$ is a shallowness function; if $s(n)$ then n should be copied, otherwise it should be shared.

- $copy_s : N \to N \cup F$ is a function with the following properties:

  1. $s(n) \equiv copy_s(n) \in F$
  2. $\neg s(n) \equiv copy_s(n) = n$
  3. $\forall n \in N, \vec{x} \in L^*, copy_s(n.\vec{x}) \equiv copy_s(n).\vec{x}$

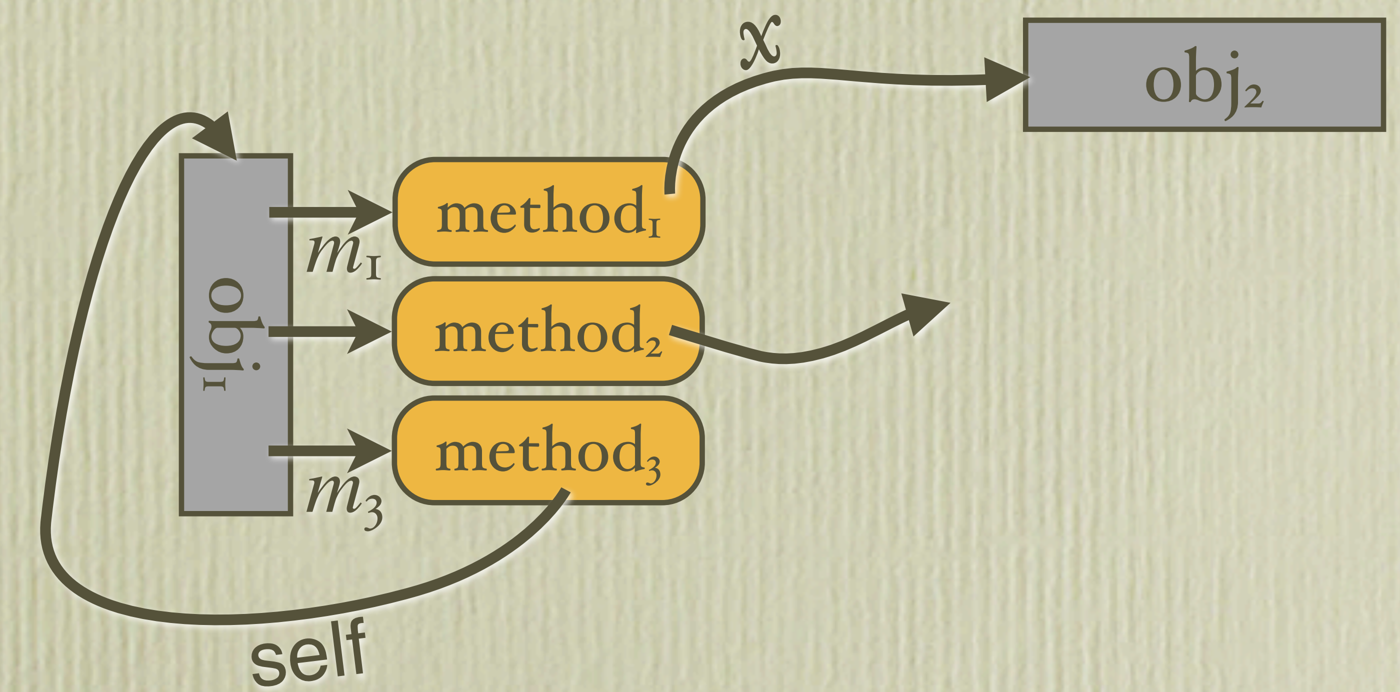
  The last equivalence means that the path on the rhs is valid exactly when the path on the lhs is valid, and that when both are valid, the object graphs commute.



- $copy_s : N \to N \cup F$ is a function with the following properties:

  1. $s(n) \equiv copy_s(n) \in F$
  2. $\neg s(n) \equiv copy_s(n) = n$
  3. $\forall n \in N, \vec{x} \in L^*, copy_s(n.\vec{x}) \equiv copy_s(n).\vec{x}$

  The last equivalence means that the path on the rhs is valid exactly when the path on the lhs is valid, and that when both are valid, the object graphs commute.



Paths from obj_1:

