b
UNIVERSITÄT
BERN

3. Understanding Classes and Metaclasses

Oscar Nierstrasz

Selected material courtesy Stéphane Ducasse

Birds-eye view

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

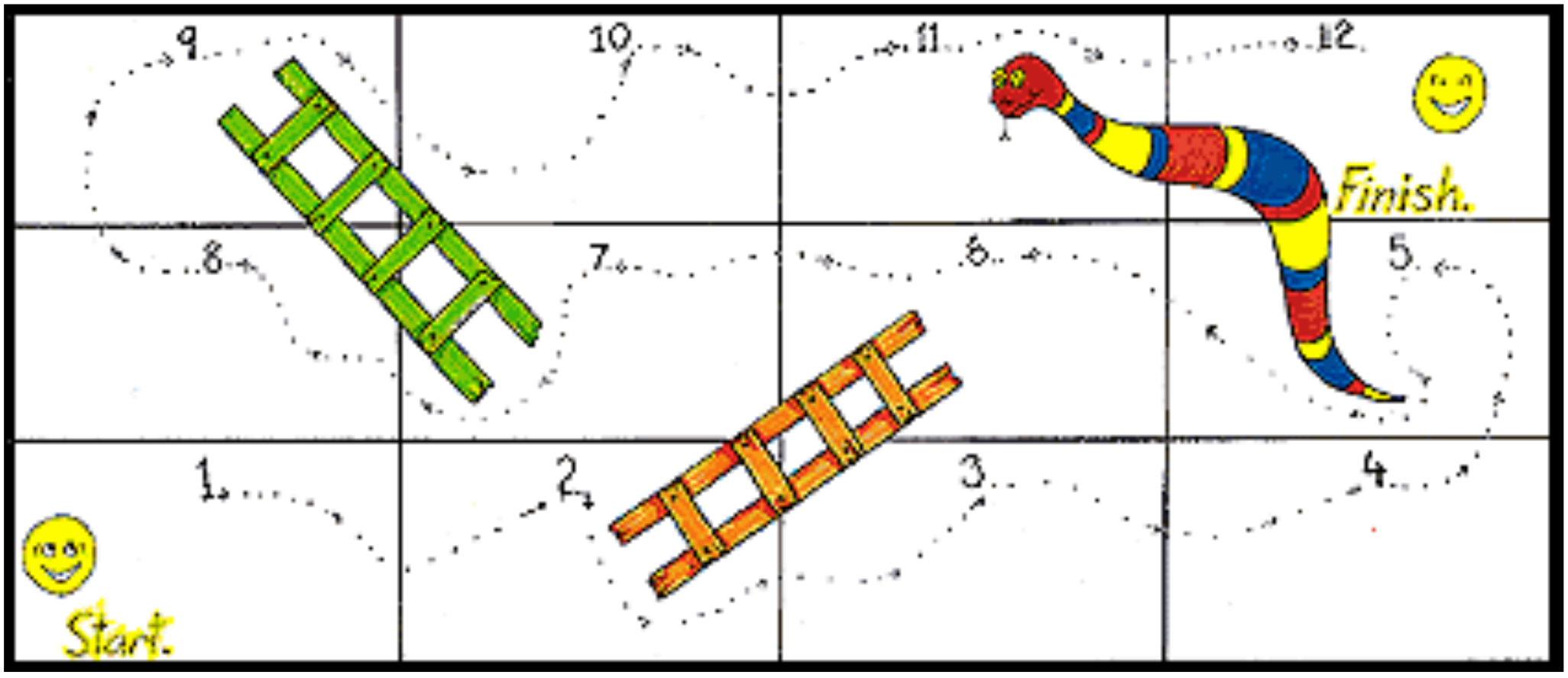
Roadmap

- Common idioms
- Self and Super
- Metaclasses in 7 points

Roadmap

- **Common idioms**
- Self and Super
- Metaclasses in 7 points

Snakes and Ladders

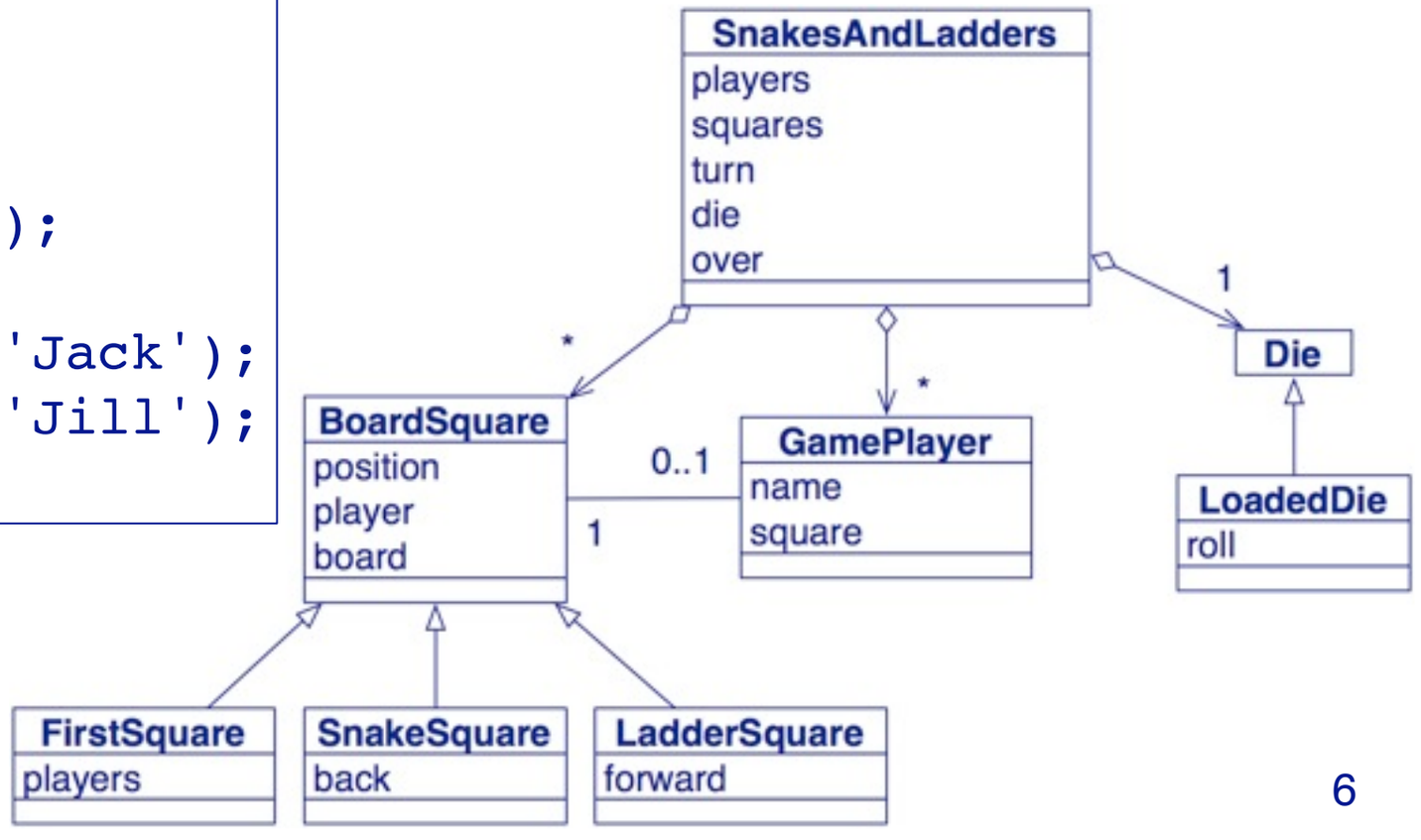


See: http://en.wikipedia.org/wiki/Snakes_and_ladders

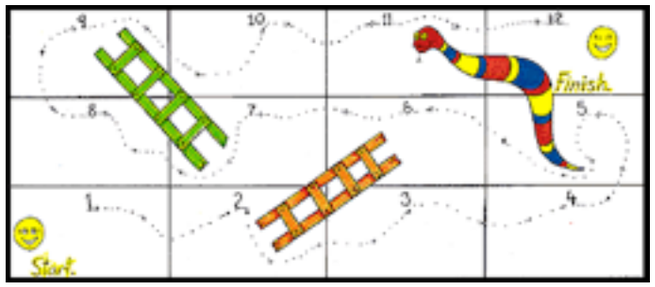
Scripting a use case

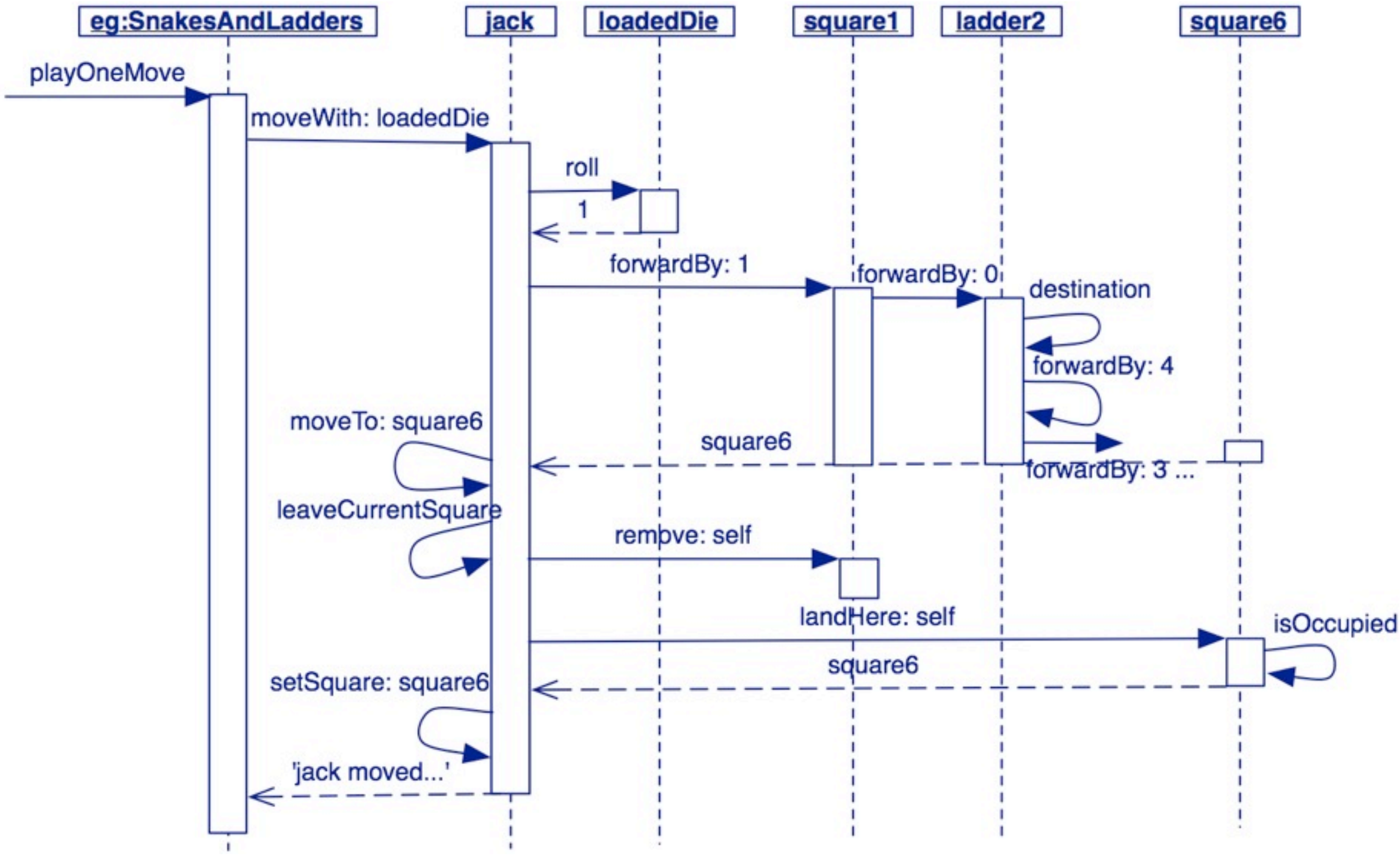
```
SnakesAndLadders class>>example
    "self example playToEnd"
    ^ (self new)
        add: FirstSquare new;
        add: (LadderSquare forward: 4);
        add: BoardSquare new;
        add: BoardSquare new;
        add: BoardSquare new;
        add: BoardSquare new;
        add: (LadderSquare forward: 2);
        add: BoardSquare new;
        add: BoardSquare new;
        add: BoardSquare new;
        add: (SnakeSquare back: 6);
        add: BoardSquare new;
        join: (GamePlayer named: 'Jack');
        join: (GamePlayer named: 'Jill');
        yourself
```

- Construct the board
- Add some players
- Play the game



Distributing responsibilities





Lots of Little Methods

- ***Once and only once***
  - —"In a program written with good style, everything is said once and only once."

