



Inheritance Semantics and Method Lookup

Stéphane Ducasse

Stephane.Ducasse@univ-savoie.fr

<http://www.iam.unibe.ch/~ducasse/>

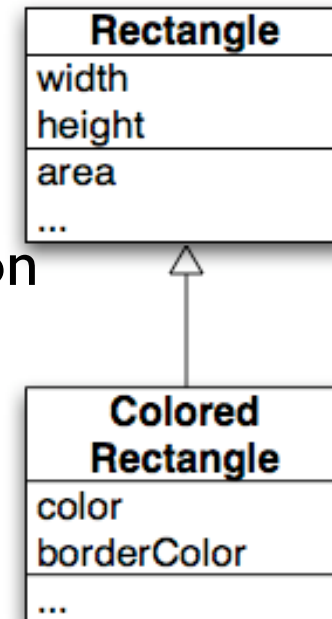
Goal

- Inheritance
- Method lookup
- Self/super difference



Inheritance

- Do not want to rewrite everything!
- Often we want small changes
- We would like to reuse and extend existing behavior
- Solution: class inheritance
- Each class defines or refines the definition of its ancestors



Inheritance

- New classes
 - Can add state and behavior:
 - color, borderColor, borderWidth,
 - totalArea
 - Can specialize ancestor behavior
 - intersect:
 - Can use ancestor's behavior and state
 - Can redefine ancestor's behavior
 - area to return totalArea



Inheritance in Smalltalk

- ***Single inheritance***
- ***Static for the instance variables***
- At class creation time the instance variables are collected from the superclasses and the class. No repetition of instance variables.
- ***Dynamic for the methods***
- Late binding (all virtual) methods are looked up at run-time depending on the dynamic type of the receiver.



Message Sending

- ***receiver selector args***
- Sending a message = looking up the method that should be executed and executing it
- ***Looking up*** a method: When a message (receiver selector args) is sent, the method corresponding to the message selector is looked up through the inheritance chain.



