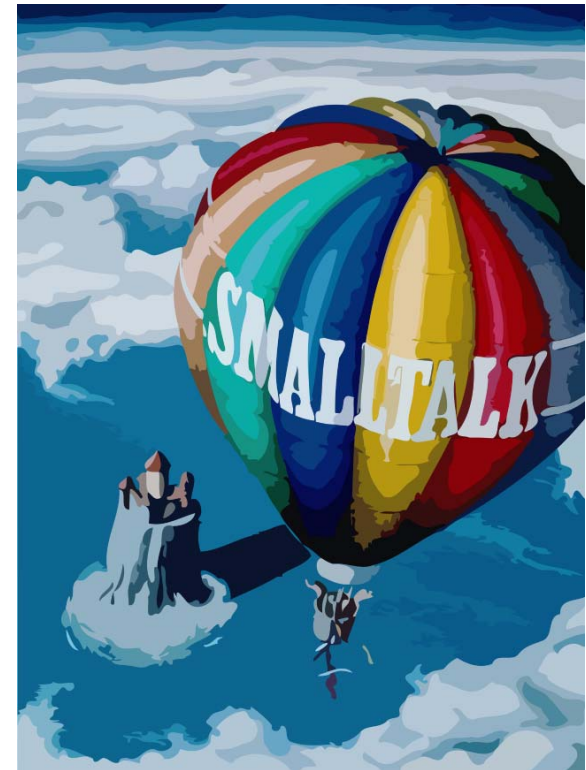


The Object-Oriented Programming Paradigm

Smalltalk

Bent Thomsen

Slides mainly based on material
by Prof. O. Nierstrasz, U. Bern and
Stephane Ducasse, INRIA



Functional programming style

```
|factorial|  
factorial :=  
  [:n |  
    (1 to: n)  
    inject: 1 into:  
    [:product :each | product * each ]].  
  
factorial value: 10
```

3628800

Perhaps not so much Functional programming style

```
inject: thisValue into: binaryBlock
    "Accumulate a running value associated with evaluating the
    binaryBlock, with the current value of the argument, the
    receiver as block arguments. For instance, to sum the numbers
    of a collection, aCollection inject: 0 into: [:subTotal value
    next]."
```

```
    | nextValue |
    nextValue := thisValue.
    self do: [:each | nextValue := binaryBlock value: nextValue
    ^nextValue
```

Functional programming style in Scheme

```
(define factorial  
  (lambda (n) (if (= n 0) 1  
                   (* n (factorial (- n 1)))))  
factorial 10
```

3628800

Functional programming style

```
|factorial|  
factorial :=  
  [:n | (n=1) ifTrue: [1]  
        ifFalse: [n * (factorial value: n-1)]  
  ].  
  
factorial value: 10
```

3628800

Procedural programming style

```
Object>>factorial: aNumber  
^aNumber = 1  
    ifTrue: [1]  
    ifFalse: [aNumber * (self factorial: aNu  
  
Object factorial: 10
```

3628800

Object Oriented programming style

```
Number>>factorial
^self = 1
  ifTrue: [self]
  ifFalse: [self * (self - 1) factorial]

10 factorial
```

3628800

Object Oriented programming style (in Pharo)

```
Integer>>factorial
  "Answer the factorial of the receiver."

  self = 0 ifTrue: [^ 1].
  self > 0 ifTrue: [^ self * (self - 1) factorial]
  self error: 'Not valid for negative integers'

10 factorial
```

3628800

Peano Arithmetic

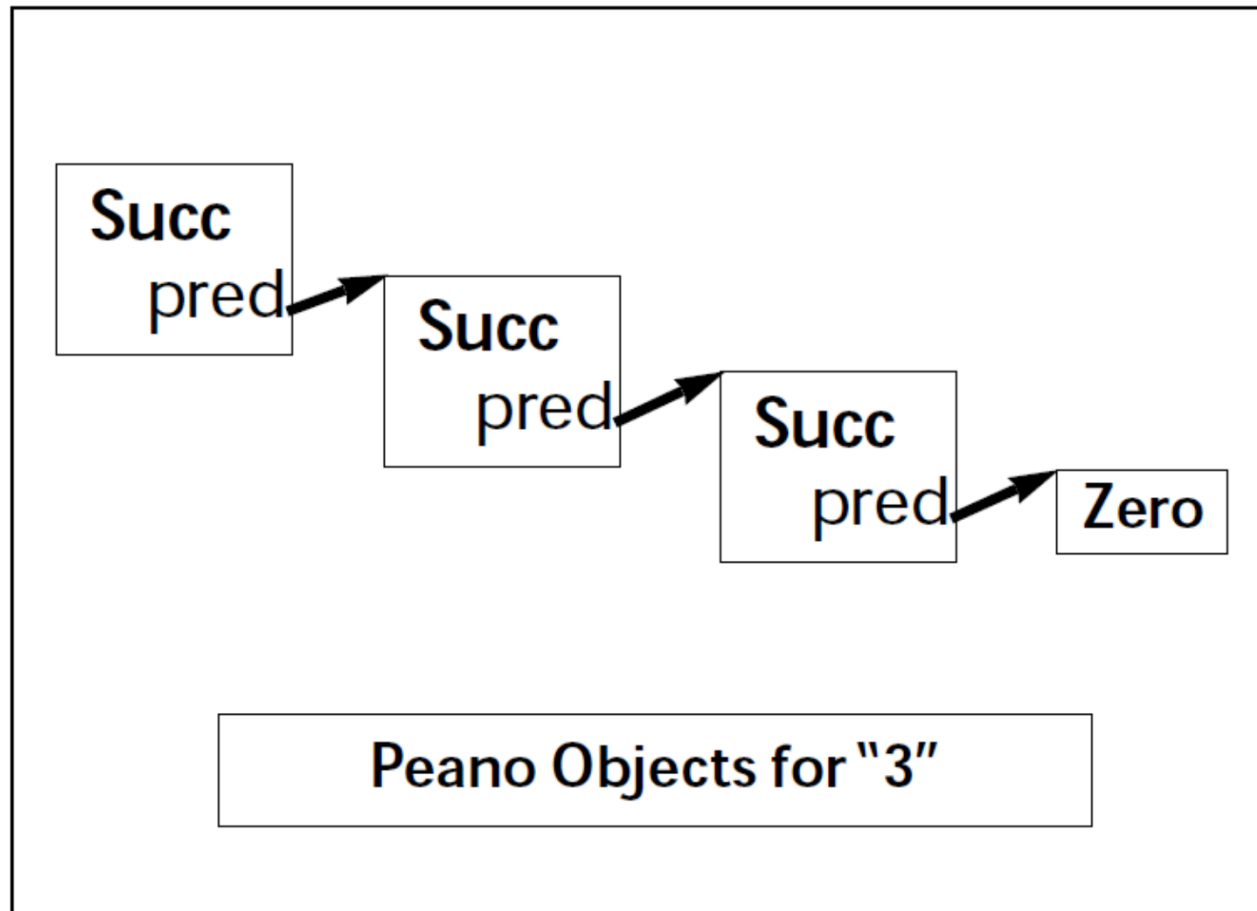
- > Zero is represented as zero
- > Other positive numbers are defined as nested invocations of the function “succ” (for “successor”)
 - E.g.: $\text{succ}(\text{succ}(\text{succ}(\text{zero})))$
- Addition (add) of Peano numbers defined through axioms
 - *Case 1: $\text{add}(X, \text{zero}) = X$*
 - E.g: $\text{add}(\text{succ}(\text{succ}(\text{succ}(\text{zero}))), \text{zero}) = \text{succ}(\text{succ}(\text{succ}(\text{zero})))$
 - i.e.: $X + 0 = X$
 - *Case 2: $\text{add}(X, \text{succ}(Y)) = \text{succ}(\text{add}(X, Y))$*
 - i.e.: $X + (1 + Y) = 1 + (X + Y)$

Adding Peano numbers

- > $\text{add}(\text{succ}(\text{zero}), \text{succ}(\text{succ}(\text{zero}))) =$
 $\text{succ}(\text{add}(\text{succ}(\text{zero}), \text{succ}(\text{zero})))$ by case 2
- > $\text{succ}(\text{add}(\text{succ}(\text{zero}), \text{succ}(\text{zero}))) =$
 $\text{succ}(\text{succ}(\text{add}(\text{succ}(\text{zero}), \text{zero})))$ by case 2
- > $\text{succ}(\text{succ}(\text{add}(\text{succ}(\text{zero}), \text{zero}))) =$
 $\text{succ}(\text{succ}(\text{succ}(\text{zero})))$ by case 1

- > Or $1 + 2 = 3!$

Peano Arithmetics as objects and messages



Peano Arithmetics as objects and messages

Class: Succ

superclass: Object

instance variables: pred

Succ class>>fromInteger: anInteger

^anInteger = 0

ifTrue: [nil]

ifFalse: [self of: (self fromInteger: anInteger - 1)]

Succ class>>of: aPeanoNumber

^self new setPred:aPeanoNumber

Succ>>setPred: aPeanoNumber

pred := aPeanoNumber

Peano Arithmetics as objects and messages

```
Succ>> pred  
  ^pred
```

```
Succ>> succ  
  ^Succ of: self
```

```
Succ>> printOn: aStream  
  aStream nextPutAll: 'succ('.  
  self pred isNil  
    ifTrue: [aStream nextPutAll: 'zero']  
    ifFalse: [self pred printOn: aStream].  
  aStream nextPutAll: ')'
```

Peano Arithmetics as objects and messages

Succ>> + aPeanoNumber

|subTotal|

subTotal:= self pred isNil

ifTrue: [aPeanoNumber]

ifFalse: [self pred+ aPeanoNumber].

^ subTotal succ

> (Succ fromInteger: 1) + (Succ fromInteger: 2)
succ(succ(succ(zero)))

Peano Arithmetics as objects and messages

- > Uses objects and messages
- > But not very object oriented!
- > Look at
 - Succ class>>fromInteger: anInteger
 - Succ>> + aPeanoNumber

A more Object Oriented Implementation of Peano Numbers

Class: Zero

superclass: Object

instance variables: <none>

Succ class>>fromInteger:anInteger

^anInteger=0

ifTrue: [Zero new]

ifFalse: [self of: (self fromInteger: anInteger - 1)]

Use Zero class instead of nil

```
Zero>>printOn: aStream  
aStream nextPutAll: 'zero'
```

```
Succ>>printOn: aStream  
aStream nextPutAll: 'succ(  
self pred printOn: aStream.  
aStream nextPutAll: ')
```

Operations in two classes

Zero>> + aPeanoNumber

^aPeanoNumber

Succ>> + aPeanoNumber

^(self pred + aPeanoNumber) succ

Operations in two classes

- > Case 1: $\text{add}(\text{zero}, X) = X$
- > Case 2: $\text{add}(\text{succ}(X), Y) = Y = \text{succ}(\text{add}(X, Y))$

Zero>>+ aPeanoNumber
^aPeanoNumber

Succ>>+ aPeanoNumber
^(self pred + aPeanoNumber) succ

Factorial on Peano Integers

Zero>>factorial
 ^self succ

Succ>>factorial
 ^self * (self pred) factorial

(You need to define * in terms of succ, pred and +)

Object Oriented Recursion

“object-oriented
recursion represents the
invocations themselves as
objects, sending the same
message to different
objects.”

Kent Beck

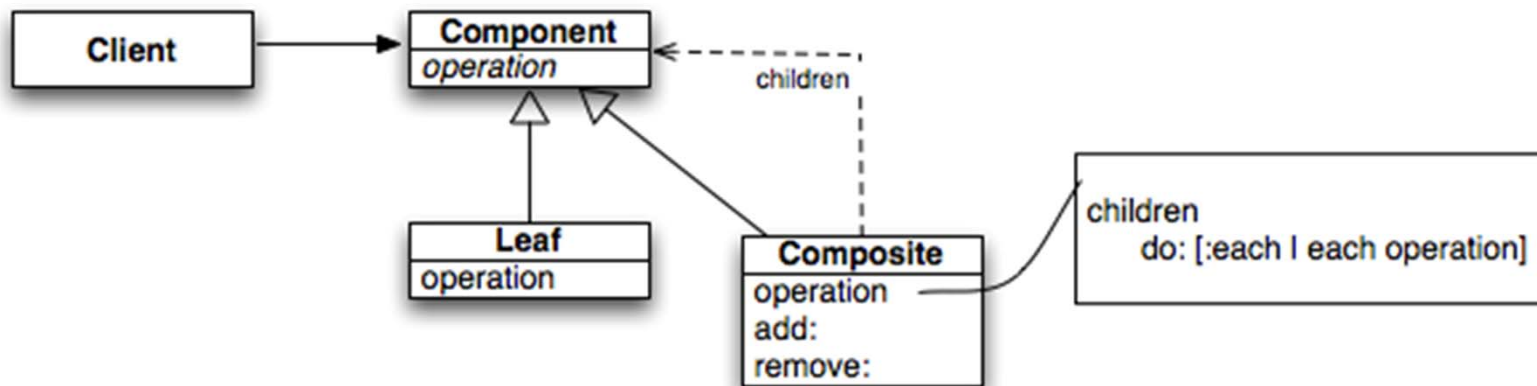
A more Obejct Oriented Implementation of Peano Numbers

Note we could have used nil instead of an instance of our own Zero class, e.g by adding + to UndefinedObject class, however that would clutter up the UndefinedObject class shared by everybody

