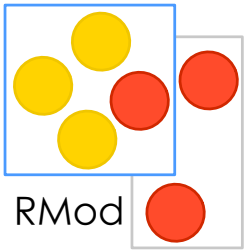


Studying a Minimal Object-Oriented Kernel

Stéphane Ducasse
Stephane.Ducasse@inria.fr
http://stephane.ducasse.free.fr

Food for thoughts

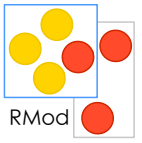
"L'idée de l'expérience ne remplace pas l'expérience"
Alain

"Give a man a fish; you have fed him for today. Teach a man to fish; and you have fed him for a lifetime"

Resources

http://books.pharo.org/booklet-ReflectiveCore/

https://github.com/SquareBracketAssociates/Booklet-AReflectiveKernel

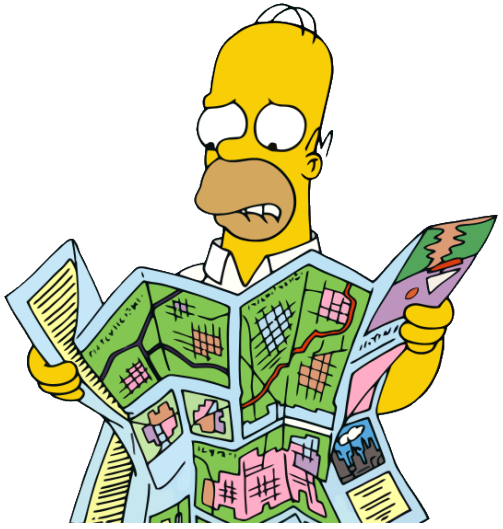


Goals

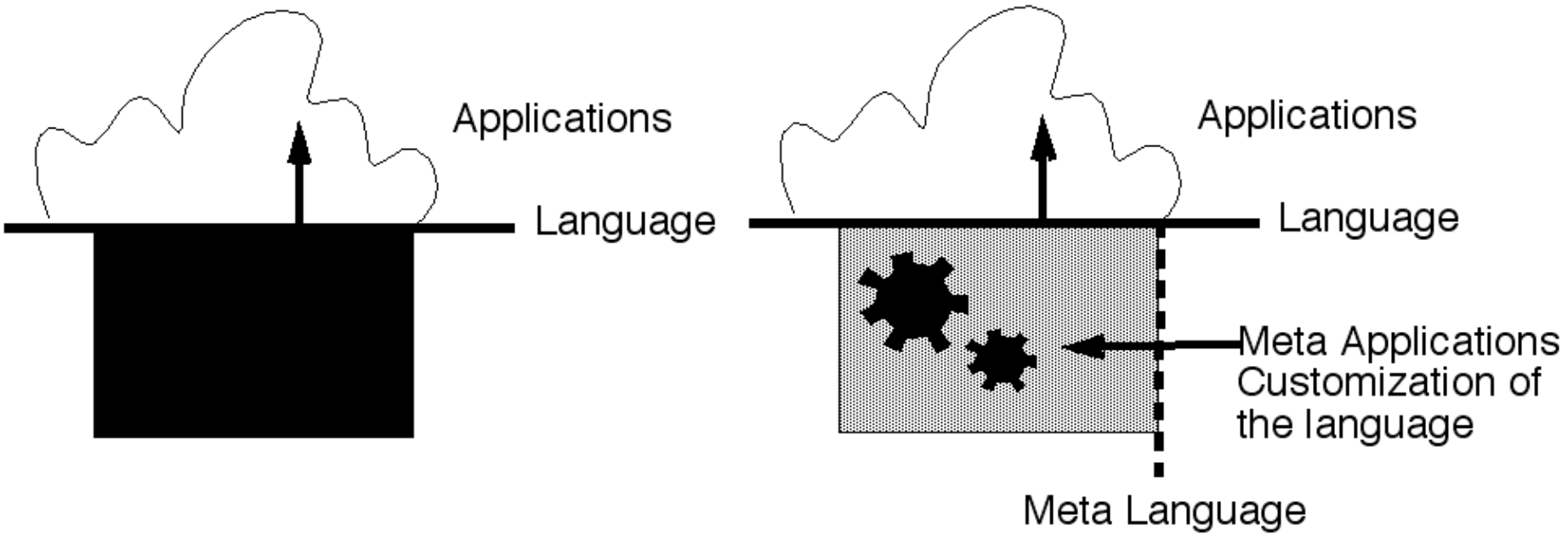
- Classes as objects
- Object and Class classes
- Semantics of inheritance
- Semantics of super and self
- Instantiation vs. Inheritance
- Allocation and Initialization
- Build your own language

Outline of the lectures

- Context
- Classes as objects
- ObjVlisp in 5 postulates
- Instances/Classes/Metaclasses
- Instance Structure and Behavior
- Class Structure
- Message Passing
- Object allocation & Initialization
- Class creation
- Inheritance Semantics
- Bootstrapping

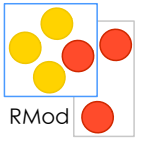


Context: Can we customise languages?



Real Cases

Nichimem (3D) corp saved 15 years of development by changing the semantics of CLOS to be close to the one Flavor.



Classes as Objects?

"The difference between classes and objects has been repeatedly emphasized. …, these concepts belong to different worlds: the program text only contains classes; at run-time, only objects exist.

…

This is not the only approach. One of the subcultures of object-oriented programming, influenced by Lisp and exemplified by Smalltalk, views ***classes as object themselves, which still have an existence at run-time***."

B. Meyer Object-Oriented Software Construction

(Classes and) Metaclasses

One metaobject

Customise point

Support language extension

They may control

- Inheritance
- Internal representation of the objects (listes, vecteurs, hash-table, ...)
- Instance variable access

Understanding instantiation

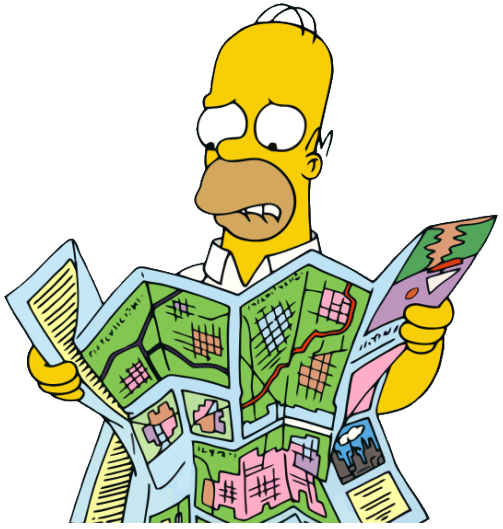
What is the relationship between

- an instance and its class?
- a class and its metaclass?
- a metaclass and its metametaclass?

What is the cost of classes as objects?

Roadmap

- Classes as objects
- ***ObjVlisp in 5 postulates***
- Instances/Classes/Metaclasses
- Instance Structure and Behavior
- Class Structure
- Message Passing
- Object allocation & Initialization
- Class creation
- Inheritance Semantics
- Bootstrapping



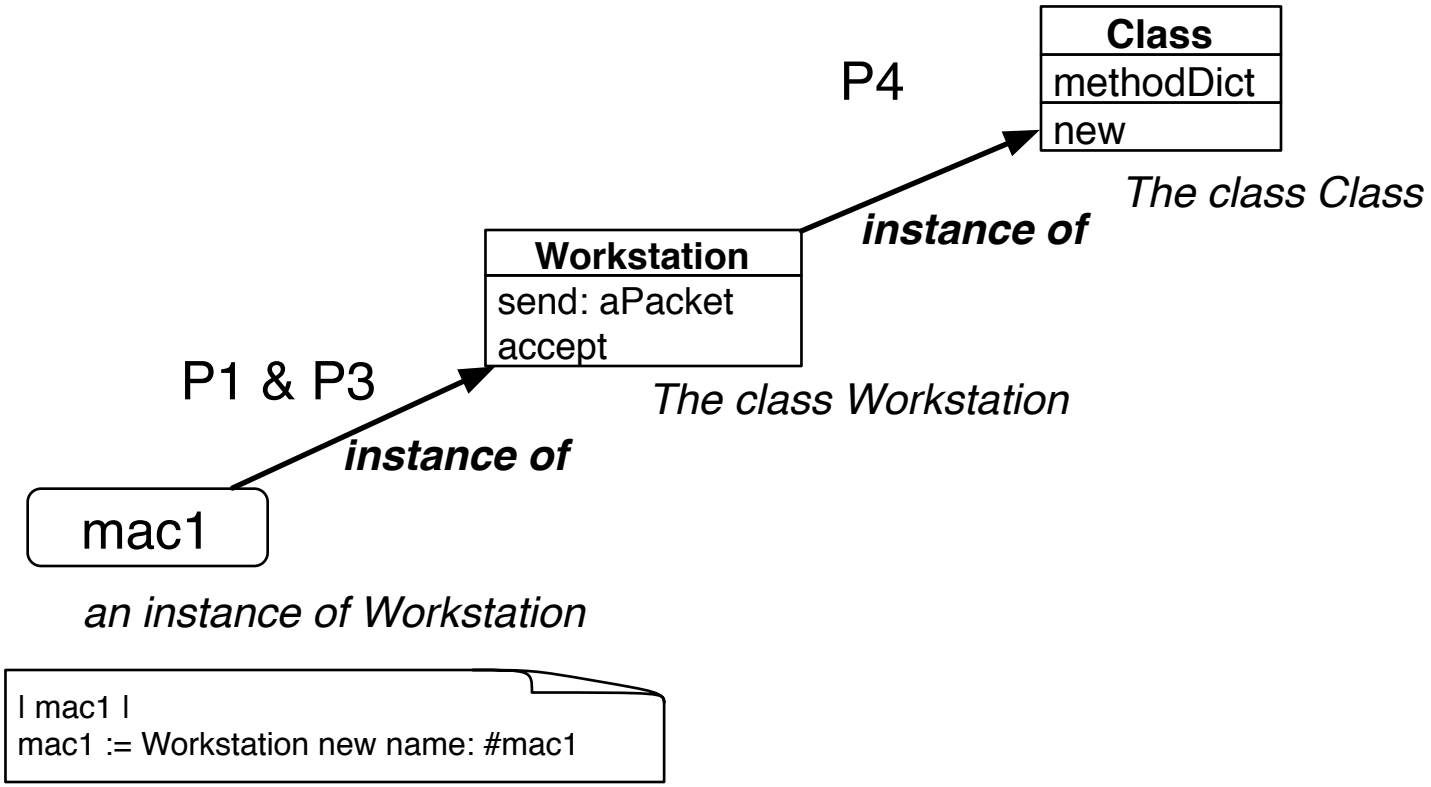
Why ObjVlisp?

- Minimal (only two classes)
- ObjVlisp self-described:
  - Object and Class
- Unified: Only one kind of object: a class is an object and a metaclass is a class that creates classes
- Simple: can be implemented with less 30 methods
- Definition a bit dated but conceptually relevant
- Equivalent of Closette (Art of MetaObject Protocol, G. Kiczales)

ObjVlisp Postulates (I)

- P1: An object represents a piece of knowledge and a set of capabilities.
- P3: Every object belongs to a class that specifies its data (instance variables) and its behavior. Objects are created dynamically from their class.
- P4: Following P3, a class is also an object therefore instance of another class *its metaclass* (that describes the behavior of a class).

ObjVlisp Postulates (II)



RMod

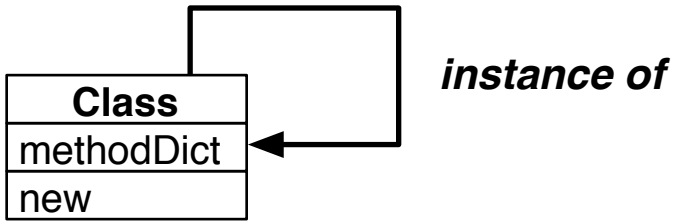
Infinite Recursion

A class is an object therefore instance of another class its metaclass that is an object too instance of a metametaclass that is an object too instance of another a metametametaclass.....

Stopping the Infinite Recursion

To stop this potential infinite recursion

- Class is the initial class and metaclass
- Class is instance of itself
- All other metaclasses are instances of Class



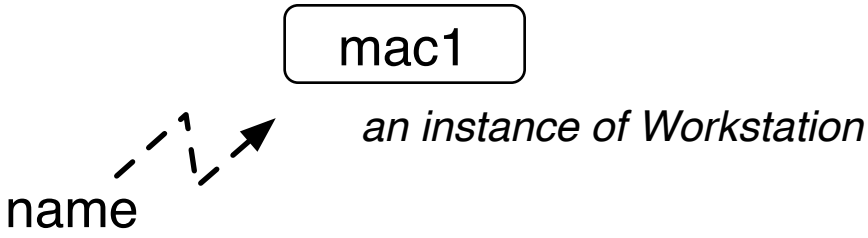
The class Class

RMod

ObjVlisp 2nd Postulate

- P2: Message passing is the only means to activate an object

[object selector args]



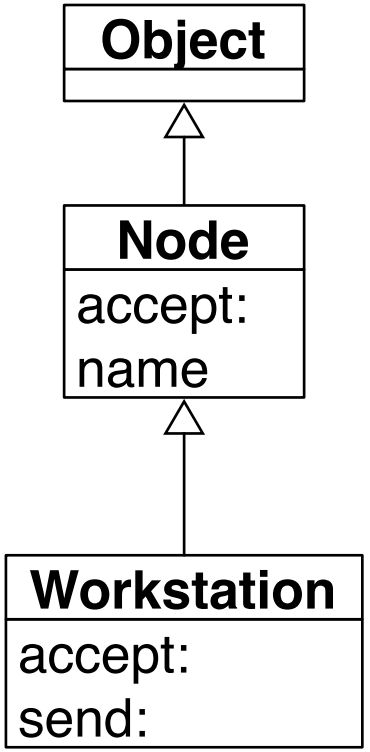

```
| mac1 |
mac1 := Workstation new name: #mac1.
mac1 name
```


RMod

ObjVlisp 5th Postulate

- P5: A class can be defined as a subclass of one or many other classes.

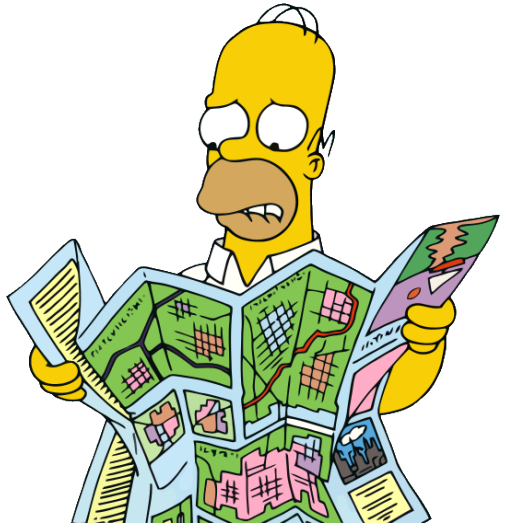
- We only implement single inheritance



RMod

Roadmap

- Context
- Classes as objects
- ObjVlisp in 5 postulates
- ***Instances/Classes/Metaclasses***
- Instance Structure and Behavior
- Class Structure
- Message Passing
- Object allocation & Initialization
- Class creation
- Inheritance Semantics
- Bootstrapping



Unifying Class/Instance

- Every object is instance of a class
- A class is an object, instance of a metaclass (P4)
  But all the objects are not classes
- Only one kind of objects without explicit distinction between classes and final instances.

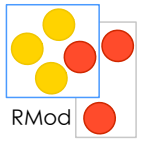
Instance/Class

- Sole difference between an instance and a class is the ability to respond to the creation message: ***new***.