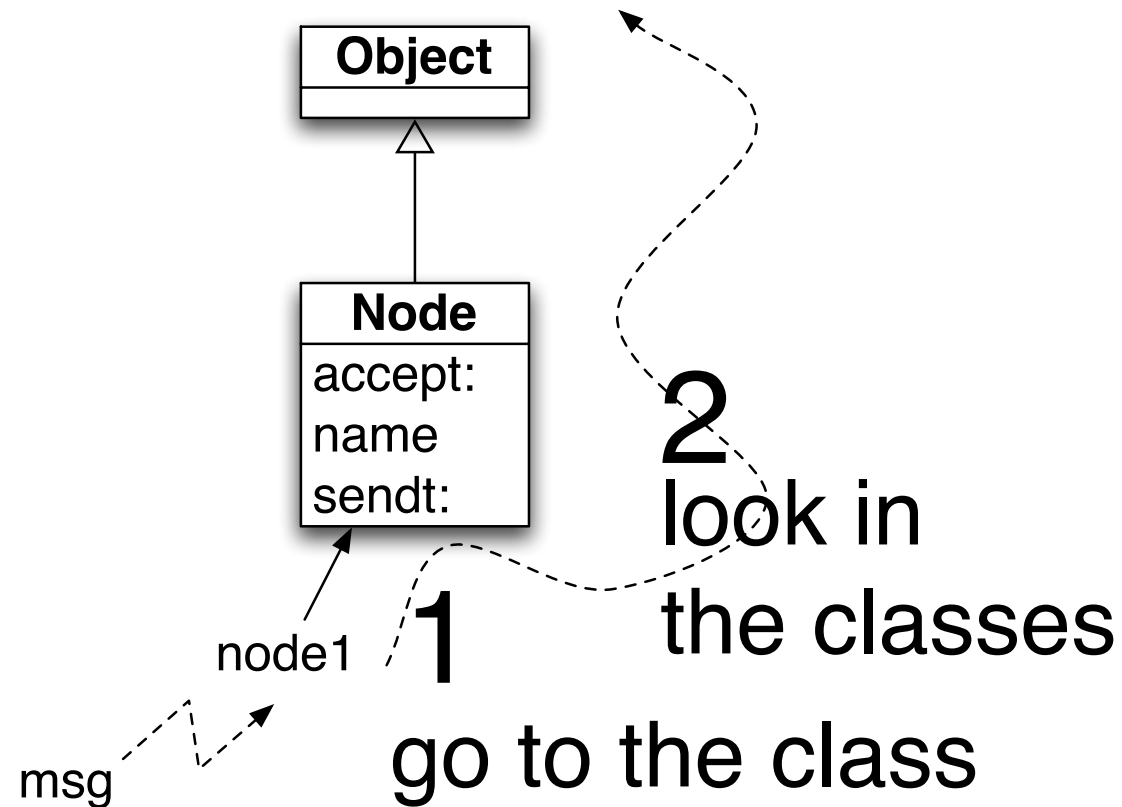
Method Lookup



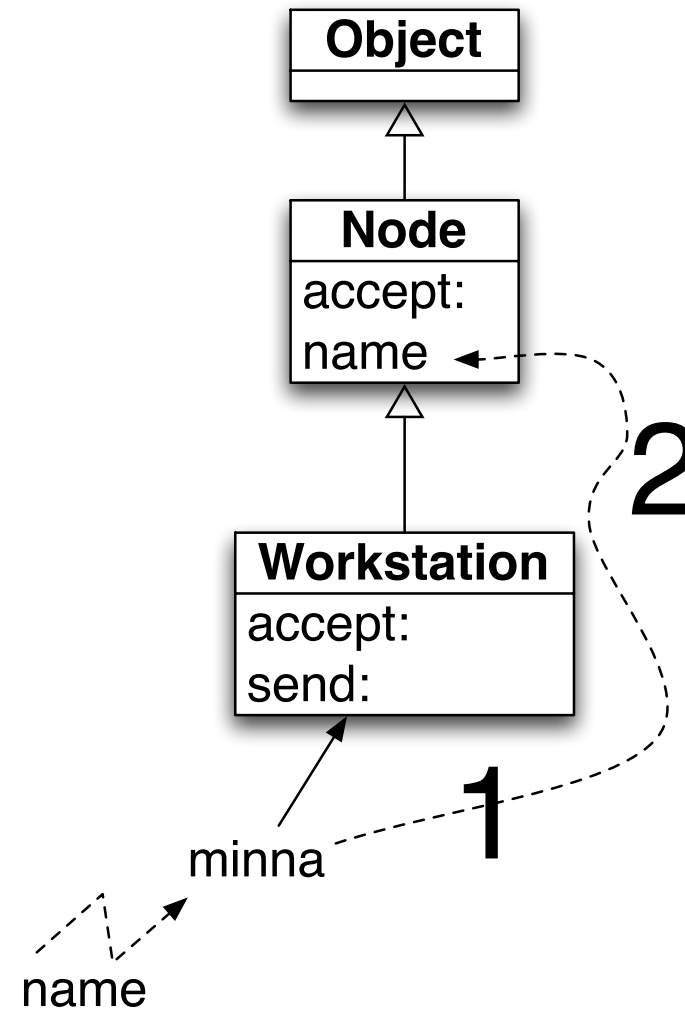
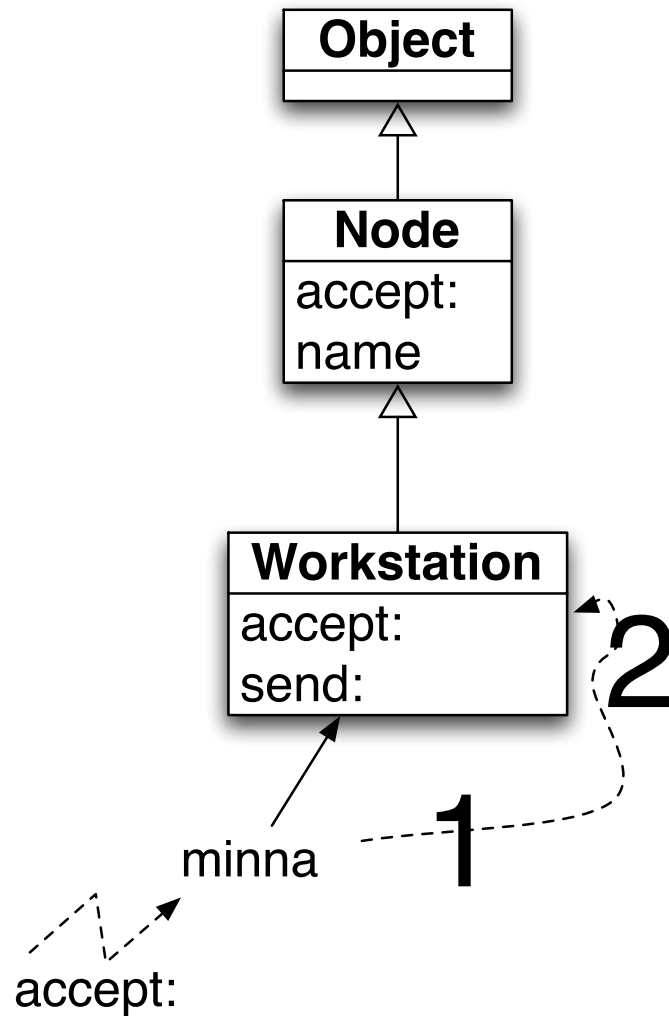
- Two steps process
- The lookup starts in the **CLASS** of the **RECEIVER**.
- If the method is defined in the method dictionary, it is returned.
- Otherwise the search continues in the superclasses of the receiver's class. If no method is found and there is no superclass to explore (class Object), this is an **ERROR**