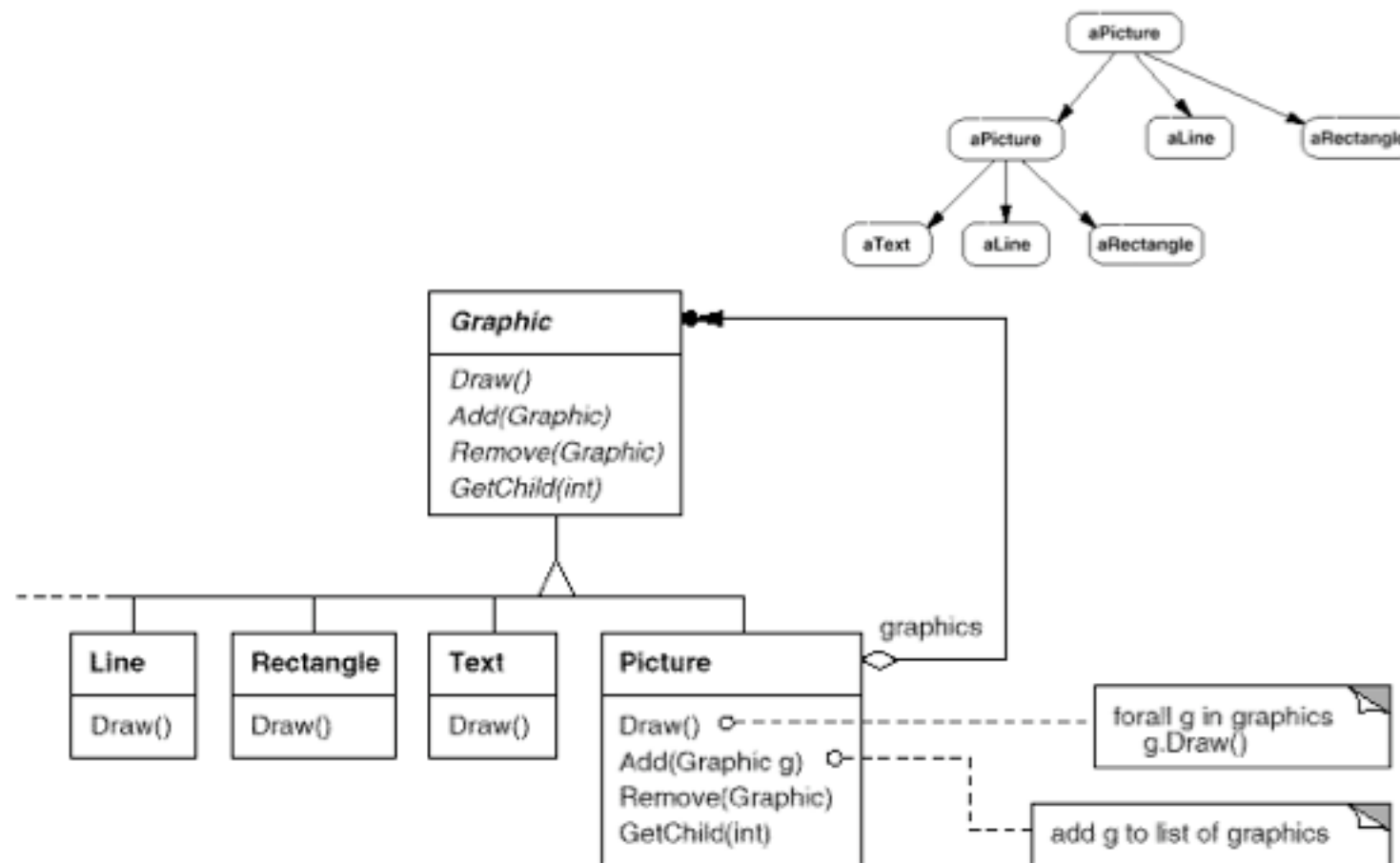
Object Oriented Recursion in use

Composite Pattern

- > Compose objects into tree structures to represent part-whole hierarchies.
- > Composite lets ***clients*** treat individual objects and compositions of objects ***uniformly***



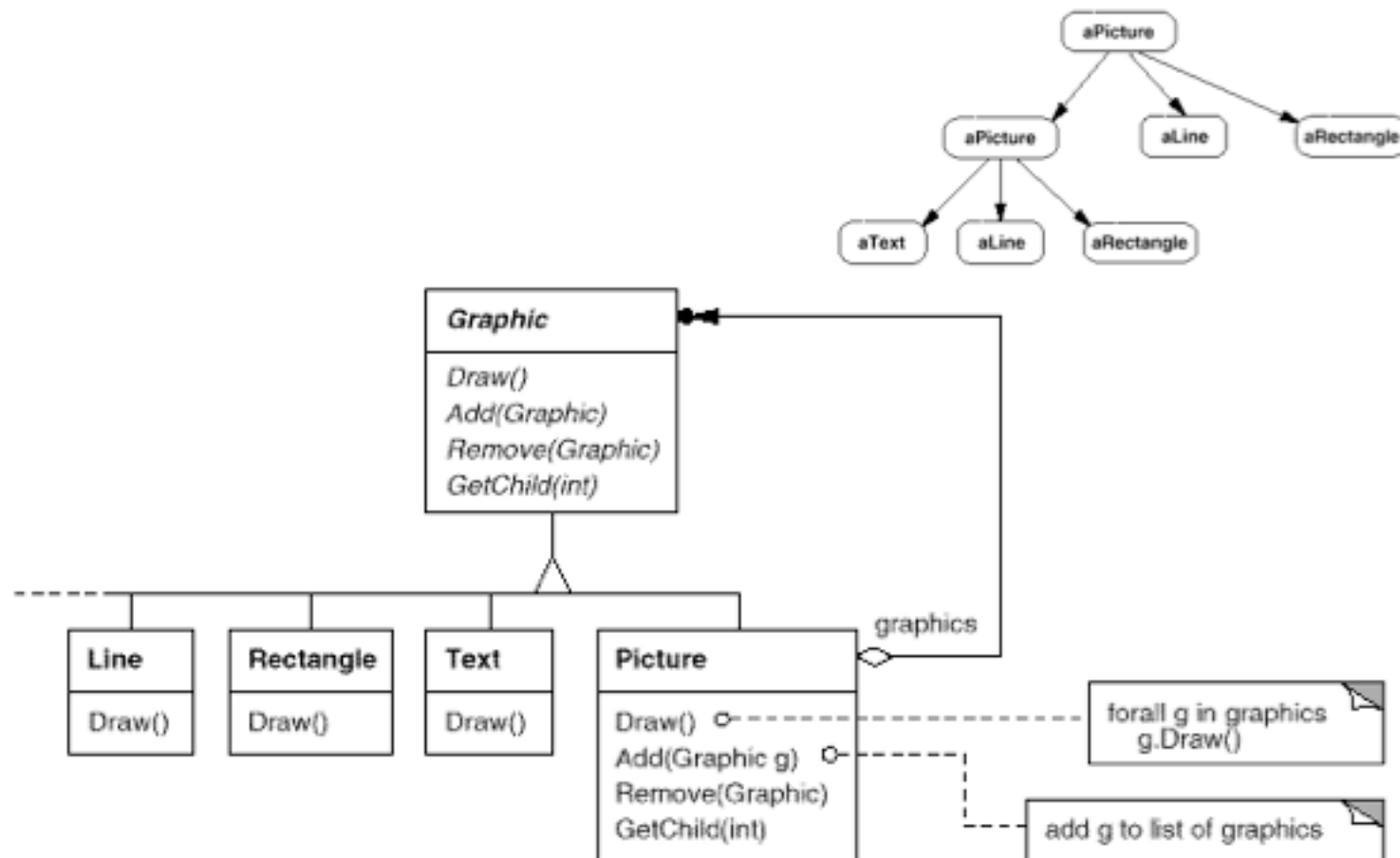
Composite Pattern Motivation



Composite Pattern Applicability

- > Use the Composite Pattern when :
 - you want to represent part-whole hierarchies of objects
 - you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly

Composite Pattern Possible Design



Composite Pattern Participants

- > **Component (Graphic)**
 - declares the interface for objects in the composition
 - implements default behavior for the interface common to all classes, as appropriate
 - declares an interface for accessing and managing its child components
- > **Leaf (Rectangle, Line, Text, ...)**
 - represents leaf objects in the composition. A leaf has no children
 - defines behavior for primitive objects in the composition

Composite Pattern

- > Composite (Picture)
 - defines behaviour for components having children
 - stores child components
 - implements child-related operations in the Component interface
- > Client
 - manipulates objects in the composition through the Component interface

Composite Pattern

Collaborations

- > Clients use the Component class interface to interact with objects in the composite structure.
- > Leaves handle requests directly.
- > Composites forward requests to its child components
- > Consequences
 - defines class hierarchies consisting of primitive and composite objects.
 - Makes the client simple. Composite and primitive objects are treated uniformly. (no cases)
 - Eases the creation of new kinds of components
 - Can make your design overly general

In Smalltalk

- Composite not only groups leaves but can also contain composites
- In Smalltalk add:, remove: do not need to be declared into Component but only on Composite. This way we avoid to have to define dummy behavior for Leaf

Composite Variations

- Use a Component superclass to define the interface and factor code there.
- Consider implementing abstract Composite and Leaf (in case of complex hierarchy)
- Only Composite delegates to children
- Composites can be nested
- Composite sets the parent back-pointer (add:/remove:)

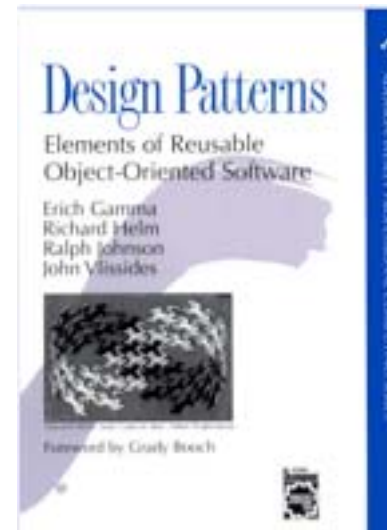
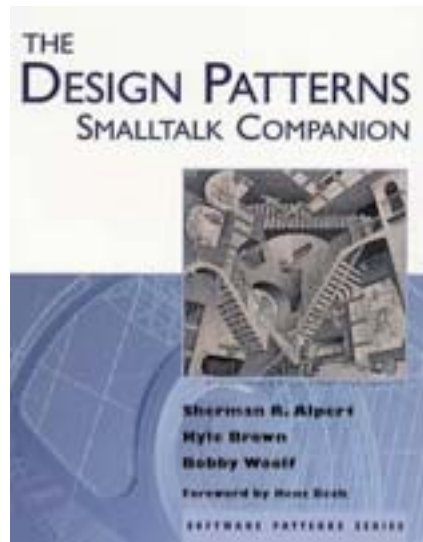
Composite Variations

- Can Composite contain any type of child? (domain issues)
- Is the Composite's number of children limited?
- Forward
 - Simple forward. Send the message to all the children and merge the results without performing any other behavior
 - Selective forward. Conditionally forward to some children
 - Extended forward. Extra behavior
 - Override. Instead of delegating

Other Patterns

- > Composite and Visitors
 - Visitors walks on structured objects
- > Composite and Factories
 - Factories can create composite elements

Many more patterns



12 of 12 people found the following review helpful:

★★★★★ **Easier to understand than the original GoF**, February 4, 2000

Reviewer: [Nicolas Weidmann](#) (Zurich, Switzerland) - [See all my reviews](#)

This book gives you a better understanding of the patterns than in its original version (the GoF one). I am not a SmallTalk programmer but a 9 years C++ one. At work I had to use the GoF book and never liked reading it. In contrast to this, the SmallTalk companion is easy to read and you can understand the patterns within the first few lines of their description. Take the Bridge pattern and compare their discussions in the two books. If you really like the Gof one then buy it. But according to me, it would be a big mistake buying the GoF in favour of the SmallTalk companion. Trust a C++ programmer :-)

Was this review helpful to you? ([Report this](#))

Object Oriented Philosophy

- > Don't do something you can ask somebody else to do !
- > Normally means: delegate to other objects
- > But also
- > use inheritance, patterns, libraries and frameworks before you write your own code!

- > E.g.
 - you may need a collection of objects in your application
 - *look in the collection library to see if there is already a suitable collection implementation*
 - You may need to traverse your collection
 - *Look in the collection library to find a suitable method*
 - *If you don't find one*
 - Look for inspiration in the collection library to implement your method

Implementation of do:

do: aBlock

"Refer to the comment in Collection|do:."

1 to: self size do:

[:index | aBlock value: (self at: index)]

Implementation of to:do:

to: stop do: aBlock

"Normally compiled in-line, and therefore not overridable.

Evaluate aBlock for each element of the interval (self to: stop by: 1)."

| nextValue |

nextValue := self.

[nextValue <= stop]

whileTrue:

[aBlock value: nextValue.

nextValue := nextValue + 1]

Implementation of WhileTrue (In Pharo)

whileTrue: aBlock

"Ordinarily compiled in-line, and therefore not overridable.

This is in case the message is sent to other than a literal block.

Evaluate the argument, aBlock, as long as the value of the receiver is true."

^ [self value] whileTrue: [aBlock value]

Implementation of While True (in VisualWorks)

whileTrue: aBlock

^self value

ifTrue:

[aBlock value.

[self value] whileTrue: [aBlock value]]

or

```
whileTrue:aBlock
```

```
self value ifFalse:[^ nil].
```

```
aBlock value.
```

```
thisContext restart
```

Remember thisContext is a pseudo-variable (like self and super)

Whenever thisContext is referred to in a running method, the entire run-time context of that method is reified and made available to the image as a series of chained MethodContext objects.