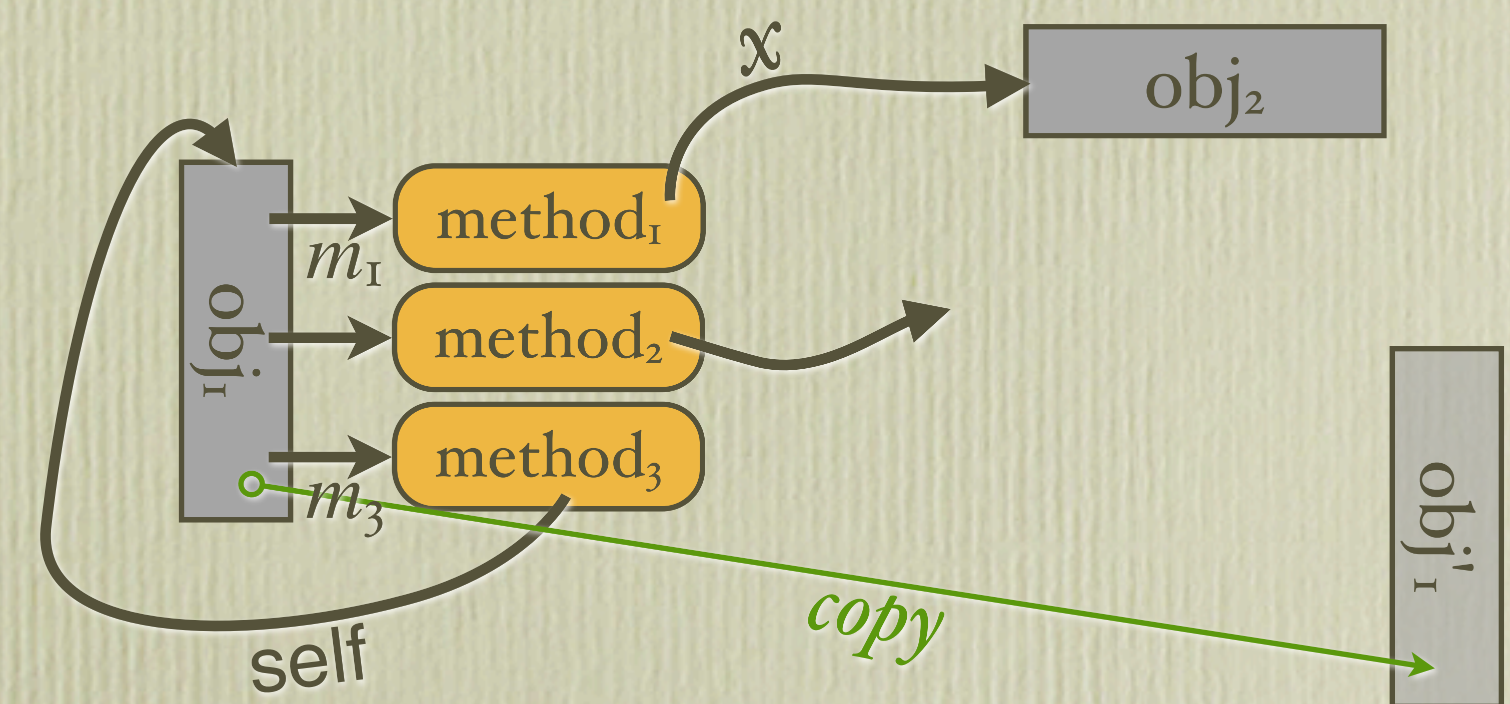
$obj_1.m_3.\mathsf{self} = obj_1$

$obj_1.m_1.x = obj_2$

- $copy_s : N \to N \cup F$ is a function with the following properties:

    1. $s(n) \equiv copy_s(n) \in F$
    2. $\neg s(n) \equiv copy_s(n) = n$
    3. $\forall n \in N, \vec{x} \in L^*, copy_s(n.\vec{x}) \equiv copy_s(n).\vec{x}$

    The last equivalence means that the path on the rhs is valid exactly when the path on the lhs is valid, and that when both are valid, the object graphs commute.



Paths from obj_1:

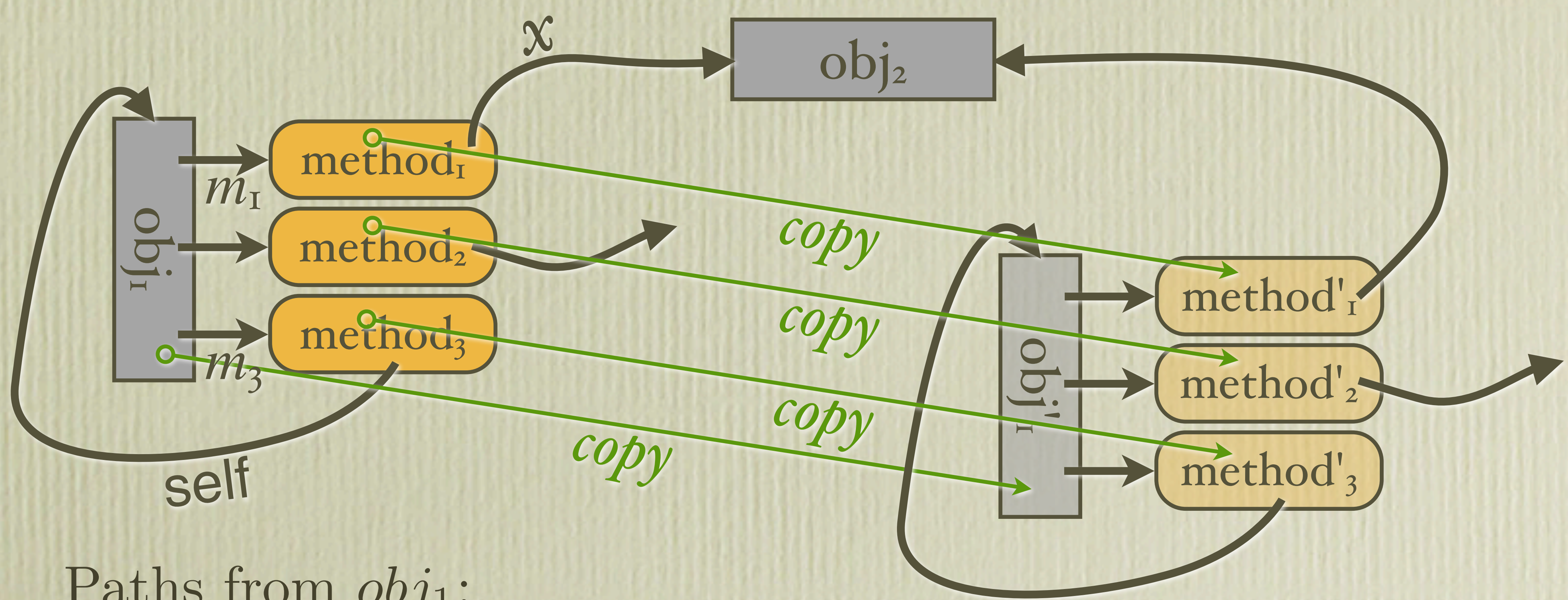
$obj_1.m_3.\mathsf{self} = obj_1$

$obj_1.m_1.x = obj_2$

- $copy_s : N \to N \cup F$ is a function with the following properties:

  1. $s(n) \equiv copy_s(n) \in F$
  2. $\neg s(n) \equiv copy_s(n) = n$
  3. $\forall n \in N, \vec{x} \in L^*, copy_s(n.\vec{x}) \equiv copy_s(n).\vec{x}$

  The last equivalence means that the path on the rhs is valid exactly when the path on the lhs is valid, and that when both are valid, the object graphs commute.