Lookup: class and inheritance



Some Cases



Method Lookup starts in Receiver Class

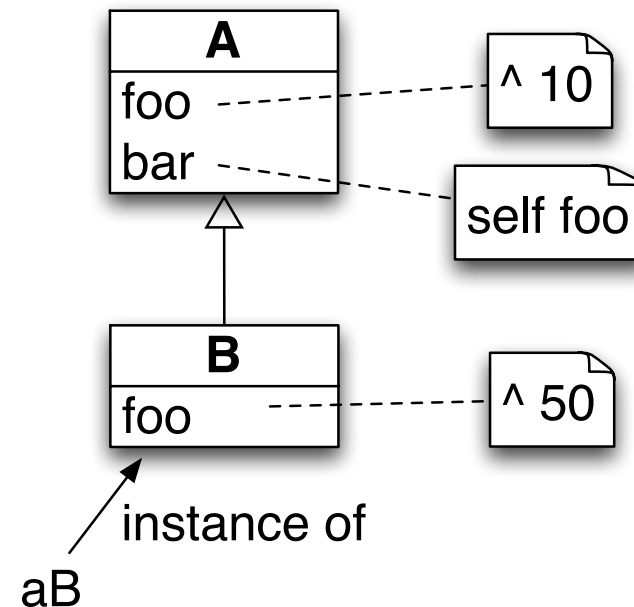


aB foo

- (1) aB class => B
- (2) Is foo defined in B?
- (3) Foo is executed -> 50

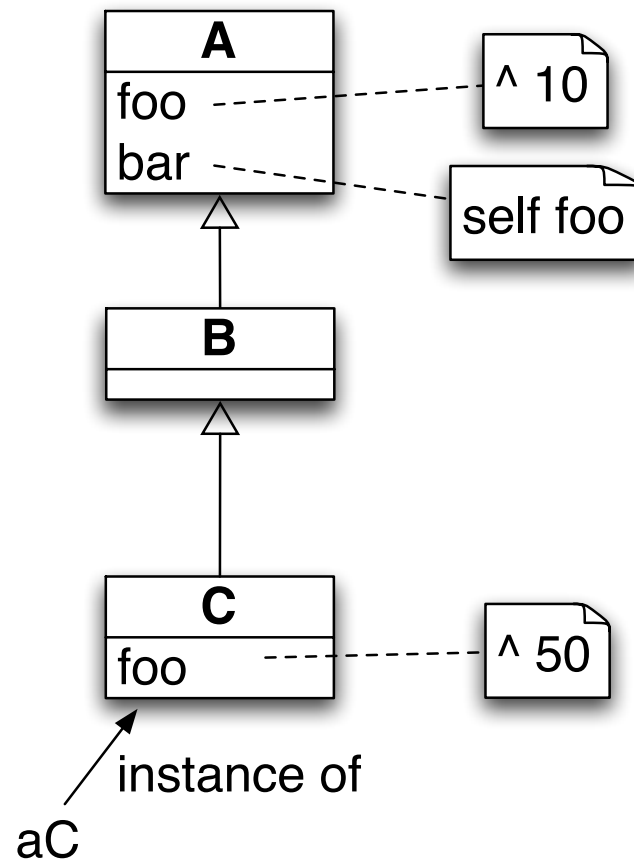
aB bar

- (1) aB class => B
- (2) Is bar defined in B?
- (3) Is bar defined in A?
- (4) bar executed
- (5) Self class => B
- (6) Is foo defined in B
- (7) Foo is executed -> 50



self **always** represents the receiver

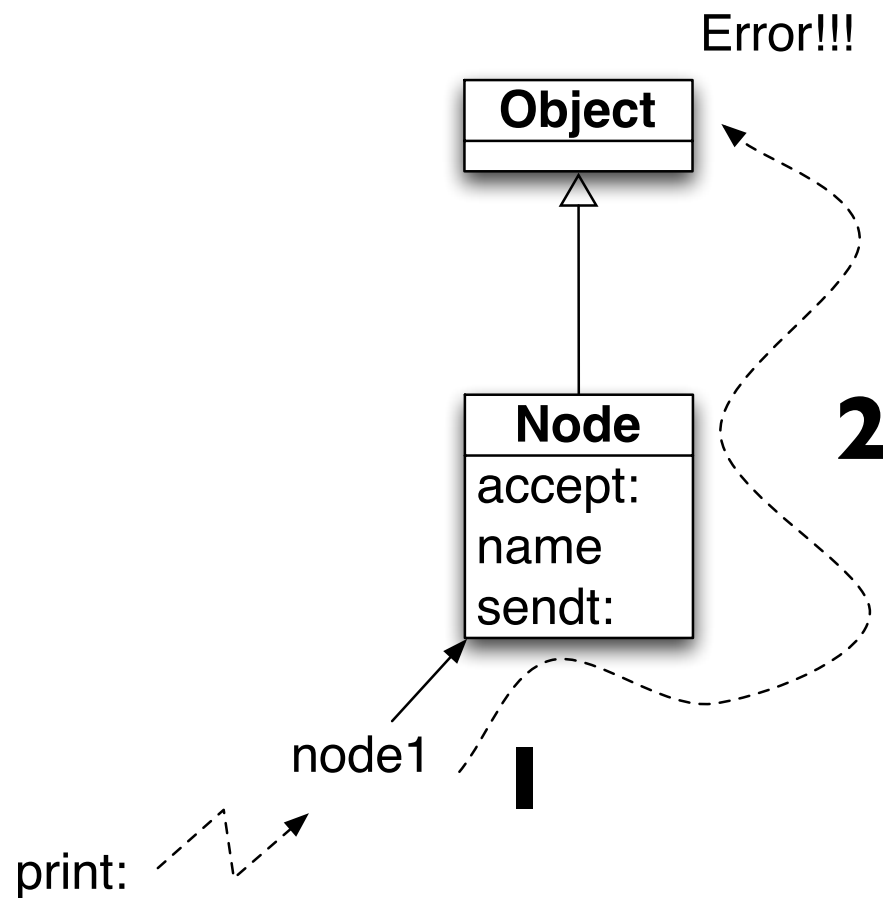
- A new foo
- -> 10
- B new foo
- -> 10
- C new foo
- -> 50
- A new bar
- -> 10
- B new bar
- -> 10
- C new bar
- -> 50