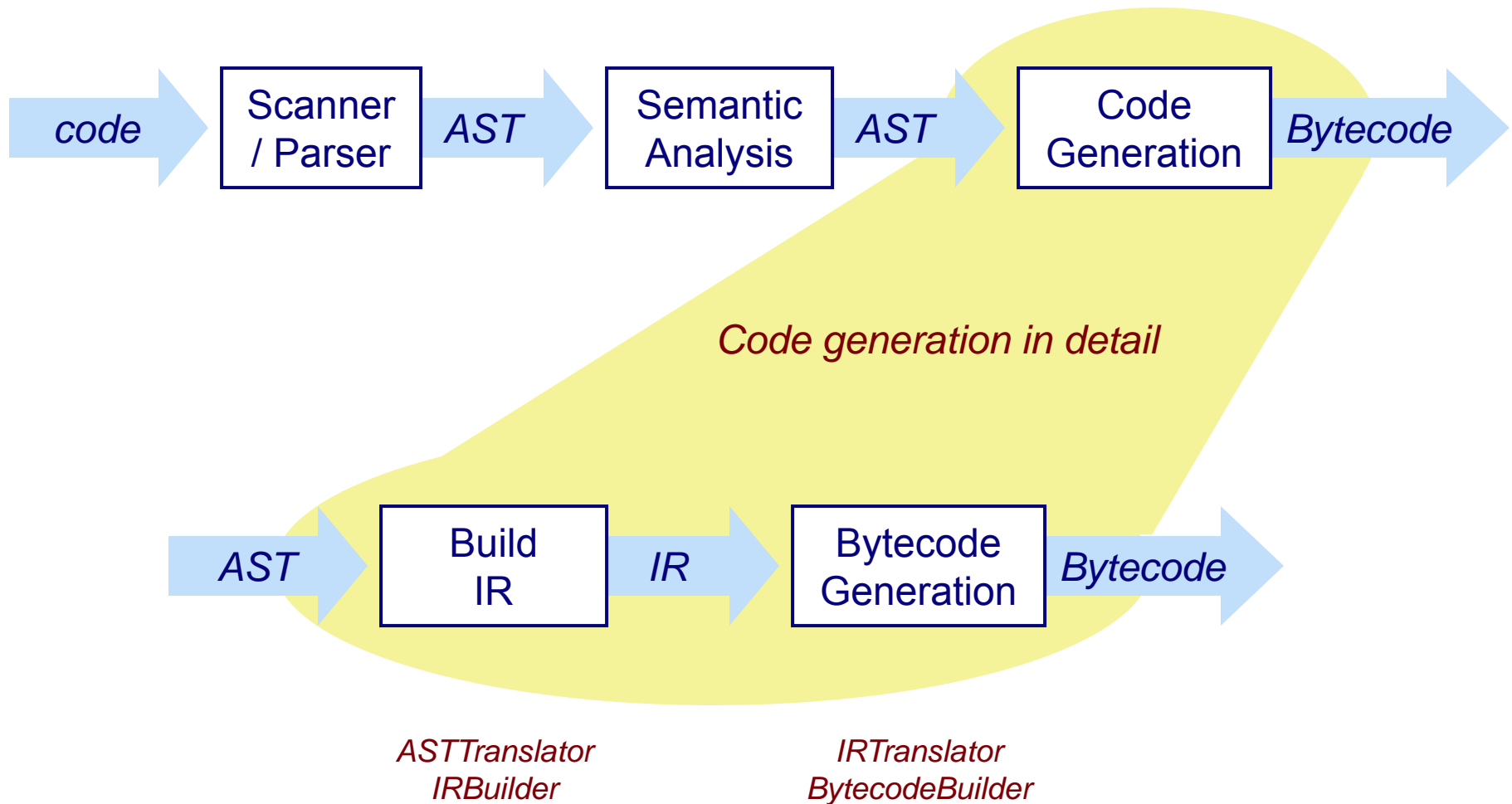
Many messages are directly interpreted

- > Other messages that may be directly interpreted by the
- > VM, depending on the receiver, include:
 - `+- < > <= >= == * / == @ bitShift: // bitAnd: bitOr: at: at:put: size next nextPut: atEnd blockCopy: value value: do: new new: x y.`
- > Selectors that are never sent, because they are inlined by the compiler and transformed to comparison and jump bytecodes:
 - `ifTrue: ifFalse: ifTrue:ifFalse: ifFalse:ifTrue: and: or: whileFalse: whileTrue: whileFalse whileTrue to:do: to:by:do: caseOf: caseOf:otherwise: ifNil: ifNotNil: ifNil:ifNotNil: ifNotNil:ifNil:`

The Pharo Compiler

- > Fully reified compilation process:
 - Scanner/Parser (built with SmaCC)
 - *builds AST (from Refactoring Browser)*
 - Semantic Analysis: ASTChecker
 - *annotates the AST (e.g., var bindings)*
 - Translation to IR: ASTTranslator
 - *uses IRBuilder to build IR (Intermediate Representation)*
 - Bytecode generation: IRTranslator
 - *uses BytecodeBuilder to emit bytecodes*

Compiler: Overview



The Pharo Virtual Machine

- > Virtual machine provides a virtual processor
 - Bytecode: The “machine-code” of the virtual machine
- > Smalltalk (like Java): Stack machine
 - easy to implement interpreters for different processors
 - most hardware processors are register machines
- > Pharo VM: Implemented in *Slang*
 - Slang: Subset of Smalltalk. (“C with Smalltalk Syntax”)
 - Translated to C

Example: Number>>asInteger

> Smalltalk code:

```
Number>>asInteger  
    "Answer an Integer nearest  
    the receiver toward zero."  
  
    ^self truncated
```

> Symbolic Bytecode

```
9 <70> self  
10 <D0> send: truncated  
11 <7C> returnTop
```

Example: Step by Step

- > 9 <70> self
 - The receiver (self) is pushed on the stack
- > 10 <D0> send: truncated
 - Bytecode 208: send literal selector 1
 - Get the selector from the first literal
 - start message lookup in the class of the object that is on top of the stack
 - result is pushed on the stack
- > 11 <7C> returnTop
 - return the object on top of the stack to the calling method

Jump Bytecodes

- > Control Flow inside one method
 - Used to implement control-flow efficiently
 - Example:

```
^ 1<2 ifTrue: ['true']
```

```
9 <76> pushConstant: 1
10 <77> pushConstant: 2
11 <B2> send: <
12 <99> jumpFalse: 15
13 <20> pushConstant: 'true'
14 <90> jumpTo: 16
15 <73> pushConstant: nil
16 <7C> returnTop
```

Generating Bytecode

- > IRBuilder: A tool for generating bytecode
 - Part of the NewCompiler
 - Pharo: Install packages AST, NewParser, NewCompiler
- > Like an Assembler for Pharo

IRBuilder: Simple Example

> *Number*>>asInteger

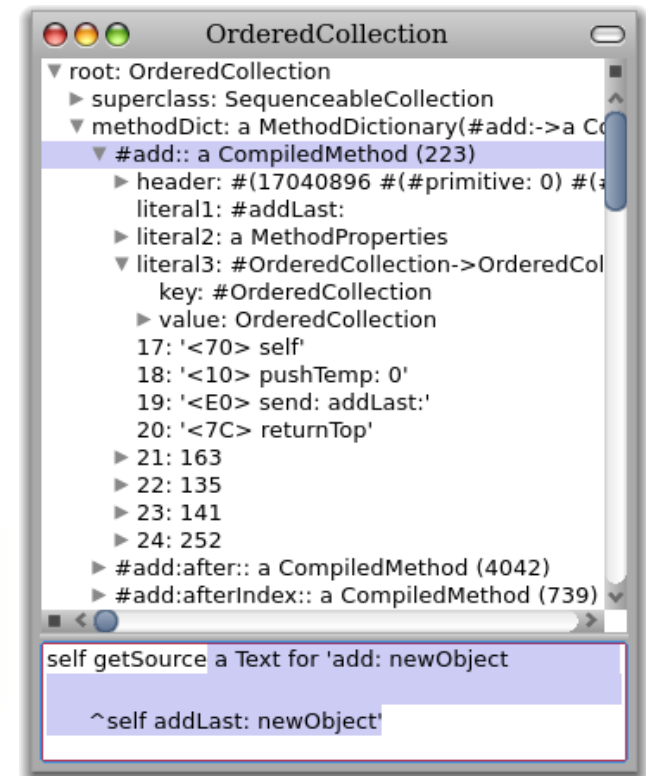
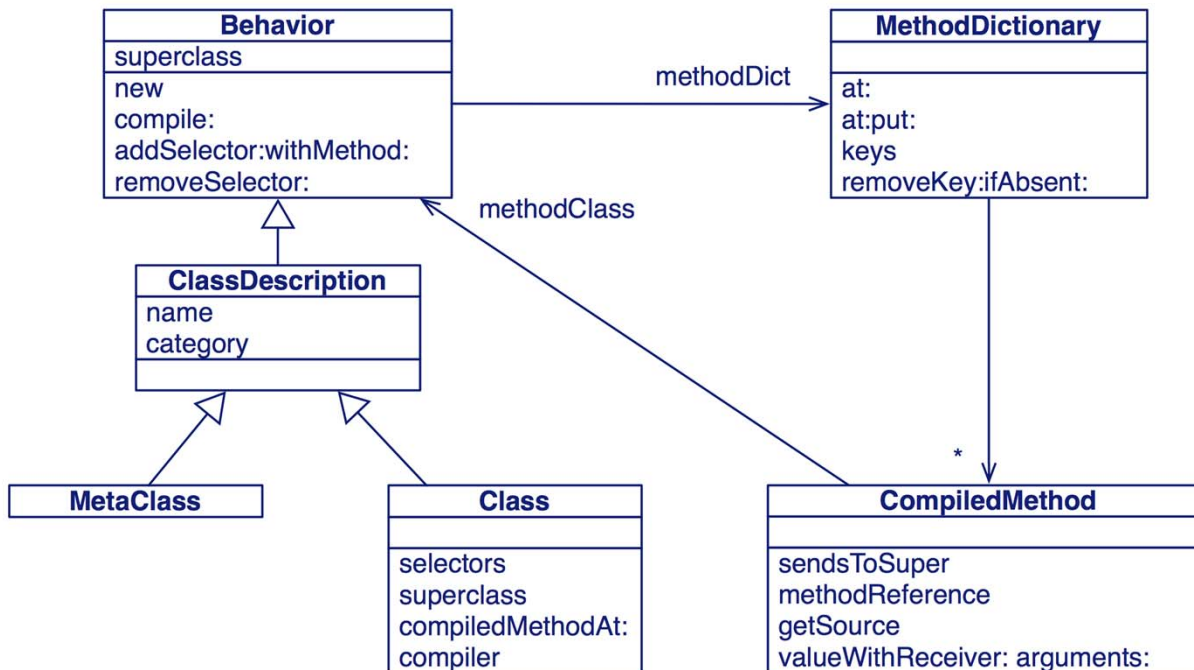
```
iRMethod := IRBuilder new
  numRargs: 1;           "receiver"
  addTemps: #(self);    "receiver and args"
  pushTemp: #self;
  send: #truncated;
  returnTop;
  ir.

aCompiledMethod := iRMethod compiledMethod.

aCompiledMethod valueWithReceiver:3.5
                    arguments: #()
```

3

Classes are Holders of CompiledMethods



Invoking a message by its name

```
Object>>perform: aSymbol  
Object>>perform: aSymbol with: arg
```

- > Asks an object to execute a message
 - Normal method lookup is performed

```
5 factorial
```

```
120
```

```
5 perform: #factorial
```

```
120
```

Executing a compiled method

```
CompiledMethod>>valueWithReceiver:arguments:
```

> No lookup is performed!

```
(SmallInteger>>#factorial)  
valueWithReceiver: 5  
arguments: #()
```

```
Error: key not found
```

```
(Integer>>#factorial)  
valueWithReceiver: 5  
arguments: #()
```

```
120
```

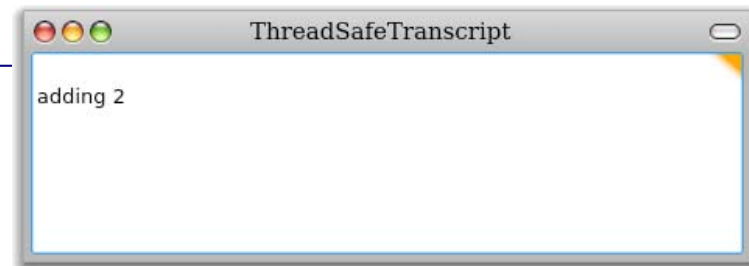
What happens when a method is executed?

