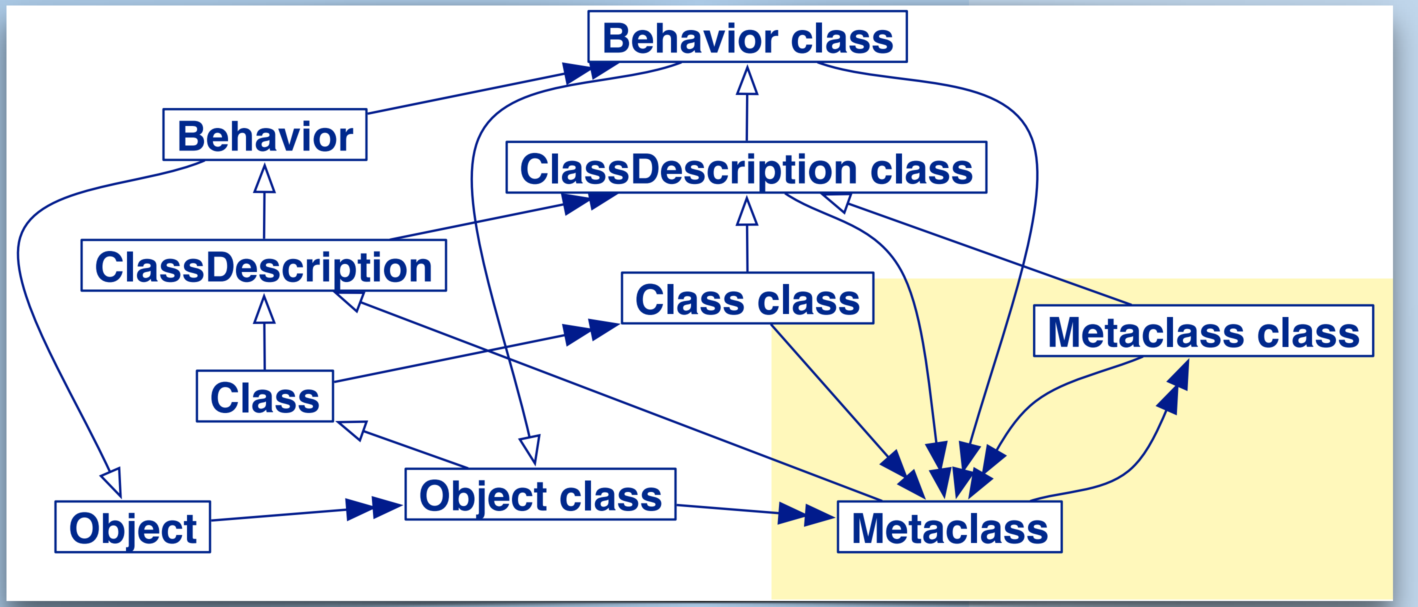
Understanding Classes and Metaclasses

Oscar Nierstrasz



Selected material courtesy Stéphane Ducasse

Birds-eye view

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

Smalltalk is a fully reflective system in which its metamodel is reified and accessible within the run-time system. You can interact with these reified entities to query and change the system at run time.

We will use a running example of a Snakes and Ladders game implementation to explore the metamodel of the game and of the Smalltalk system itself.

Roadmap

> OO modeling idioms
> Understanding `self` and `super`
> Metaclasses in 7 points

Roadmap

> **OO modeling idioms**
> Understanding `self` and `super`
> Metaclasses in 7 points