When message is not found

- If no method is found and there is no superclass to explore (class Object), a new method called `#doesNotUnderstand:` is sent to the receiver, with a representation of the initial message.

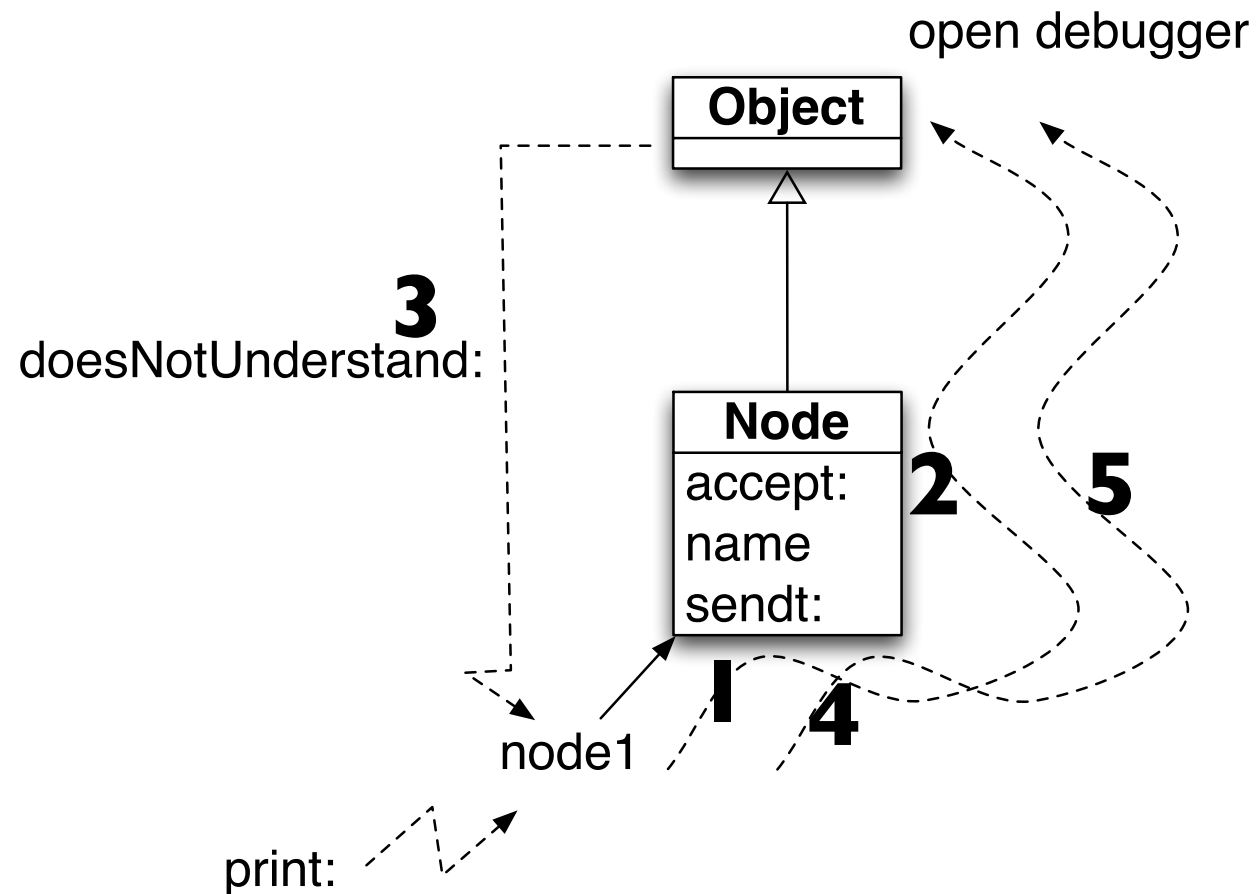
Graphically...