- > We need space for:
 - The temporary variables
 - Remembering where to return to
- > Everything is an Object!
 - So: we model this space with objects
 - Class `MethodContext`

```
ContextPart variableSubclass: #MethodContext  
  instanceVariableNames: 'method closureOrNil receiver'  
  classVariableNames: ''  
  poolDictionaries: ''  
  category: 'Kernel-Methods'
```

Anonymous class in Pharo

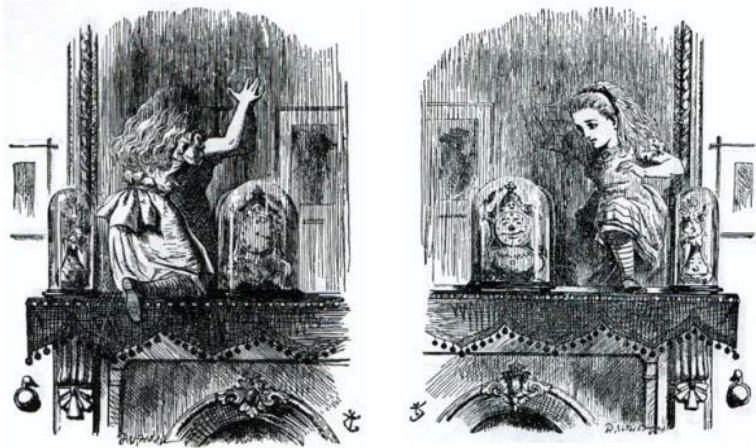
```
| anonClass set |  
anonClass := Behavior new.  
anonClass superclass: Set;  
    setFormat: Set format.  
  
anonClass compile:  
    'add: anObject  
      Transcript show: ''adding '', anObject printString; cr.  
      ^ super add: anObject'.  
  
set := Set new.  
set add: 1.  
  
set primitiveChangeClassTo: anonClass basicNew.  
set add: 2.
```



Reflection



Birds-eye view



Reflection allows you to both *examine* and *alter* the meta-objects of a system.

Using reflection to modify a running system requires some care.



Why we need reflection

As a programming language becomes *higher and higher level*, its implementation in terms of underlying machine involves *more and more tradeoffs*, on the part of the implementor, about what cases to optimize at the expense of what other cases. ... the *ability to cleanly integrate* something outside of the language's scope *becomes more and more limited*

Kiczales, in Paepcke 1993

What is are Reflection and Reification?

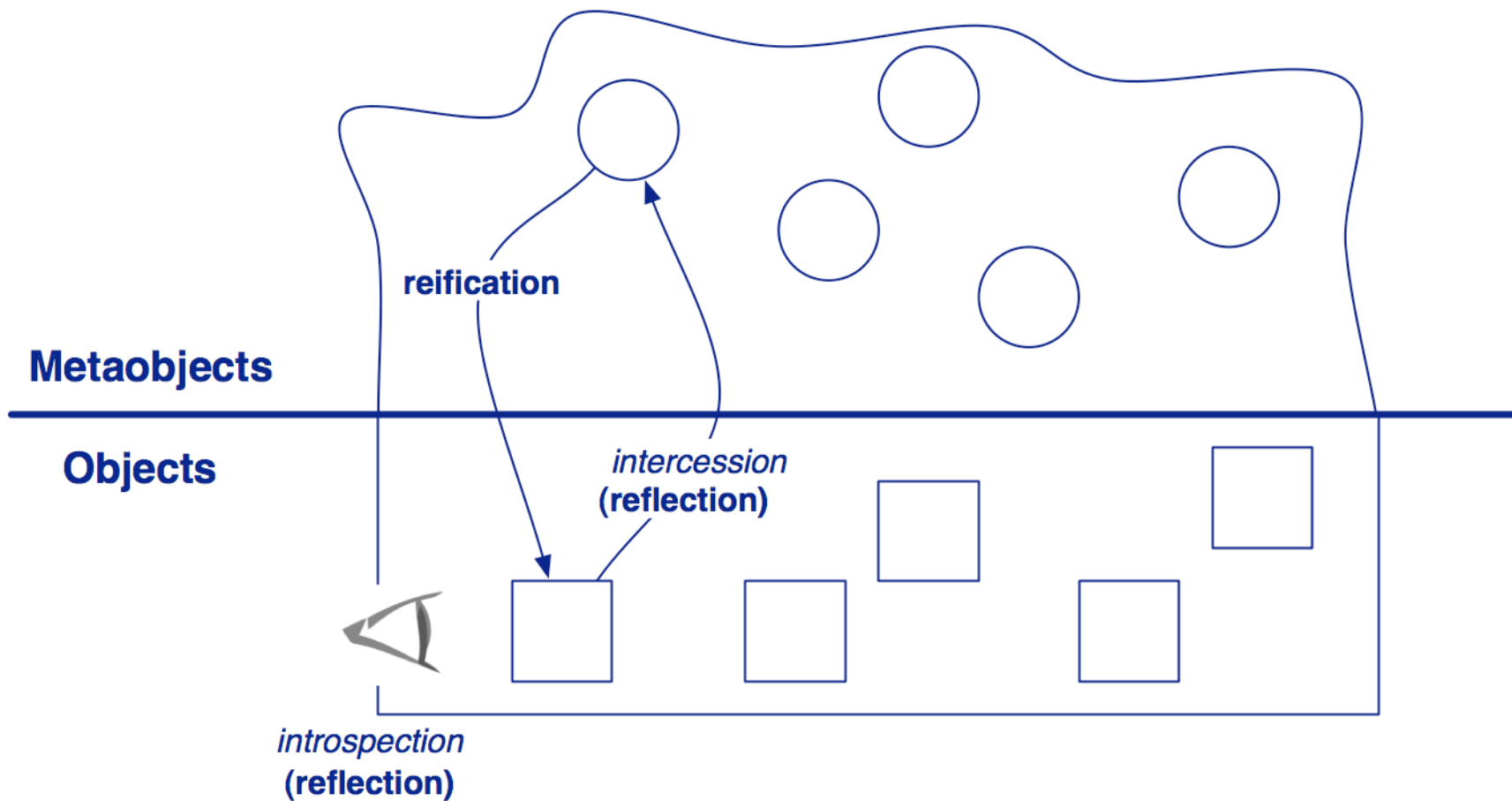
> Reflection is the ability of a program to *manipulate as data* something representing the *state of the program* during its own execution.

- Introspection is the ability for a program to *observe* and therefore *reason* about its own state.
- Intercession is the ability for a program to *modify* its own execution state or *alter its own interpretation* or meaning.

> Reification is the mechanism for encoding execution state as data

— Bobrow, Gabriel & White, 1993

Reflection and Reification

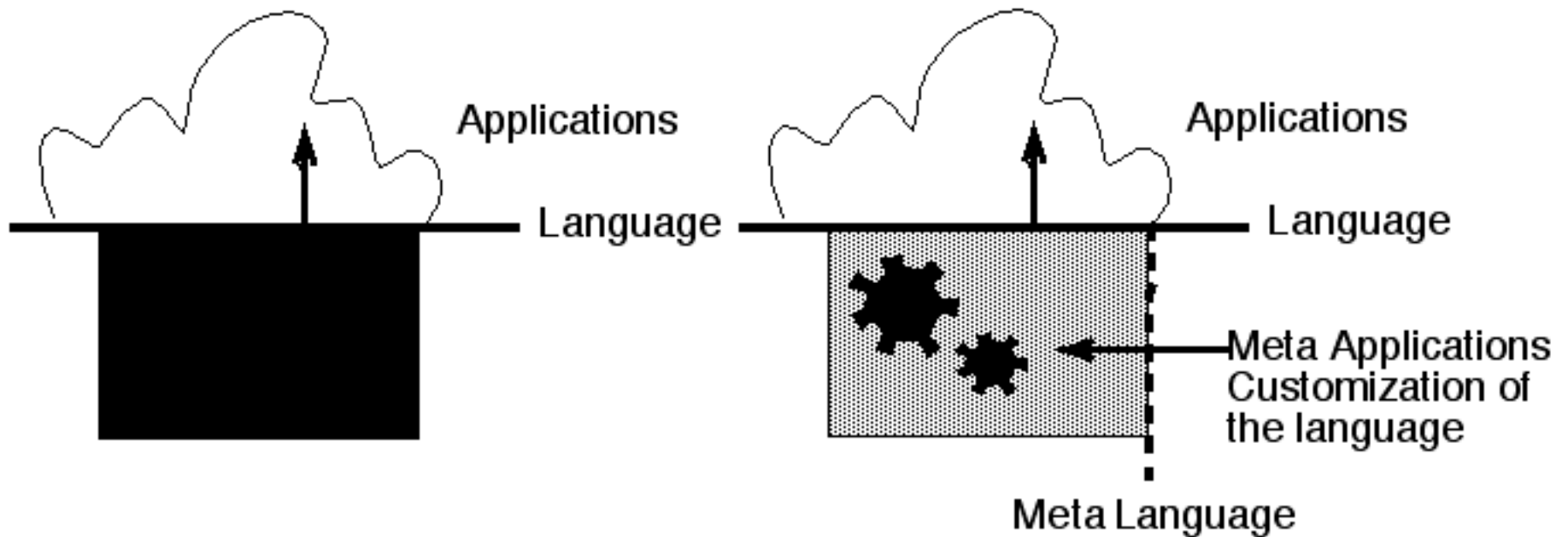


Consequences

- > “A system having itself as application domain and that is *causally connected* with this domain can be qualified as a reflective system”
 - Maes, OOPSLA 1987
- A reflective system has an *internal representation of itself*.
- A reflective system is able to *act on itself* with the ensurance that its representation will be causally connected (up to date).
- A reflective system has some static capacity of *self-representation* and dynamic *self-modification* in constant synchronization

Metaprogramming in Programming Languages

- > The meta-language and the language can be different:
 - Scheme and an OO language
- > The meta-language and the language can be same:
 - Smalltalk, CLOS
 - In such a case this is a *metacircular architecture*



Structural and behavioral reflection

- > Structural reflection is concerned with the ability of the language to provide a complete *reification* of both
 - the *program* currently executed
 - as well as its *abstract data types*.
- > Behavioral reflection is concerned with the ability of the language to provide a complete reification of
 - its own *semantics* and *implementation* (processor)
 - as well as the data and implementation of the *run-time system*.

Malenfant et al., *A Tutorial on Behavioral Reflection and its Implementation*, 1996

The Essence of a Class

1. A format
 - I.e., a number of instance variables and types
2. A superclass
3. A method dictionary

Behavior class>> new

> In Pharo:

```
Behavior class>>new
| classInstance |
classInstance := self basicNew.
classInstance methodDictionary:
    classInstance emptyMethodDictionary.
classInstance superclass: Object.
classInstance setFormat: Object format.
^ classInstance
```

NB: not to be confused with Behavior>>new!

The Essence of an Object

1. Class pointer
 2. Values
- > Can be special:
- `SmallInteger`
 - Indexed rather than pointer values
 - Compact classes (`CompiledMethod`, `Array ...`)

Metaobjects vs metaclasses

- > Need distinction between metaclass and metaobject!
 - A metaclass is a class whose instances are classes
 - A metaobject is an object that describes or manipulates other objects
 - *Different metaobjects can control different aspects of objects*

Some MetaObjects

- > **Structure:**
 - Behavior, ClassDescription, Class, Metaclass, ClassBuilder
- > **Semantics:**
 - Compiler, Decompiler, IRBuilder
- > **Behavior:**
 - CompiledMethod, BlockContext, Message, Exception
- > **ControlState:**
 - BlockContext, Process, ProcessorScheduler
- > **Resources:**
 - WeakArray
- > **Naming:**
 - SystemDictionary
- > **Libraries:**
 - MethodDictionary, ClassOrganizer

Meta-Operations

“Meta-operations are operations that provide information about an object as opposed to information directly contained by the object ...They permit things to be done that are not normally possible”

Inside Smalltalk

Accessing state

- > *Object*>>instVarNamed: aString
- > *Object*>>instVarNamed: aString put: anObject
- > *Object*>>instVarAt: aNumber
- > *Object*>>instVarAt: aNumber put: anObject

```
pt := 10@3.  
pt instVarNamed: 'x'.  
pt instVarNamed: 'x' put: 33.  
pt
```

```
10  
  
33@3
```

Accessing meta-information

- > *Object*>>class
- > *Object*>>identityHash

```
'hello' class  
(10@3) class  
Smalltalk class  
Class class  
Class class class  
Class class class class
```

```
'hello' identityHash  
Object identityHash  
5 identityHash
```

```
ByteString  
Point  
SystemDictionary  
Class class  
Metaclass  
Metaclass class
```

```
2664  
2274  
5
```

Introspection

Ex: Code metrics

Collection allSuperclasses size.	2
Collection allSelectors size.	610
Collection allInstVarNames size.	0
Collection selectors size.	163
Collection instVarNames size.	0
Collection subclasses size.	9
Collection allSubclasses size.	101
Collection linesOfCode.	864

Changes

- > `Object>>become: anotherObject`
 - Swap the object pointers of the receiver and the argument.
 - All variables in the entire system that used to point to the receiver now point to the argument, and vice-versa.
 - Fails if either object is a `SmallInteger`

- > `Object>>primitiveChangeClassTo: anObject`
 - both classes should have the same format, *i.e.*, the same physical structure of their instances
 - *“Not for casual use”*

become:

- > Swap all the pointers from one object to the other and back (symmetric)

```
ReflectionTest>>testBecome
| pt1 pt2 pt3 |

pt1 := 0@0.
pt2 := pt1.
pt3 := 100@100.
pt1 become: pt3.

self assert: pt1 = (100@100).
self assert: pt1 == pt2.
self assert: pt3 = (0@0).
```

Making factorial more Object Oriented

- > SmallInteger subclass: #Zero
- > 0 primitiveChangeClassTo: Zero
- > Zero>>factorial
- > ^1
- > SmallInteger>>factorial
- > ^self * (self - 1) factorial
- > Note: This is the principle, but it will crash Pharo !!!