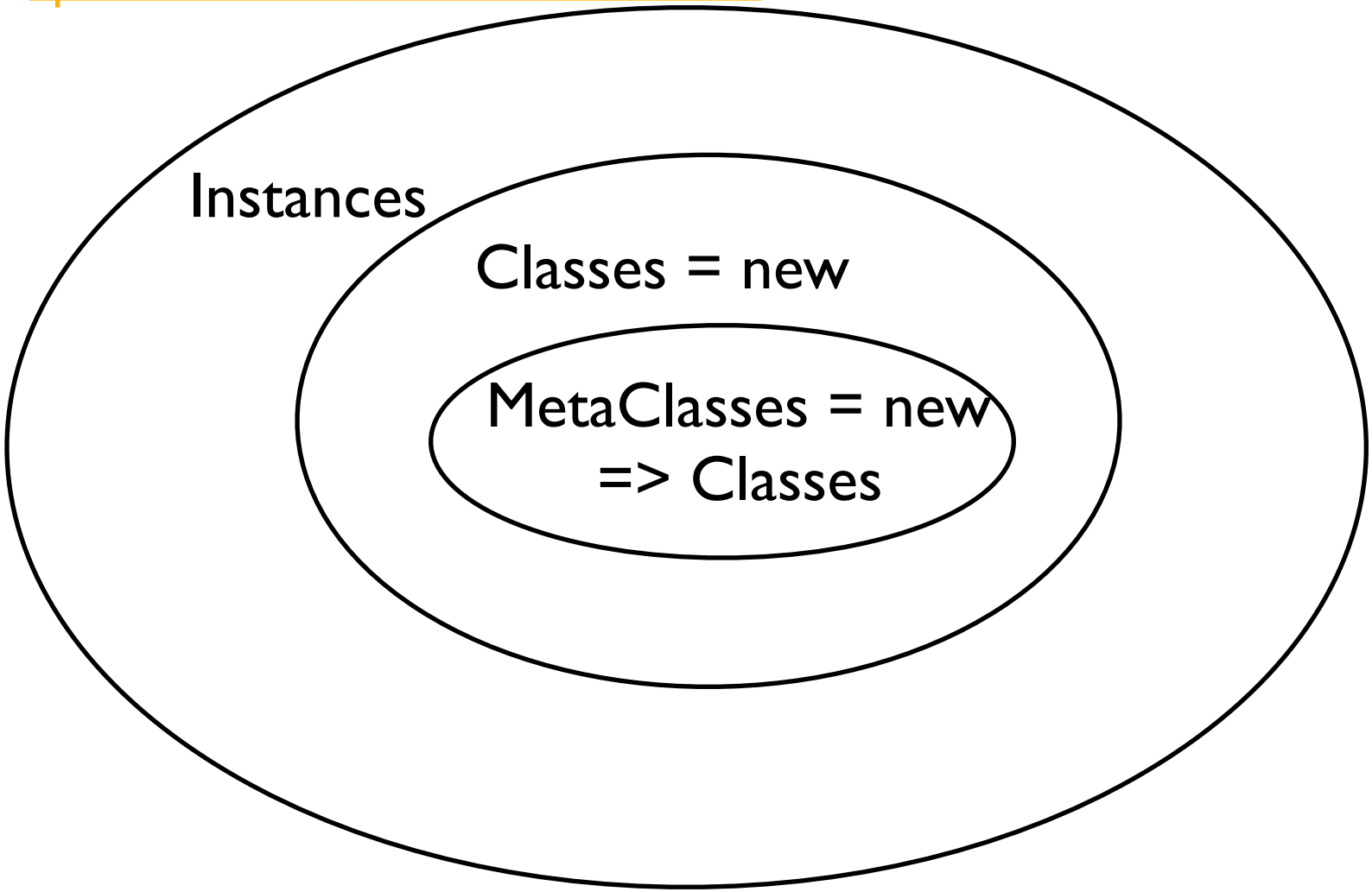
- Only a class responds to new

Class/Metaclass

A ***metaclass*** is only a class whose instance are classes

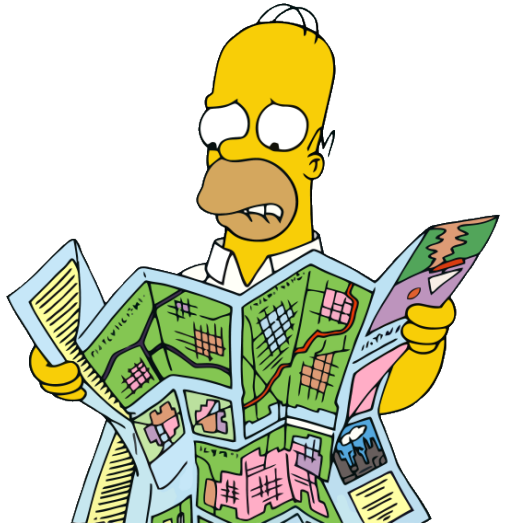


Instance/Class/Metaclass



RoadMap

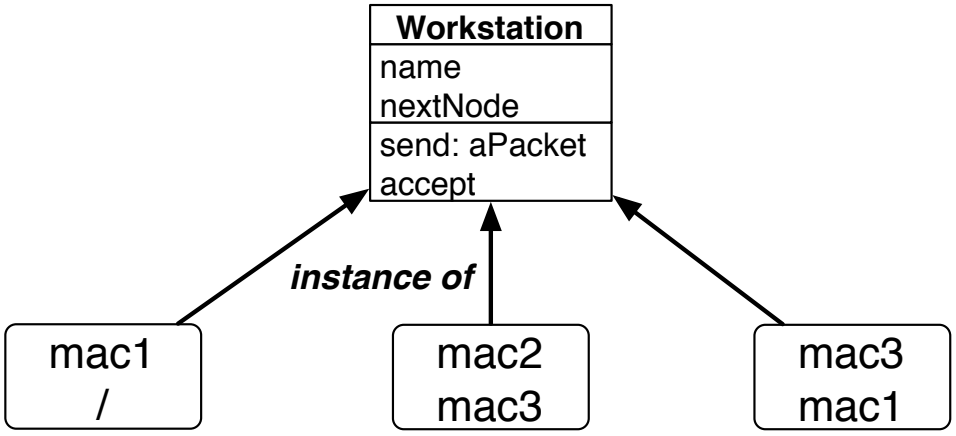
- Context
- Classes as objects
- ObjVlisp in 5 postulates
- Instances/Classes/Metaclasses
- ***Instance Structure and Behavior***
- Class Structure
- Message Passing
- Object allocation & Initialization
- Class creation
- Inheritance Semantics
- Bootstrapping



Instance Structure

Instance variables

- an ordered sequence of instance variables ***defined*** by a class
- ***shared*** by all instances
- values ***specific*** to each instance



some instances of Workstation

RMod

Impact on metaclass

The value of the i-v instance variable of a class is the list of instance variables of its instances

Point i-v
>>> #('x''y')

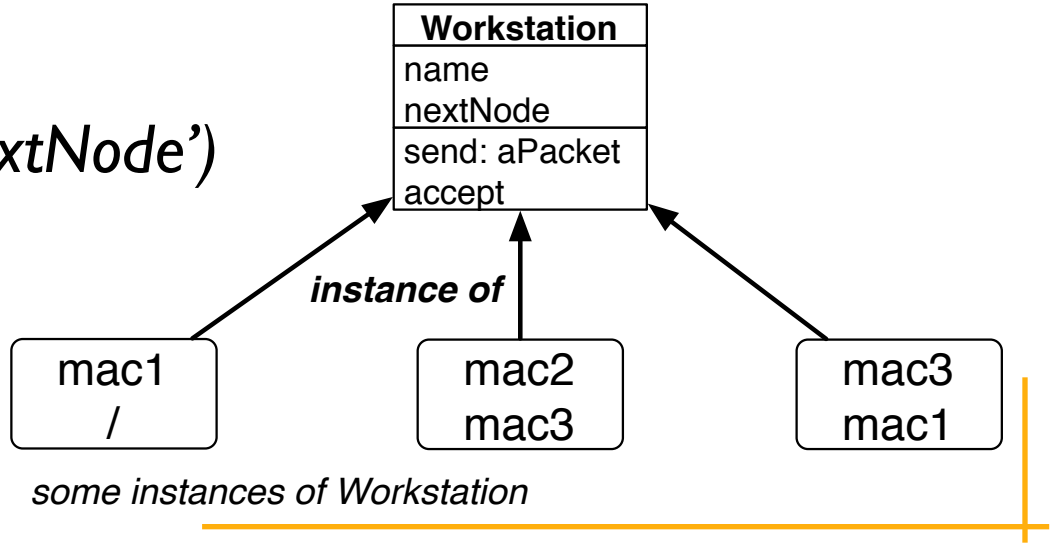
Workstation i-v
>>> #('name' 'nextNode')

In presence of Inheritance

In particular, every object possesses an instance variable ***class*** *(inherited from Object) that points to its class.*

mac1 class
>>> Workstation

Workstation i-v
>>> #('class' 'name' 'nextNode')



Instance Behavior

A method

- belongs to a class
- defines the behavior of ***all the instances*** of the class
- is stored into a dictionary that associates a key (the method selector) and the method body

Impact on metaclass

The method dictionary of a class is the value of the instance variable ***methodDict*** *defined on the metaclass* ***Class***.

RMod

Method implementation choices

- Do not want to have to write a parser and AST

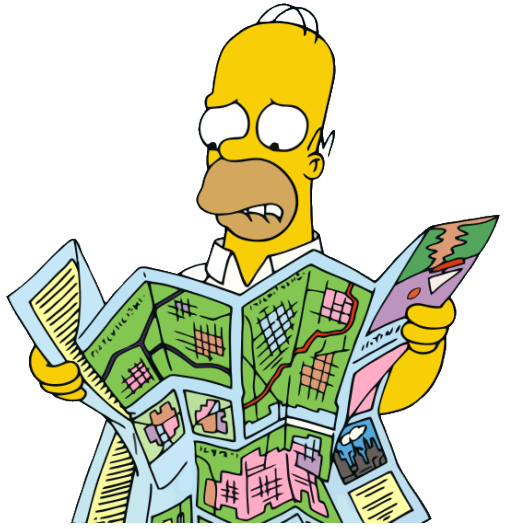
- Let's use a pharo block
- In methodDict

```
display -> [:objself |
                Transcript show: (objself unary: #name) ;
                cr ]
```

- No support to directly access to instance variables
- Use accessors

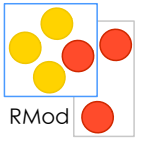
RoadMap

- Context
- Classes as objects
- ObjVlisp in 5 postulates
- Instances/Classes/Metaclasses
- Instance Structure and Behavior
- ***Class Structure***
- Message Passing
- Object allocation & Initialization
- Class creation
- Inheritance Semantics
- Bootstrapping



Class as an Object

- How would you represent a class?
- What state do you need to represent a class?



Class as an Object

As an instance factory responsible for holding instance behavior, a Class has 4 instance variables that describe a class:

- ***name*** the class name
- ***superclass*** its superclass
- ***i-v*** the list of its instance variables (used during instance creation)
- ***methodDict*** a method dictionary (used during lookup)

Class as an Object

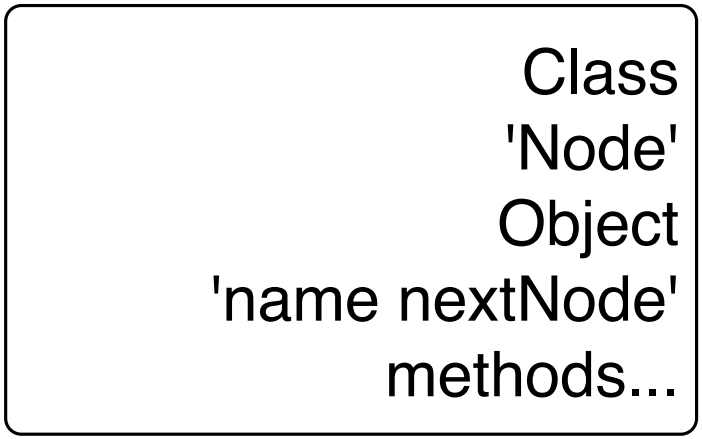
- Workstation class -> Class

- A class possesses the instance variable ***class*** inherited from the class Object that refers to its class (as any object!)

- The value of the *class* instance variable is an identifier of the class

Class Node as Object

The class Node



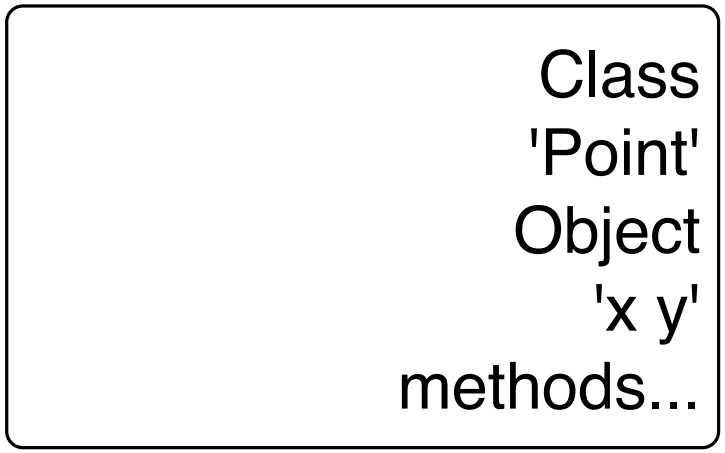
is instance of Class
named Node
inherits from Object
has instance variables
defines some methods

- Node is instance of class Class because we can create instances of Node sending it the message new

RMod

Class Point as Object

The class Point



is instance of Class
named Point
inherits from Object
has instance variables
defines some methods

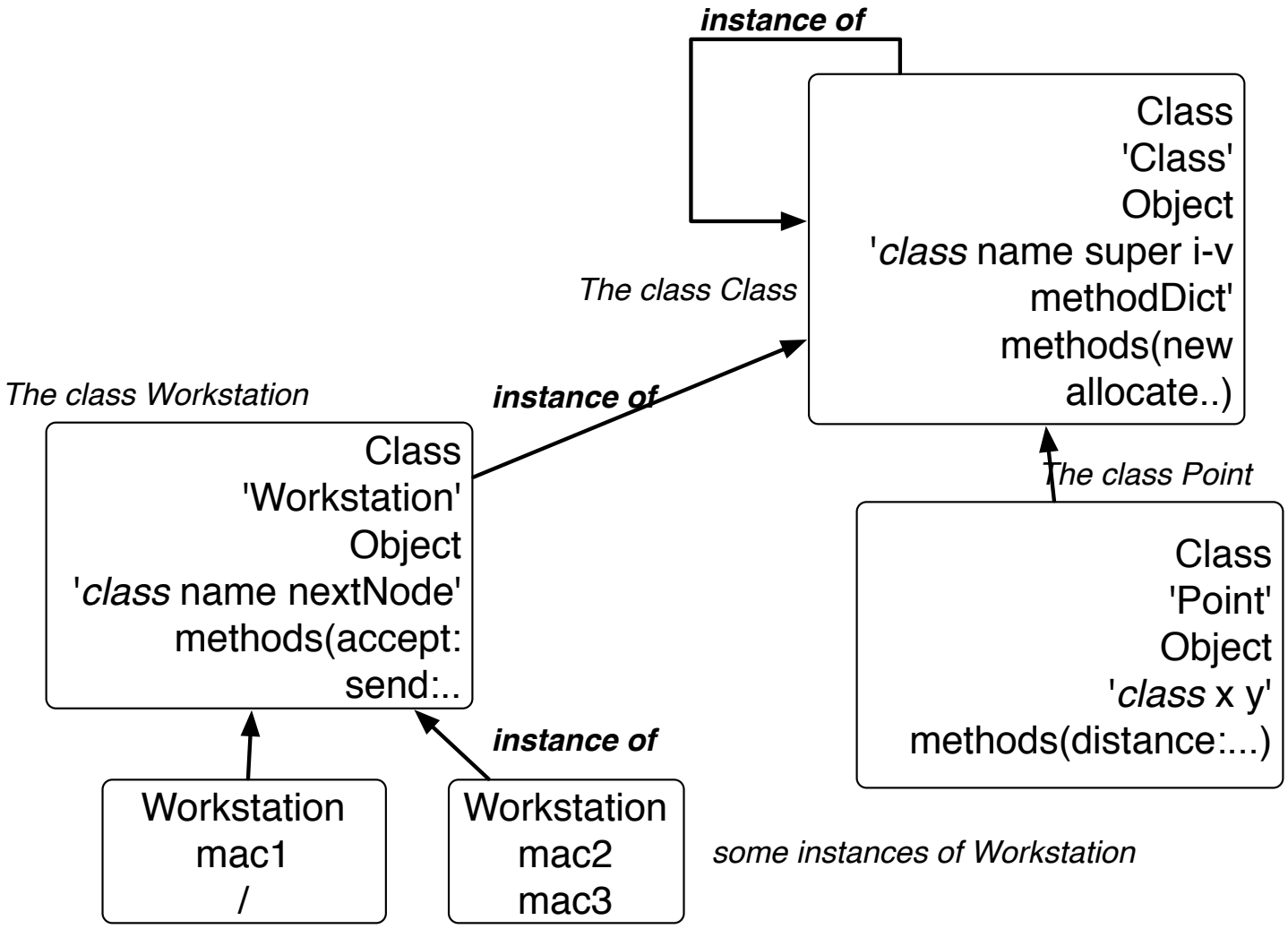
RMod

The class Class

The class Class

Class	*is instance of Class*
'Class'	*named Class*
Object	*inherits from Object*
'name super i-v	*has instance variables*
methodDict'	
methods...	*defines some methods*

Instantiation graph



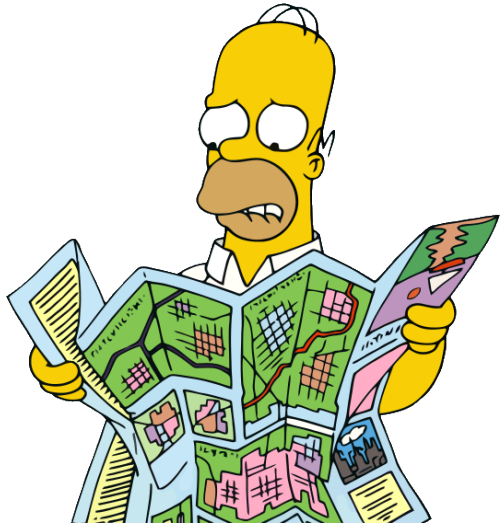
RMod

Sum up: The class Class

- Initial metaclass
- Defines the structure and behavior of all the ***classes***

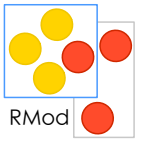
RoadMap

- Context
- Classes as objects
- ObjVlisp in 5 postulates
- Instances/Classes/Metaclasses
- Instance Structure and Behavior
- Class Structure
- ***Inheritance and its Semantics***
- Object allocation & Initialization
- Class creation
- Bootstrapping



Inheritance

- Inheritance is an **incremental** definition
- We defined a class by reusing its superclass



Two kinds of inheritance

Static for the state

- Subclasses get superclass state
- *At compilation time* ***(class-creation time)***

Dynamic for behavior

- Inheritance tree walked ***at run-time***

Instance Variable Inheritance

- Static for the instances variables
- Done once at the class creation

- When C is created, its instance variables are the union of the instance variables of its superclass with the instance variables defined in C.