Paths from obj_1:

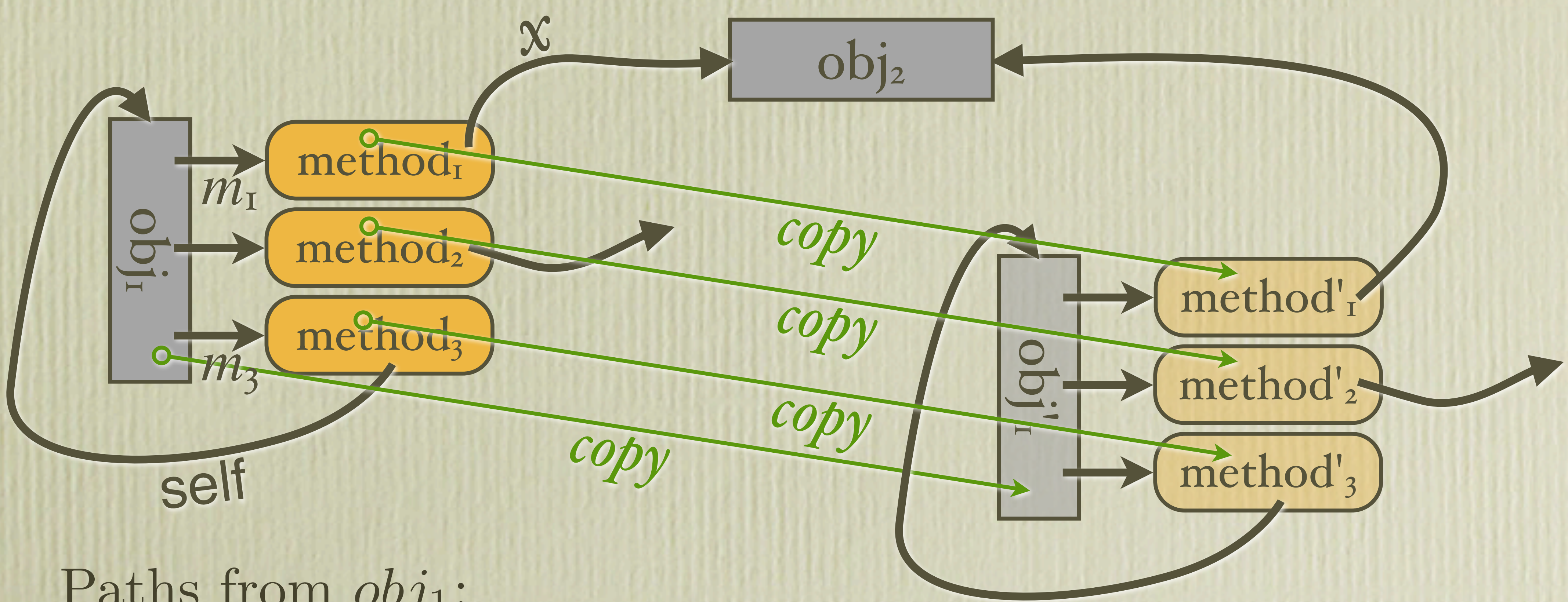
$obj_1.m_3.\mathsf{self} = obj_1$

$obj_1.m_1.x = obj_2$

- $copy_s : N \to N \cup F$ is a function with the following properties:

  1. $s(n) \equiv copy_s(n) \in F$
  2. $\neg s(n) \equiv copy_s(n) = n$
  3. $\forall n \in N, \vec{x} \in L^*, copy_s(n.\vec{x}) \equiv copy_s(n).\vec{x}$

  The last equivalence means that the path on the rhs is valid exactly when the path on the lhs is valid, and that when both are valid, the object graphs commute.



Paths from obj_1:

$obj_1.m_3.\mathsf{self} = obj_1$

$obj_1.m_1.x = obj_2$

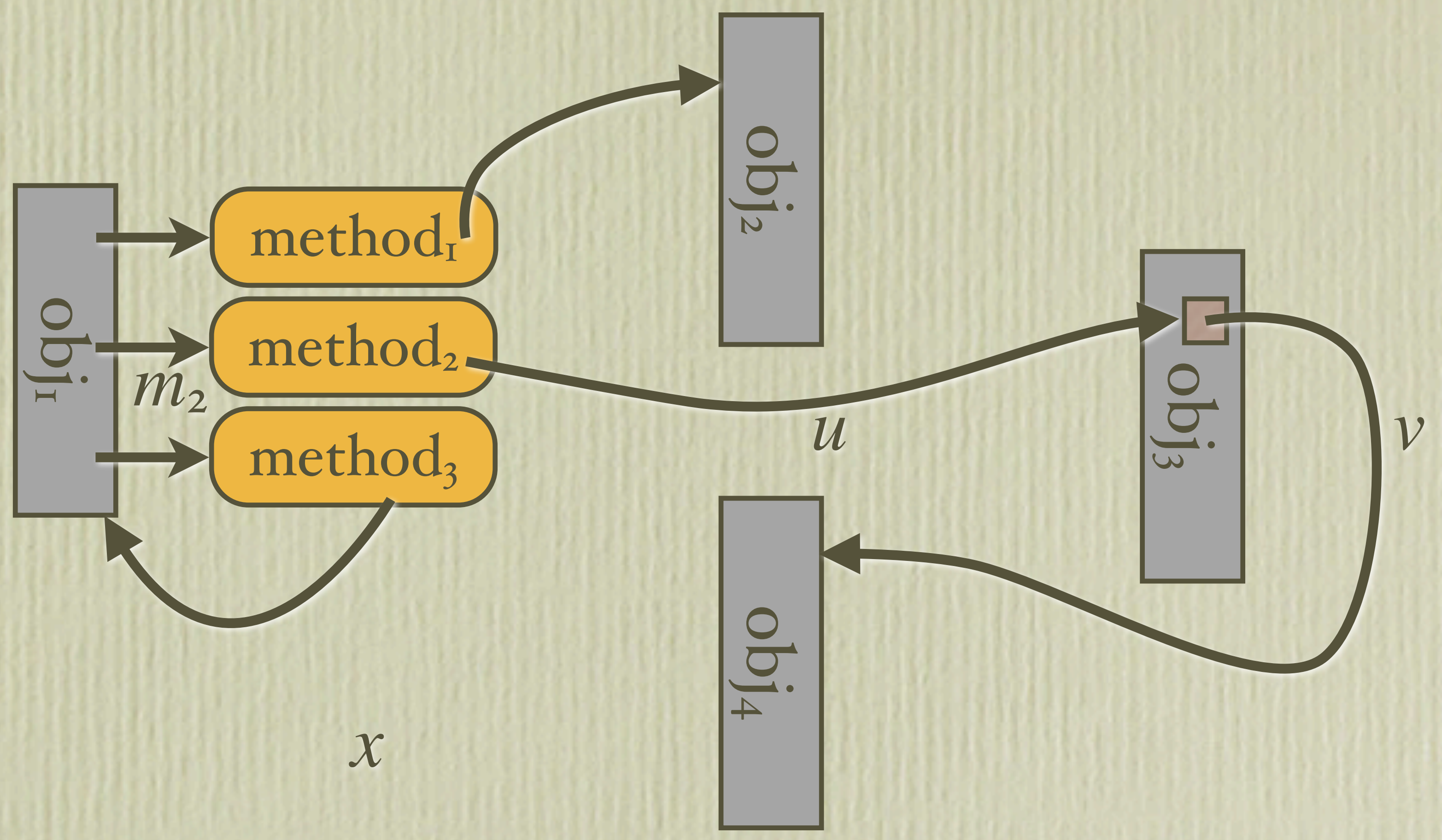
Paths from $copy(obj_1) = obj'_1$:

$obj'_1.m_3.\mathsf{self} = obj'_1$

$obj'_1.m_1.x = obj_2$

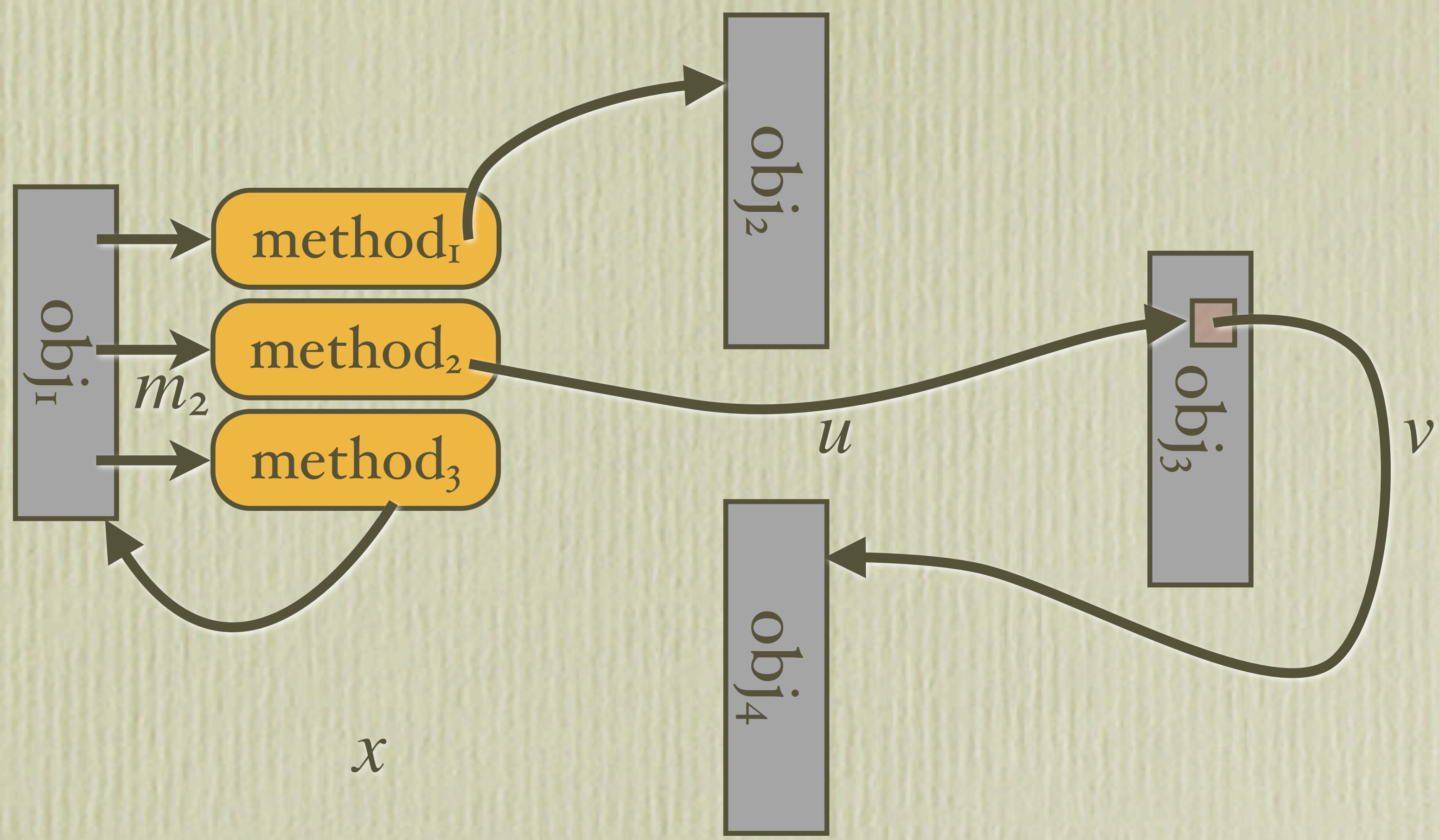
obj$_1$.copy ?

Which objects to copy and which to share?