Implementing Instance Specific Methods

```
ReflectionTest>>testPrimitiveChangeClassTo
| behavior browser |

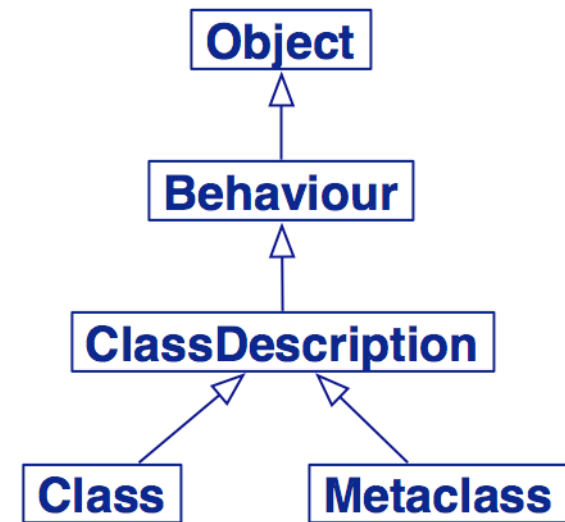
behavior := Behavior new. "an anonymous class"
behavior superclass: Browser.
behavior setFormat: Browser format.
browser := Browser new.

browser primitiveChangeClassTo: behavior new.
behavior compile: 'thisIsATest ^ 2'.

self assert: browser thisIsATest = 2.
self should: [Browser new thisIsATest]
raise: MessageNotUnderstood.
```

Classes are objects too

- > **Object**
 - Root of inheritance
 - Default Behavior
 - Minimal Behavior
- > **Behavior**
 - Essence of a class
 - Anonymous class
 - Format, methodDict, superclass
- > **ClassDescription**
 - Human representation and organization
- > **Metaclass**
 - Sole instance



Understanding Classes and Metaclasses



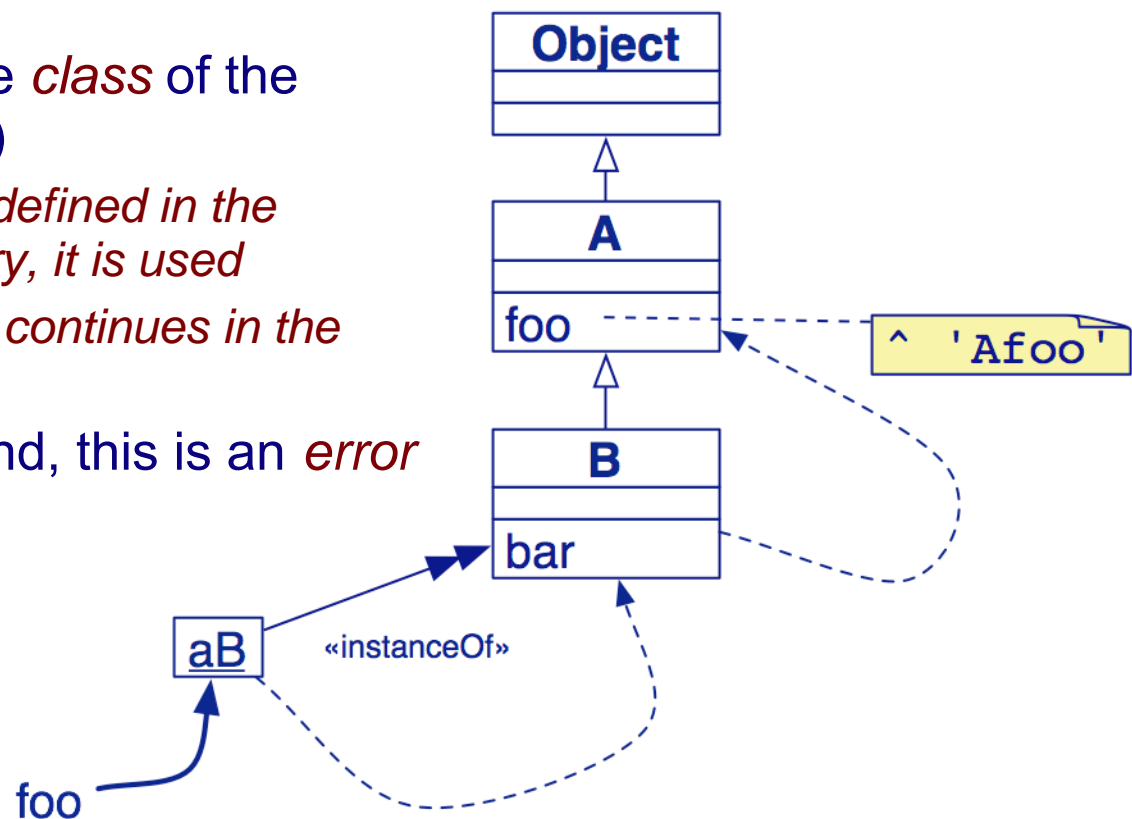
SmallTalk -- Everything is an object

- > Rule 1. Everything is an object.
- > Rule 2. Every object is an instance of a class.
- > Rule 3. Every class has a superclass.
- > Rule 4. Everything happens by sending messages.
- > Rule 5. Method lookup follows the inheritance chain.

Normal method lookup

Two step process:

- Lookup starts in the *class* of the *receiver* (an object)
 1. *If the method is defined in the method dictionary, it is used*
 2. *Else, the search continues in the superclass*
- If no method is found, this is an *error*
- ...



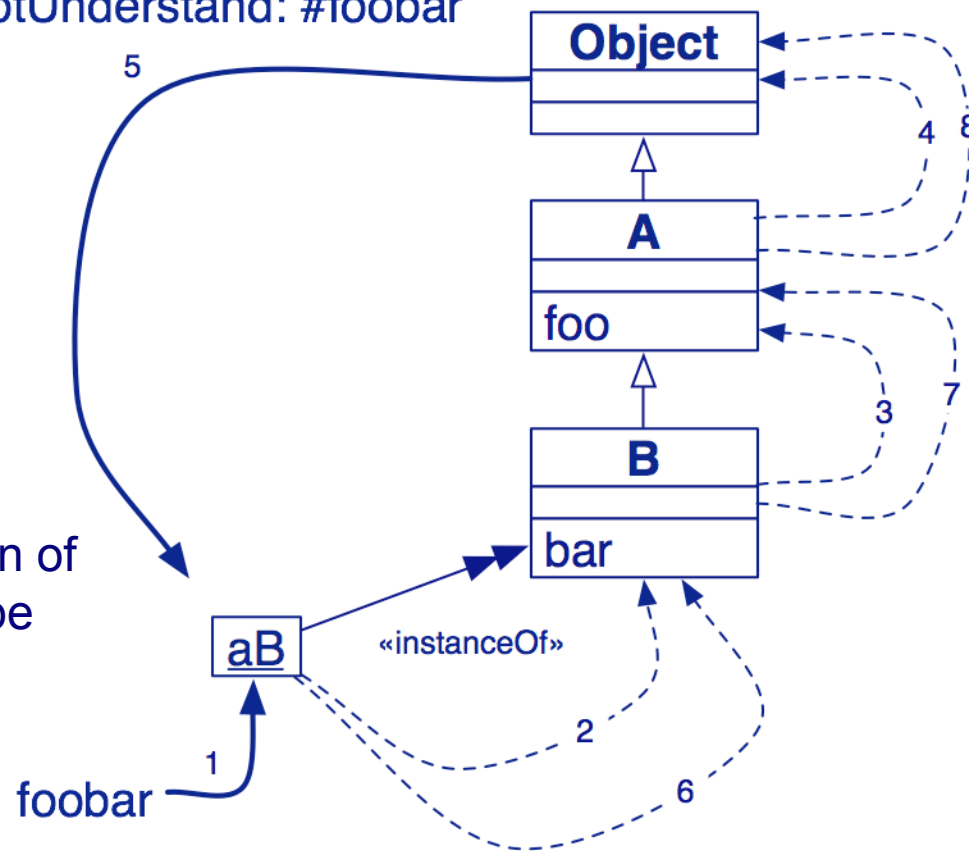
Message not understood

When method lookup fails, an error message is sent to the object and lookup starts again with this new message.

`self doesNotUnderstand: #foobar`

open debugger

NB: The default implementation of `doesNotUnderstand:` may be overridden by any class.



Birds-eye view



Reify your metamodel — A fully reflective system models its own metamodel.

Reify:

make (something abstract)
more concrete or real.

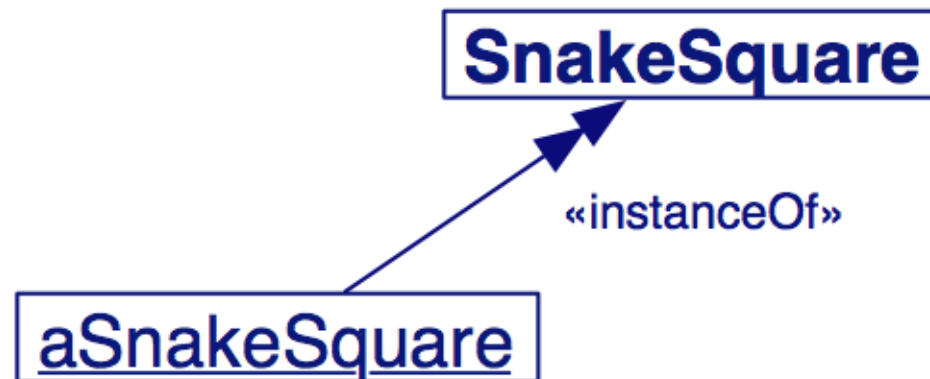
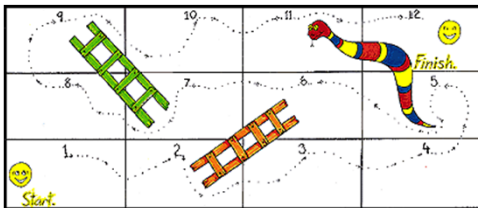


Metaclasses in 7 points

1. Every object is an instance of a class
2. Every class eventually inherits from Object
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass
7. The metaclass of Metaclass is an instance of Metaclass

Adapted from Goldberg & Robson, *Smalltalk-80 — The Language*

1. Every object is an instance of a class



Example from the Snakes and Ladders Board Game

...

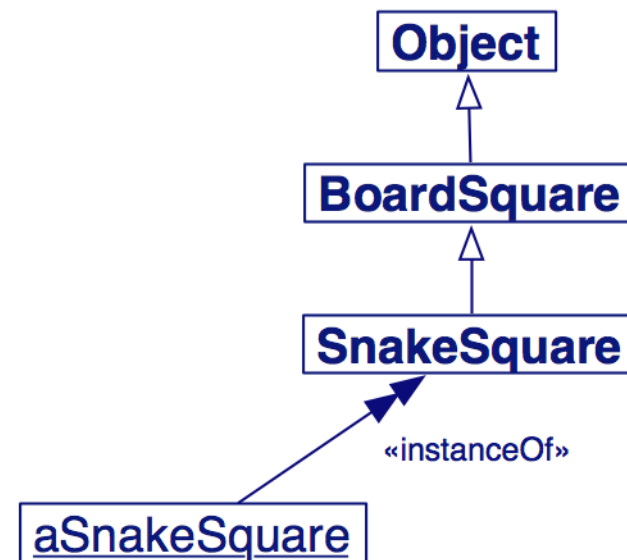
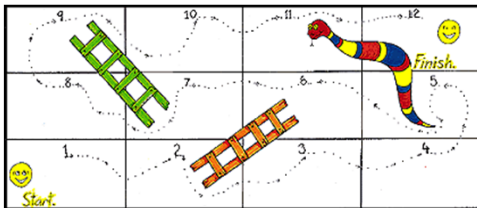
Metaclasses in 7 points

1. Every object is an instance of a class
2. **Every class eventually inherits from Object**
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass
7. The metaclass of Metaclass is an instance of Metaclass

2. Every class inherits from Object

- > ***Every object is-an Object =***
 - The class of every object ultimately inherits from `Object`

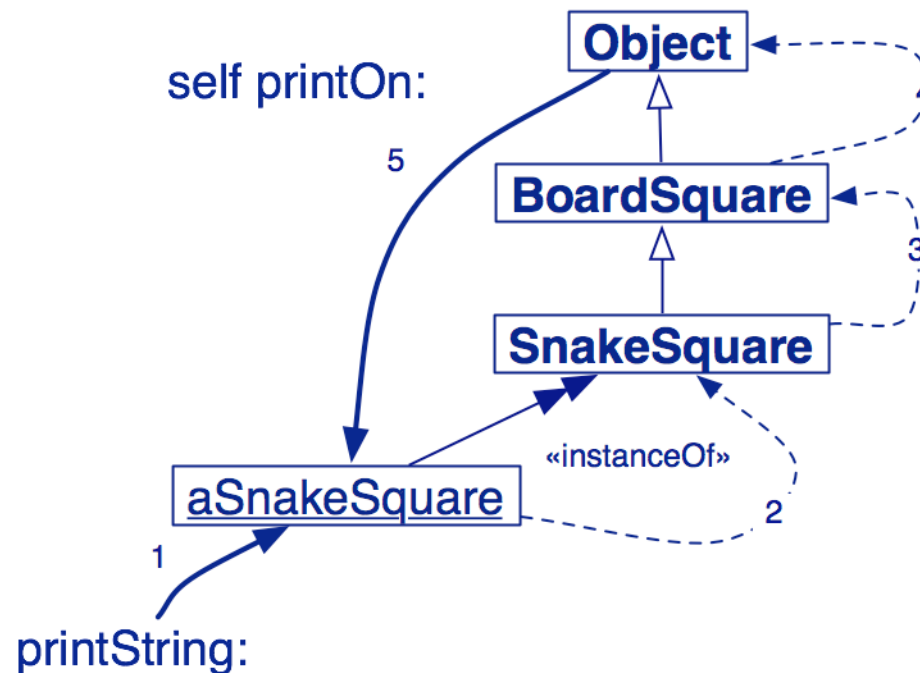
aSnakeSquare is-a SnakeSquare
and is-a BoardSquare
and is-an Object