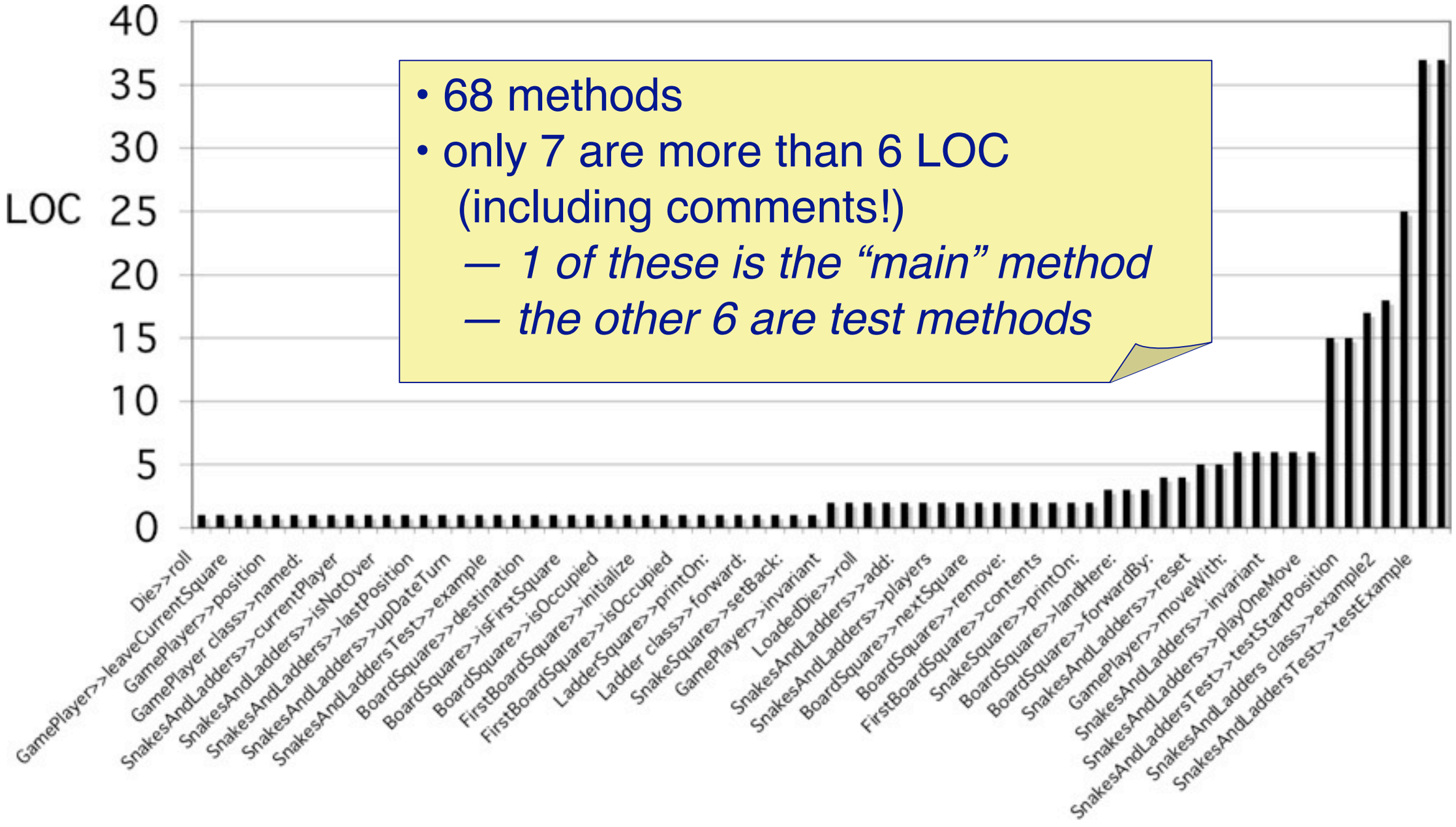
- ***Lots of little pieces***
  - —"Good code invariably has small methods and small objects. Only by factoring the system into many small pieces of state and function can you hope to satisfy the 'once and only once' rule."

Composed Method

How do you divide a program into methods?

- Divide your program into methods that perform one identifiable task.
  - Keep all of the operations in a method at the *same level of abstraction*.
  - This will naturally result in programs with *many small methods, each a few lines long*.

Snakes and Ladders methods



How to initialize objects?

In Smalltalk,
— *methods are public, and*
— *instance variables are private*

So, how can a class (an object) initialize the instance variables of its instances (other objects)?

Explicit Initialization

How do you initialize instance variables to their default values?

- Implement a method `initialize` that sets all the values explicitly.
- Override the class message `new` to invoke it on new instances

```
SnakesAndLadders>>initialize
    super initialize.
    die := Die new.
    squares := OrderedCollection new.
    players := OrderedCollection new.
    turn := 1.
    over := false.
```

Who calls initialize?

- In Pharo, the method `new` calls `initialize` by default.

```
Behavior>>new
   ^ self basicNew initialize
```

- ***NB:*** You can override `new`, but you should *never* override `basicNew`!

Constructor Method

How do you represent instance creation?

- Provide methods in the class side “instance creation” protocol that create well-formed instances. Pass all required parameters to them.

```
LadderSquare class>>forward: number
    ^ self new setForward: number
```

```
SnakeSquare class>>back: number
    ^ self new setBack: number
```

Constructor Parameter Method

How do you set instance variables from the parameters to a Constructor Method?

- Code a single method that sets all the variables. Preface its name with “set”, then the names of the variables.

```
SnakeSquare>>setBack: aNumber
   back := aNumber.
```

```
LadderSquare>>setForward: aNumber
   forward := aNumber.
```

```
BoardSquare>>setPosition: aNumber board: aBoard
   position := aNumber.
   board := aBoard
```

Constructor Parameter Method

How do you set instance variables from the parameters to a Constructor Method?

- Code a single method that sets all the variables. Preface its name with “set”, then the names of the variables.

```
SnakeSquare>>setBack: aNumber
   back := aNumber.
```

```
LadderSquare>>setForward: aNumber
   forward := aNumber.
```

```
BoardSquare>>setPosition: aNumber board: aBoard
   position := aNumber.
   board := aBoard
```

Better yet, use “initialize” as the prefix

Debug Printing Method

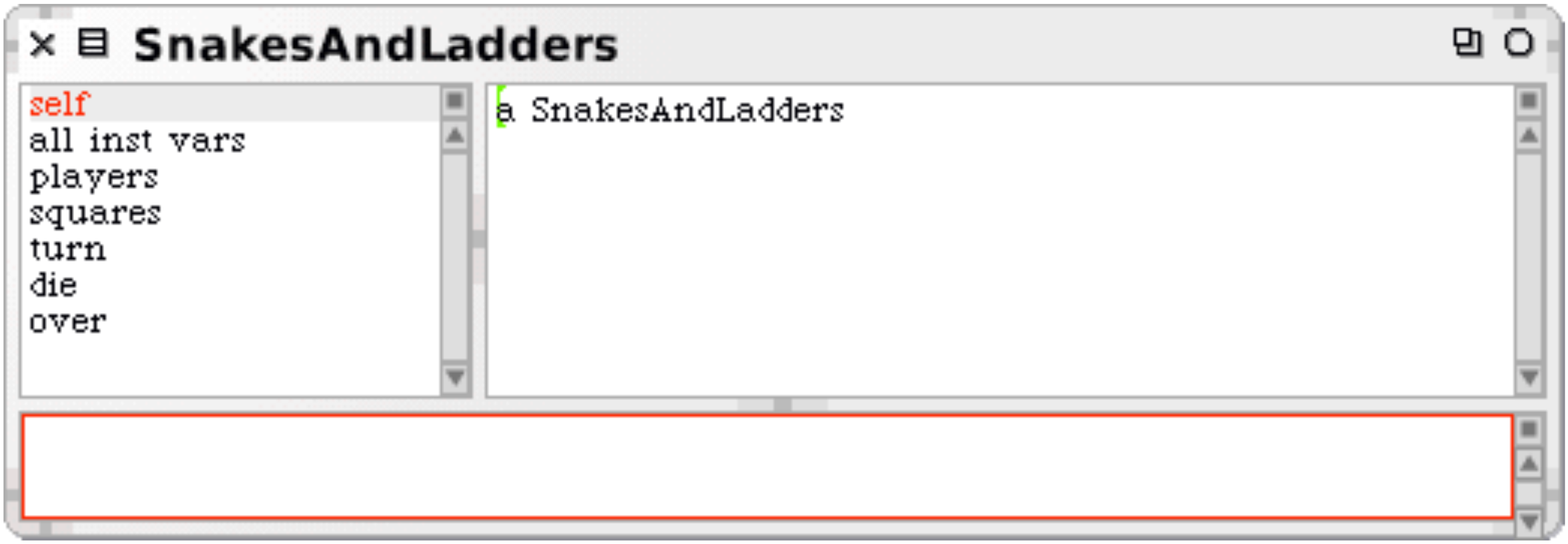
How do you code the default printing method?

- There are two audiences:
  - you (wanting a lot of information)
  - your clients (wanting only external properties)

- Override `printOn:` to provide information about object's structure to the programmer
  - Put printing methods in the "printing" protocol

Viewing the game state

```
SnakesAndLadders example inspect
```



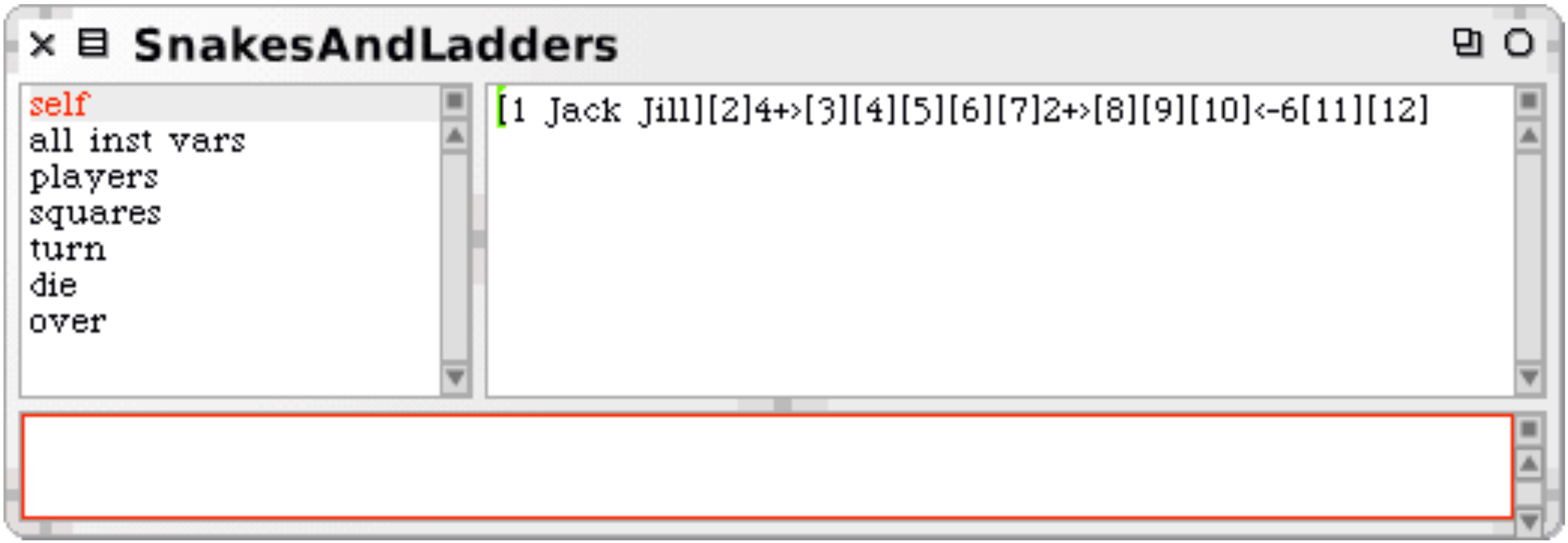
In order to provide a simple way to monitor the game state and to ease debugging, we need a textual view of the game