- final-instance-variables (C) =

    OrderedUnion (iv (super C)),

        local-instance-variables(C))

Instance Variable Inheritance

Point iv
>>> #('class' 'x' 'y')

3DPoint iv
>>> #('class' 'x' 'y' 'z')

No repetition of equally name instance variables
Reuse method definition of superclass

In particular from Object

The class Object defines the instance variable ***class*** so that any object can know its class

(10@10) class -> Point
Point class -> Class

RMod

BTW: What is Object?

Object: Minimal Shared Behavior

- Represents the common behavior shared by all the objects:
  - classes
  - final instances

- Every object knows its class: class instance variable

- Methods:
  - initialize (instance variable initialization)
  - error, class, metaclass?, class?
  - iv-set, iv-ref (meta operations)

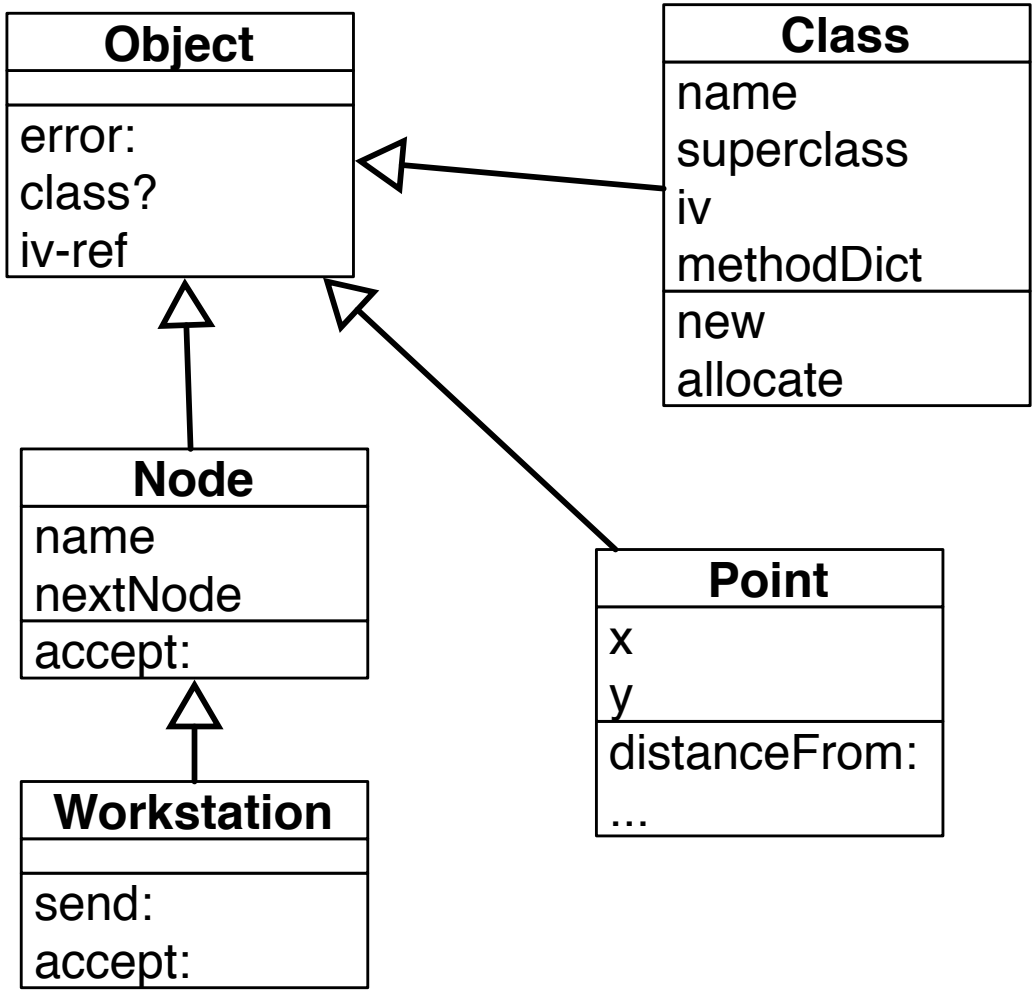
Inheritance Graph

- ***Object*** is the root of the hierarchy.

- a Workstation is an object (should at least understand the minimal behavior), so ***Workstation*** inherits from ***Object***

- a class is an object so ***Class*** inherits from ***Object***
- In particular, class instance variable is inherited from Object class.

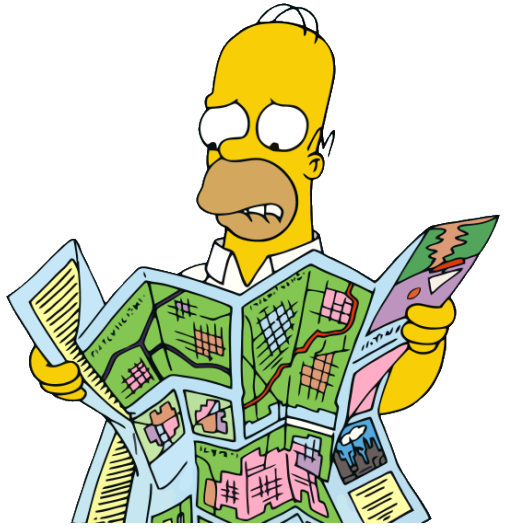
Inheritance Graph



RMod

RoadMap

- Context
- Classes as objects
- ObjVlisp in 5 postulates
- Instances/Classes/Metaclasses
- Instance Structure and Behavior
- Class Structure
- **Inheritance and its Semantics *(Method Lookup)***
- Object allocation & initialization
- Class creation
- Bootstrapping

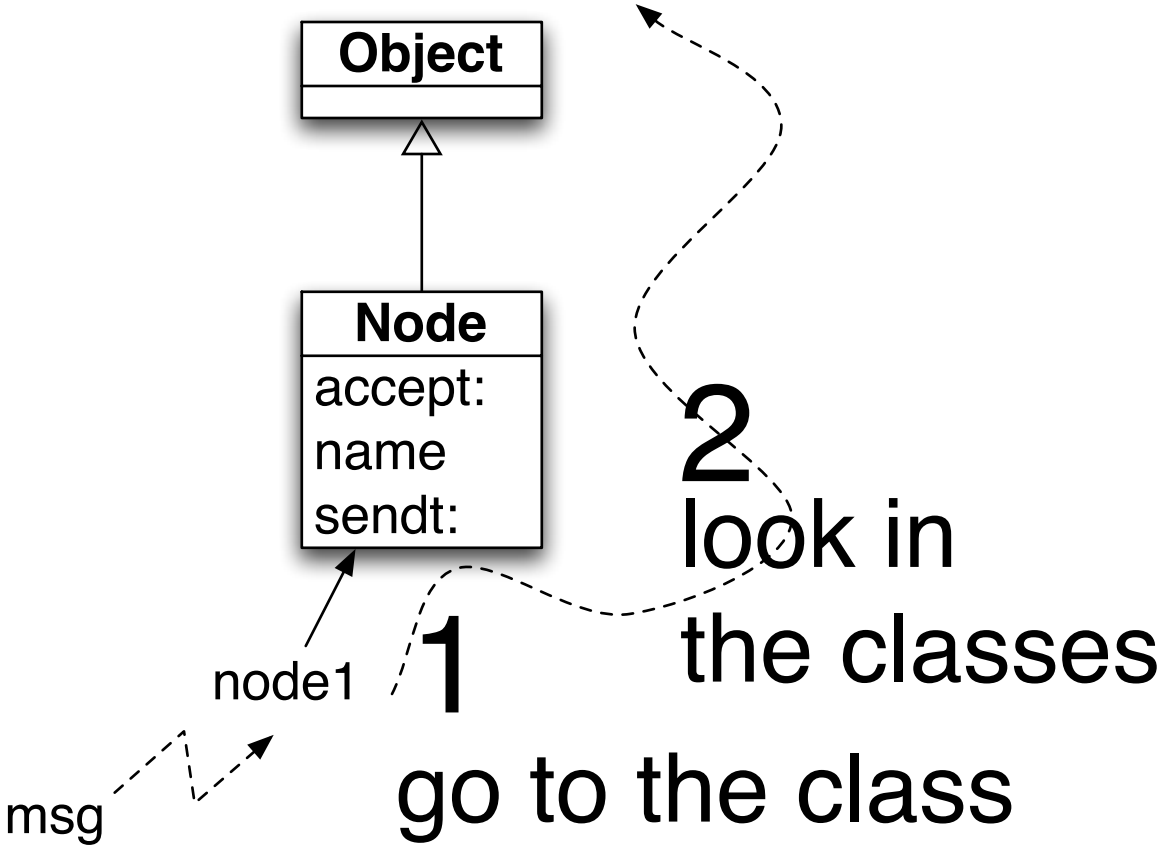


Sending a message

Two steps:

- ***Lookup for the method*** corresponding to the message walking the inheritance tree
- Applying the method ***to the receiver***

Lookup!


Object
Node
accept:
name
sendt:
node1
msg
1
go to the class
2
look in
the classes

Method Inheritance

Walks through the inheritance graph between classes using the super instance variable

lookup (selector class receiver):

if the method is found then return it

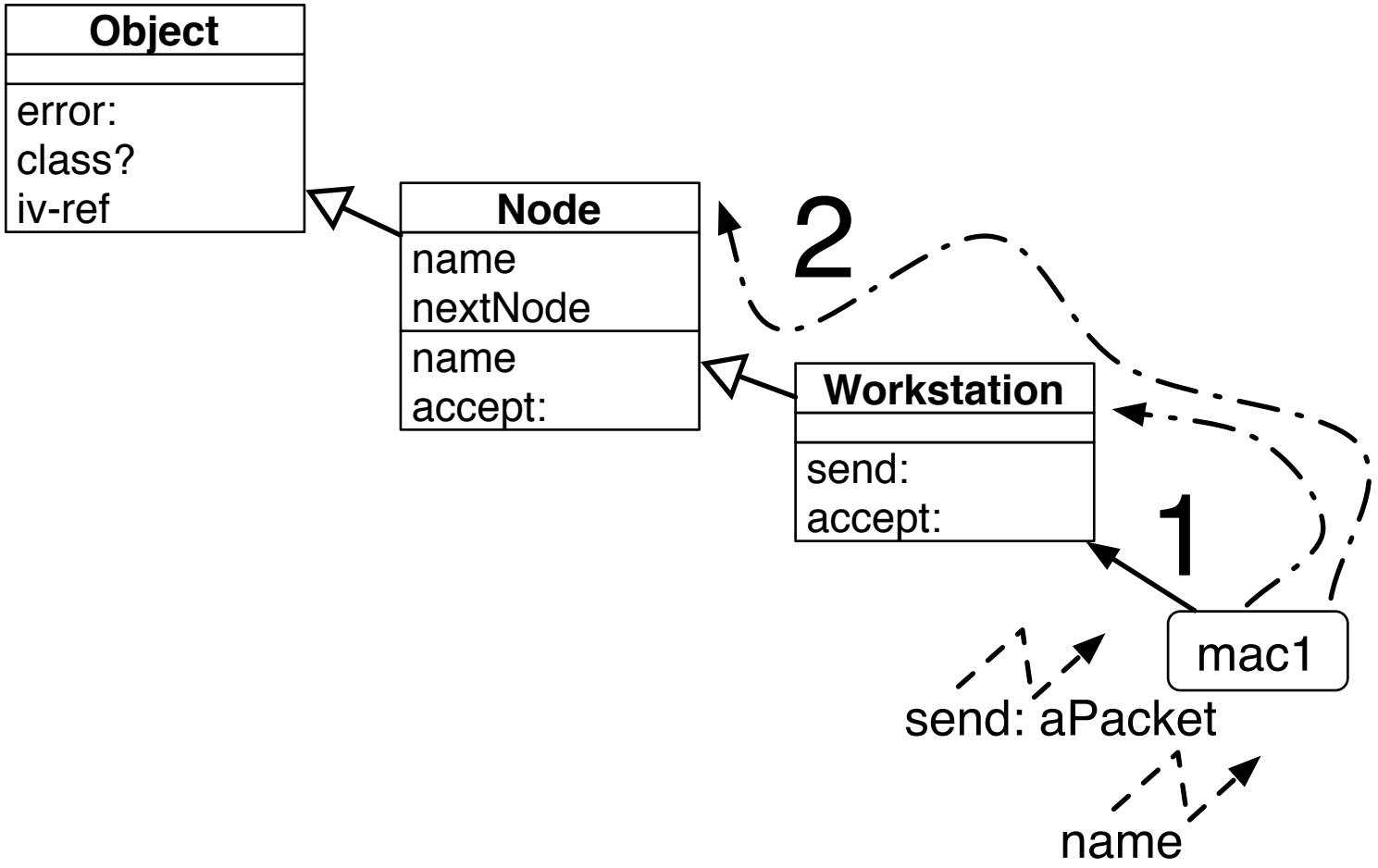
else if class == Object

then [receiver error selector]

else lookup (selector super(class) receiver)