Caveat: in Pharo, *Object* has a superclass called *ProtoObject*

The Meaning of is-a

- > When an object receives a message, the method is looked up in the method dictionary of its class, and, if necessary, its superclasses, up to Object



Responsibilities of Object

> Object

- represents the common object behavior
 - *error-handling, halting ...*
- all classes should inherit ultimately from Object

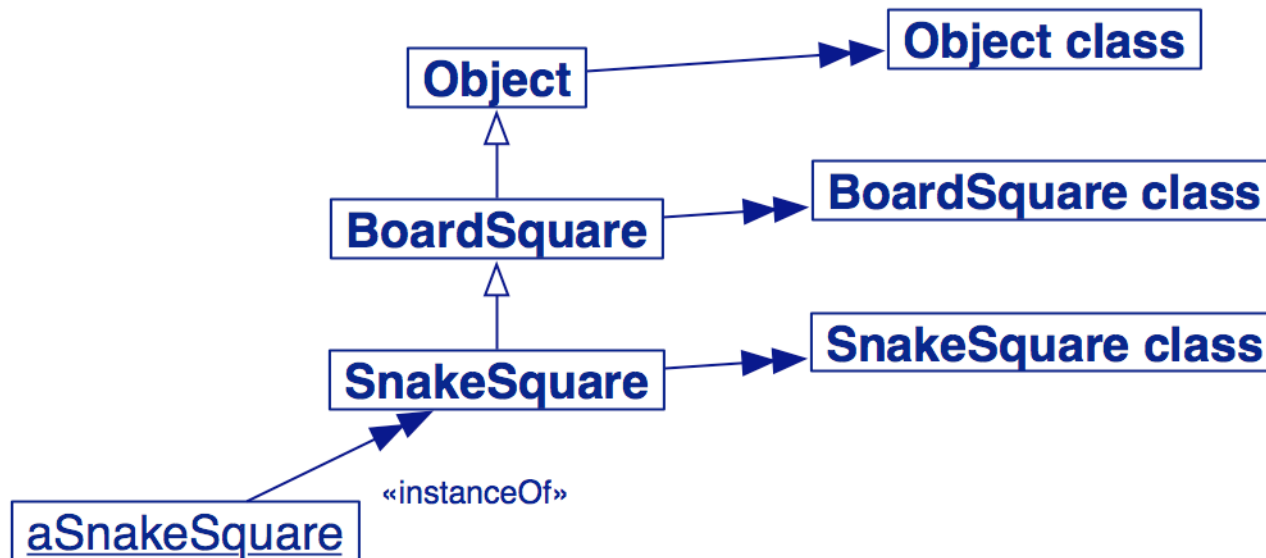
Metaclasses in 7 points

1. Every object is an instance of a class
2. Every class eventually inherits from Object
3. **Every class is an instance of a metaclass**
4. The metaclass hierarchy parallels the class hierarchy
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass
7. The metaclass of Metaclass is an instance of Metaclass

3. Every class is an instance of a metaclass

> ***Classes are objects too!***

- Every class `x` is the unique instance of its metaclass, called `x class`



Metaclasses are implicit

- > ***There are no explicit metaclasses***
 - Metaclasses are created implicitly when classes are created
 - No sharing of metaclasses (unique metaclass per class)

Metaclasses by Example

```
BoardSquare allSubclasses
SnakeSquare allSubclasses

SnakeSquare allInstances
SnakeSquare instVarNames

SnakeSquare back: 5

SnakeSquare selectors

SnakeSquare canUnderstand: #new
SnakeSquare canUnderstand: #setBack:
```

```
a Set(SnakeSquare FirstSquare LadderSquare)
a Set()

an Array(<-2[6] <-4[11] <-6[11])
#('back')

<-5[nil]

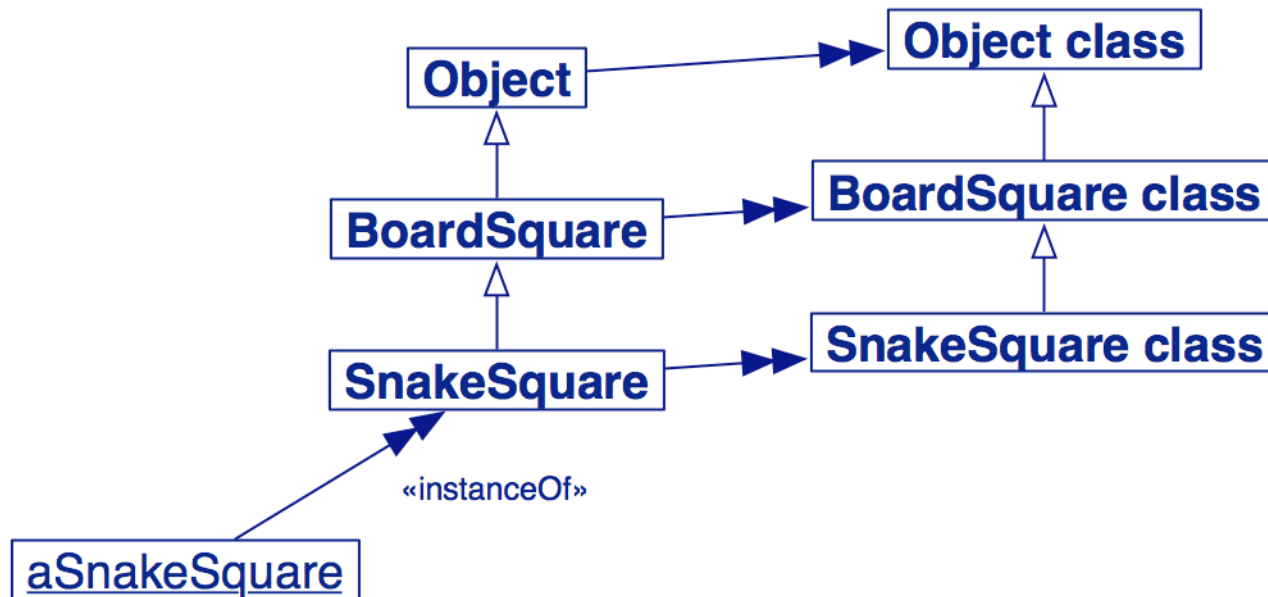
an IdentitySet(#setBack:
                #printOn: #destination)
false
true
```

NB: canUnderstand: tells you which messages *instances* can understand.
SnakeSquare class canUnderstand: #new -> true

Metaclasses in 7 points

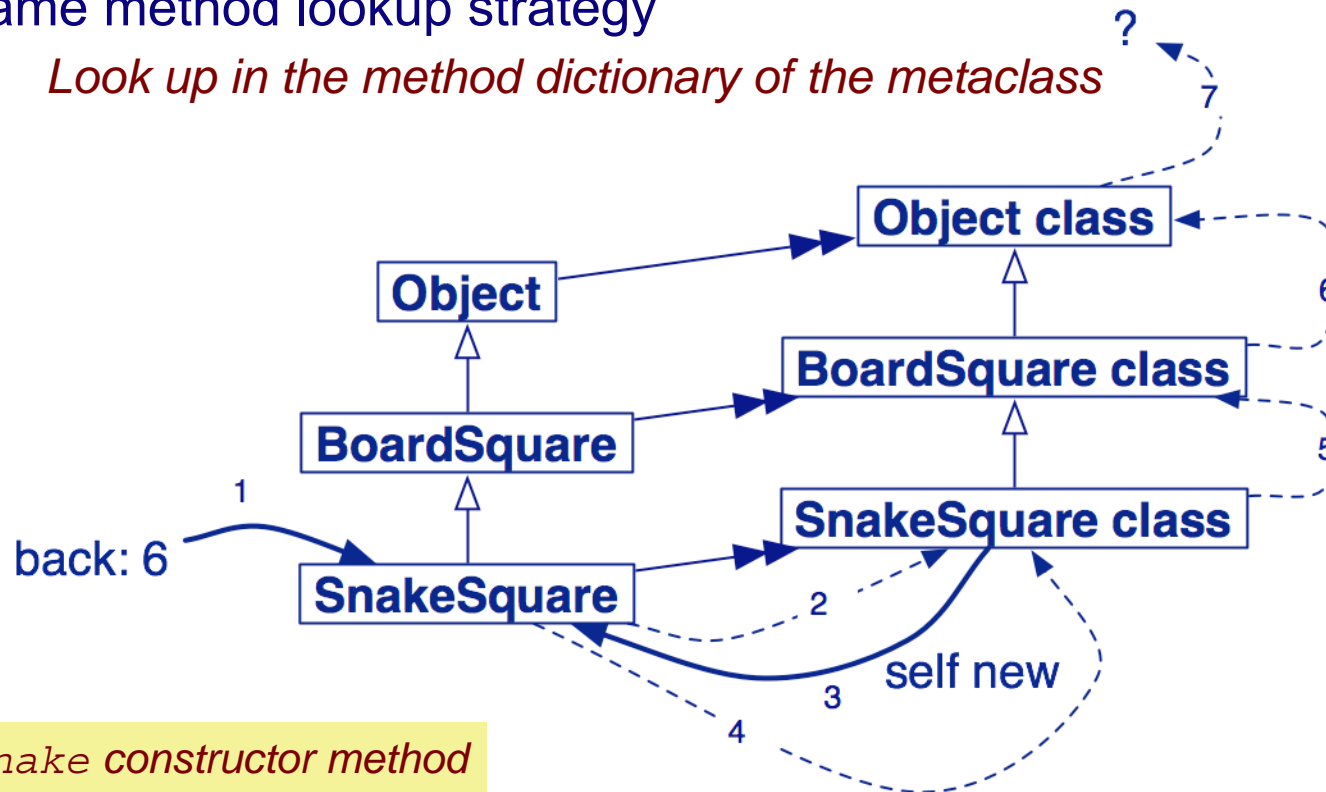
1. Every object is an instance of a class
2. Every class eventually inherits from Object
3. Every class is an instance of a metaclass
4. **The metaclass hierarchy parallels the class hierarchy**
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass
7. The metaclass of Metaclass is an instance of Metaclass

4. The metaclass hierarchy parallels the class hierarchy



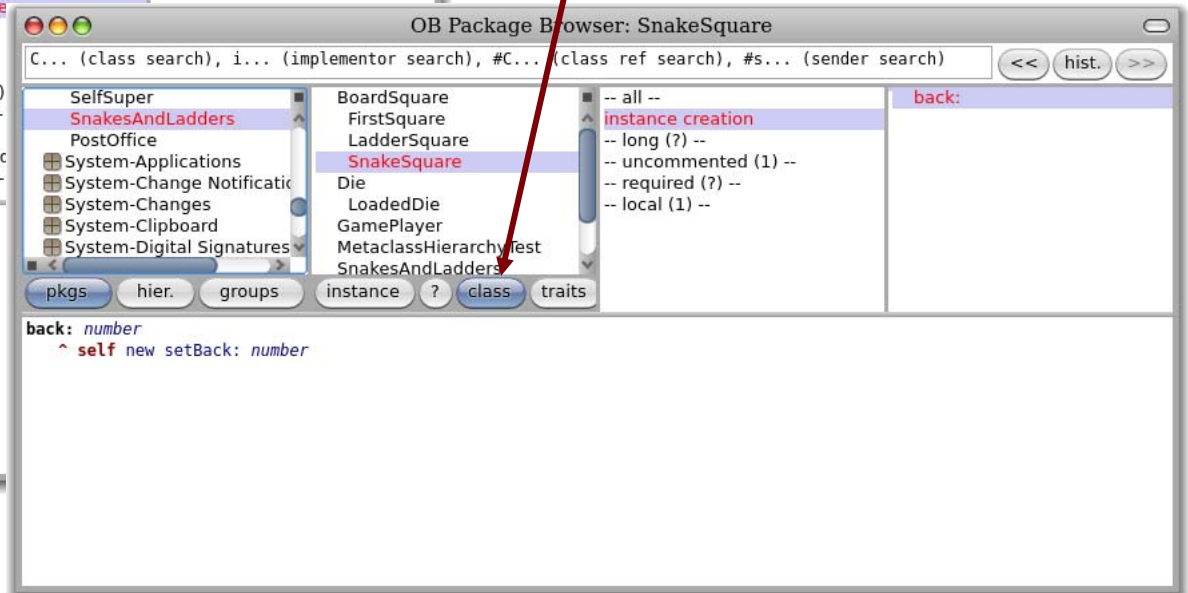
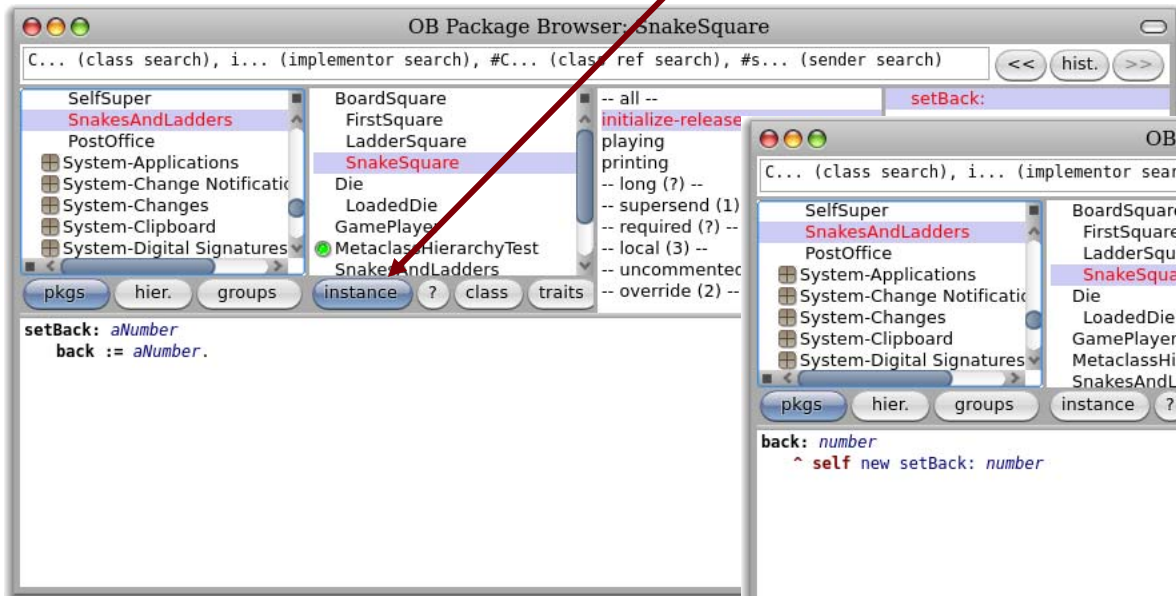
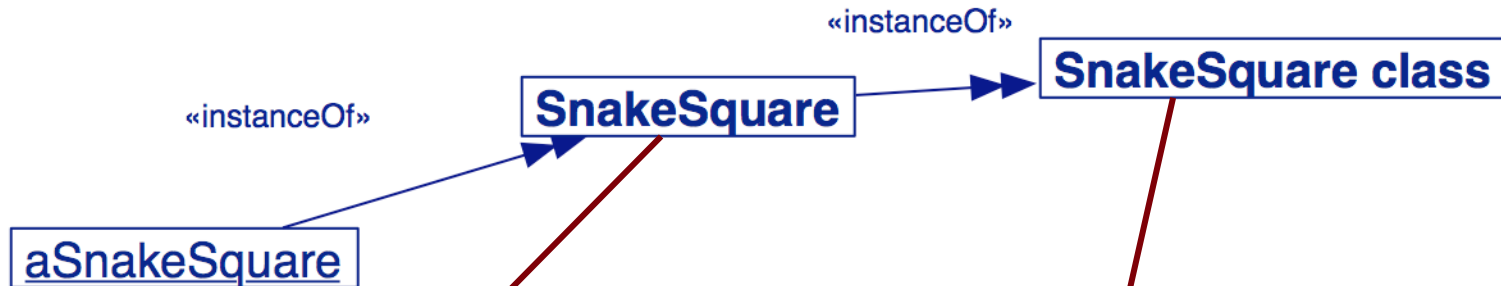
Uniformity between Classes and Objects