Implementing printOn:

```
SnakesAndLadders>>printOn: aStream
   squares do: [:each | each printOn: aStream]

BoardSquare>>printOn: aStream
   aStream nextPutAll: '[', position printString, self contents, ']'

LadderSquare>>printOn: aStream
   super printOn: aStream.
   aStream nextPutAll: forward asString, '+>'

SnakeSquare>>printOn: aStream
   aStream nextPutAll: '<-', back asString.
   super printOn: aStream

GamePlayer>>printOn: aStream
   aStream nextPutAll: name
```

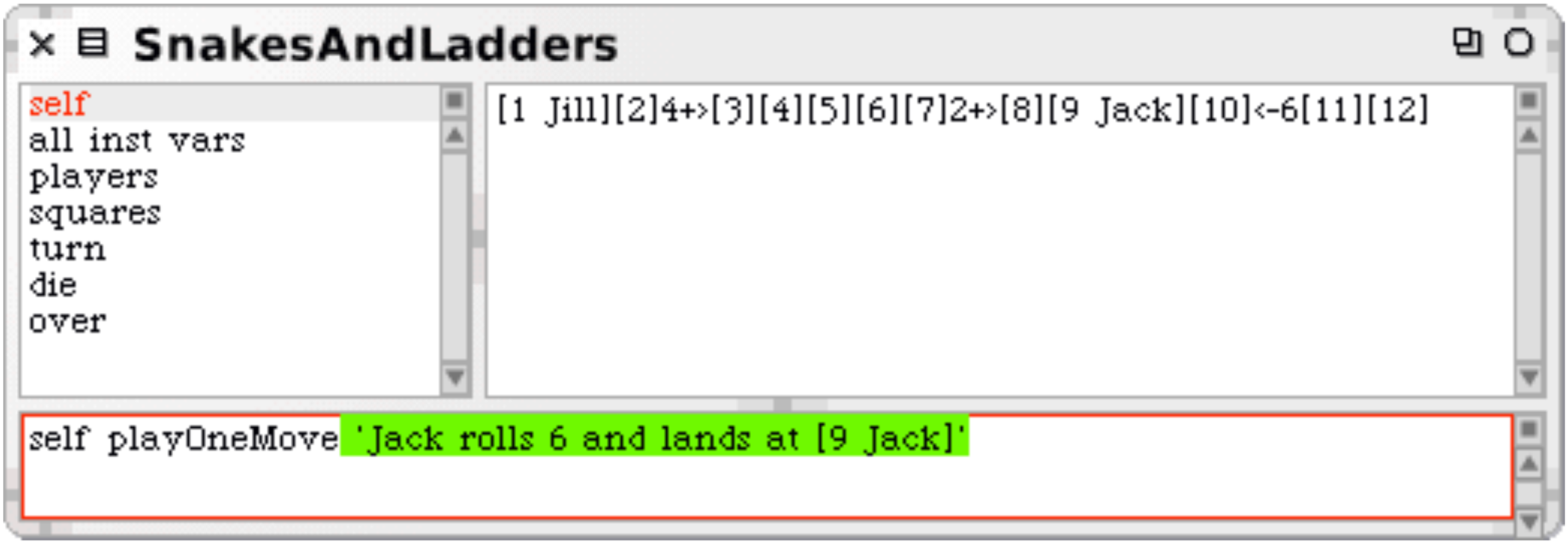
Viewing the game state

```
SnakesAndLadders example inspect
```



Interacting with the game

With a bit of care, the Inspector can serve as a basic GUI for objects we are developing



Query Method

How do you represent testing a property of an object?

- Provide a method that returns a `Boolean`.
  - Name it by prefacing the property name with a form of “be” — is, was, will etc.

Some query methods

```
SnakesAndLadders>>isNotOver
   ^ self isOver not

BoardSquare>>isFirstSquare
   ^ position = 1

BoardSquare>>isLastSquare
   ^ position = board lastPosition

BoardSquare>>isOccupied
   ^ player notNil

FirstSquare>>isOccupied
   ^ players size > 0
```

Constant Method

How do you code a constant?

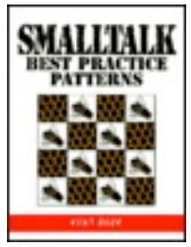
- Create a method that returns the constant

```
Fraction>>one
   ^ self numerator: 1 denominator: 1
```

Roadmap

- Common idioms
- **Self and Super**
- Metaclasses in 7 points

Super



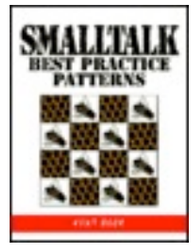
How can you invoke superclass behaviour?

> Invoke code in a superclass explicitly by sending a message to `super` instead of `self`.

— The method corresponding to the message will be found in the *superclass of the class implementing the sending method*.

— Always check code using super carefully. Change super to self if doing so does not change how the code executes!

— **Caveat:** If subclasses are *expected* to call super, consider using a Template Method instead!



Extending Super

How do you add to the implementation of a method inherited from a superclass?

> Override the method and send a message to super in the overriding method.

A closer look at super

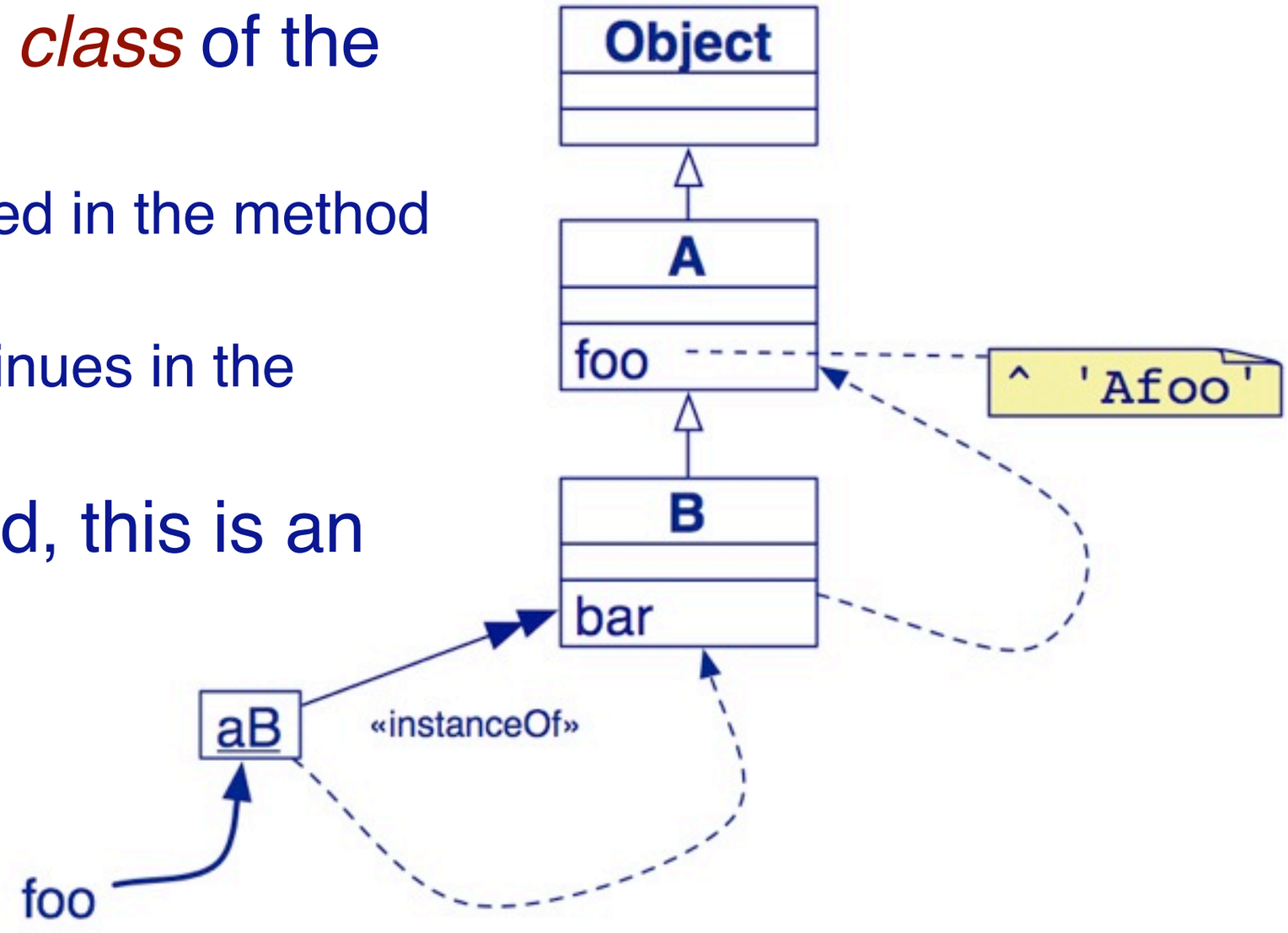
- Snake and Ladder both extend the printOn: method of their superclass

```
BoardSquare>>printOn: aStream
   aStream nextPutAll:
      '[', position printString, self contents, ']'

LadderSquare>>printOn: aStream
   super printOn: aStream.
   aStream nextPutAll: forward asString, '+>'

SnakeSquare>>printOn: aStream
   aStream nextPutAll: '<-', back asString.
   super printOn: aStream.
```