Programming is modeling

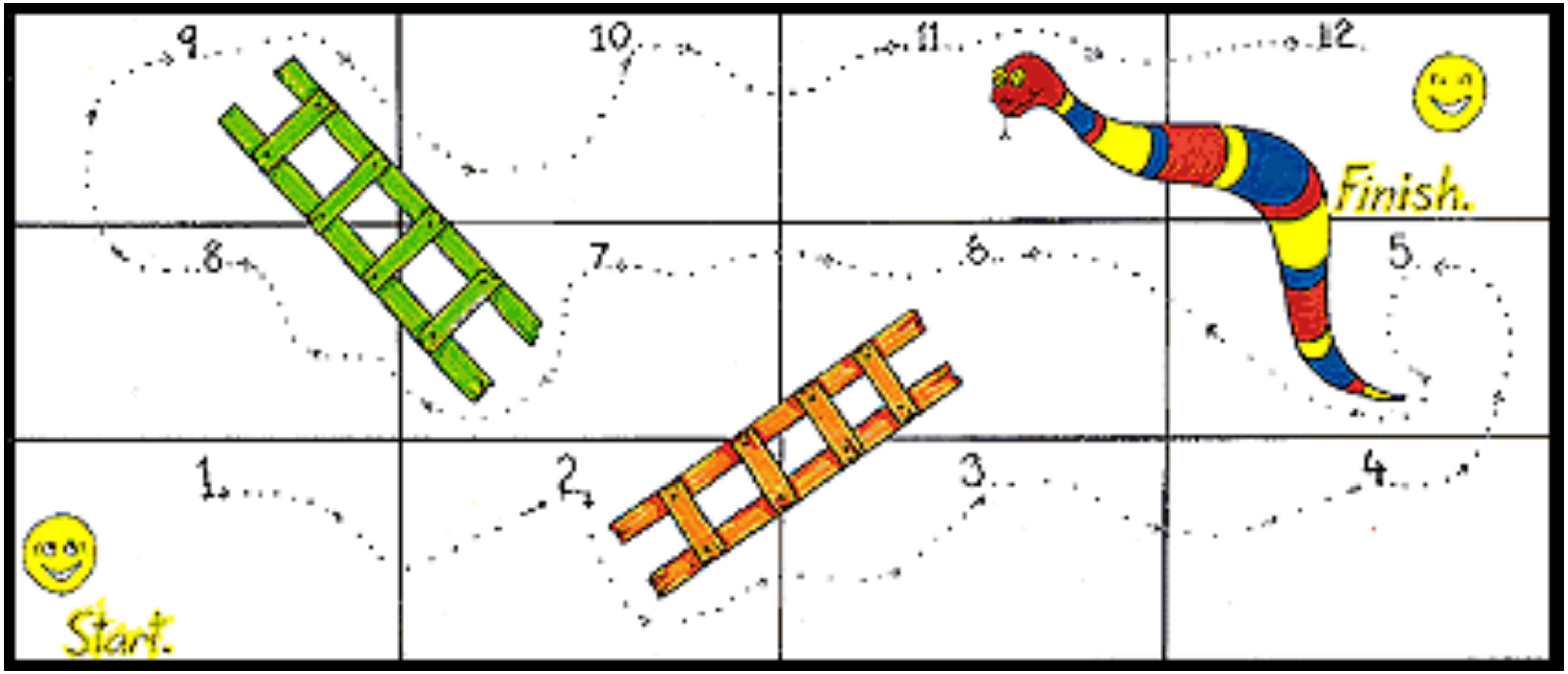
- Well-designed OO code obeys certain quality principles:
  - Model domain concepts
  - Use inheritance to model specialization and polymorphism
  - Distribute responsibilities
  - Code behaviour declaratively
  - Test comprehensively
  - ...

All software programs can be thought of as *executable models*. They apply a particular programming paradigm (OO, functional, logic-based etc.) to express domain concepts as well as to model a solution to the task at hand.

In order to analyze a software system we need to understand what principles are being used to express models in software.

In this lecture we will first look at some of the idioms and design principles used to model the Snakes and Ladders game in code, and then afterwards we will explore the OO metamodel (of Smalltalk) behind it.

Snakes and Ladders



Snakes and Ladders is a children's game that is not very interesting for adults to play as no strategy is required — the players just alternate in rolling a die and following the rules of the game to get to the final square first. It is just interesting for children to learn how to play a game with fixed rules. On the other hand it is sufficiently complex to make it interesting to implement as an object-oriented programming exercise.