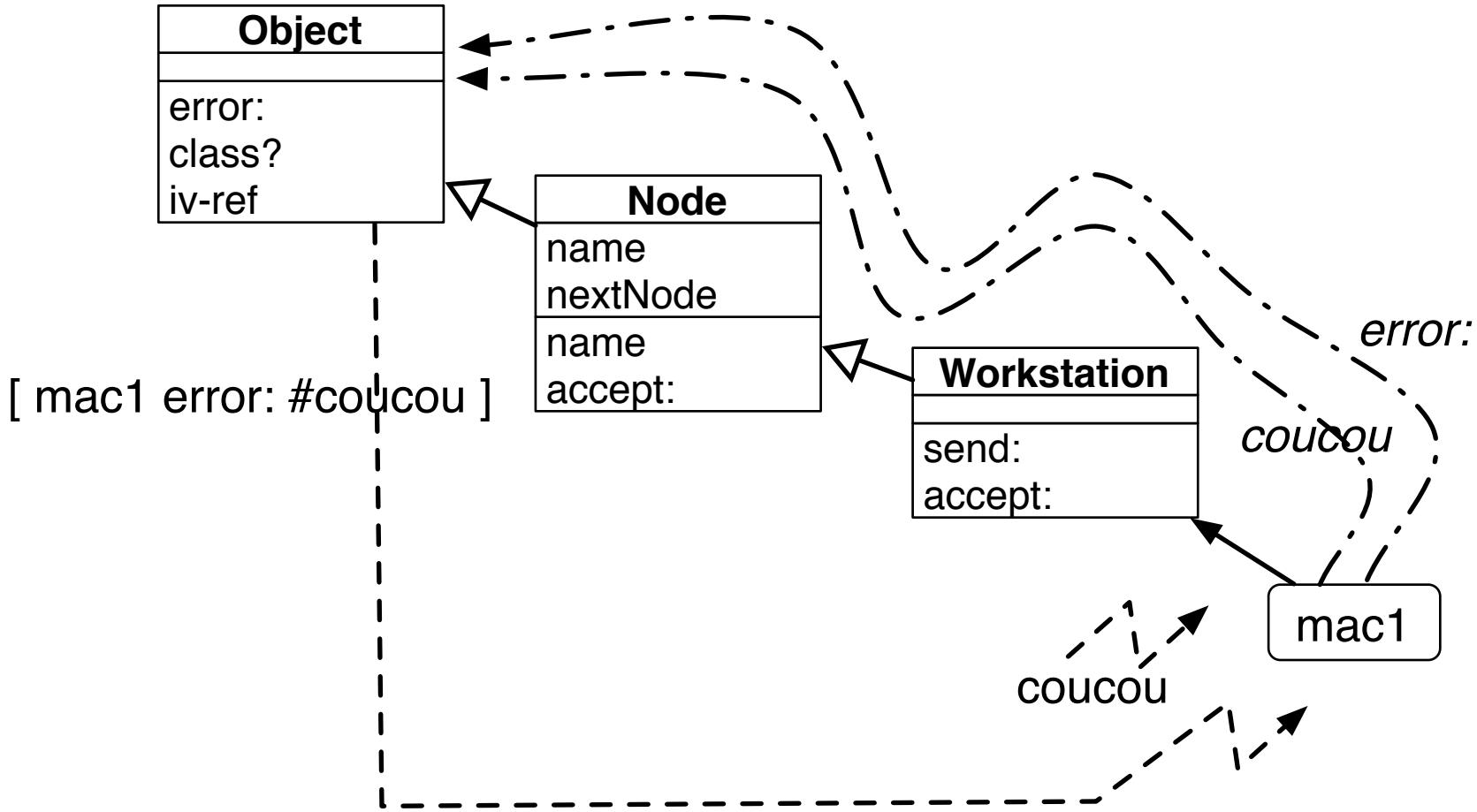
the error method can be specialized to handle the error.

Lookup (I)



RMod

Lookup (II)



RMod

Method Lookup

Two steps process

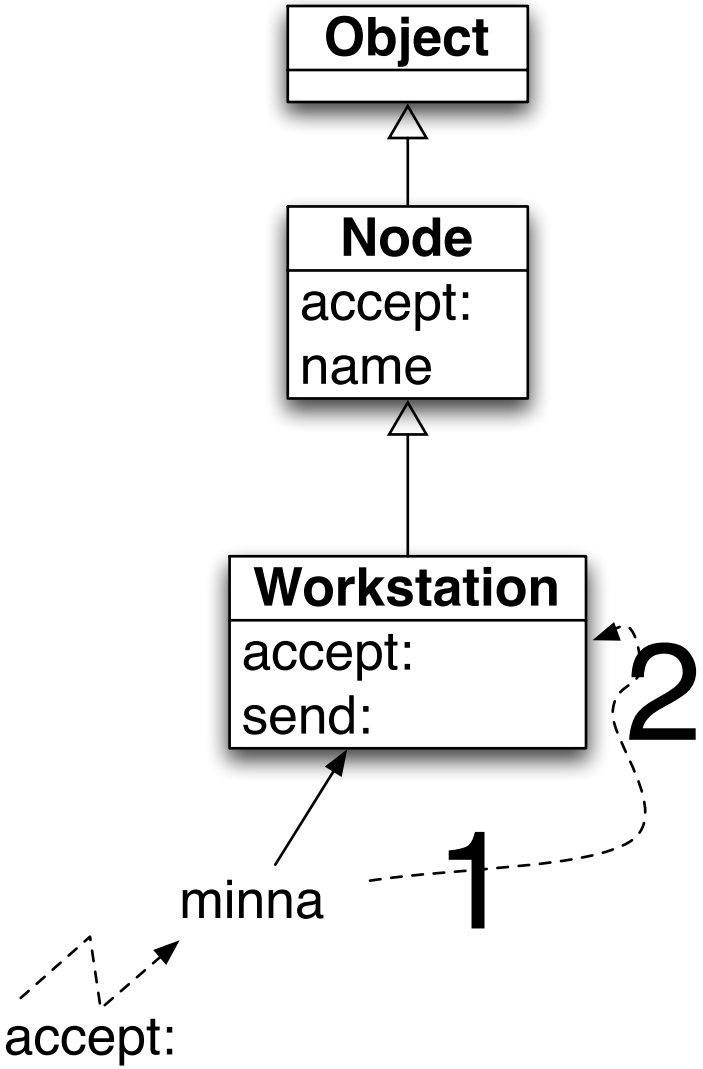
The lookup **starts** in the ***CLASS*** of the ***RECEIVER***.

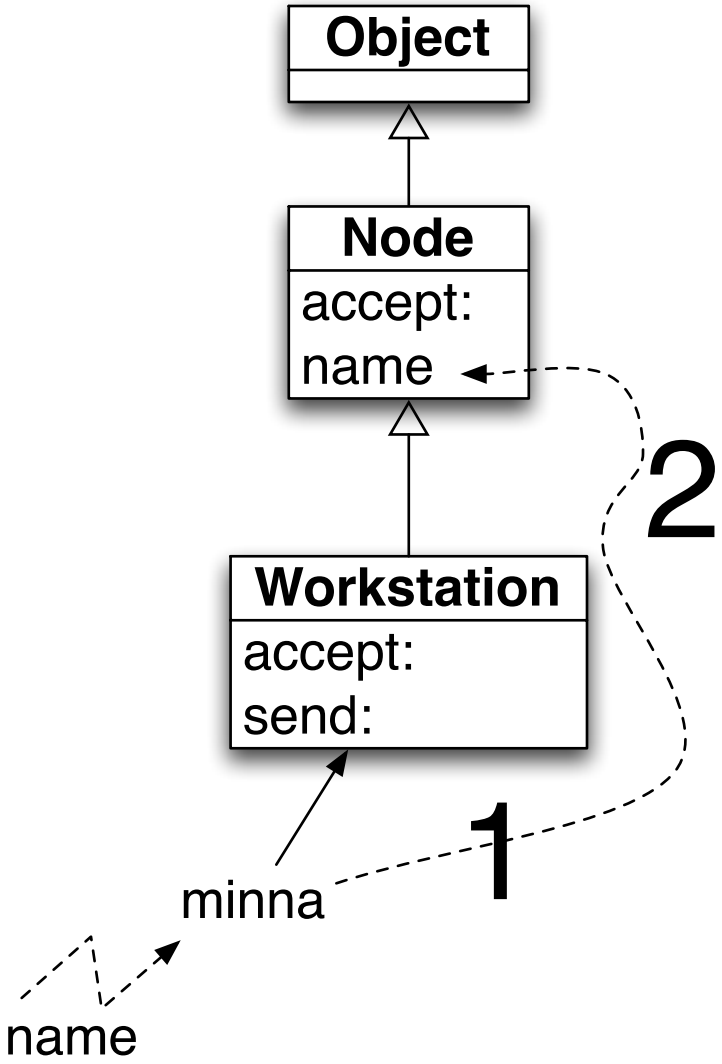
If the method is defined in the method dictionary,

Returns it

Otherwise the ***search continues*** in the superclass of the receiver's class. If no method is found and there is no superclass to explore (class Object), this is an error

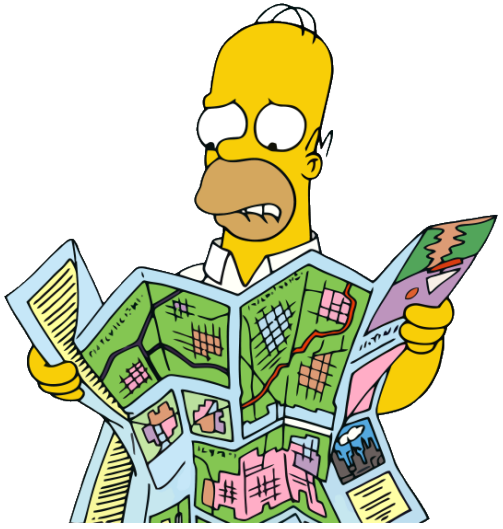
Some Cases


Object
Node
accept:
name
Workstation
accept:
send:
2
minna
1
accept:


Object
Node
accept:
name
2
Workstation
accept:
send:
minna
1
name

RoadMap

- Context
- Classes as objects
- ObjVlisp in 5 postulates
- Instances/Classes/Metaclasses
- Instance Structure and Behavior
- Class Structure
- ***Inheritance and its Semantics (self)***
- Object allocation & Initialization
- Class creation
- Bootstrapping



Do you understand self?

What is self?

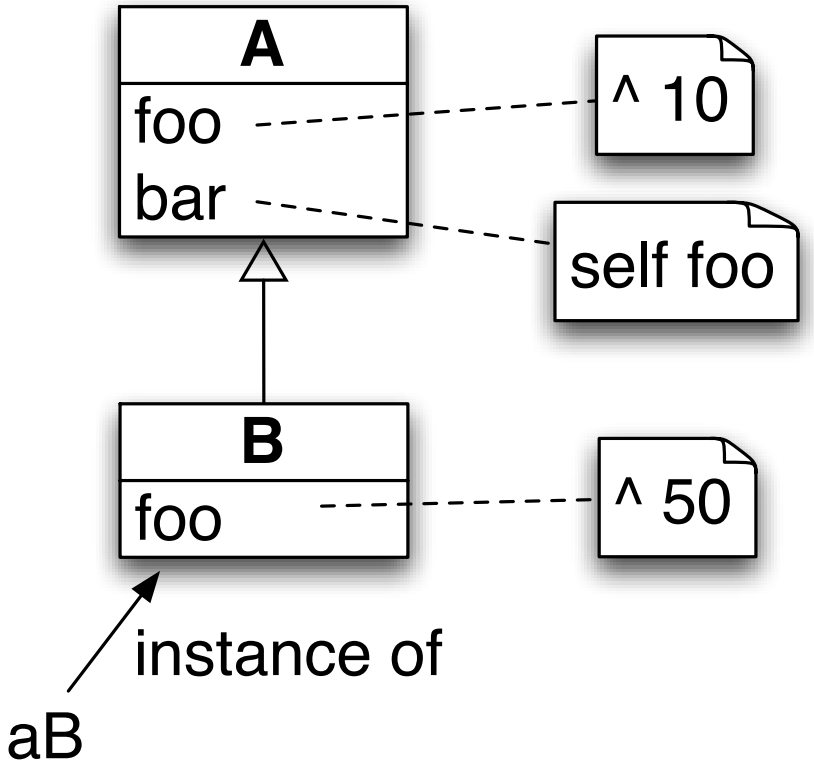
Method lookup starts in receiver class

A new foo
>>>

B new foo
>>>

A new bar
>>>

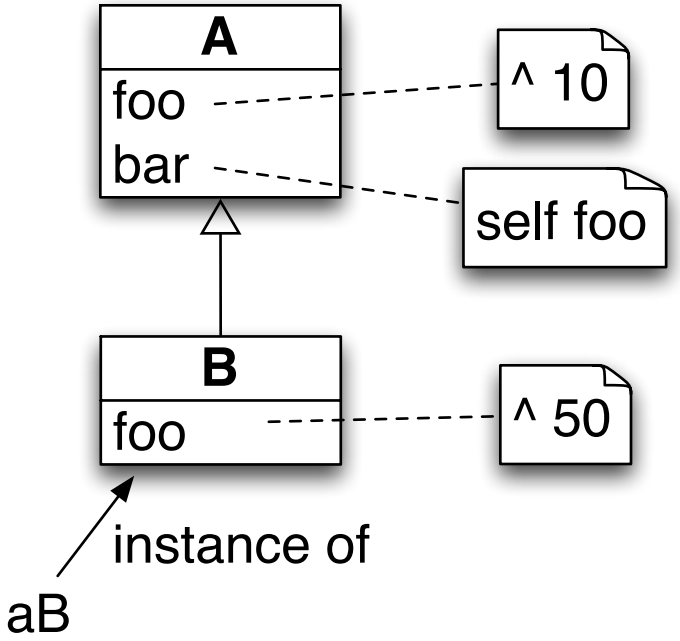
B new bar
>>>



Method Lookup starts in Receiver Class

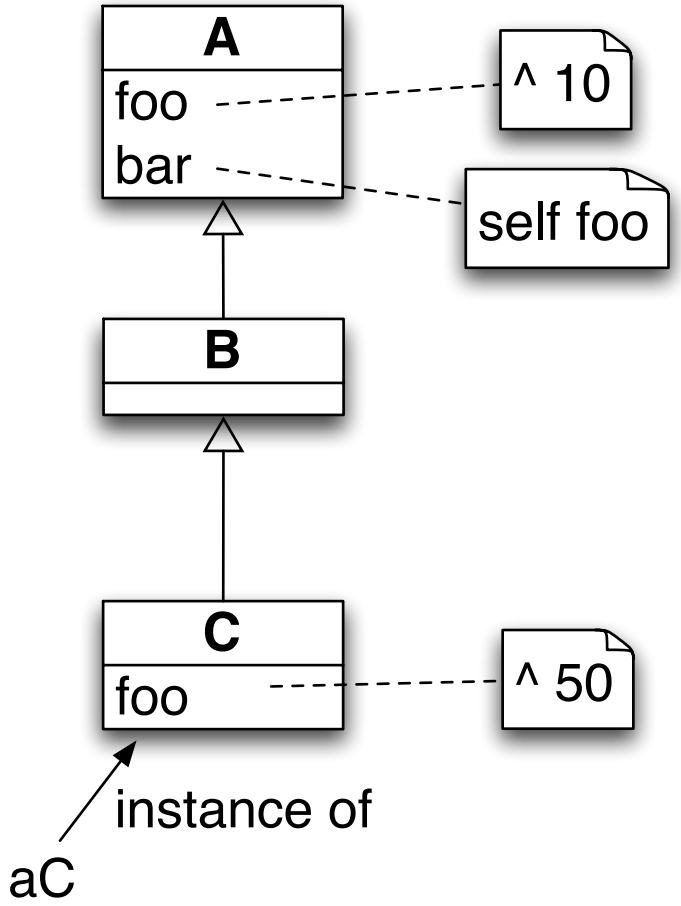
aB foo
(1) aB class => B
(2) Is foo defined in B?
(3) Foo is executed -> 50

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



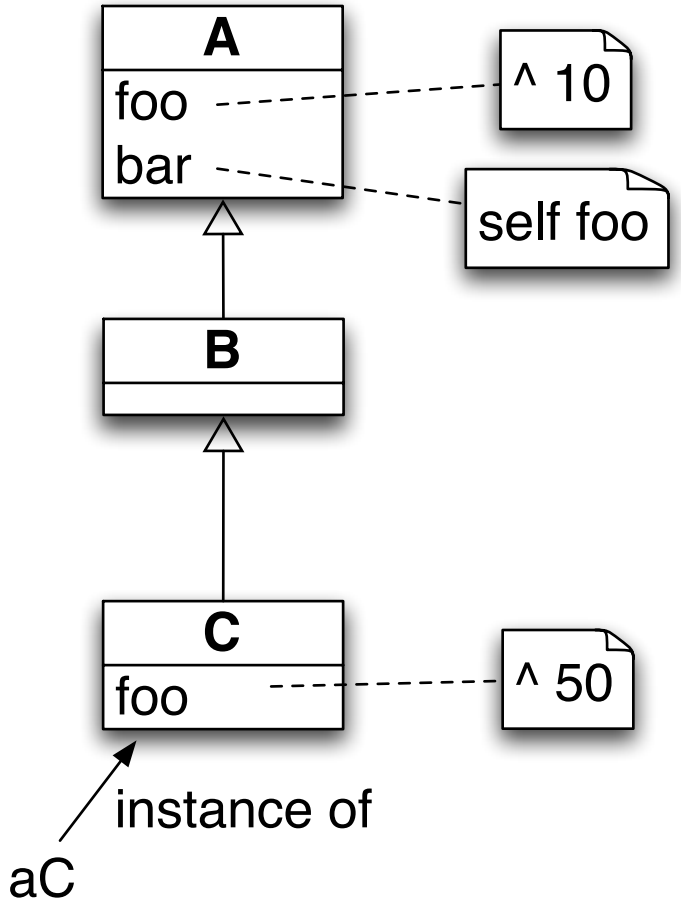
self **always** represents the receiver

A new foo
>>>
B new foo
>>>
C new foo
>>>
A new bar
>>>
B new bar
>>>
C new bar
>>>