Normal method lookup

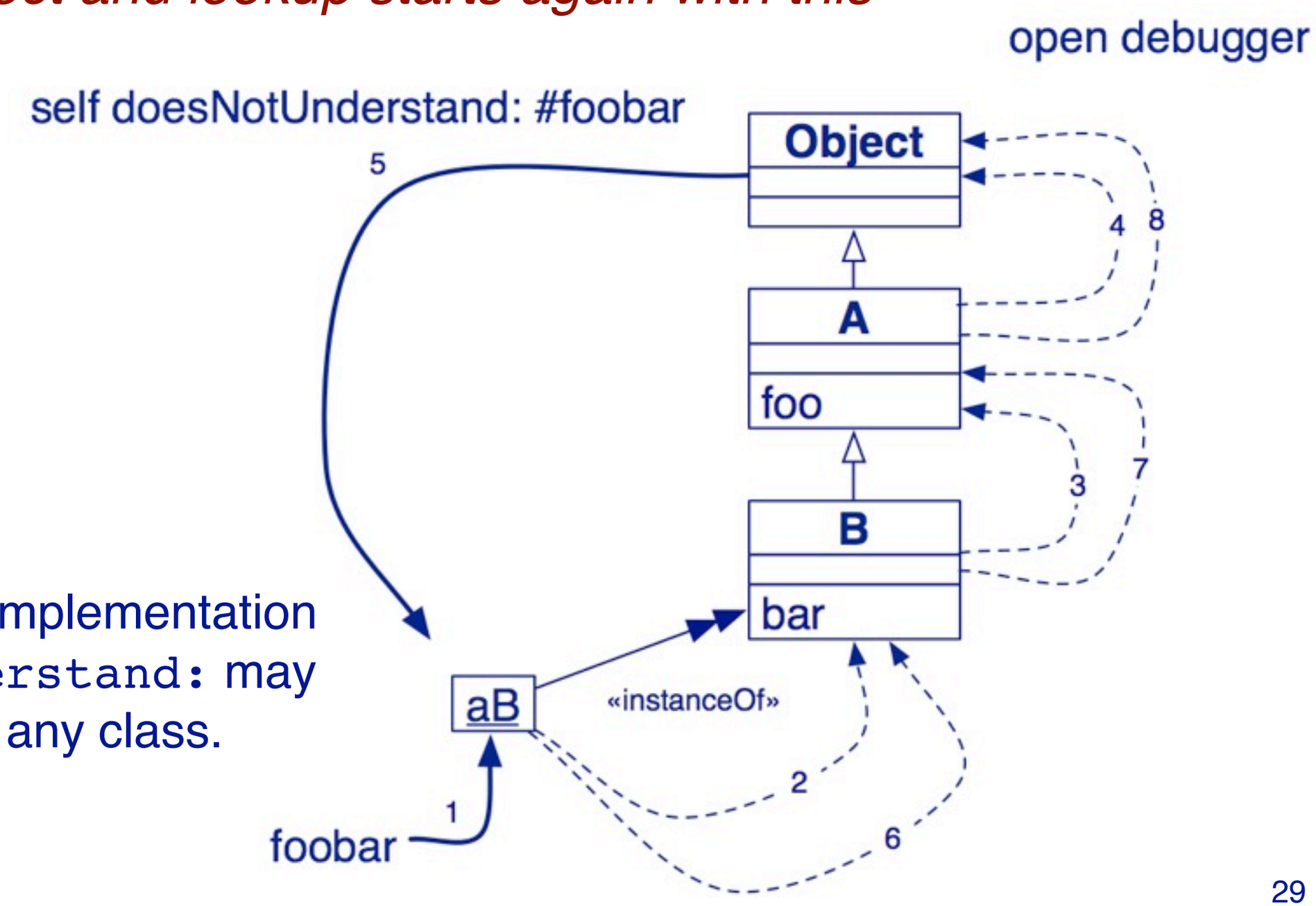
Two step process:

- Lookup starts in the *class* of the *receiver* (an object)
  - If the method is defined in the method dictionary, it is used
  - 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.



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

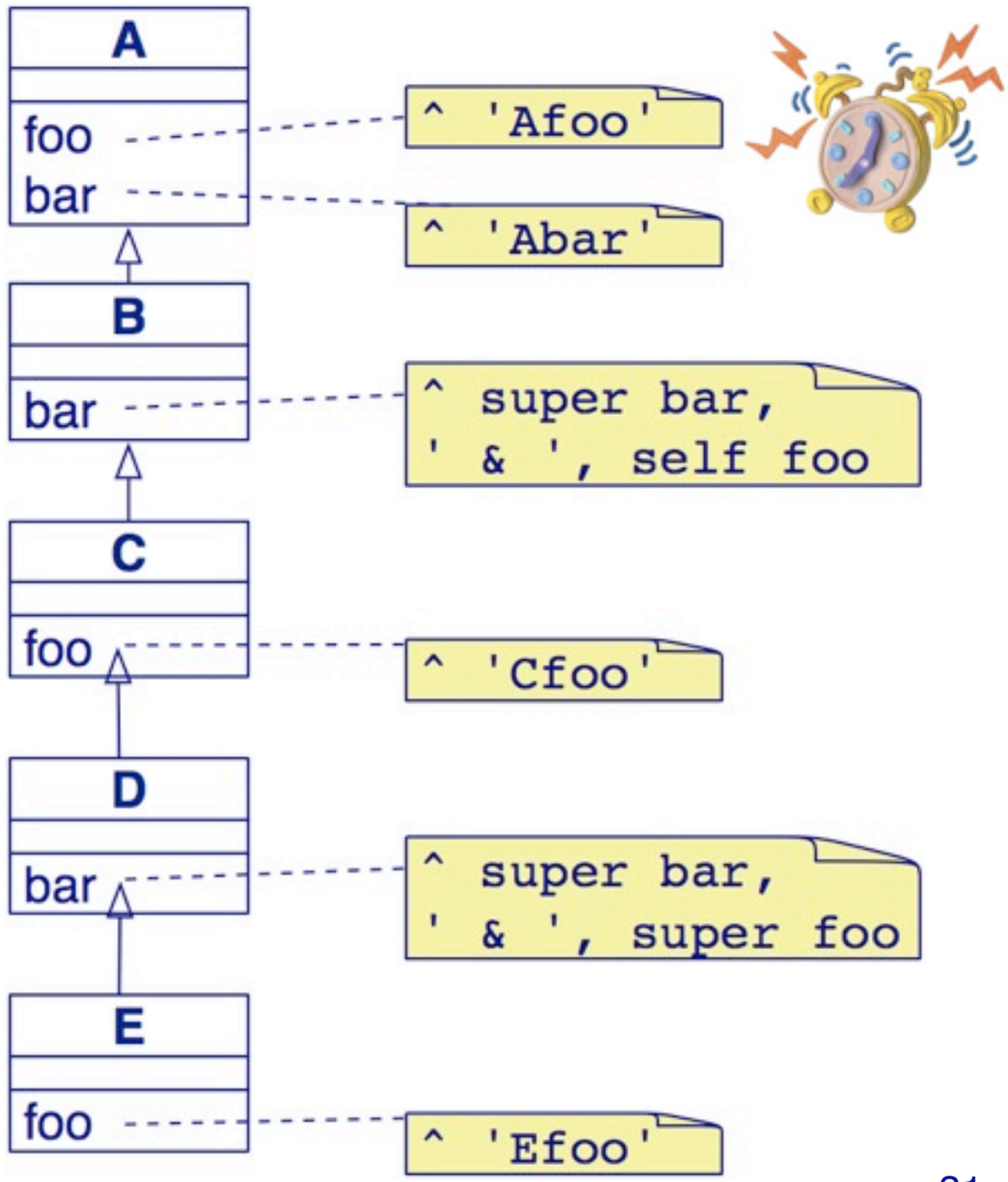
Super

- Super modifies the usual method lookup to *start in the superclass of the class whose method sends to super*
  - **NB:** lookup does *not* start in the superclass of the receiver!
    - *Cf.* `C new bar` *on next slide*
  - Super is not the superclass!

Super sends

```
A new bar
B new bar
C new bar
D new bar
E new bar
```

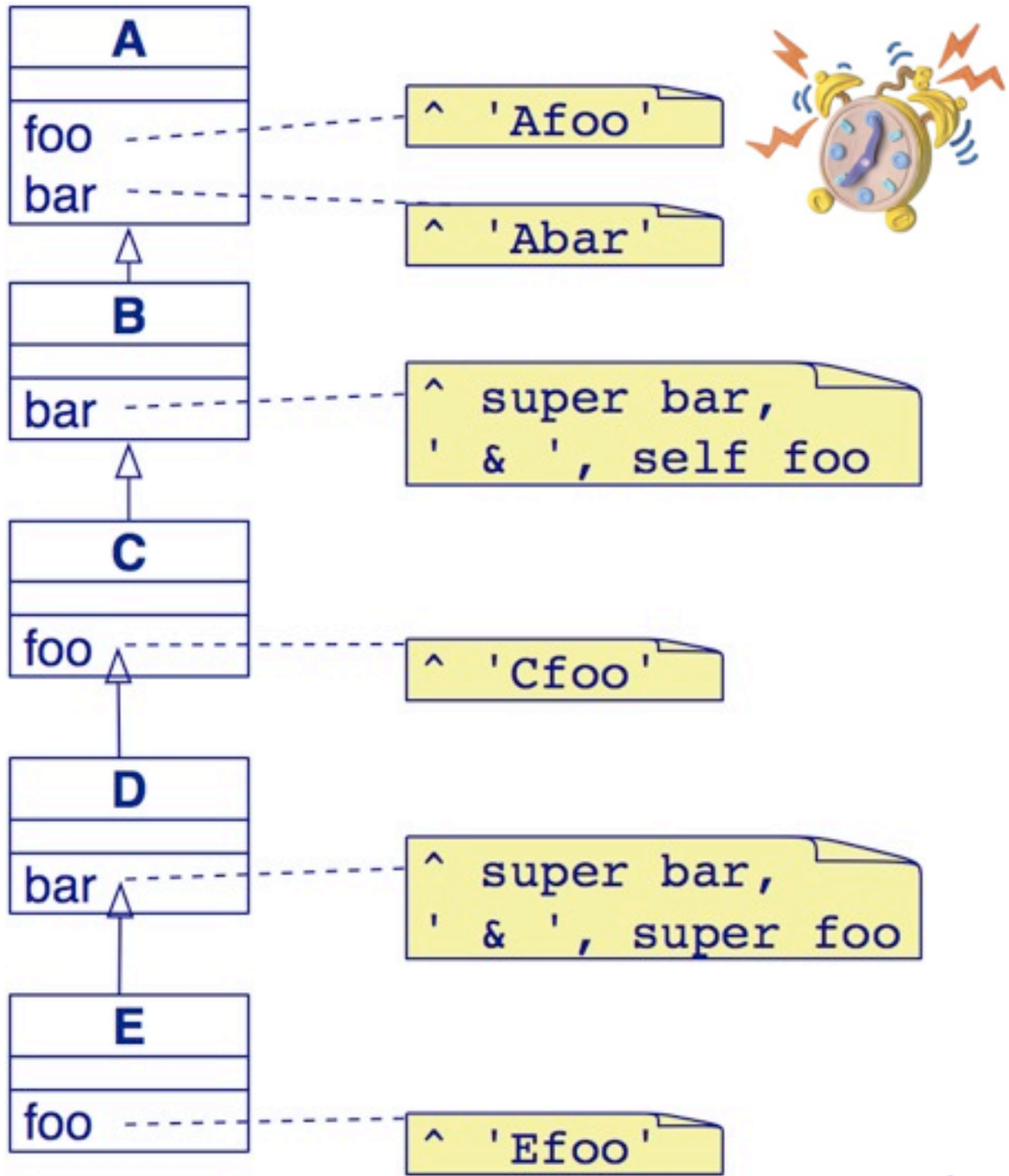
NB: It is usually a *mistake* to super-send to a different method. *D*>>`bar` should probably do `self foo`, not `super foo`!