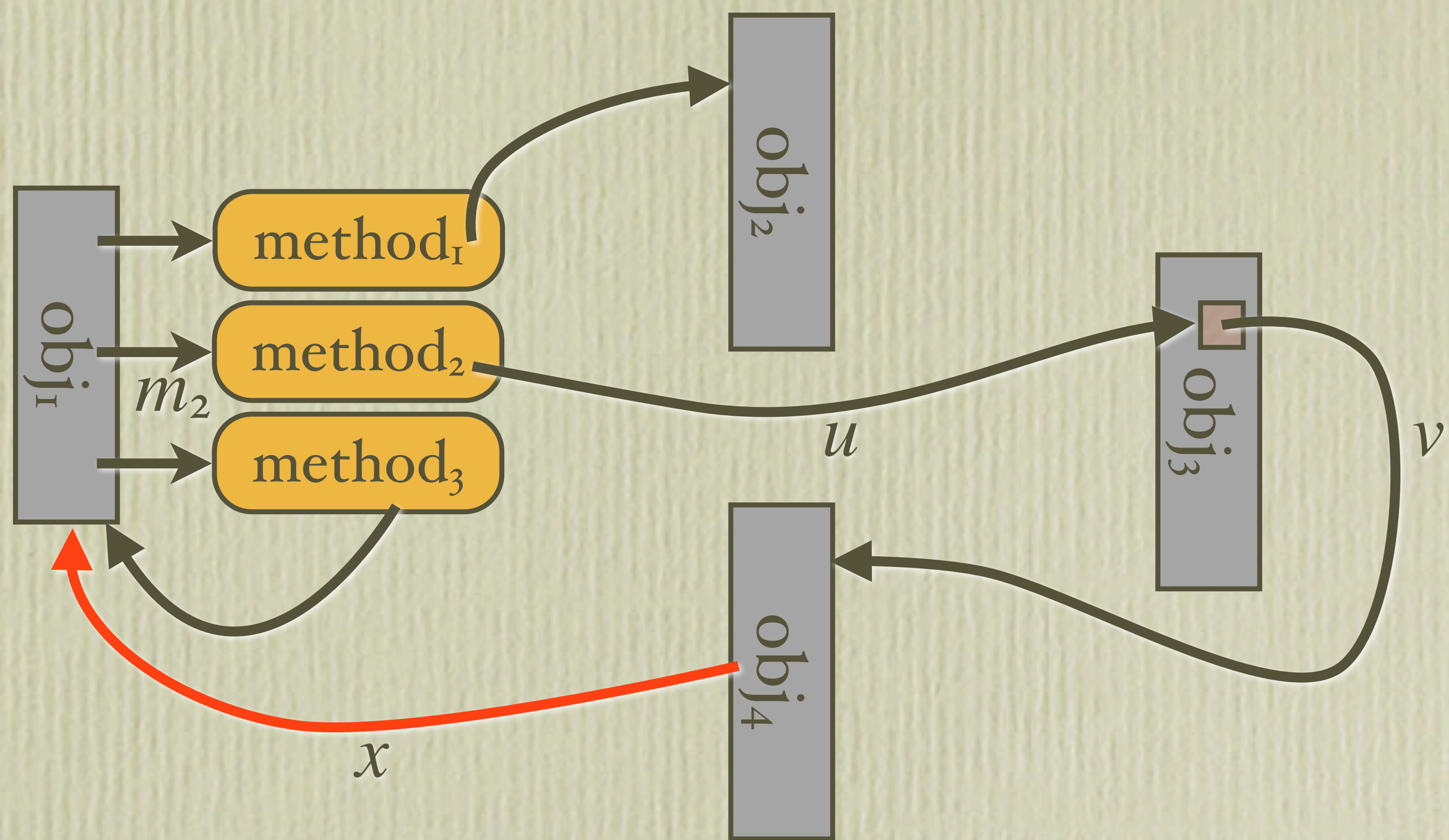
$s(\mathrm{x}) = \mathrm{x} \in \{$ obj$_1$, obj$_3$, method$_1$, method$_2$, method$_3$ $\}$

obj_1.copy ?