- > Classes are objects too, so ...
 - Everything that holds for objects holds for classes as well
 - Same method lookup strategy
 - *Look up in the method dictionary of the metaclass*



back: is a Snake constructor method

About the Buttons



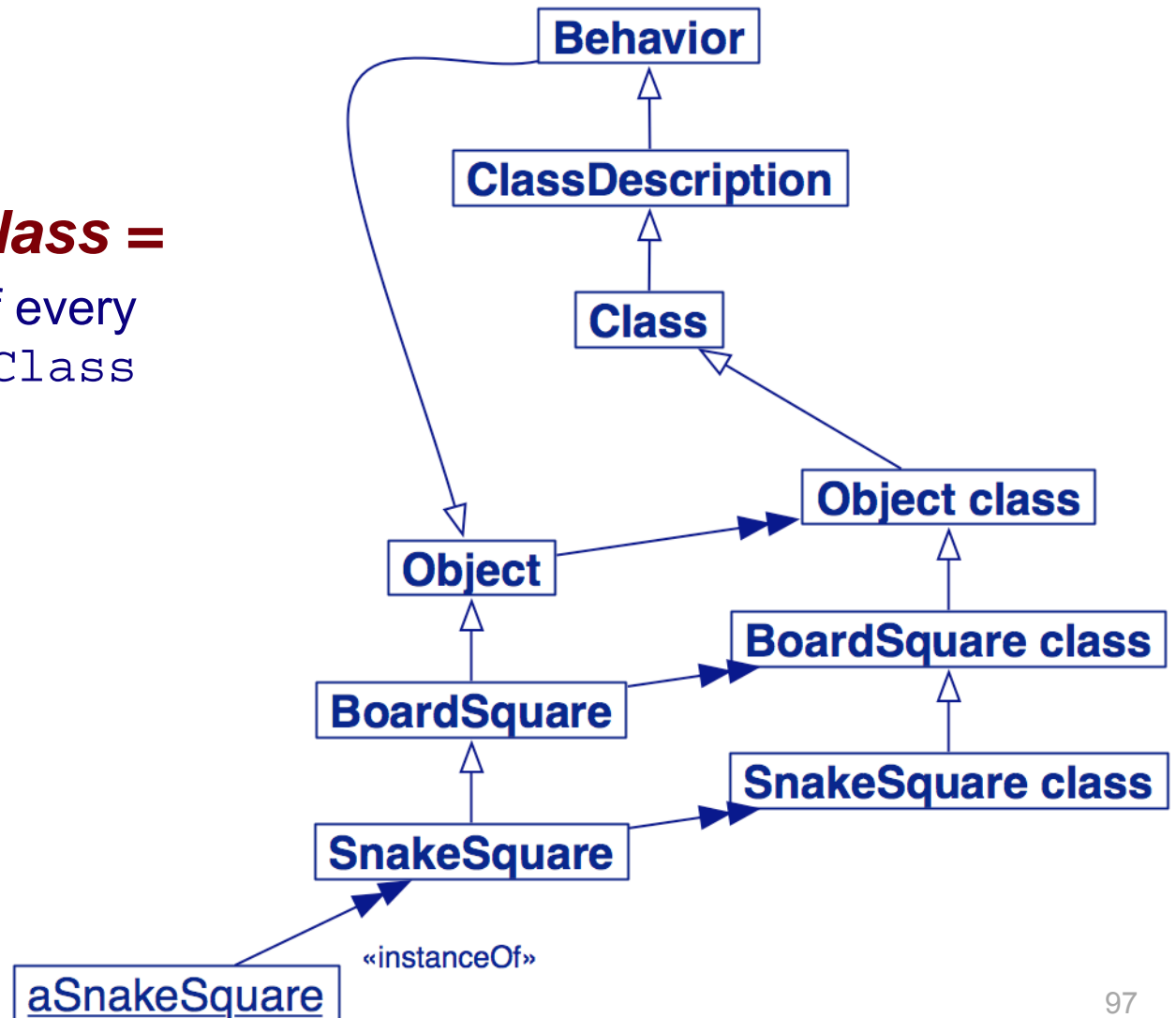
Metaclasses in 7 points

1. Every object is an instance of a class
2. Every class eventually inherits from Object
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. **Every metaclass inherits from Class and Behavior**
6. Every metaclass is an instance of Metaclass
7. The metaclass of Metaclass is an instance of Metaclass

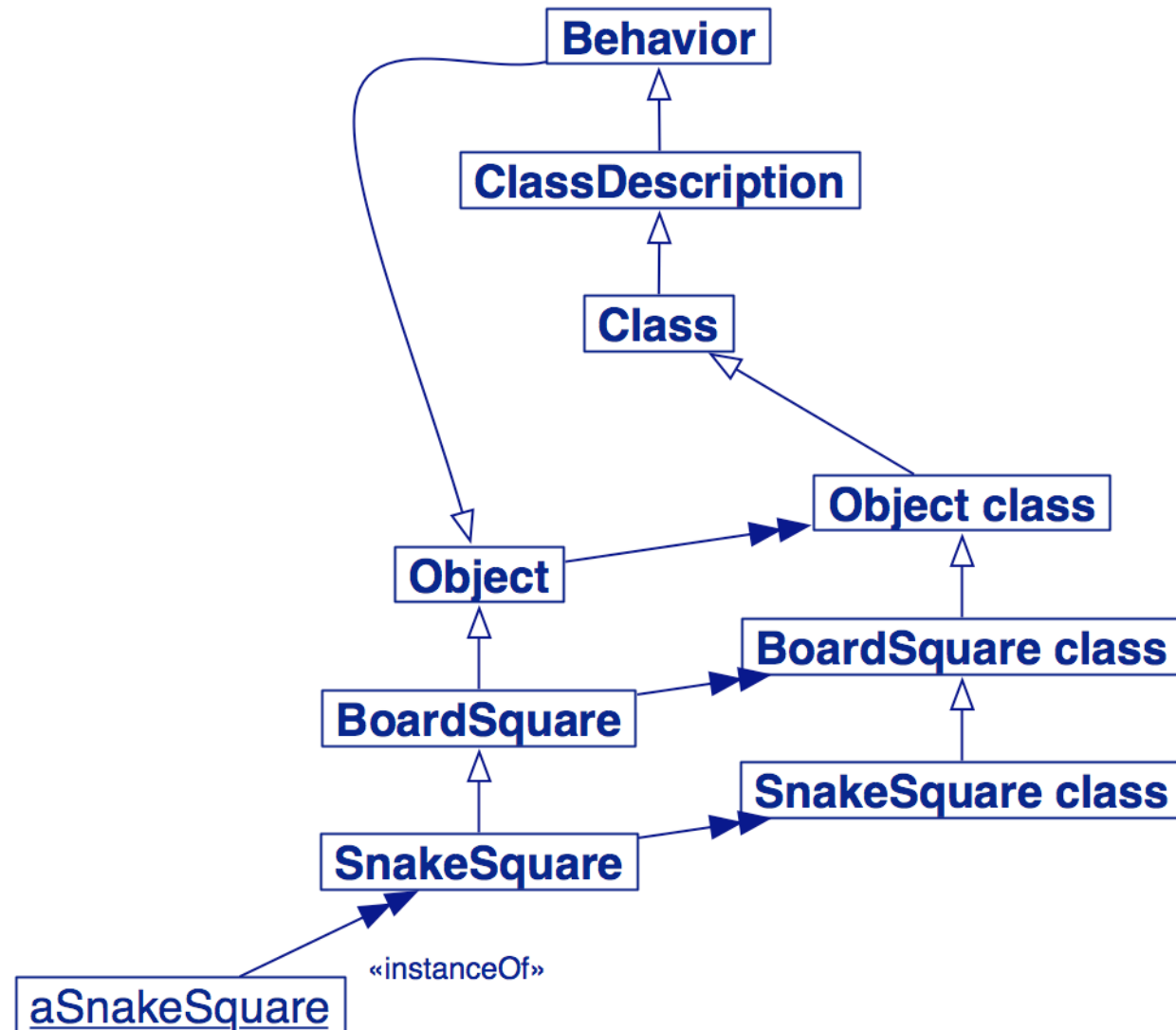
5. Every metaclass inherits from Class and Behavior

Every class is-a Class =

—The metaclass of every class inherits from Class



Where is new defined?



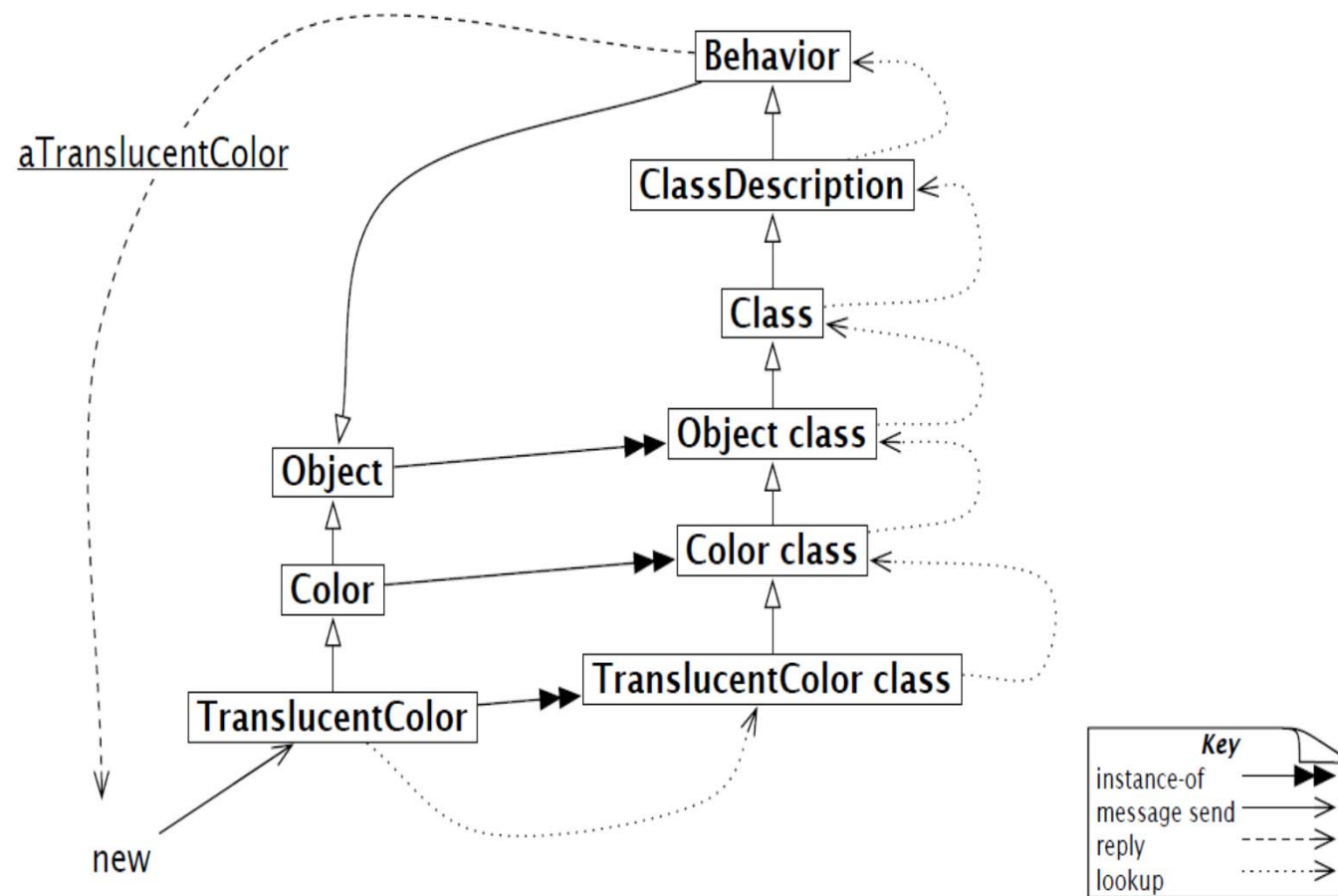


Figure 13.8: new is an ordinary message looked up in the metaclass chain.