...in Smalltalk

- nodeI print: aPacket
 - node is an instance of Node
 - print: is looked up in the class Node
 - print: is not defined in Node > lookup continues in Object
 - print: is not defined in Object => lookup stops + exception
 - message: nodeI doesNotUnderstand: #(#print aPacket) is executed
 - nodeI is an instance of Node so doesNotUnderstand: is looked up in the class Node
 - doesNotUnderstand: is not defined in Node => lookup continues in Object
 - doesNotUnderstand: is defined in Object => lookup stops + method executed (open a dialog box)

Graphically...



Roadmap

- Inheritance
- Method lookup
- ***Self/super difference***



How to Invoke Overridden Methods?

- Solution: Send messages to super
- When a packet is not addressed to a workstation, we just want to pass the packet to the next node, i.e., we want to perform the default behavior defined by Node.

```
Workstation>>accept: aPacket  
    (aPacket isAddressedTo: self)  
    ifTrue:[Transcript show: 'Packet accepted by the Workstation ',  
self name asString]  
    ifFalse: [super accept: aPacket]
```

- Design Hint: Do not send messages to super with different selectors than the original one. It introduces implicit dependency between methods with different names.

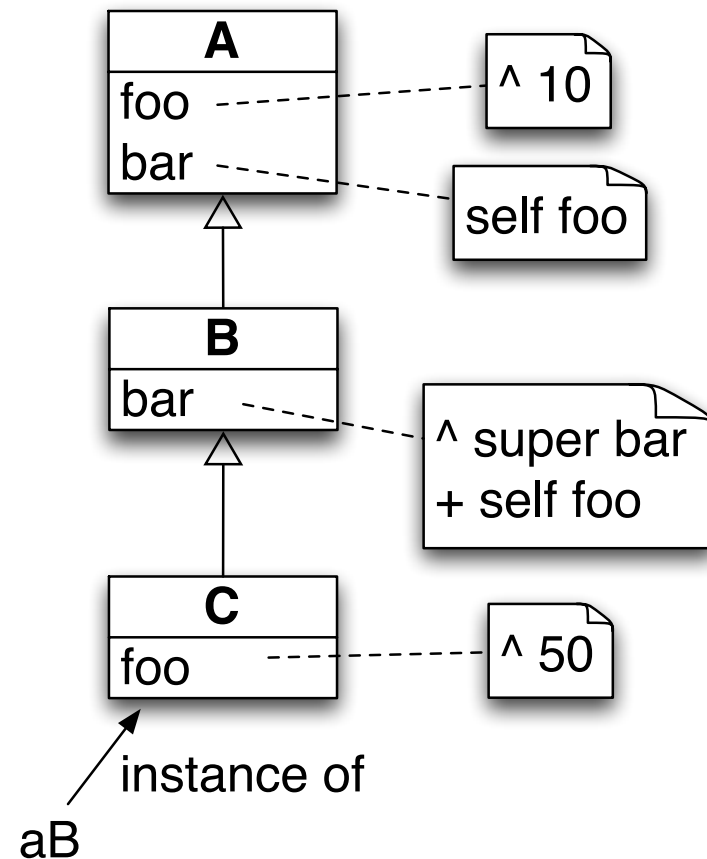
The semantics of super

- Like self, **super** is a pseudo-variable that refers to the **receiver** of the message.
- It is used to invoke overridden methods.
- When using self, the lookup of the method begins in the class of the receiver.
- When using super, the lookup of the method begins in the **superclass of the class of the method containing** the super expression