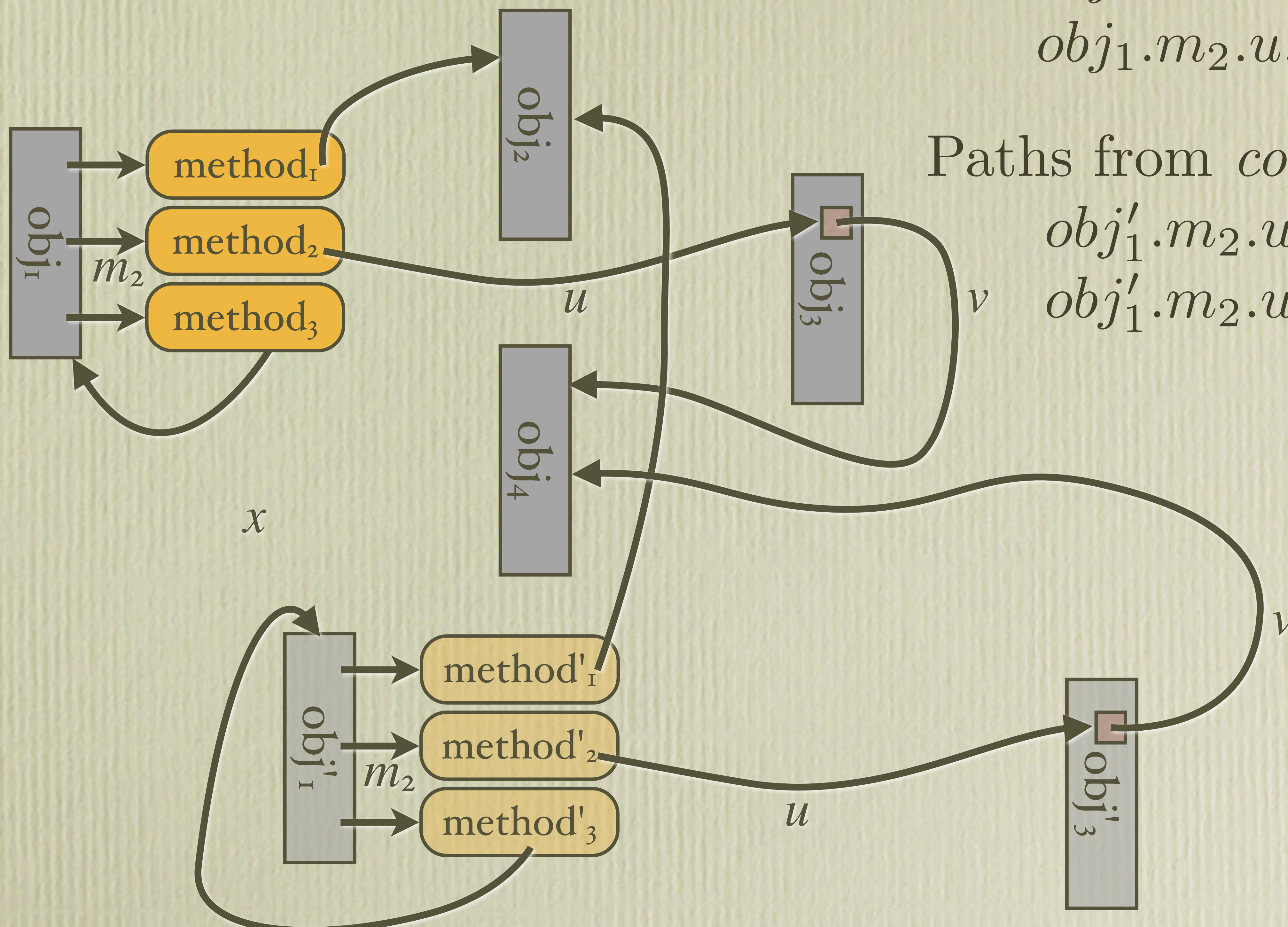
s(x) = x $\in$ { obj_1, obj_3, $method_1$, $method_2$, $method_3$ }

obj_1.copy ?



$s(x) = x \in \{\, obj_1, obj_3, method_1,$
$method_2, method_3 \,\}$

obj_1.copy ?



Paths from obj_1:

$$obj_1.m_2.u.v = obj_4$$
$$obj_1.m_2.u.v.x = obj_1$$

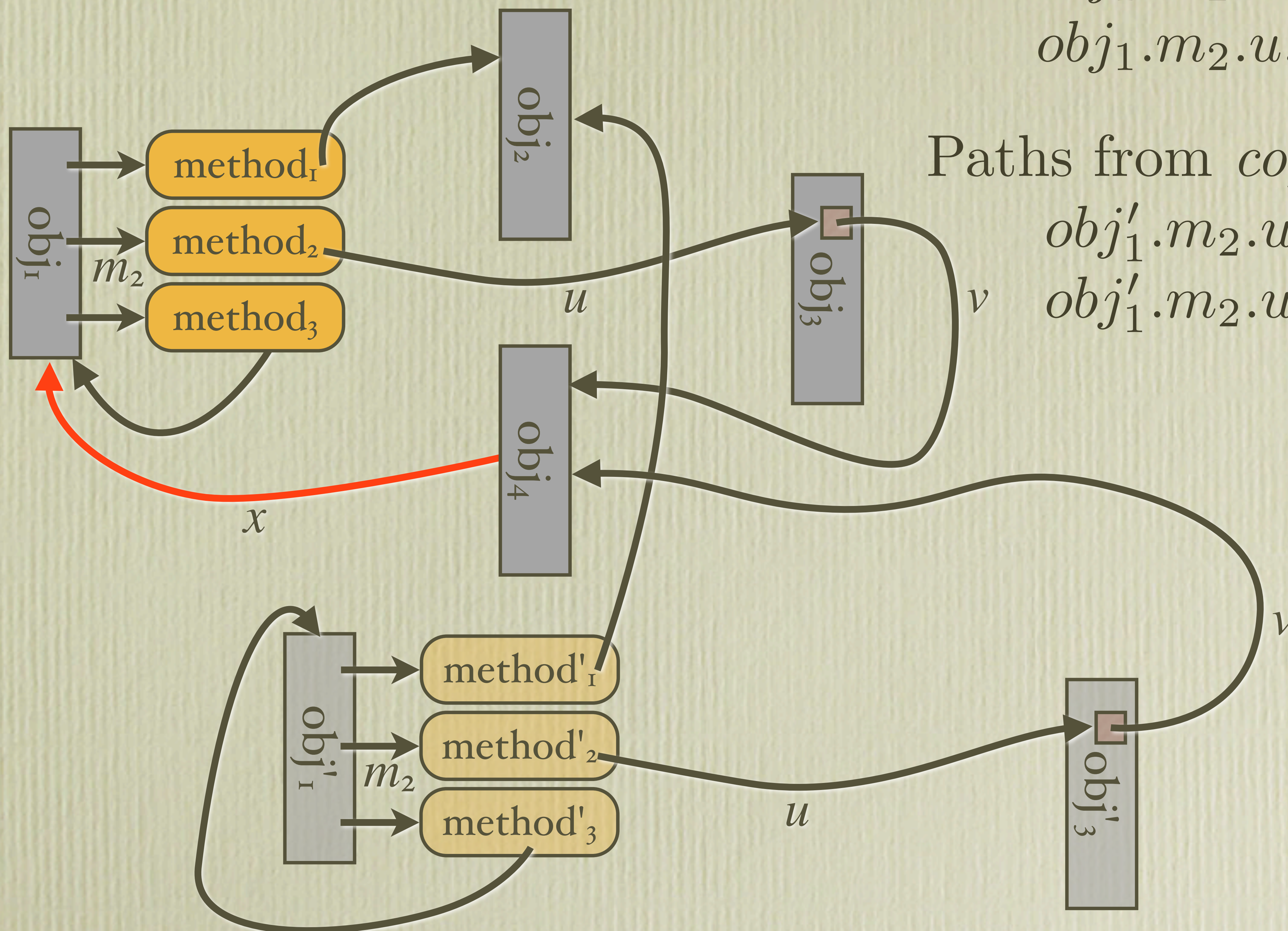
Paths from $copy_s(obj_1)$:

$$obj'_1.m_2.u.v = obj_4$$
$$obj'_1.m_2.u.v.x = obj_1$$

wrong!

s(x) = x ∈ { obj_1, obj_3, $method_1$, $method_2$, $method_3$ }

obj$_1$.copy ?