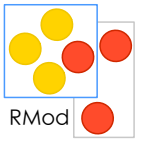
self **always** represents the receiver

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

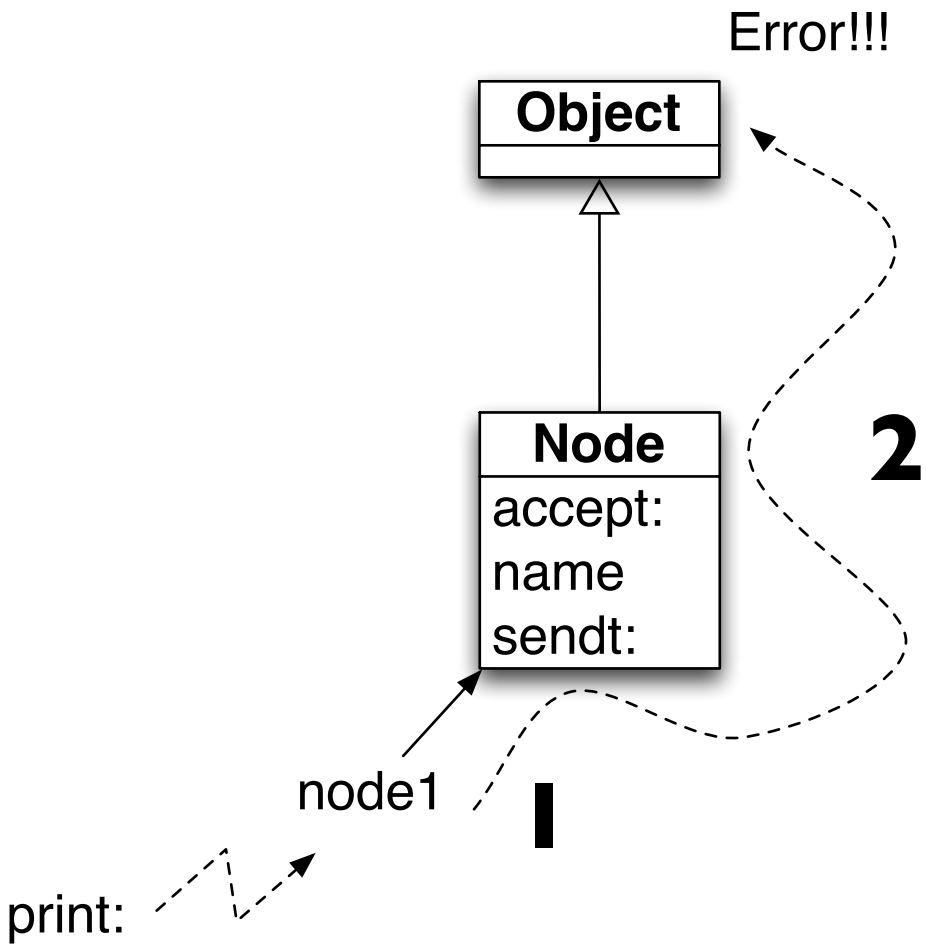


When message is not found

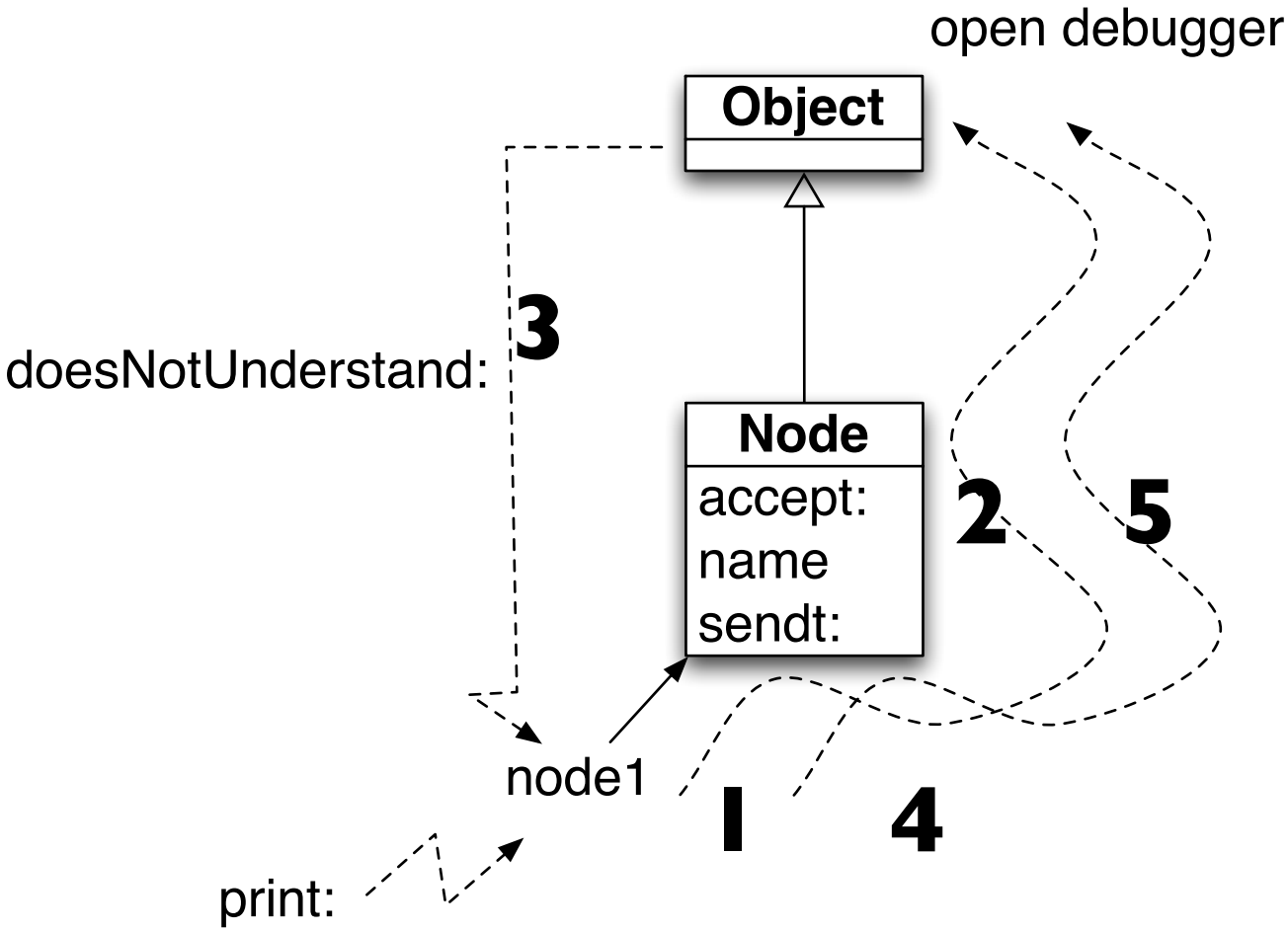
If no method is found and there is no superclass to explore (class Object), a new method called #doesNotUnderstand: is sent to the receiver, with a representation of the initial message.



Graphically…



Graphically…



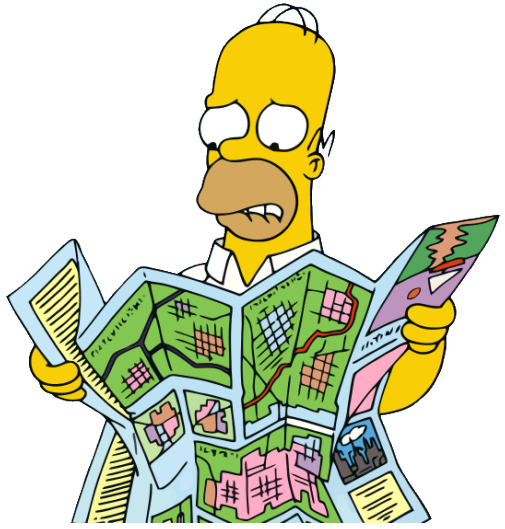
Why sending a message for error

We could manage the error in the low-level language.
Why sending an ObjVlisp message?

- It lets the class manages its error
- It supports proxy creation and many design possibility
- Important reflective hook

RoadMap

- Context
- Classes as objects
- ObjVlisp in 5 postulates
- Instances/Classes/Metaclasses
- Instance Structure and Behavior
- Class Structure
- ***Inheritance and its Semantics (super)***
- Object allocation & Initialization
- Class creation
- Bootstrapping



What is super?

Let us take two mins….