Super sends

```
A new bar
B new bar
C new bar
D new bar
E new bar
```

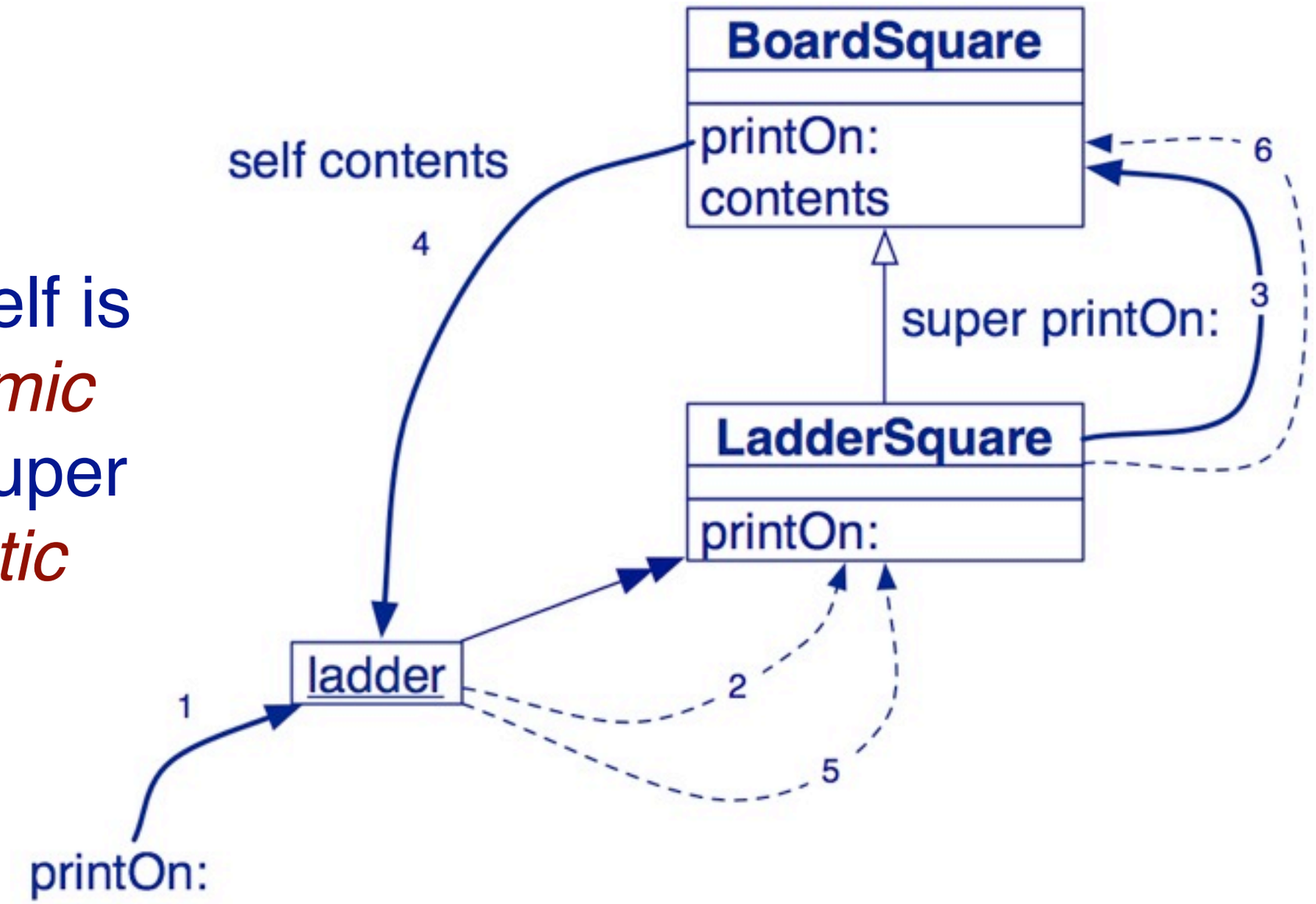
```
'Abar'
'Abar & Afoo'
'Abar & Cfoo'
'Abar & Cfoo & Cfoo'
'Abar & Efoo & Cfoo'
```



NB: It is usually a *mistake* to super-send to a different method. *D*>>`bar` should probably do `self foo`, not `super foo`!

Self and super

Sending to self is always *dynamic*
Sending to super is always *static*



Roadmap

- Common idioms
- Self and Super
- **Metaclasses in 7 points**

Metaclasses in 7 points

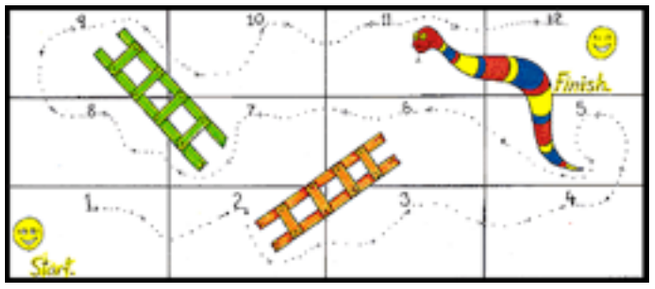
1. Every object is an instance of a class
2. Every class eventually inherits from Object
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass
7. The metaclass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

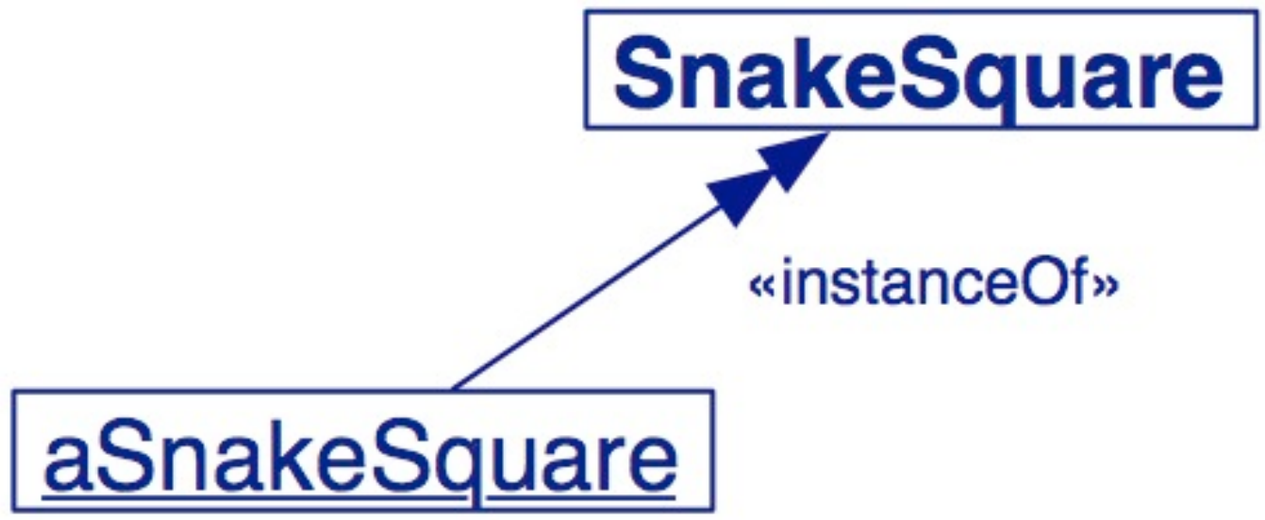
Adapted from Goldberg & Robson, Smalltalk-80 — The Language

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

1. Every object is an instance of a class





Remember the Snakes and Ladders Board Game …

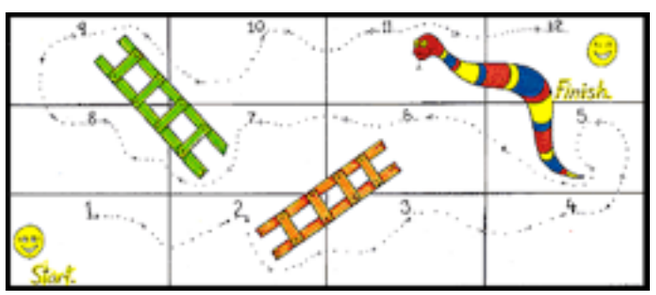
Metaclasses in 7 points

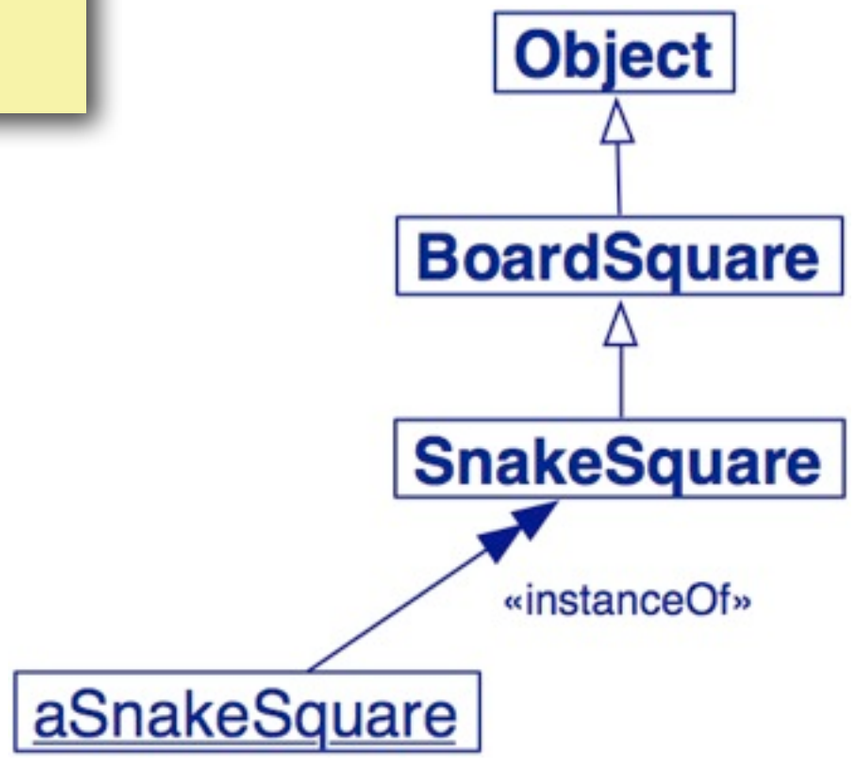
1. Every object is an instance of a class
2. **Every class eventually inherits from Object**
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass
7. The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