super changes lookup starting class

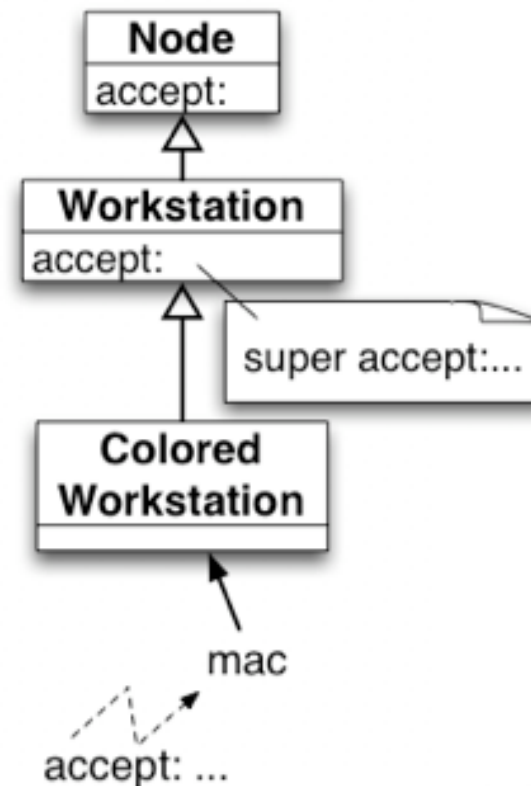


- A new bar
- -> 10
- B new bar
- -> 10 + 10
- C new bar
- -> 50 + 50



super is NOT the superclass of the receiver class

Suppose the WRONG hypothesis: “*The semantics of super is to start the lookup of a method in the superclass of the receiver class*”



super is NOT the superclass of the receiver class

mac is instance of ColoredWorkStation
Lookup starts in ColoredWorkStation
Not found so goes up...

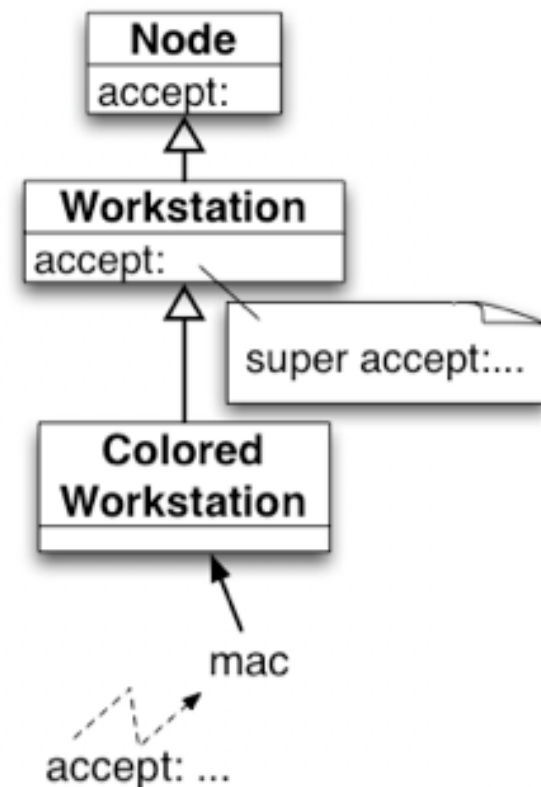
accept: is defined in Workstation
lookup stops
method accept: is executed

Workstation>>accept: does a super
send

Our hypothesis: start in the super of the
class of the receiver

=> superclass of class of a ColoredWorkstation
is ... **Workstation !**

Therefore we look in workstation **again!!!**



What you should know

- Inheritance of instance variables is made at class definition time.
- Inheritance of behavior is dynamic.
- `self` ****always**** represents the receiver.
- Method lookup starts in the class of the receiver.
- **`super`** represents the **receiver** but method lookup starts in the superclass of the class **using** it.
- ***Self is dynamic vs. super is static.***