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

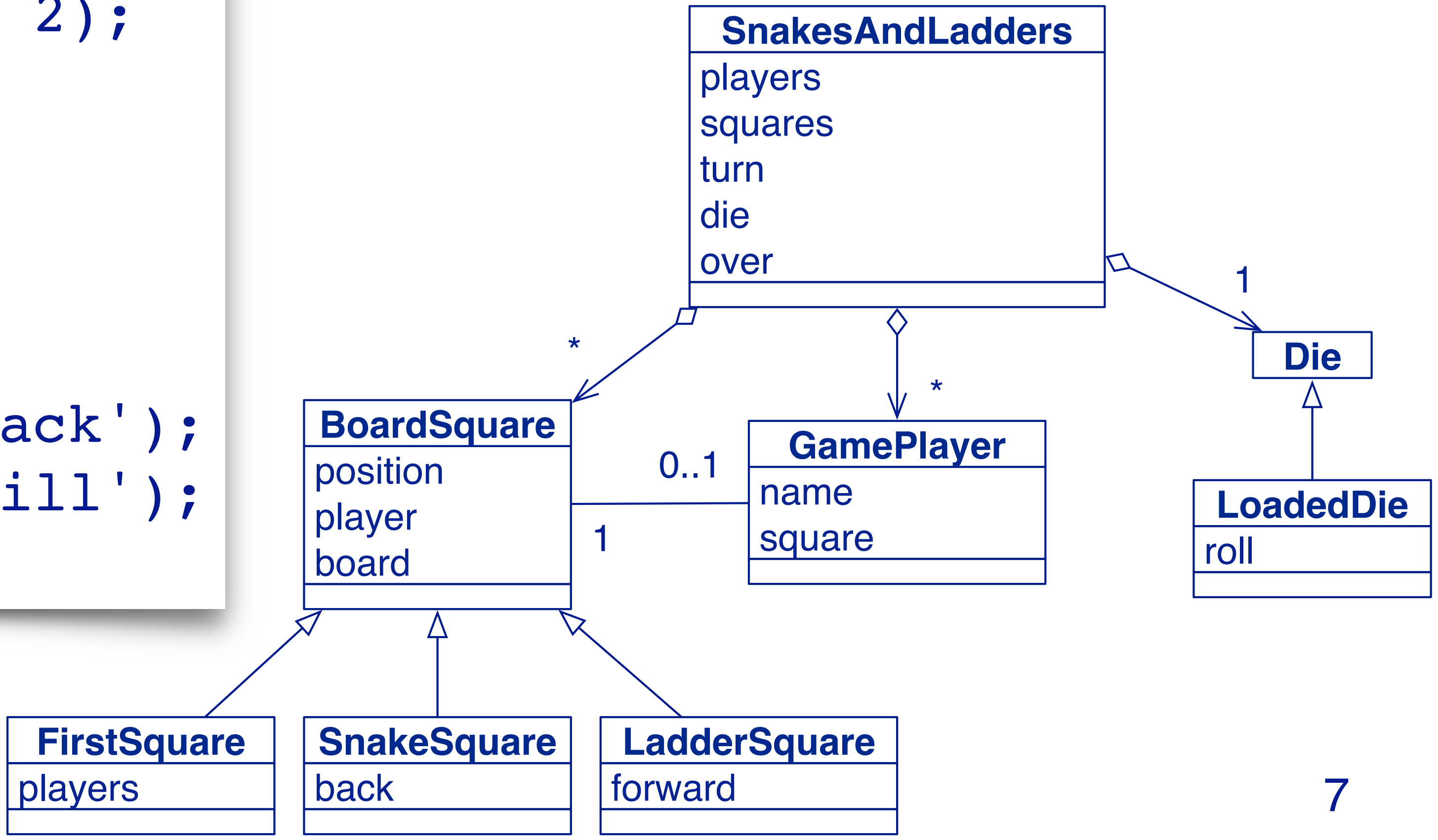
Students who have followed the first-year Bachelor's Object-Oriented Programming course, P2, may recall the Java version of the game.

We will implement the game in Smalltalk to illustrate and explore the relationships between *objects*, *classes* and *metaclasses* in the Smalltalk language metamodel.