Responsibilities of Behavior

> Behavior

- Minimum state necessary for objects that have instances.
- Basic interface to the compiler.
- **State:**
 - *class hierarchy link, method dictionary, description of instances (representation and number)*
- **Methods:**
 - *creating a method dictionary, compiling method*
 - *instance creation (new, basicNew, new:, basicNew:)*
 - *class hierarchy manipulation (superclass:, addSubclass:)*
 - *accessing (selectors, allSelectors, compiledMethodAt:)*
 - *accessing instances and variables (allInstances, instVarNames)*
 - *accessing class hierarchy (superclass, subclasses)*
 - *testing (hasMethods, includesSelector, canUnderstand:, inheritsFrom:, isVariable)*

Responsibilities of ClassDescription

> ClassDescription

- adds a number of facilities to basic Behavior:
 - *named instance variables*
 - *category organization for methods*
 - *the notion of a name (abstract)*
 - *maintenance of Change sets and logging changes*
 - *most of the mechanisms needed for fileOut*
- ClassDescription is an abstract class: its facilities are intended for inheritance by the two subclasses, Class and Metaclass.

Responsibilities of Class

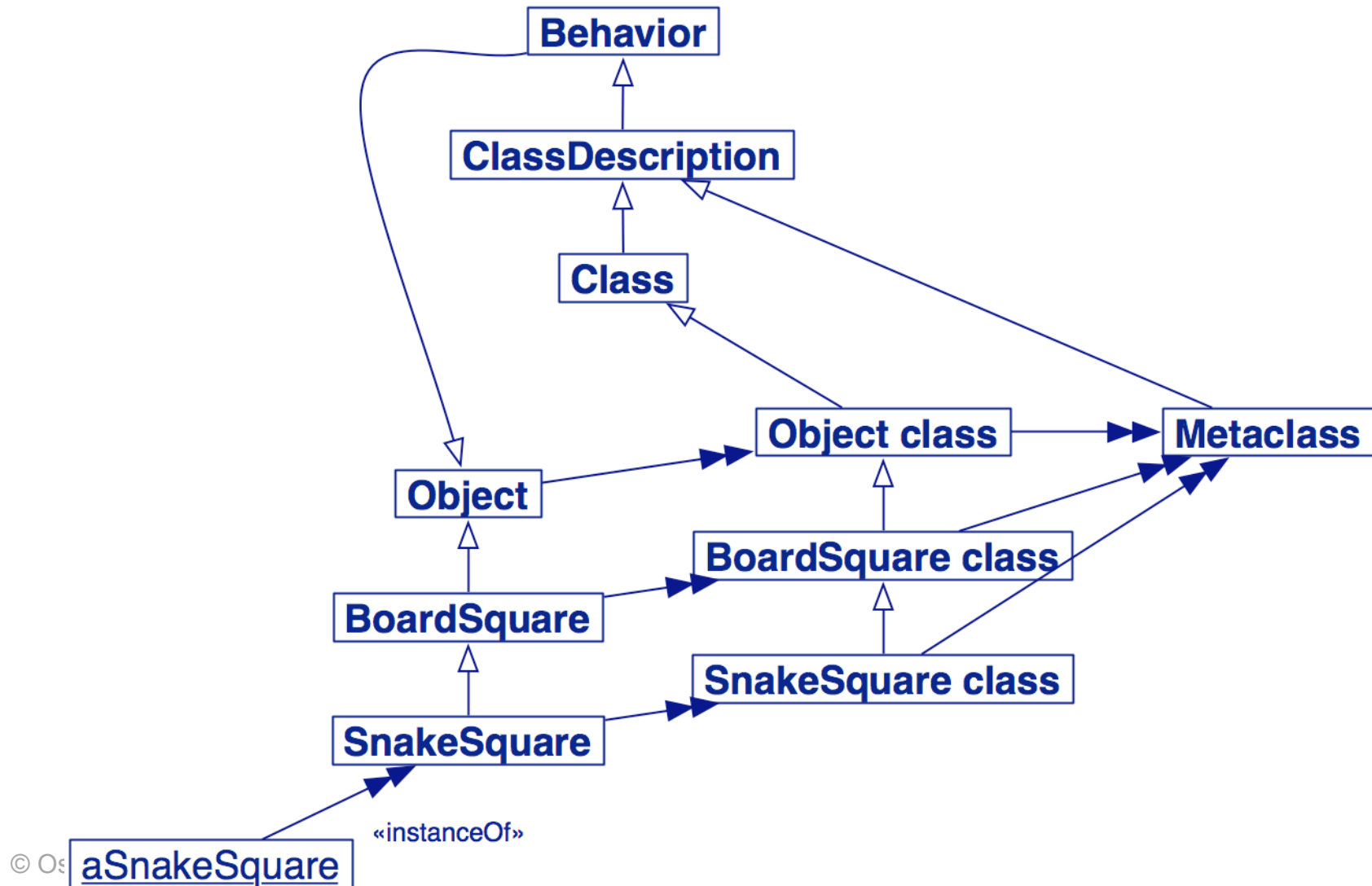
> **Class**

- represents the common behavior of all classes
 - *name, compilation, method storing, instance variables ...*
- representation for classVariable names and shared pool variables (`addClassVarName:`, `addSharedPool:`, `initialize`)
- Class inherits from Object because Class is an Object
 - *Class knows how to create instances, so all metaclasses should inherit ultimately from Class*

Metaclasses in 7 points

1. Every object is an instance of a class
2. Every class eventually inherits from Object
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. Every metaclass inherits from Class and Behavior
- 6. Every metaclass is an instance of Metaclass**
7. The metaclass of Metaclass is an instance of Metaclass

6. Every metaclass is an instance of Metaclass



Metaclass Responsibilities

> Metaclass

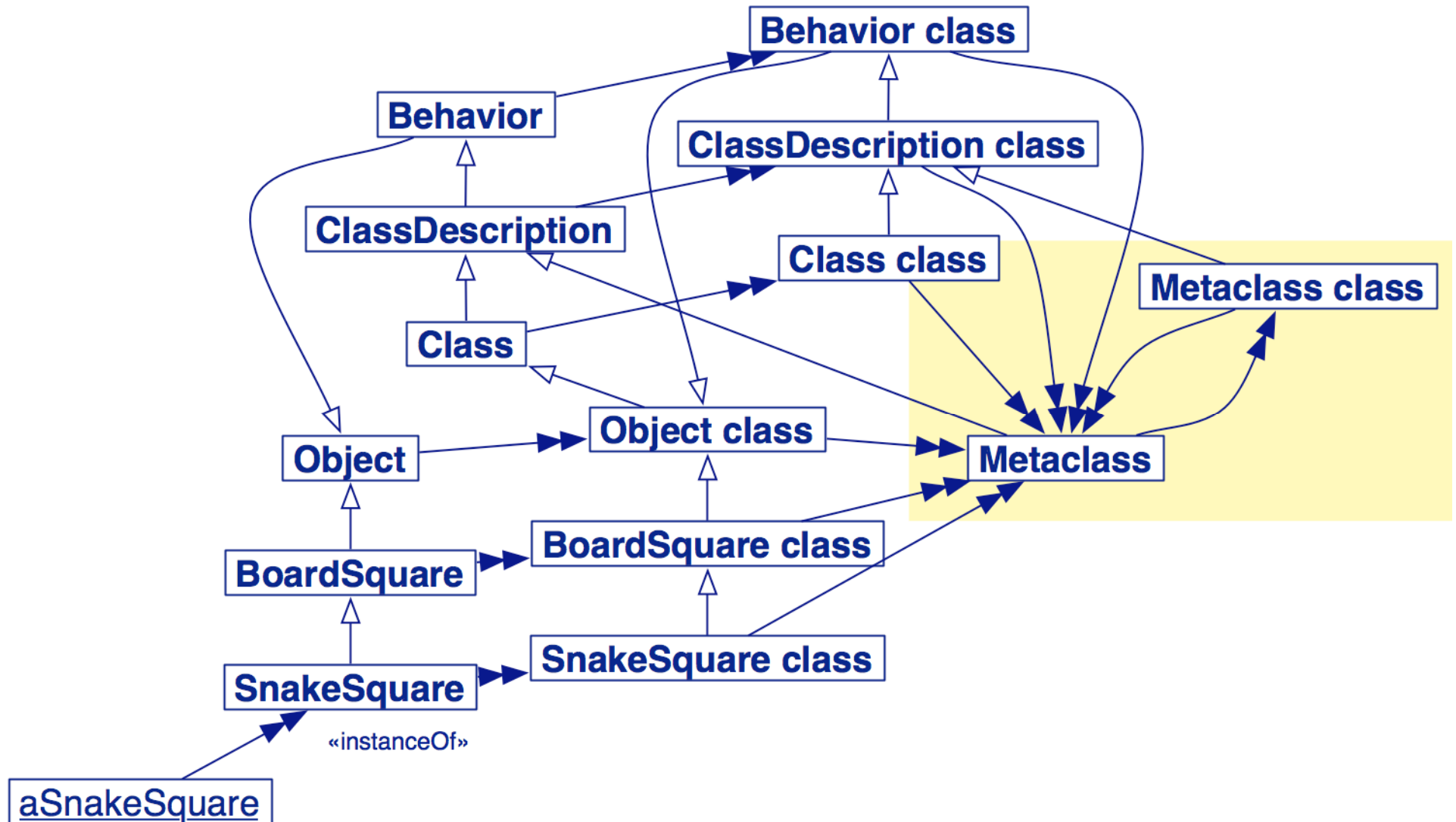
— Represents common metaclass Behavior

- *instance creation (subclassOf:)*
- *creating initialized instances of the metaclass's sole instance*
- *initialization of class variables*
- *metaclass instance protocol (name:inEnvironment:subclassOf:.....)*
- *method compilation (different semantics can be introduced)*
- *class information (inheritance link, instance variable, ...)*

Metaclasses in 7 points

1. Every object is an instance of a class
2. Every class eventually inherits from Object
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass
7. **The metaclass of Metaclass is an instance of Metaclass**

7. The metaclass of Metaclass is an instance of Metaclass








Navigating the metaclass hierarchy

```
MetaclassHierarchyTest>>testHierarchy
  "The class hierarchy"
self assert: SnakeSquare superclass = BoardSquare.
self assert: BoardSquare superclass = Object.
self assert: Object superclass superclass = nil.
  "The parallel metaclass hierarchy"
self assert: SnakeSquare class name = 'SnakeSquare class'.
self assert: SnakeSquare class superclass = BoardSquare class.
self assert: BoardSquare class superclass = Object class.
self assert: Object class superclass superclass = Class.
self assert: Class superclass = ClassDescription.
self assert: ClassDescription superclass = Behavior.
self assert: Behavior superclass = Object.
  "The Metaclass hierarchy"
self assert: SnakeSquare class class = Metaclass.
self assert: BoardSquare class class = Metaclass.
self assert: Object class class = Metaclass.
self assert: Class class class = Metaclass.
self assert: ClassDescription class class = Metaclass.
self assert: Behavior class class = Metaclass.
self assert: Metaclass superclass = ClassDescription.
  "The fixpoint"
self assert: Metaclass class class = Metaclass
```







Navigating the metaclass hierarchy

```
3 class.  
3 class class  
3 class class class  
3 class class class class  
3 class class class class class  
3 class class class class class class
```

What you should know!

-  *What does is-a mean?*
-  *What is the difference between sending a message to an object and to its class?*
-  *What are the responsibilities of a metaclass?*
-  *What is the superclass of `Object` class?*
-  *Where is `new` defined?*

Can you answer these questions?

-  *Why are there no explicit metaclasses?*
-  *When should you override `new`?*
-  *Why don't metaclasses inherit from `Class`?*
-  *Are there any classes that don't inherit from `Object`?*
-  *Is `Metaclass` a `Class`? Why or why not?*
-  *Where are the methods `class` and `superclass` defined?*

License

<http://creativecommons.org/licenses/by-sa/3.0/>



Attribution-ShareAlike 3.0 Unported

You are free:

- to Share** — to copy, distribute and transmit the work
- to Remix** — to adapt the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.