super *changes* lookup starting class

A new foo

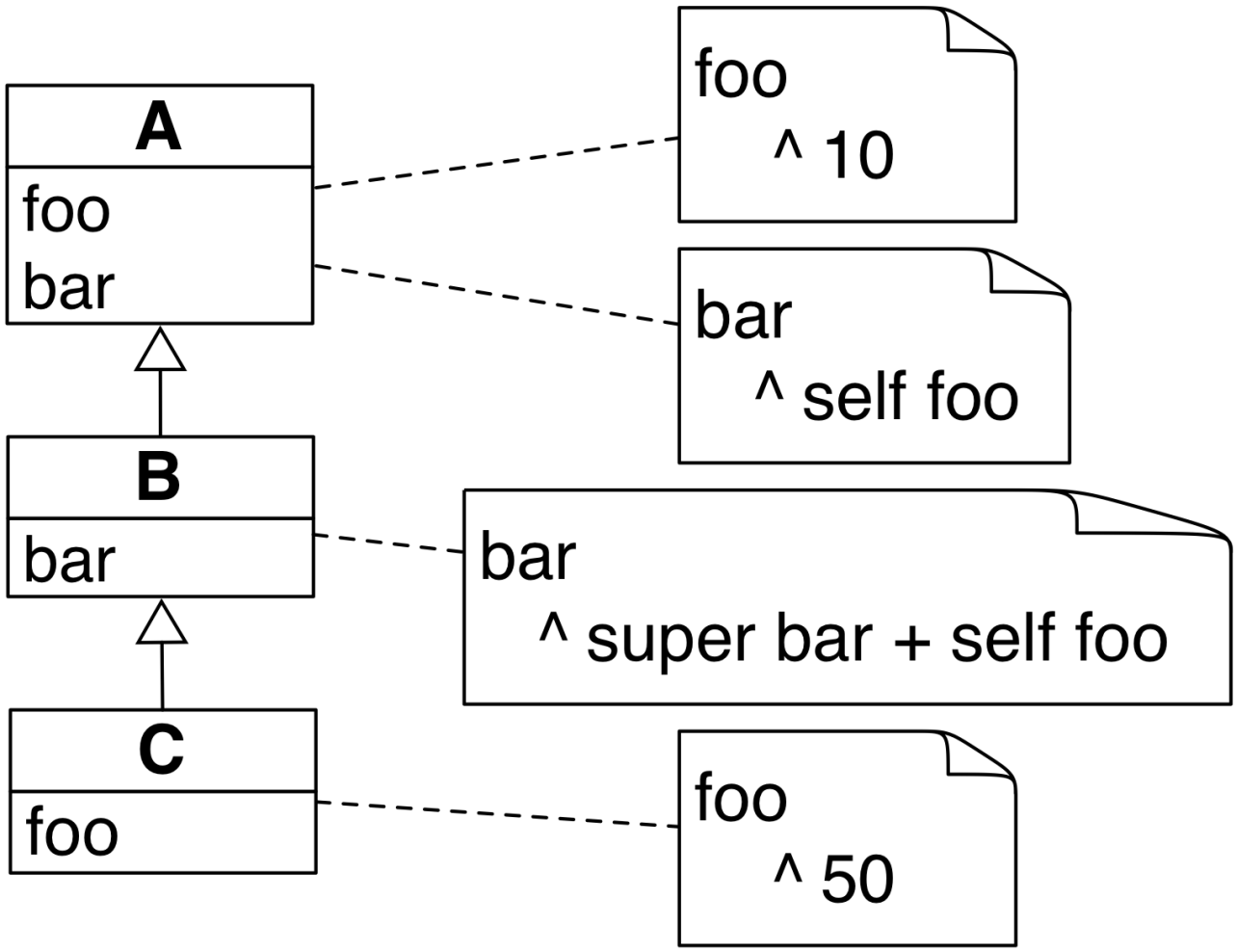
A new bar

B new foo

B new bar

C new foo

C new bar

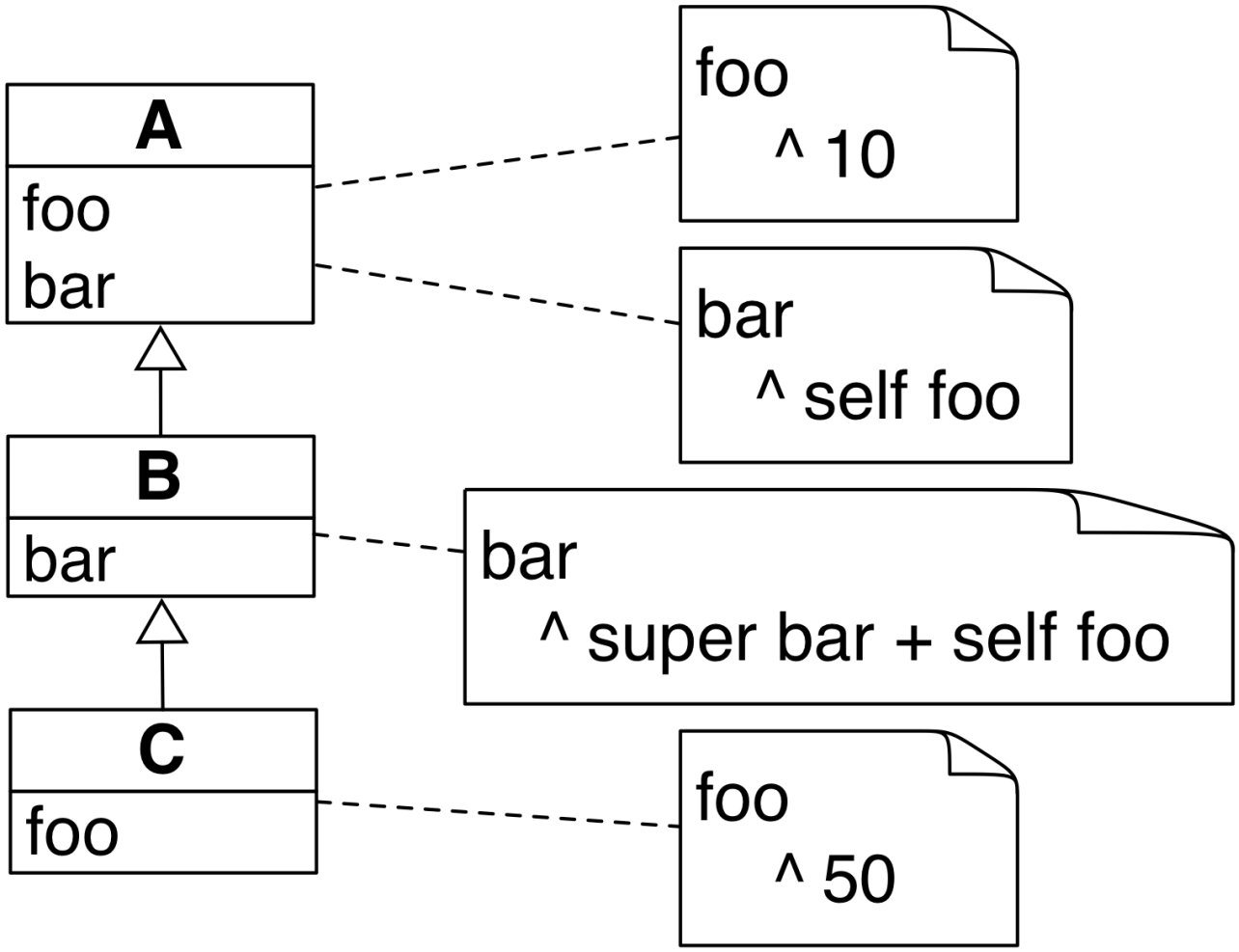


super *changes* lookup starting class

A new bar
>>> 10

B new bar
>>> 10 + 10

C new bar
>>> 50 + 50



super is NOT

the superclass of the receiver class

an instance of the superclass

NO

No

no

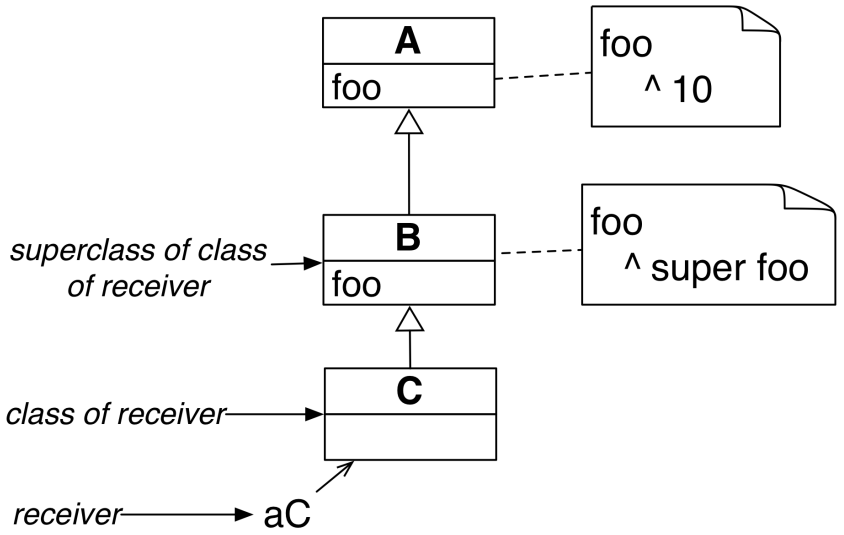
o

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

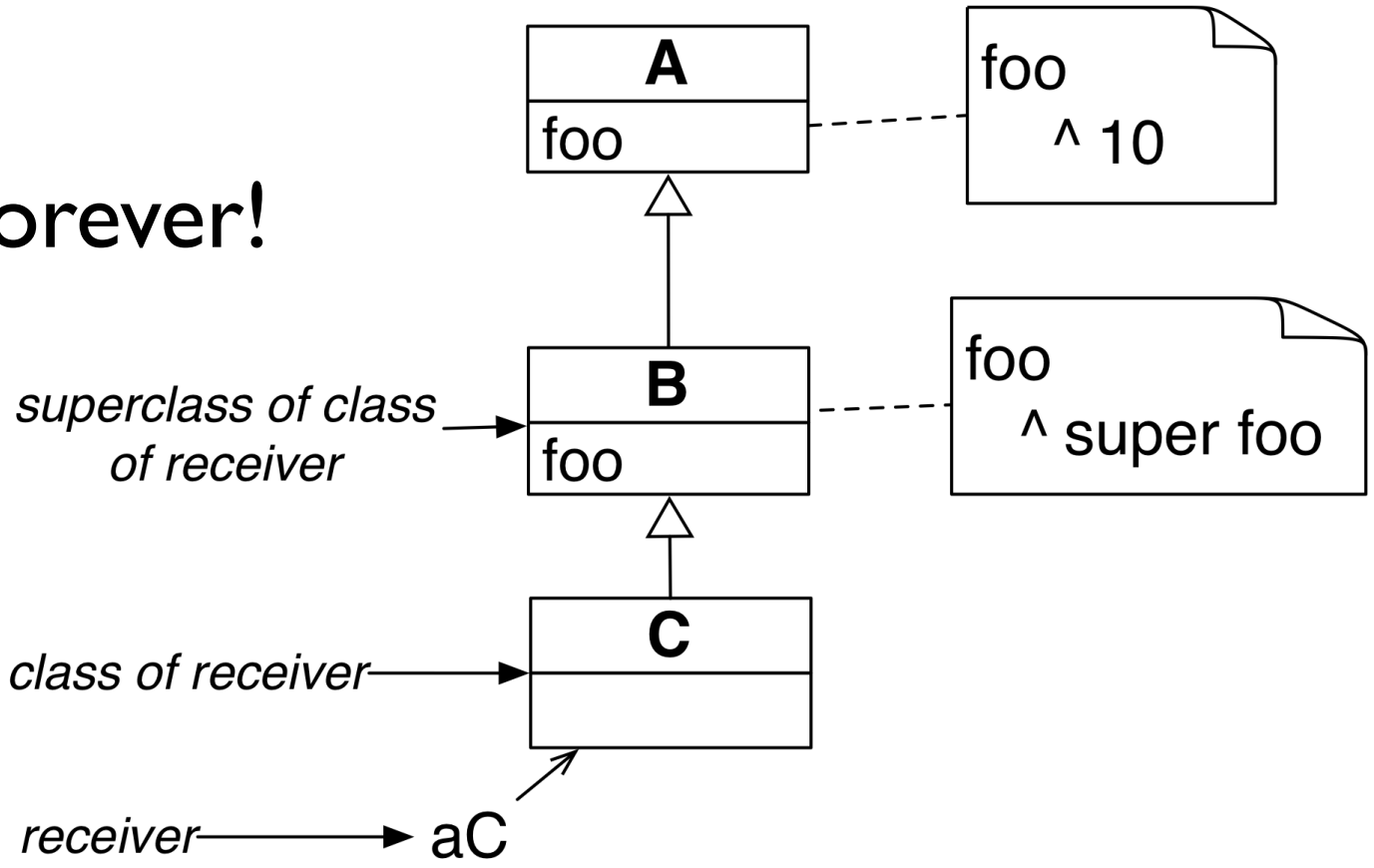
Some books are PLAIN WRONG!

Suppose the **WRONG** definition: *"The semantics of super is to start the lookup of a method in the superclass of the receiver class"*



Some books are PLAIN WRONG!

aC foo
loops forever!

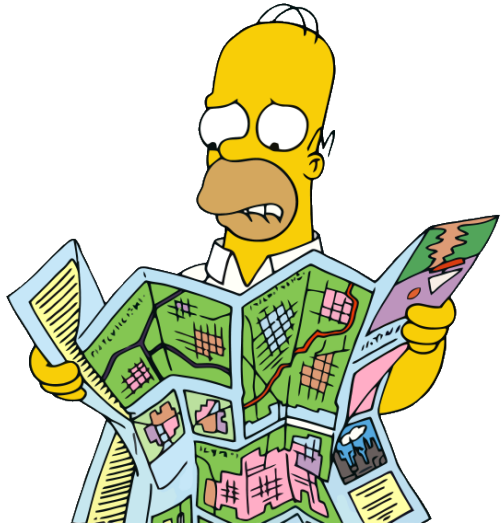


Dynamic vs. Static

- self is dynamic:
  - Using self the lookup of the method begins in the class of the receiver.
  - Bound at execution-time
- super is static:
  - Using super the lookup of the method begins in the superclass of the class of the method containing the super expression (not in the superclass of the receiver class).
  - Bound at compile-time

RoadMap

- Context
- Classes as objects
- ObjVlisp in 5 postulates
- Instances/Classes/Metaclasses
- Instance Structure and Behavior
- Class Structure
- Inheritance and its Semantics
- ***Object allocation & Initialization***
- Class creation
- Bootstrapping



Object Creation

Creation of **instances** of the class Point

- [Point new :x 24 :y 6]
- [Point new]
- [Point new :y 10 :y 15]

Object Creation

Creation of the **class** Point instance of Class

- [Class new
  :name 'Point'
  :super 'Object'
  :i-v #(x y)
  :methods (x ...display ...)
  ]

RMod

One way to create objects

Send the message new to a class!

Object Creation: new

- Object Creation = initialisation O allocation

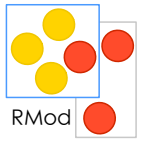
- Creating an instance is the composition of two actions:
  memory allocation: ***allocate*** method
  object initialisation: ***initialize*** method

RMod

Instance creation

[aClass new args] = [[aClass allocate] initialize args]

- new creates an object: class or final instances
- new is a class method



Object Allocation

Should return:

- Object with empty instance variables
- Object with an identifier to its class

- Done by the method ***allocate*** defined on the metaclass Class

- The ***allocate*** method is a *class* method (it is applied to classes in respond to the message ***allocate***)

RMod

Allocation Examples

[Point allocate]
>>> #(Point nil nil) for x and y

[Workstation allocate]
>>>#(Workstation nil nil) for ‘name’ and ‘nextNode’

[Class allocate]
>>>#(Class nil nil nil nil nil) for name, super, iv, keywords and methodDict

Object Initialization

- instance variable values are given by means of keywords (:x ,:y) associated with the instances variables

- [Point new :y 6 :x 24]
  -> [#(Point nil nil) initialize (:y 6 :x 24)]
  -> #(Point 24 6)

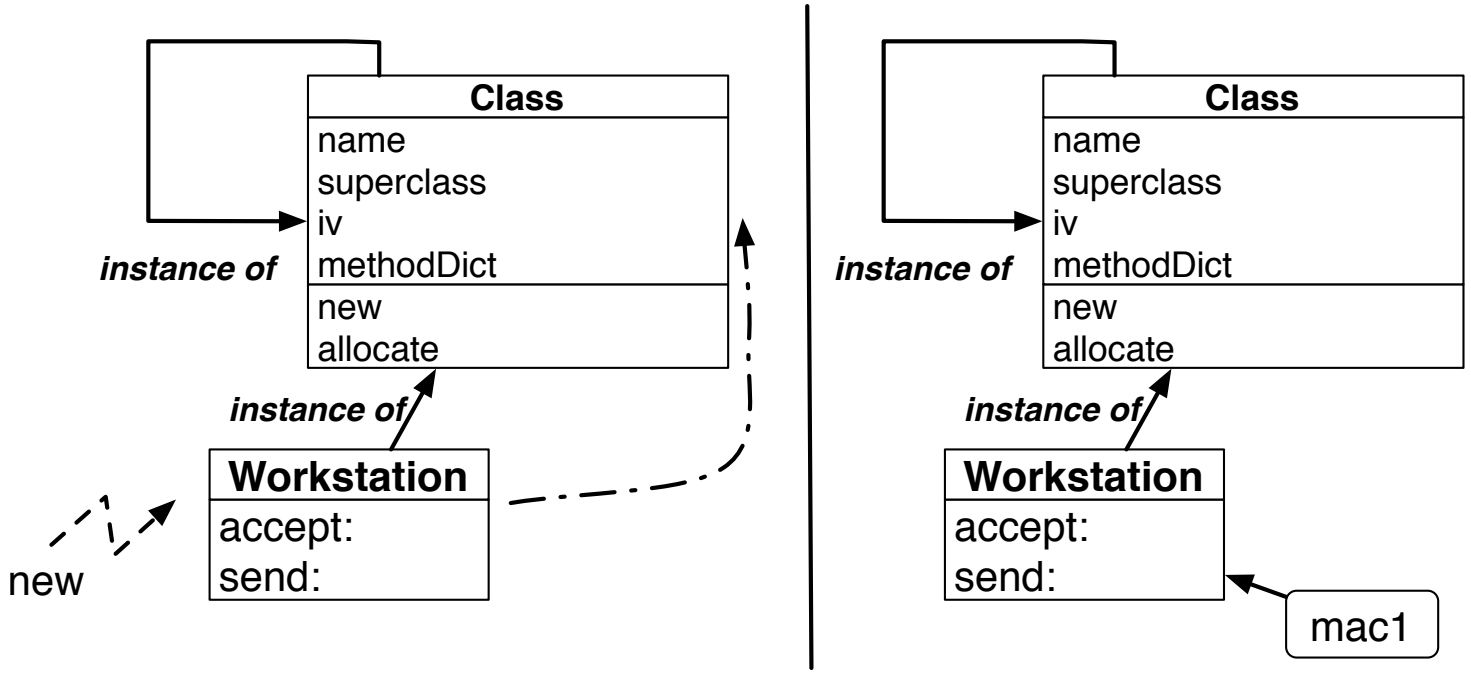
Object Initialization

- [Point new :y 6 :x 24]
  -> [#(Point nil nil) initialize (:y 6 :x 24)]
  -> #(Point 24 6)

- Two steps
  - get the values specified during the creation (y -> 6, x -> 24)
  - assign the values to the instance variables of the created object.

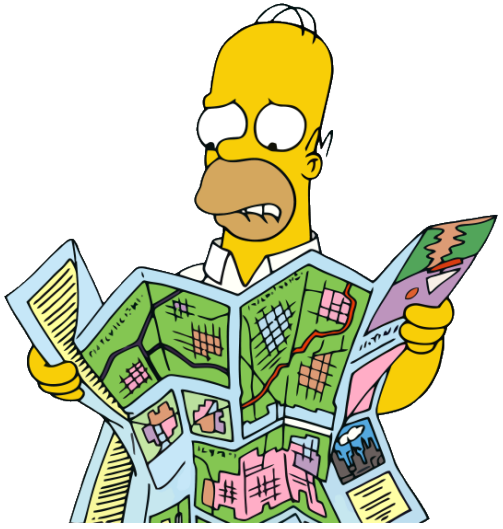
Instance Creation: Metaclass Role

Lookup method in the class of the receiver then we apply it to the receiver.



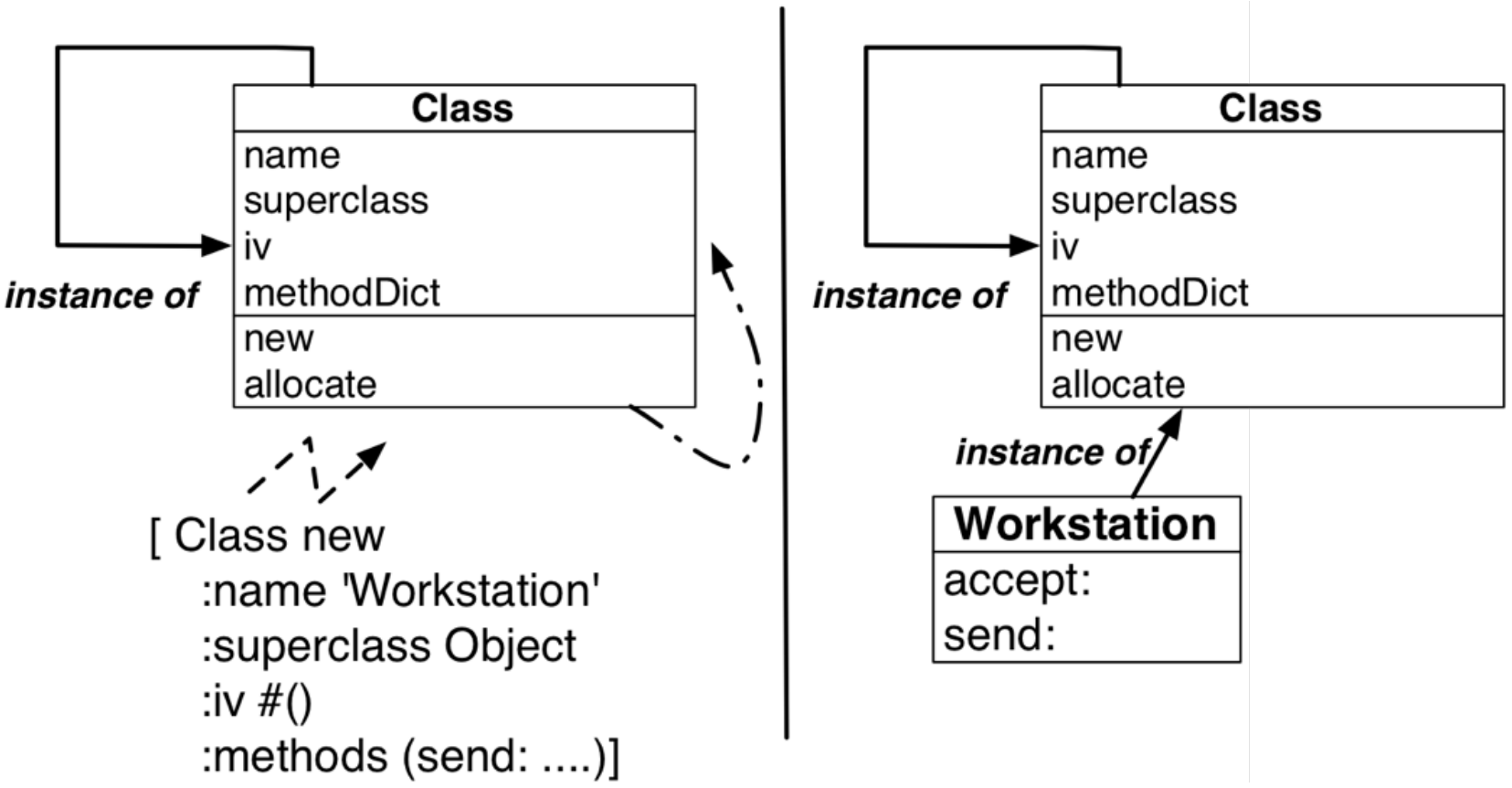
RoadMap

- Context
- Classes as objects
- ObjVlisp in 5 postulates
- Instances/Classes/Metaclasses
- Instance Structure and Behavior
- Class Structure
- Inheritance and its Semantics
- Object allocation & Initialization
- ***Class creation***
- Bootstrapping