Paths from obj_1:

$$obj_1.m_2.u.v = obj_4$$

$$obj_1.m_2.u.v.x = obj_1$$

Paths from $copy_s(obj_1)$:

$$obj_1'.m_2.u.v = obj_4$$

$$obj_1'.m_2.u.v.x = obj_1$$

wrong!

$s(\mathrm{x}) = \mathrm{x} \in \{ \mathrm{obj}_1, \mathrm{obj}_3, \mathrm{method}_1, \mathrm{method}_2, \mathrm{method}_3 \}$

Definition of Copy

- Nothing you didn't know
  - but I've never seen this formalized
  - and we got it wrong in the Grace compiler
- For some *s*, there can be no correct $copy_s$
- Copy can be implemented in Grace
  - with sufficient meta-level operations
  - includes reflecting on the bound variables of methods.

Trait Proposal

- trait { ... } means the same as object { ... }
  - with the restriction that methods can't close over variables other than self
    - no useful object-local variables
- using a trait is essentially equivalent to *delegating* to the trait methods
  - self is bound dynamically to the object receiving the method request

```
class SuccessfulMatch.new(result', bindings') {
    inherits true
    def result = result'
    def bindings = bindings'
    method asString {
        "SuccessfulMatch(result = {result}, bindings = {bindings})"
    }
}
```

```
class SuccessfulMatch.new(result', bindings') {
    inherits true
    def result = result'
    def bindings = bindings'
    method asString {
        "SuccessfulMatch(result = {result}, bindings = {bindings})"
    }
}
```

- Perfectly OK — `true` has no state
  - just methods like:

```
method or(another:Block) { self }
method and(another:Block) { another.apply }
```

Objects as Traits

- What about objects with captured state?
  - all objects using them get to share the same state
- Not what you want?
  - copy the object, or generate a fresh object

```
object serialNumber {
    def rawSerial = aRandom.between(10^12)and((10^13) -1)
    def checkDigits = calculateCheckDigitsFor(rawSerial)
    def serial is public, readable
       = rawSerial.asString ++ checkDigits.asString
}
```

```
object serialNumber {
    def rawSerial = aRandom.between(10^12)and((10^13) -1)
    def checkDigits = calculateCheckDigitsFor(rawSerial)
    def serial is public, readable
       = rawSerial.asString ++ checkDigits.asString
}
```

```
class engine.ofSize(volume) {
    uses serialNumber
    def displacement is public, readable = volume
    def cylinders is public, readable = 6
}
```

```
object serialNumber {
    def rawSerial = aRandom.between(10^12)and((10^13) -1)
    def checkDigits = calculateCheckDigitsFor(rawSerial)
    def serial is public, readable
       = rawSerial.asString ++ checkDigits.asString
}
```

```
class engine.ofSize(volume) {
    uses serialNumber
    def displacement is public, readable = volume
    def cylinders is public, readable = 6
}
```

- Every engine has the same serial

```
object serialNumber {
    def rawSerial = aRandom.between(10^12)and((10^13) -1)
    def checkDigits = calculateCheckDigitsFor(rawSerial)
    def serial is public, readable
       = rawSerial.asString ++ checkDigits.asString
}
```

```
class engine.ofSize(volume) {
    uses serialNumber
    def displacement is public, readable = volume
    def cylinders is public, readable = 6
}
```

- Every engine has the same serial

```
class serialNumber.new {
    def rawSerial = aRandom.between(10^12)and((10^13) -1)
    def checkDigits = calculateCheckDigitsFor(rawSerial)
    def serial is public, readable
       = rawSerial.asString ++ checkDigits.asString
}
```

```
class serialNumber.new {
    def rawSerial = aRandom.between(10^12)and((10^13) -1)
    def checkDigits = calculateCheckDigitsFor(rawSerial)
    def serial is public, readable
       = rawSerial.asString ++ checkDigits.asString
}
```

```
class engine.ofSize(volume) {
    uses serialNumber.new
    def displacement is public, readable = volume
    def cylinders is public, readable = 6
}
```

```
class serialNumber.new {
    def rawSerial = aRandom.between(10^12)and((10^13) -1)
    def checkDigits = calculateCheckDigitsFor(rawSerial)
    def serial is public, readable
       = rawSerial.asString ++ checkDigits.asString
}
```

```
class engine.ofSize(volume) {
    uses serialNumber.new
    def displacement is public, readable = volume
    def cylinders is public, readable = 6
}
```

```
class serialNumber.new {
    def rawSerial = aRandom.between(10^12)and((10^13) -1)
    def checkDigits = calculateCheckDigitsFor(rawSerial)
    def serial is public, readable
       = rawSerial.asString ++ checkDigits.asString
}
```

```
class engine.ofSize(volume) {
    uses serialNumber.new
    def displacement is public, readable = volume
    def cylinders is public, readable = 6
}
```

```
class serialNumber.new {
    def rawSerial = aRandom.between(10^12)and((10^13) -1)
    def checkDigits = calculateCheckDigitsFor(rawSerial)
    def serial is public, readable
       = rawSerial.asString ++ checkDigits.asString
}
```

```
class engine.ofSize(volume) {
    uses serialNumber.new
    def displacement is public, readable = volume
    def cylinders is public, readable = 6
}
```