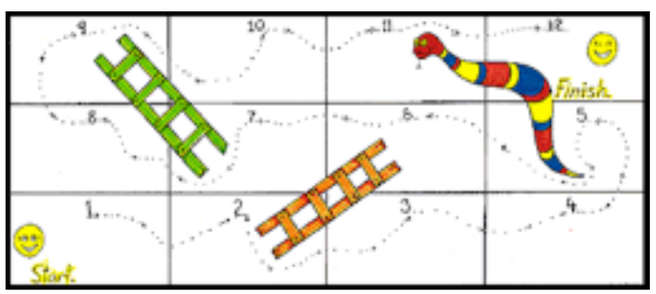


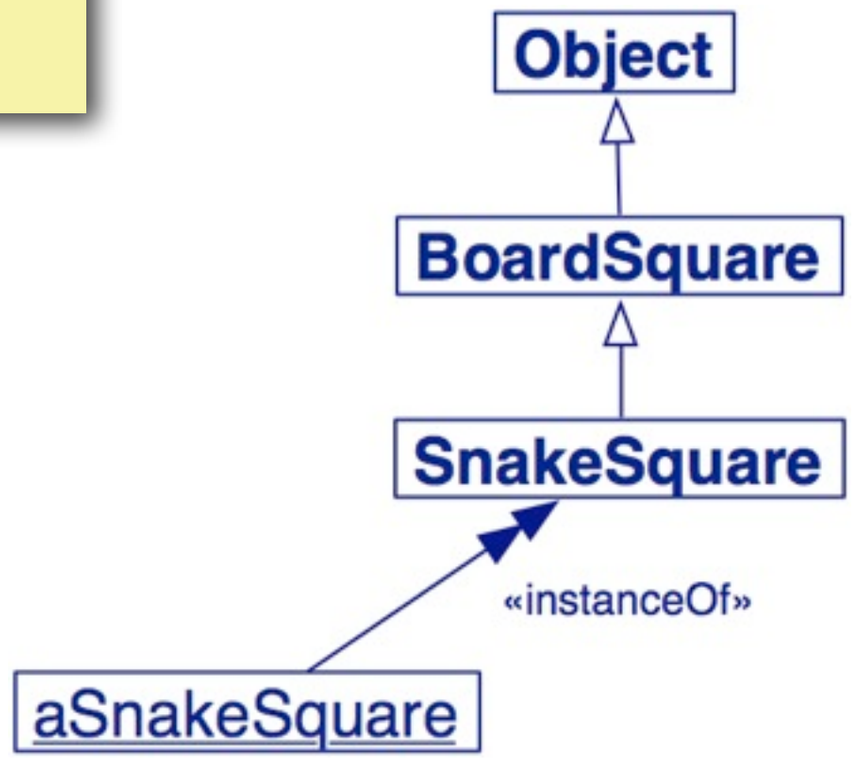


2. Every class inherits from Object

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

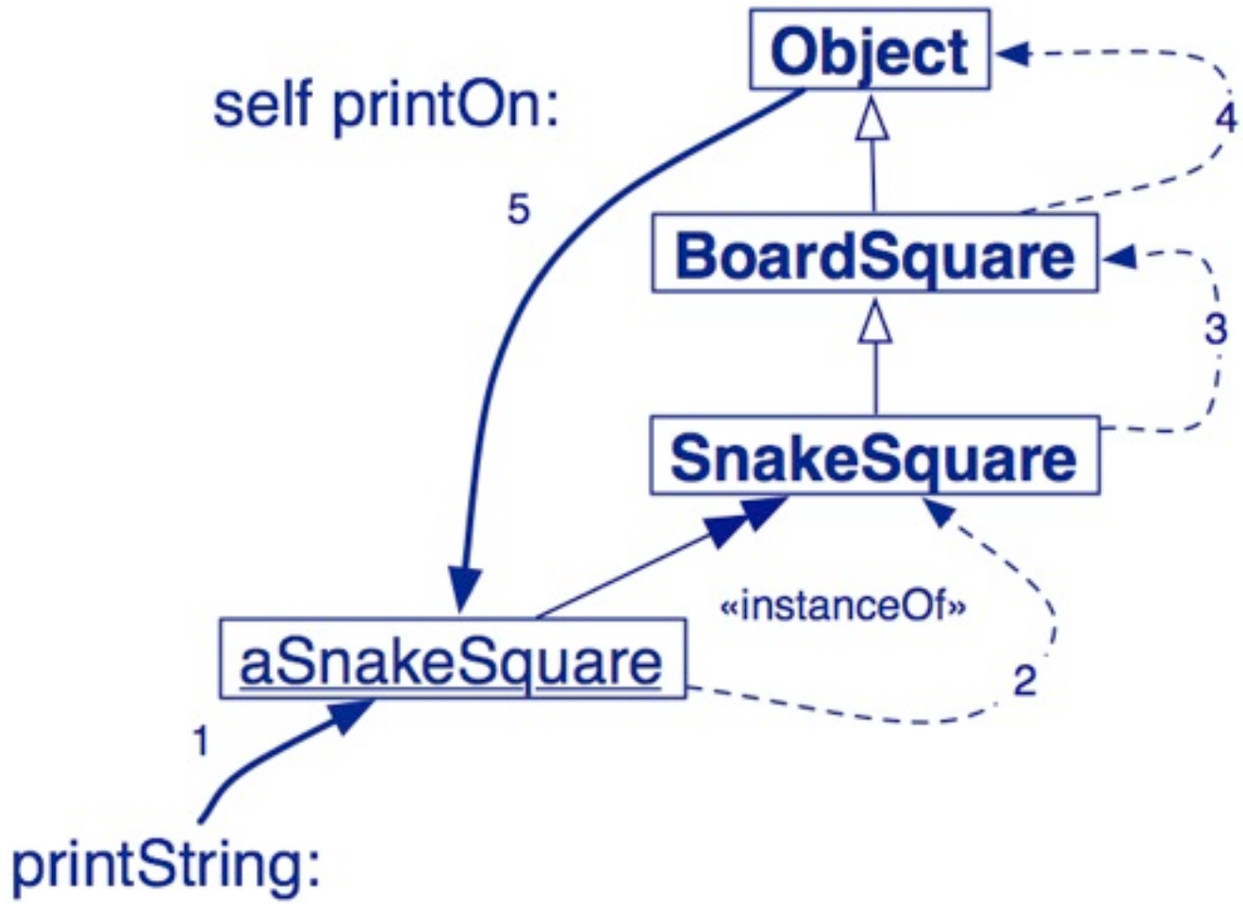




Caveat: in Pharo, Object has a superclass called ProtoObject

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

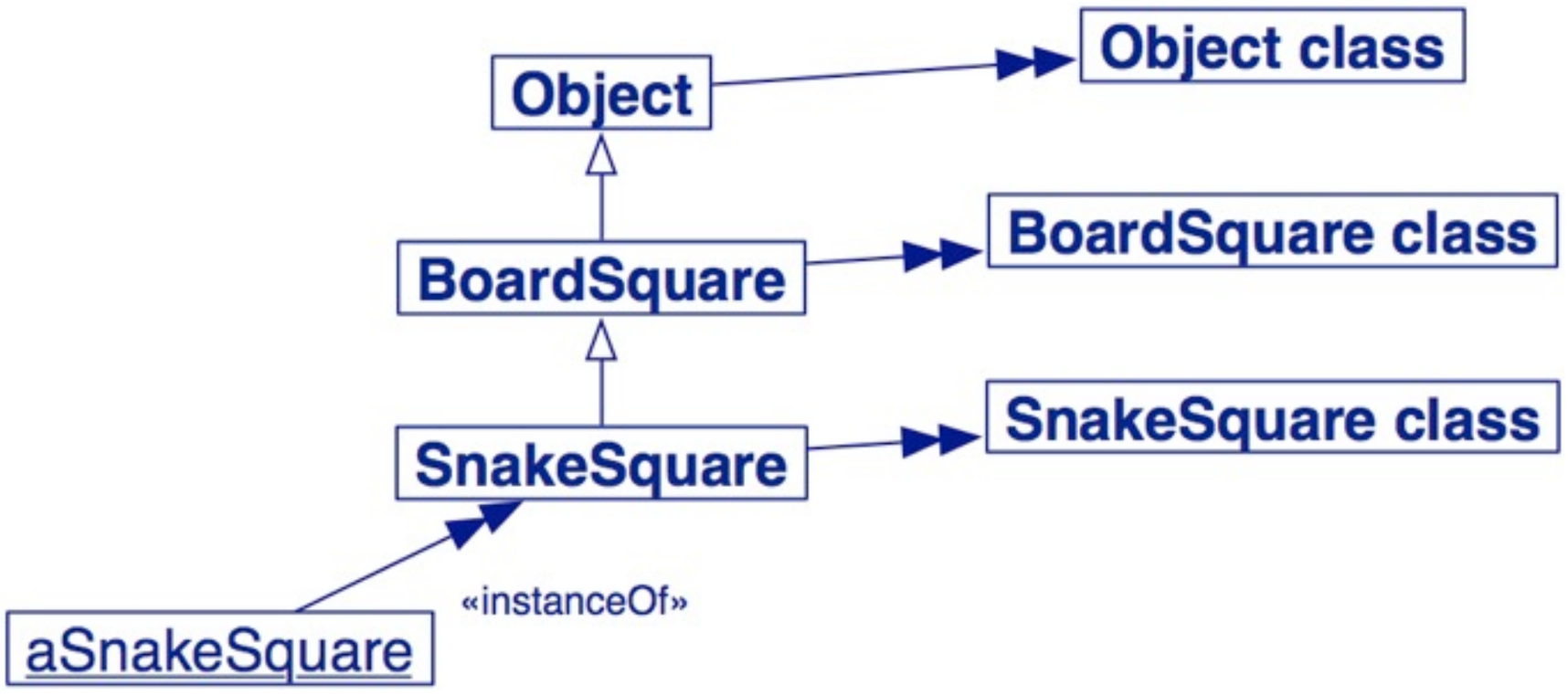
- **Object**
  - represents the common object behavior
  - error-handling, halting …
  - all classes should inherit ultimately from Object

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
```

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

```
SnakeSquare allInstances
SnakeSquare instVarNames
```

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

```
SnakeSquare back: 5
```

```
<-5[nil]
```

```
SnakeSquare selectors
```

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

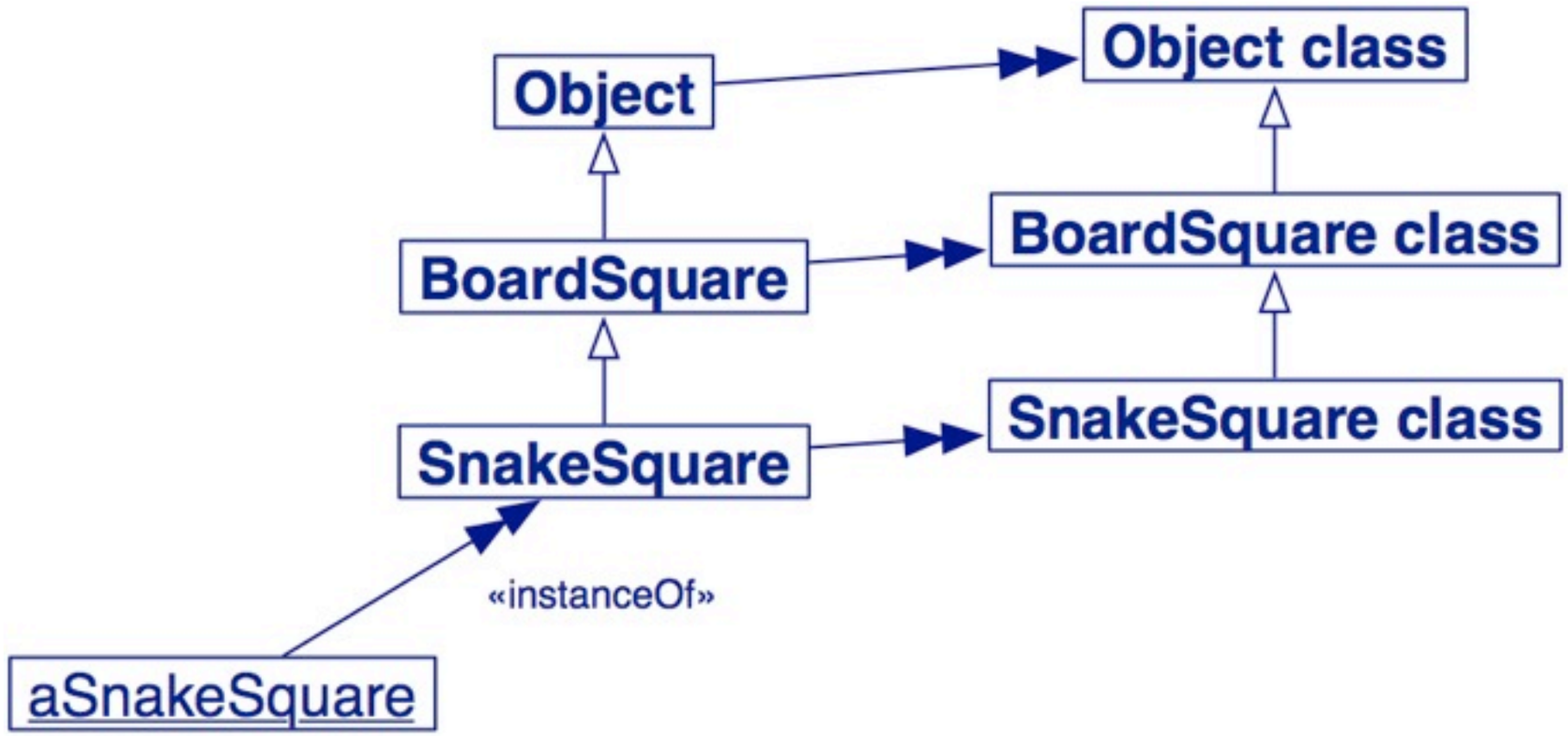
```
SnakeSquare canUnderstand: #new
SnakeSquare canUnderstand: #setBack:
```