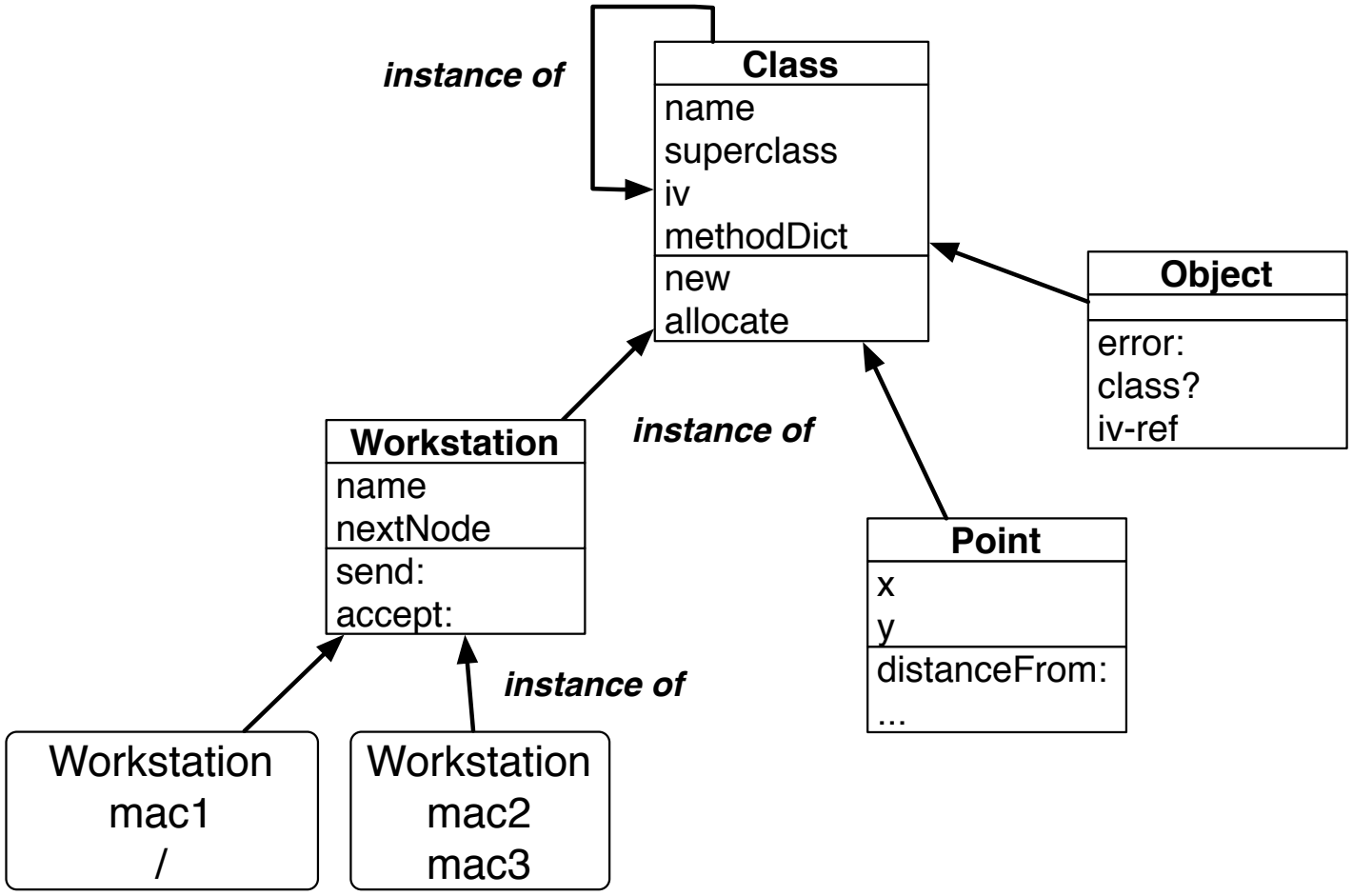


Class Creation

Look in the *class of the receiver*

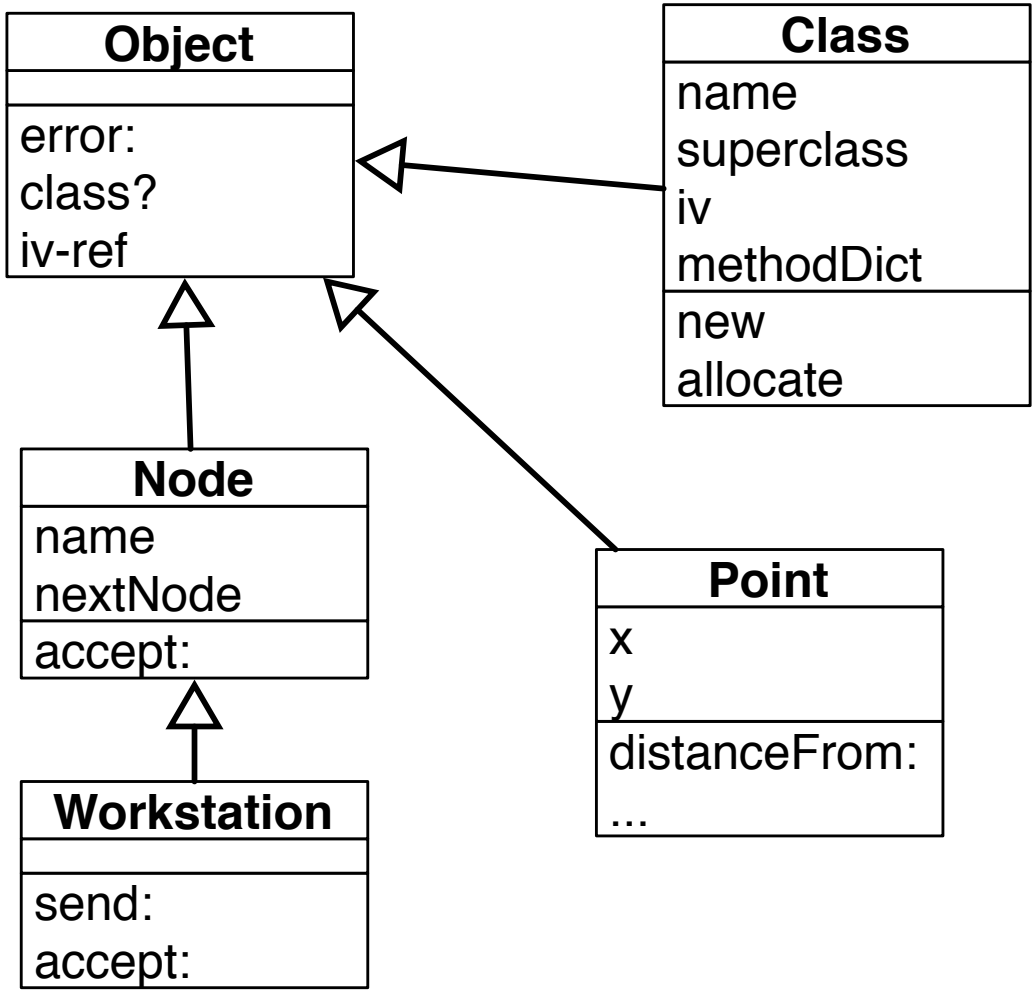


Instantiation Graph



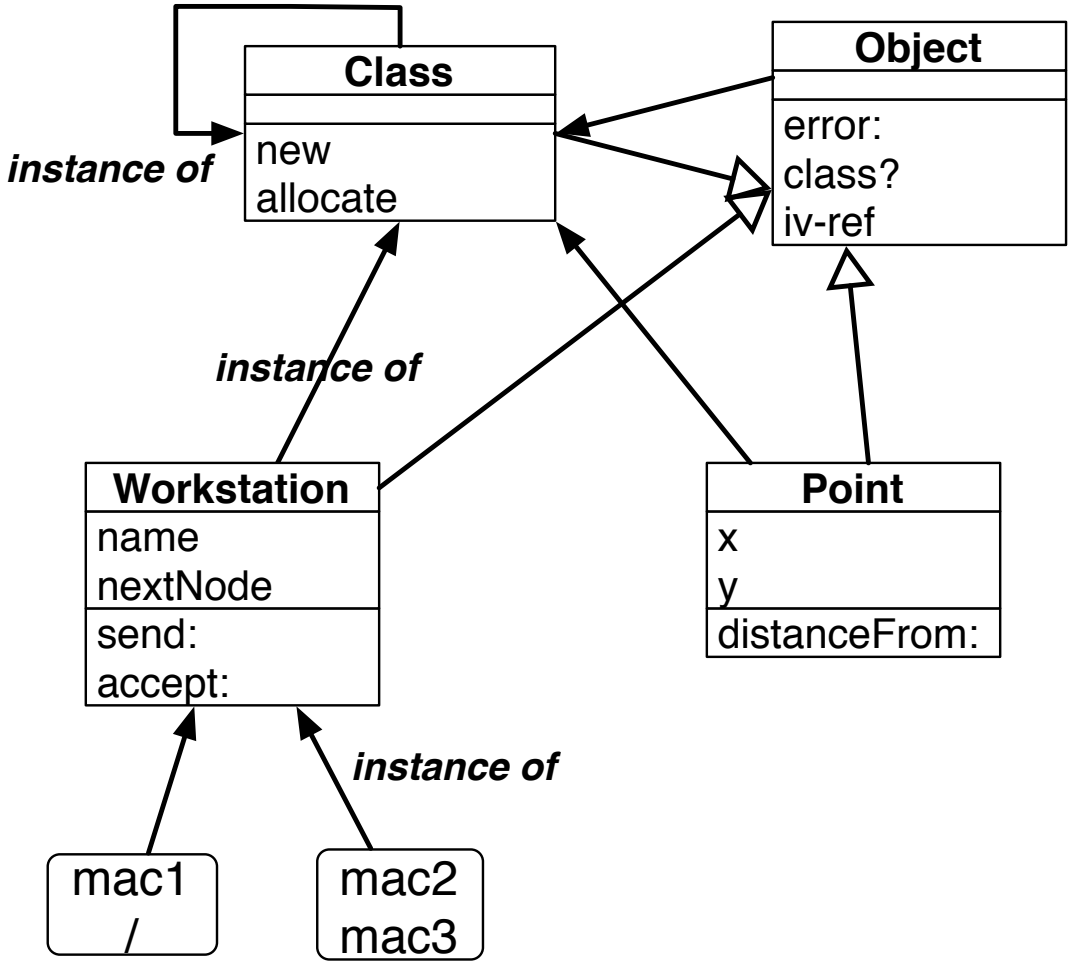
RMod

Inheritance Graph



RMod

A Simple Kernel



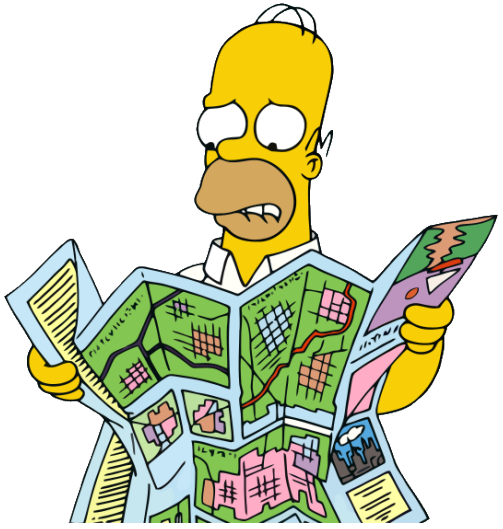
RMod

Instantiation Graph

- ***Class*** is the root of instantiation graph
- ***Object*** is a class that represents the minimal behavior of an object
- ***Object*** is a class so it is instance of ***Class***

Examples

- …
- ***One example***
- Some points
- Bootstrapping



Abstract Classes

- Prb. Abstract classes should not create instances
- Sol. Redefine the new method

Abstract Classes

- Prb. Abstract classes should not create instances
- Sol. Redefine the new method

Metaclass Use

[***Abstract*** new :name 'Node' :super 'Object' ….]

[Node new]
>>> Cannot create instance of class Node

[***Abstract*** new :name Abstract-Stack :super Object ....]

RMod

How do we create a metaclass?

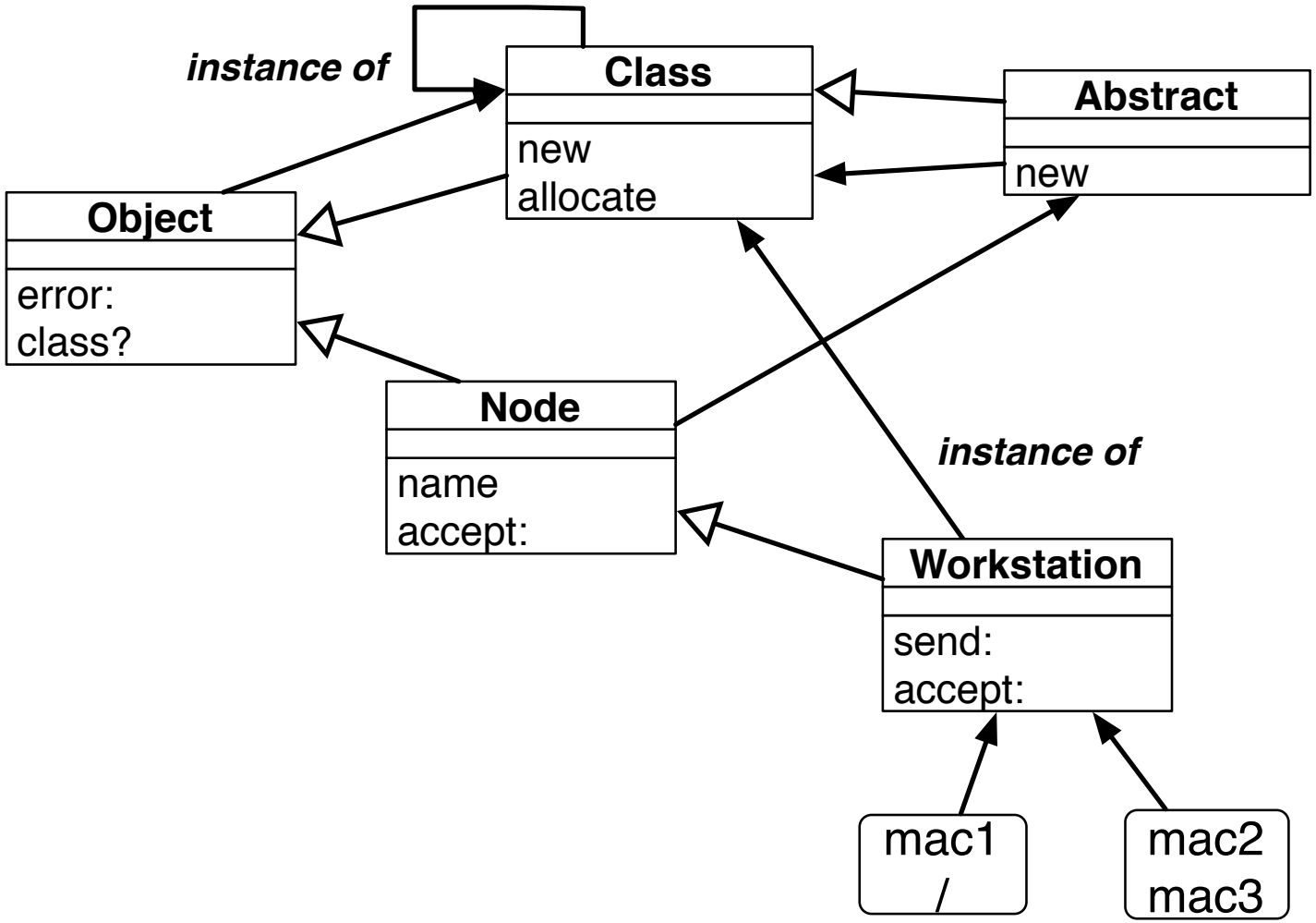
Make a class inherits from Class

Metaclass Definition

- [Class new
  :name 'Abstract'
  :super 'Class'
  :methods
    (new (lambda (self initargs)
      (self error "Cannot create instance
        of class %s" self name))]