- Every engine has a new serial

```
class serialNumber.new {
    def rawSerial = aRandom.between(10^12)and((10^13) -1)
    def checkDigits = calculateCheckDigitsFor(rawSerial)
    def serial is public, readable
       = rawSerial.asString ++ checkDigits.asString
}
```

```
class engine.ofSize(volume) {
    uses serialNumber.new
    def displacement is public, readable = volume
    def cylinders is public, readable = 6
}
```

- Every engine has a new serial

Inheriting Initialization

- Object *initialization* is not the same as object *creation*
- Smalltalk makes this clear:

```
Behavior ›› new
    "Answer a new initialized instance of the
    receiver (which is a class) ..."
    ↑ self basicNew initialize
```

  - Behavior ›› basicNew creates the object
  - Instance ›› initialize assigns to fields, registers it, *etc.*

Inheriting Initialization

- Easy: inherit the `initialize` method
  - in Smalltalk, this is a real method
  - in Java, it's a "special" method called `"<init>"`
- *Pharo* Smalltalk and Java classes both invoke initialization automatically
  - *after* the object has been created
- If we want to inherit initialization in Grace, we can do the same thing

Inheriting Initialization

```
def initializable is public, readable = trait {
  method create { done }
  method new is public {
    def instance = self.create
    instance.initialize
    instance
  }
}
```

- Captures the separation of creation and initialization as a trait

Inheriting Initialization

```
def initializable is public, reada
  method create { done }
  method new is public {
    def instance = self.create
    instance.initialize
    instance
  }
}
```

new method creates
and initializes

- Captures the separation of creation and initialization as a trait

Using the Initialization Trait

```
 def aWindow = object {
    uses initializable
    method create  is override {
        object {
            var bounds is public, readable, writable
            method paint(c) is public { … }
            method initialize is public {
                world.register(self) }
            method minimize is public { … }        }
    }
    method withBounds(b) is public {
        def instance = self.create
        instance.bounds := b
        instance.initialize
    }
}
```

Using the Initialization Trait

```
def aWindow = object {
  uses initializable
  method create  is override {
     object {
        var bounds is public, readable, writable
        method paint(c) is public { … }
        method initialize is public {
           world.register(self) }
        method minimize is public { … }       }
  }
  method withBounds(b) is public {
     def instance = self.create
     instance.bounds := b
     instance.initialize
  }
}
```

gets us the create and new methods

Using the Initialization Trait

```
 def aWindow = object {
    uses initializable
    method create  is override {
        object {
            var bounds is public, readable, writable
            method paint(c) is public { … }
            method initialize is public {
                world.register(self) }
            method minimize is public { … }        }
    }
    method withBounds(b) is public {
        def instance = self.create
        instance.bounds := b
        instance.initialize
    }
}
```

Using the Initialization Trait

```
 def aWindow = object {
uses initializable
method create  is override {
    object {
        var bounds is public, readable, writable
        method paint(c) is public { … }
        method initialize is public {
            world.register(self) }
        method minimize is public { … }        }
}
method withBounds(b) is public {
    def instance = self.create
    instance.bounds := b
    instance.initialize
}
}
```

```
aWindow.withBounds(aRectangle.
topLeft(100@100)diagonal(50@50))
```

Portland State UNIVERSITY

Using the Initialization Trait

```
 def aWindow = object {
    uses initializable
    method create  is override {
        object {
            var bounds is public, readable, writable
            method paint(c) is public { … }
            method initialize is public {
                world.register(self) }
            method minimize is public { … }
    }
    method withBounds(b) is public {
        def instance = self.create
        instance.bounds := b
        instance.initialize
    }
}
```

creates and
sets bounds of a new
window

aWindow.withBounds(aRectangle.
topLeft(100@100)diagonal(50@50))

Portland State
UNIVERSITY

Using the Initialization Trait

```
 def aWindow = object {
    uses initializable
    method create  is override {
        object {
            var bounds is public, readable, writable
            method paint(c) is public { … }
            method initialize is public {
                world.register(self) }
            method minimize is public { … }        }
    }
    method withBounds(b) is public {
        def instance = self.create
        instance.bounds := b
        instance.initialize
    }
}
```

```
aWindow.withBounds(aRectangle.
topLeft(100@100)diagonal(50@50))
```

Using the Initialization Trait

```
 def aWindow = object {
  uses initializable
  method create  is override {
     object {
        var bounds is public, readable, writable
        method paint(c) is public { … }
        method initialize is public {
           world.register(self) }
        method minimize is public { … }       }
  }
  method withBounds(b) is public {
     def instance = self.create
     instance.bounds := b
     instance.initialize
  }
}
```

aWindow.new

aWindow.withBounds(aRectangle.
topLeft(100@100)diagonal(50@50))

Using the Initialization Trait

```
def aWindow = object {
  uses initializable
  method create  is override {
    object {
      var bounds is public, readable, writable
      method paint(c) is public { … }
      method initialize is public {
        world.register(self) }
      method minimize is public { … }
  }
  method withBounds(b) is public {
    def instance = self.create
    instance.bounds := b
    instance.initialize
  }
}
```

creates a new, initialized window with undefined bounds

```
aWindow.new
```

```
aWindow.withBounds(aRectangle.
topLeft(100@100)diagonal(50@50))
```

Using the Initialization Trait

```
 def aWindow = object {
 uses initializable
 method create  is override {
     object {
         var bounds is public, readable, writable
         method paint(c) is public { … }
         method initialize is public {
             world.register(self) }
         method minimize is public { … }        }
 }
 method withBounds(b) is public {
     def instance = self.create
     instance.bounds := b
     instance.initialize
 }
}
```

aWindow.new

aWindow.withBounds(aRectangle.
topLeft(100@100)diagonal(50@50))

What About Classes?

Currently

```
class A.name {
        inherits S
        defs, vars and methods }
```

means

```
def A = object {
          method name {  object {
            inherits S
            defs, vars and methods }
        }}
```

We can change this!

Summary

- Don't "build in" complex features
- Start with general-purpose building blocks
  - Complex features can be fabricated from the building blocks
  - They will inevitably be consistent