```
false
true
```

Metaclasses in 7 points

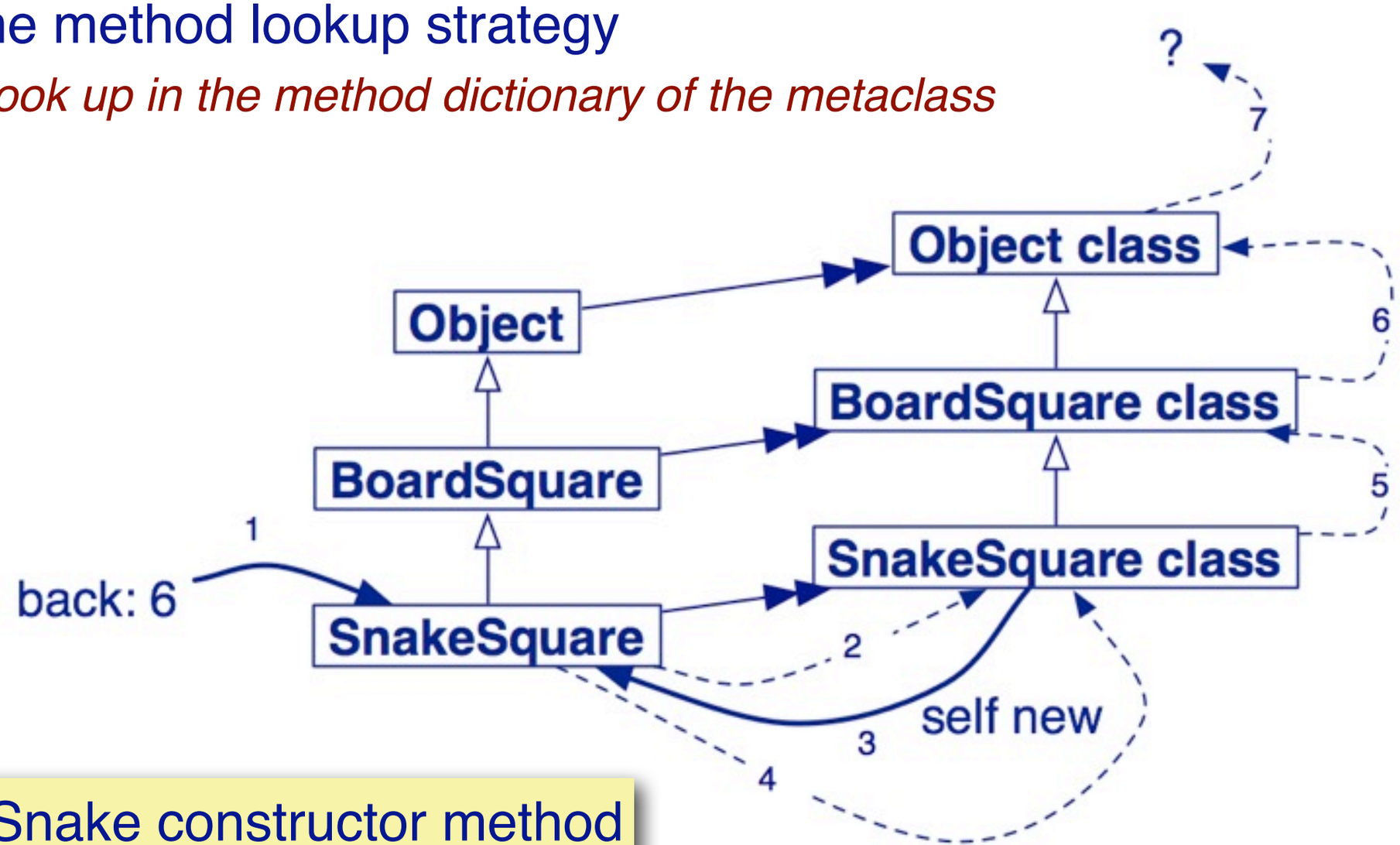
1. Every object is an instance of a class
2. Every class eventually inherits from Object
3. Every class is an instance of a metaclass
4. **The metaclass hierarchy parallels the class hierarchy**
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass
7. The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

4. The metaclass hierarchy parallels the class hierarchy



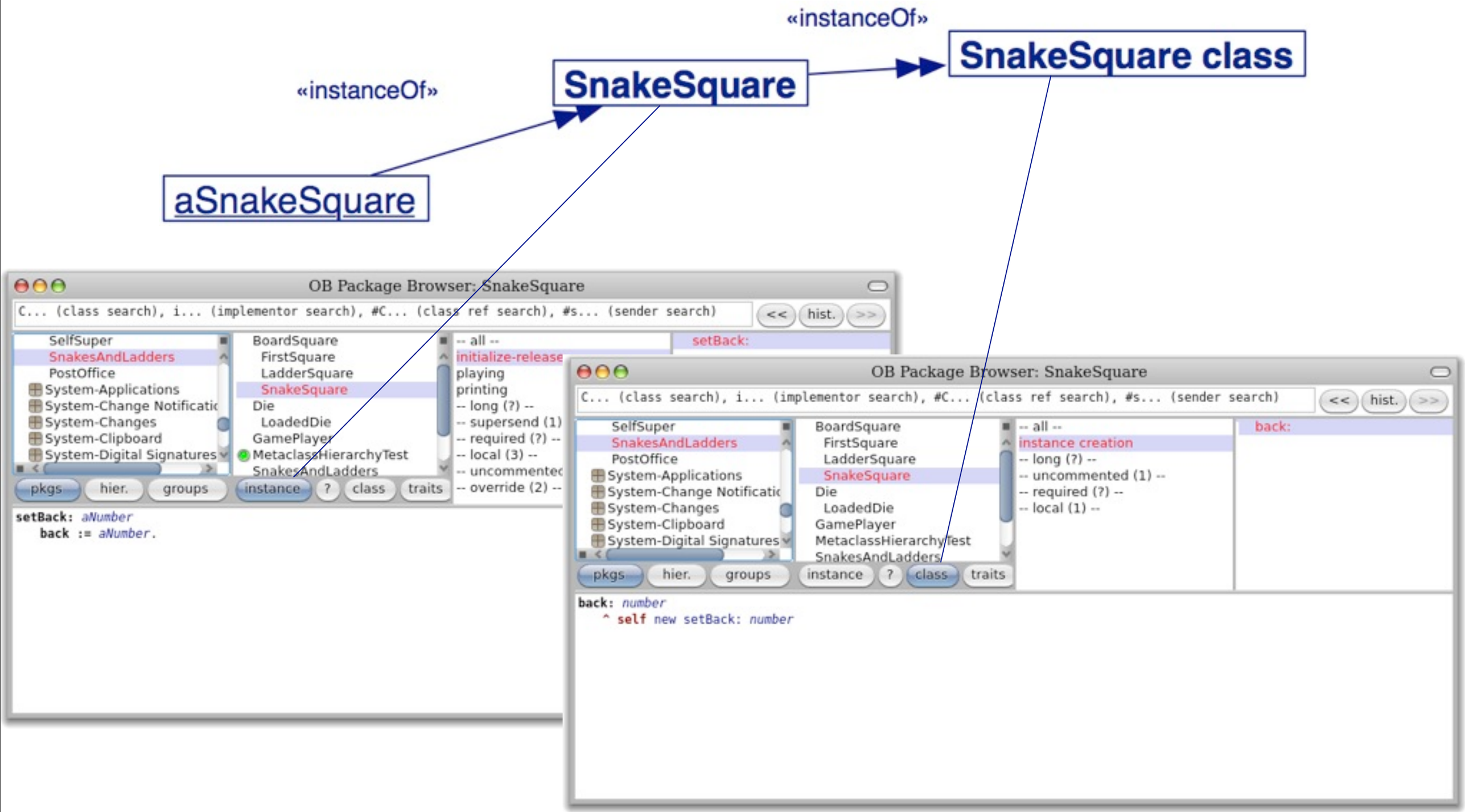
Uniformity between Classes and Objects

- 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

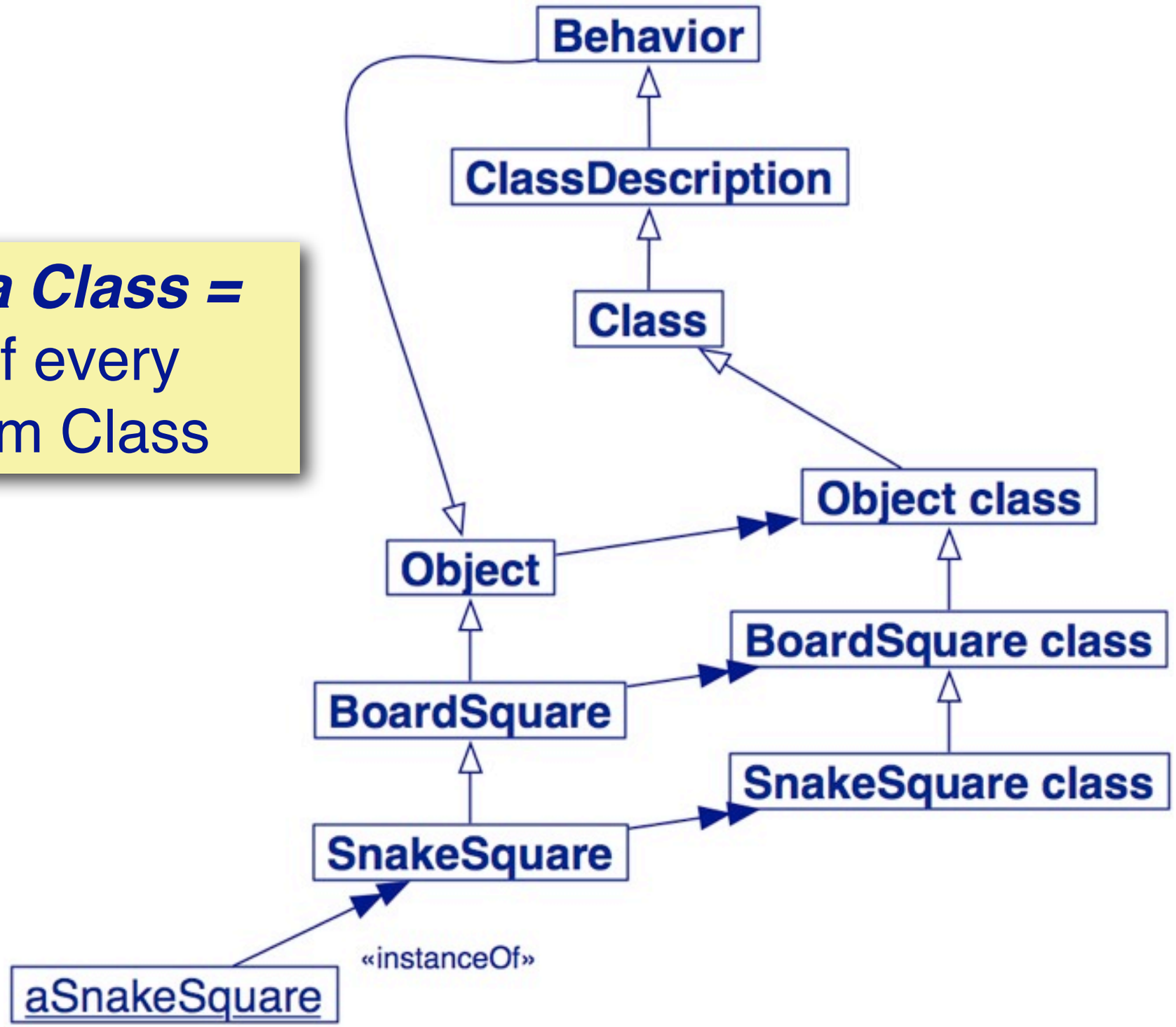


Metaclasses in 7 points

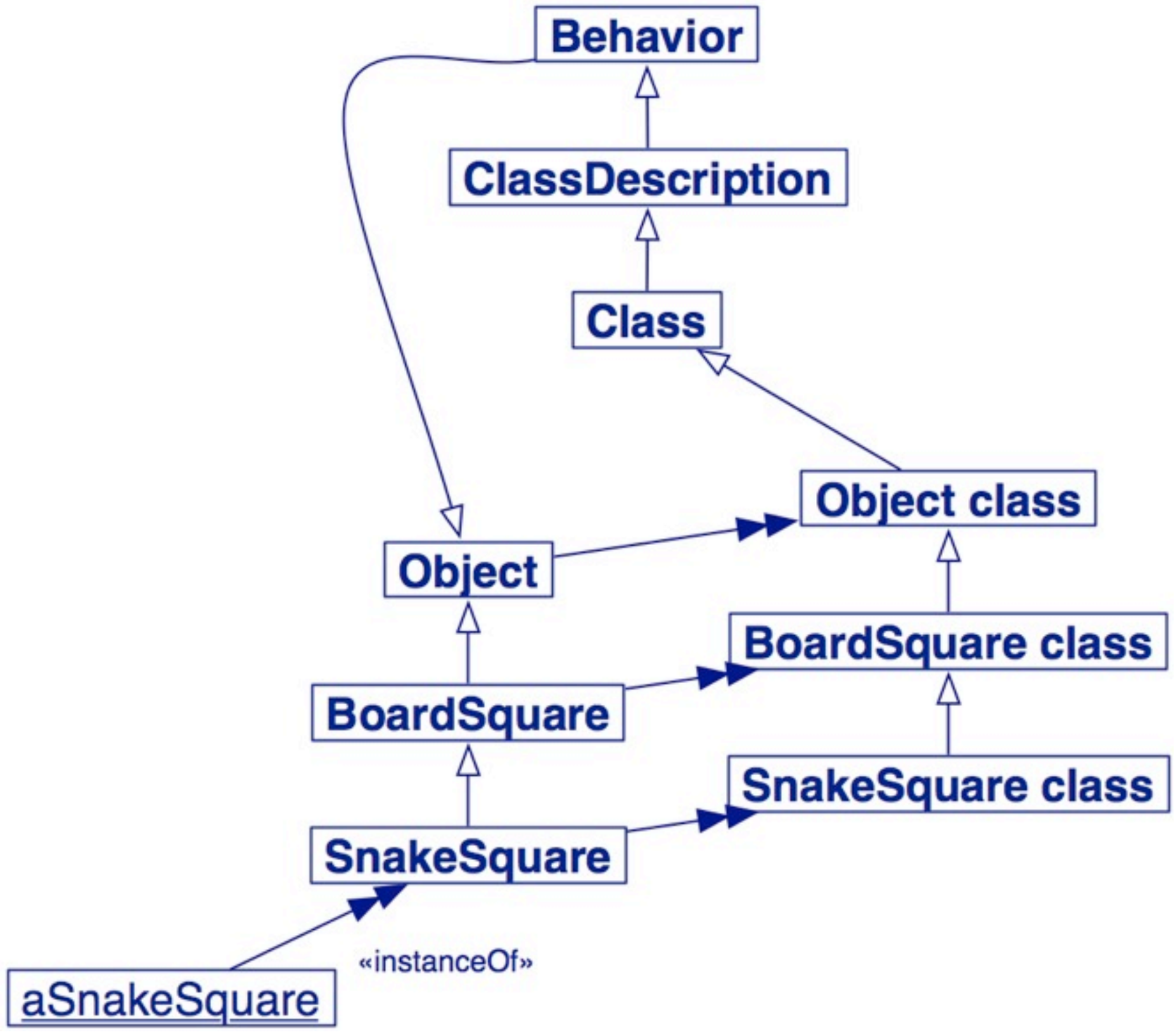
1. Every object is an instance of a class
2. Every class eventually inherits from Object
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. **Every metaclass inherits from Class and Behavior**
6. Every metaclass is an instance of Metaclass
7. The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