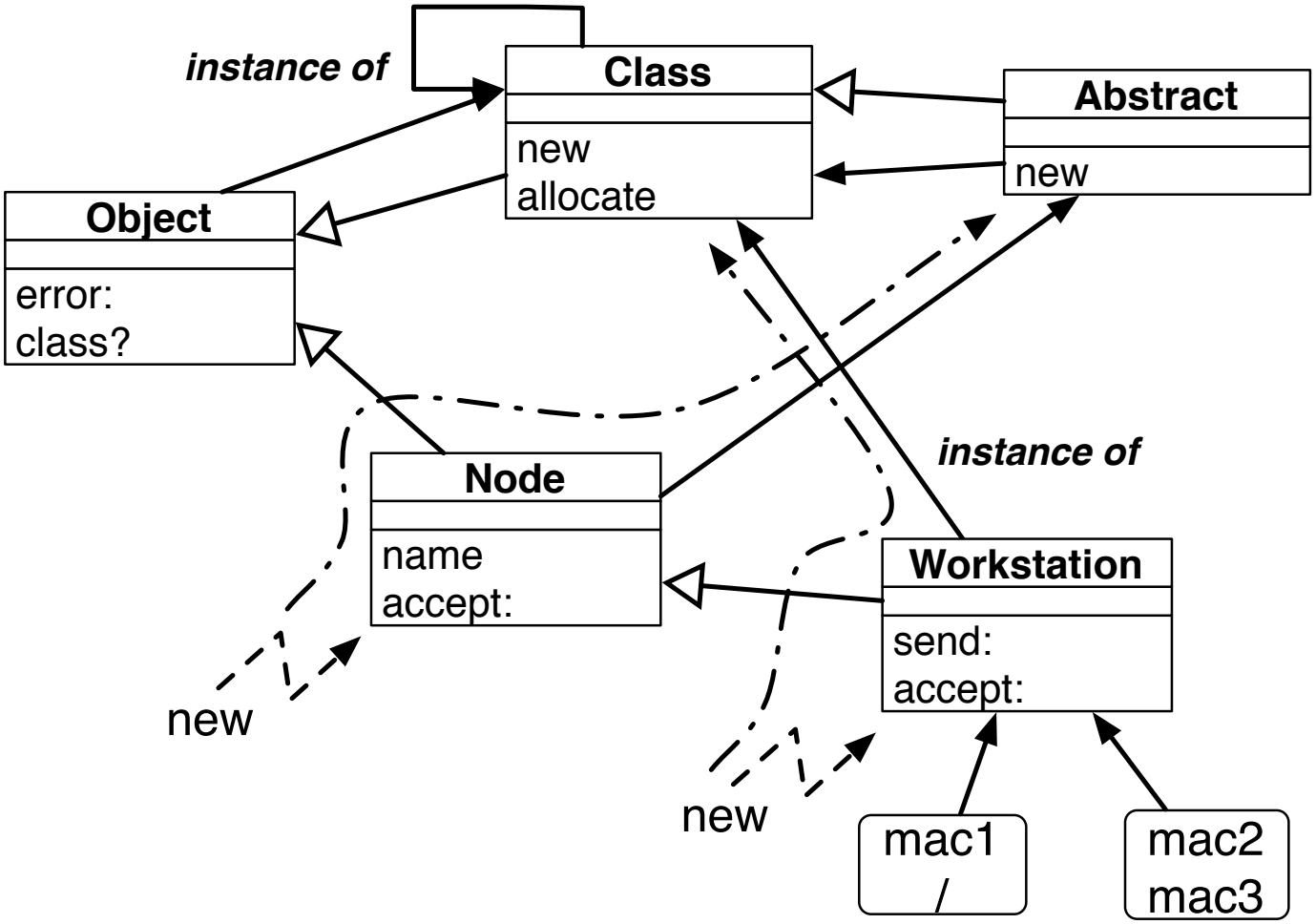
- Abstract is a class: It is instance of ***Class***

Complete Picture



RMod

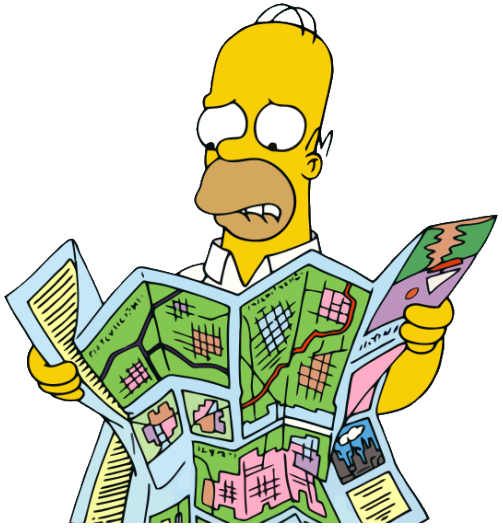
Method Lookup



RMod

RoadMap

- …
- one example
- ***Some points***
- Bootstrapping



initialization

- initialize is defined on both classes ***Class*** and ***Object***

Object initialization

Remember

```
[#(Point nil nil) initialize (:y 6 :x 24)]
=> #(Point 6 24)
```

- Two steps:
  - Extract bindings
  - Set the values

Class initialization

[Class new *:name 'Point' :super Object :i-v (x y)*...]
[#(Class nil nil nil...) initialize (*:name Point :super Object :i-v (x y)*…]

(1) a class as an object (executing initialize method)

[#(Class 'Point' Object (x y) nil #(x: (mkmethod...) y: (mkmethod ...)]

(2) inheritance of instance variables

keyword definition,
class declaration to the env
[#(Class Point Object (*class* x y) (:x :y) #(x: (...) y: (...)]

RMod

About the 6th Postulate

6th Postulate: class variable of anObject = instance variable of anObject's class

Example:

Pig color is always pink

Pig class

name super i-v ... **color**

So class variables are shared by all the instances of a class.

Why the 6th is wrong!

Semantically class variables are not instance variables of object'class!

Instance variable of metaclass should represent class information not instance information shared at the meta-level.

Metaclass information should represent classes not domain objects

Solution

A class possesses an instance variable that stores structure that represents instance ***shared-variable*** and their values.

[Class new
:name 'Pig' :super Object
:i-v (weigth name) ***:shared-var: #(color)***]

A class has the possibility to define shared variables

Bootstrapping

- Mandatory to have ***Class*** instance of itself
- Be lazy: Use as much as possible of the system to define itself
- Idea: Cheat the system so that it believes that ***Class*** already exists as instance of itself, then create ***Object*** and ***Class*** inherits from Object as normal classes

Three Steps Bootstrap

1- Manual creation of the instance that represents the class ***Class*** with

inheritance simulation (class instance variable from ***Object*** class)

only the necessary methods for the creation of the classes (new, allocate and initialize)

Creation of the class

Object [Class new :name 'Object'....]

definition of all the method of Object

Redefinition of ***Class***

[***Class*** new :name 'Class' :super ***Object***.....]

definition of all the methods of Class

RMod

Recap: Class class

- Initial metaclass
- Reflective: its instance variable values describe instance variables of any classes in the system (itself too)
- Defines the behavior of all the classes
- Inherits from Object class
- Root of the instantiation graph
- Instance variables: name, super, iv, methodDict
- Some Methods
  - new, allocate, initialize (instance variable inheritance, keywords, method compilation)
  - class?, subclass-of?

Recap: Object class

- Defines the behavior shared by all the objects of the system
- Instance of Class
- Root of the inheritance tree: all the classes inherit directly or indirectly from Object
- Its instance variable: class
- Its methods:
- initialize (initialisation les variables d'instance), error, class, metaclass?, class?, iv-set, iv-ref

References

- [Bobrow'83] D.Bobrow and M. Stefik: "The LOOPS Manual, Xerox Parc, 1983.
- [Goldberg'83] A. Goldberg and D. Robson: "Smalltalk-80: The Language", Addison-Welsey, 1983.
- [Cointe'87] P. Cointe: "Metaclasses are First Class: the ObjVlisp Model", OOPSLA'87.
- [Graube'89] N. Graube: "Metaclass compatibility", OOPSLA'89, 1989.
- [Briot'89] J.-P. Briot and P. Cointe, "Programming with Explicit Metaclasses inSmalltalk-80", OOPSLA'89.
- [Danforth'94] S. Danforth and I. Forman: "Reflection on Metaclass Programming in SOM", OOPSLA'94.
- [Rivard'96] F. Rivard, "A New Smalltalk Kernel Allowing Both Explicit and Implicit Metclass Programming" OOPSLA'96 Workshop Extending the Smalltalk Language, 1996
- [Bouraqadi'98] M.N. Bouraqadi-Saadani, T. Ledoux and F. Rivard: "Safe Metaclass Programming", OOPSLA'98

Summary

Classes are objects too

Instantiation = initialize(allocate())

Class is the instantiation root

Object is the inheritance root

One single method lookup for classes and instances

- first go to the class
- then follow inheritance chain

super and self are referring to the message receiver but super changes the method lookup

Implementation

#(#ObjPoint 10 20)

1 = classId
self offsetForClass

+.... ivs

RMod

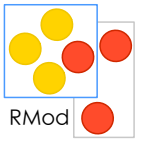
Structure of Classes

#(class name superclass ivs keys

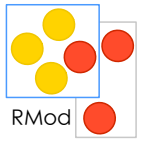
#(#ObjClass #ObjPoint #ObjObject #(class x y) ....

RMod

```
lookupMethodInClass: class
    | currentClass dictionary found |
    <inline: false>
    self assert: class ~= objectMemory nilObject.
    currentClass := class.
    [currentClass ~= objectMemory nilObject]
        whileTrue:
        [dictionary := objectMemory fetchPointer: MethodDictionaryIndex ofObject:
currentClass.
        found := self lookupMethodInDictionary: dictionary.
        found ifTrue: [^currentClass].
        currentClass := self superclassOf: currentClass].
    "Cound not find a normal message -- raise exception #doesNotUnderstand:"
    self createActualMessageTo: class.
    messageSelector := objectMemory splObj: SelectorDoesNotUnderstand.
    self sendBreak: messageSelector + BaseHeaderSize
        point: (objectMemory lengthOf: messageSelector)
        receiver: nil.
    ^self lookupMethodInClass: class
```

```
lookupMethodInClass: class
    | currentClass dictionary found |
    <inline: false>
    self assert: class ~= objectMemory nilObject.
    currentClass := class.
    [currentClass ~= objectMemory nilObject]
        whileTrue:
        [dictionary := objectMemory fetchPointer: MethodDictionaryIndex ofObject:
currentClass.
        found := self lookupMethodInDictionary: dictionary.
        found ifTrue: [^currentClass].
        currentClass := self superclassOf: currentClass].
    "Cound not find a normal message -- raise exception #doesNotUnderstand:"
    self createActualMessageTo: class.
    messageSelector := objectMemory splObj: SelectorDoesNotUnderstand.
    self sendBreak: messageSelector + BaseHeaderSize
        point: (objectMemory lengthOf: messageSelector)
        receiver: nil.
    ^self lookupMethodInClass: class
```

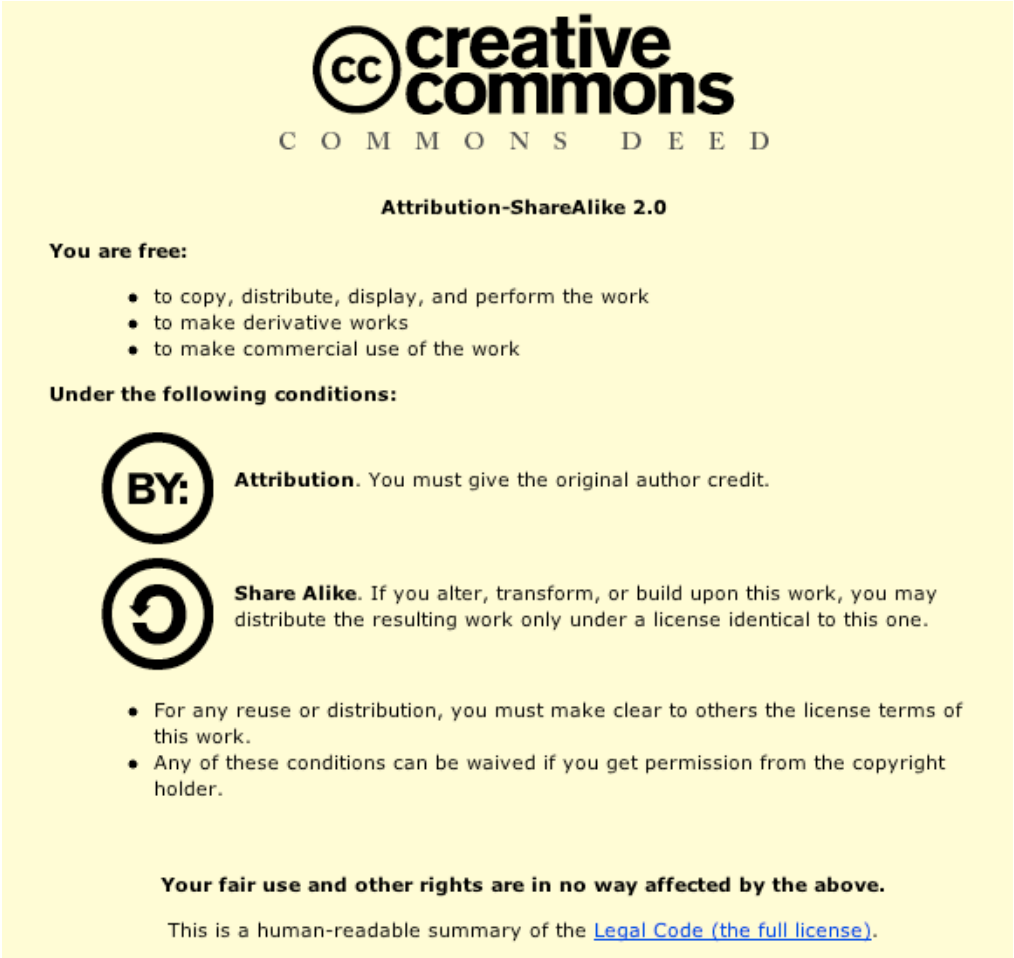
```
ookupMethodInDictionary: dictionary
    "This method lookup tolerates integers as Dictionary keys to suoport
     execution of images in which Symbols have been compacted out."
    | length index mask wrapAround nextSelector methodArray |

    length := objectMemory fetchWordLengthOf: dictionary.
            ...
    index := SelectorStart + (mask bitAnd: ((objectMemory isIntegerObject: messageSelector)
                                ifTrue: [objectMemory integerValueOf: messageSelector]
                                ifFalse: [objectMemory hashBitsOf: messageSelector])).

    "It is assumed that there are some nils in this dictionary, and search will
     stop when one is encountered. However, if there are no nils, then wrapAround
     will be detected the second time the loop gets to the end of the table."
    wrapAround := false.
    [true] whileTrue:
        [nextSelector := objectMemory fetchPointer: index ofObject: dictionary.
         nextSelector = objectMemory nilObject ifTrue: [^ false].
         nextSelector = messageSelector ifTrue:
            [methodArray := objectMemory fetchPointer: MethodArrayIndex ofObject: dictionary.
             newMethod := objectMemory fetchPointer: index - SelectorStart ofObject:
              methodArray.
            ^true].
         index := index + 1.
         index = length ifTrue:
            [wrapAround ifTrue: [^false].
             wrapAround := true.
             index := SelectorStart]].
```

License: CC-Attribution-ShareAlike 2.0

http://creativecommons.org/licenses/by-sa/2.0/



creative commons

COMMONS DEED

Attribution-ShareAlike 2.0

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

BY: **Attribution**. You must give the original author credit.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license).