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.

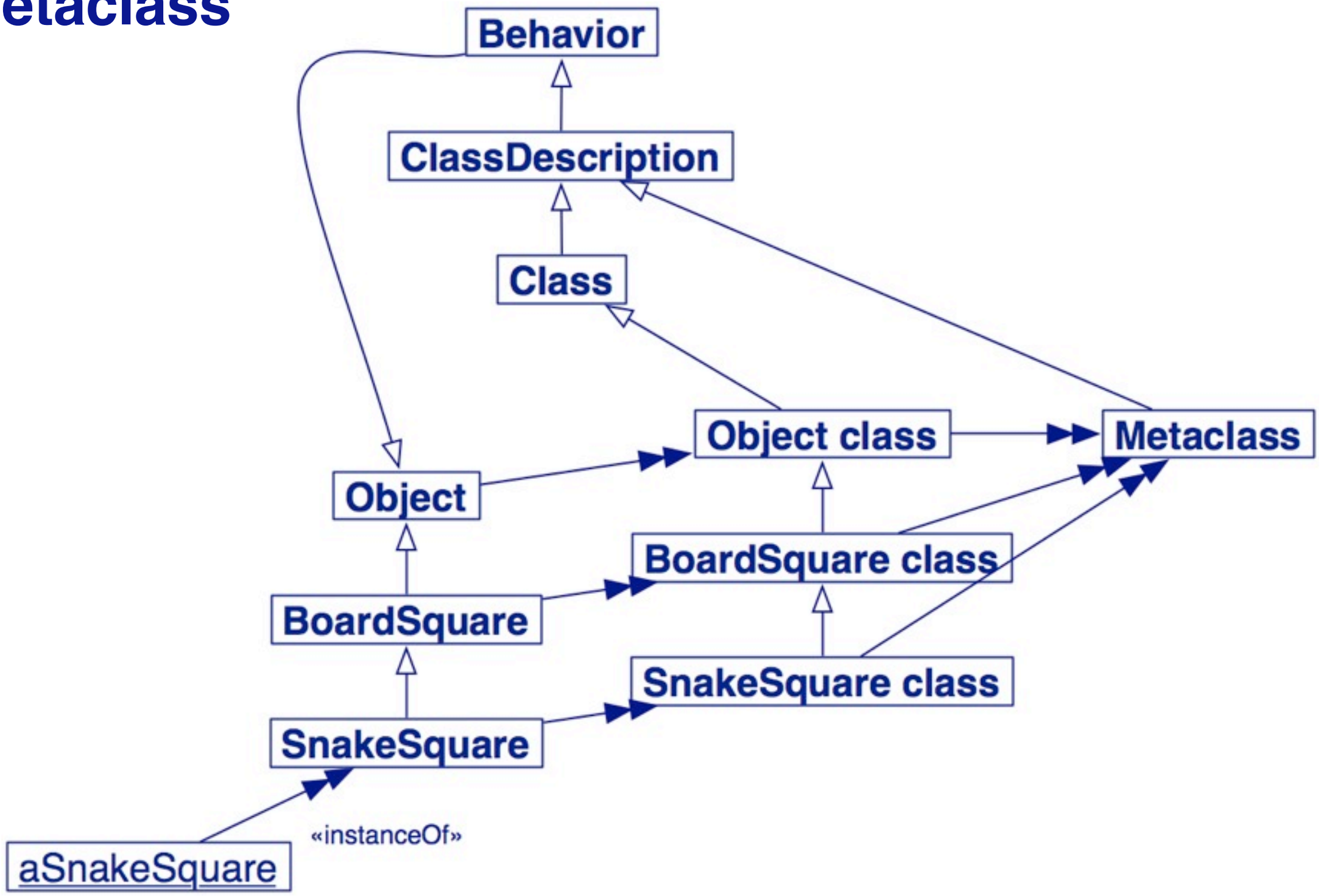
Responsibilities of Class

- **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



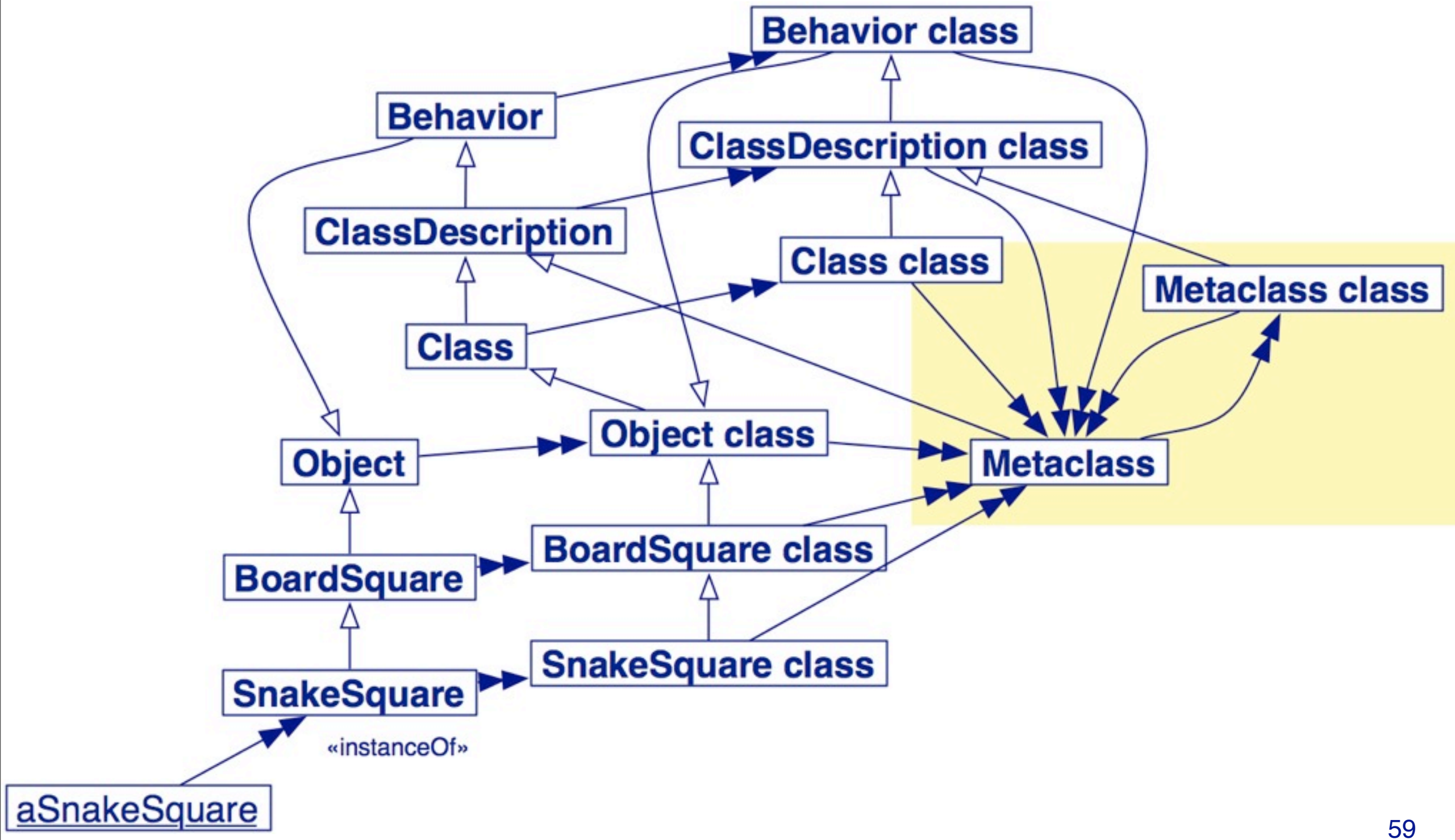
Metaclass Responsibilities

- **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
```

What you should know!

- How is a new instance of a class initialized?
- How is super static and self dynamic?
- Why is it usually a mistake for a method to super-send a different message?
- 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?

- When should you override new?
- When does self = super? When does super = self?
- What does self refer to in the method SnakesAndLadders class>>example?
- Why are there no explicit metaclasses?
- Why don’t metaclasses inherit from Class?
- Are there any classes that don’t inherit from Object?
- Is Metaclass a Class? Is it a Metaclass? Why or why not?
- Where are the methods class and superclass defined?