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
```

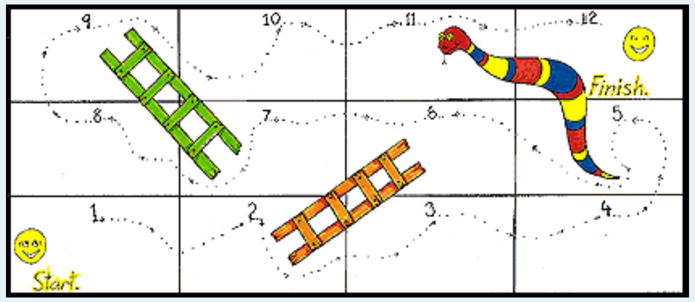
- > Construct the board
- > Add some players
- > Play the game

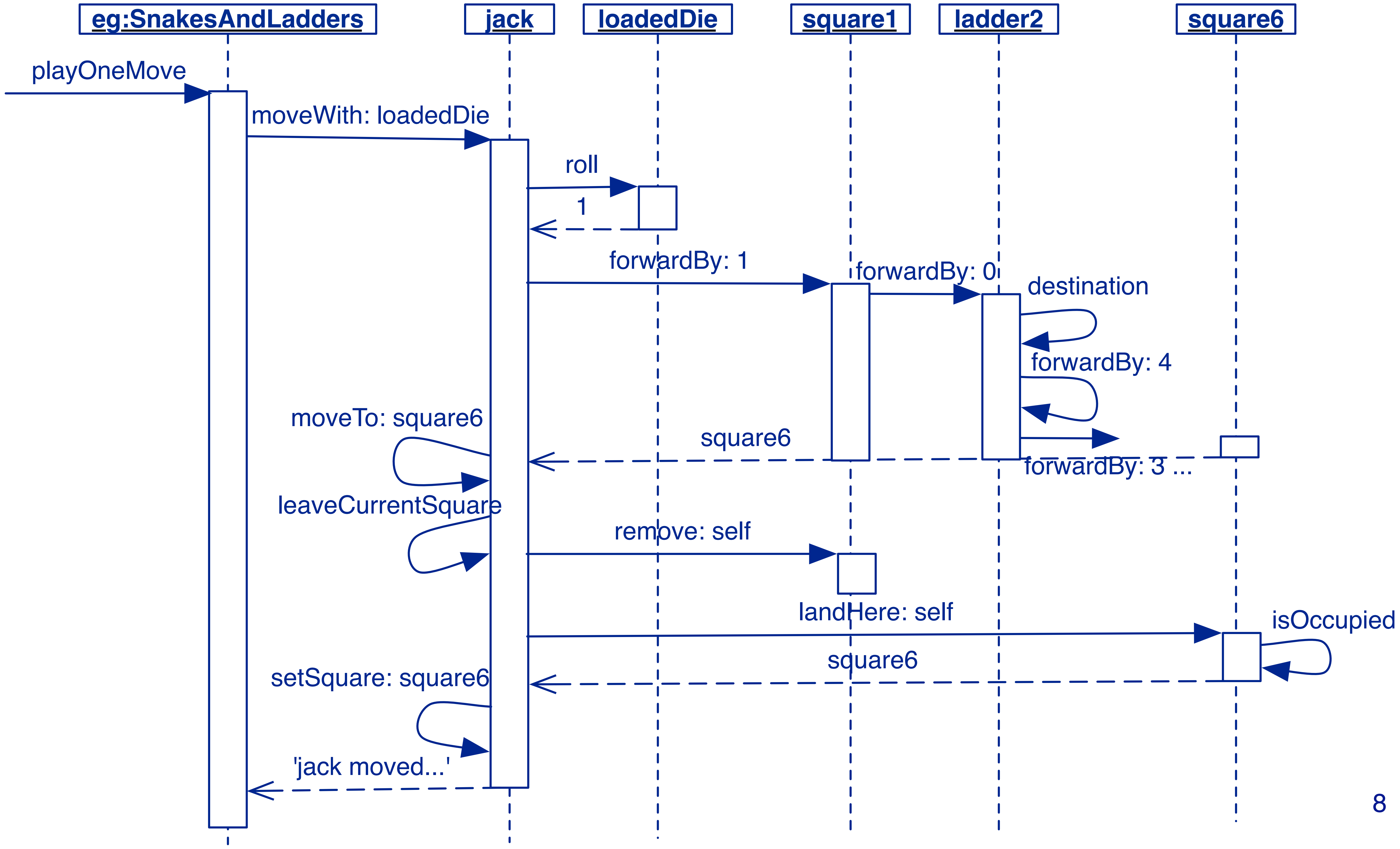


We develop the game top-down be specifying a class-side script as a concrete scenario. To play the game, we must first *configure the board* as a sequence of different kinds of squares (the first square, regular squares, snakes, and ladders), then we *add the players*, and finally we send the message `playToEnd`.

Note the use of a cascade (;) to send a series of messages to an object. Also note the final message `yourself`, which returns the object itself, rather than whatever the previous message might return as a result. (This is a common Smalltalk idiom.)

Distributing responsibilities

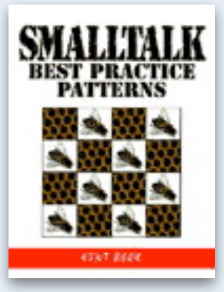




There are many possible ways to implement snakes and ladders. Here we adopt an extreme OO design in which we model all the objects of the domain and distribute responsibilities to each of them. To play the game, the message `playOneMove` is sent repeatedly to the game until the game is over. Then:

- the example game (eg) receives the message `playOneMove`
- the game asks the current play to move with a die (a loaded die is used for test cases)
- the player rolls the die, yielding a number N
- it asks its current square to help it move forward N squares
- that square asks the next square, and so on
- the target square then computes the destination (itself, unless it is a snake or a ladder)
- the player leaves its current square and moves to its destination
- if the destination is occupied, the player is transported to the first square

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***

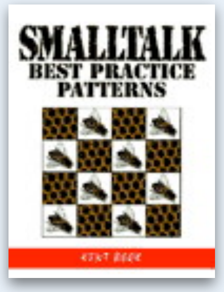
— “Good code invariably has small methods and small objects. Only by factoring the system into many small pieces of state and function can you hope to satisfy the ‘once and only once’ rule.”

Kent beck has written a wonderful book of Smalltalk design patterns and idioms, called “Smalltalk best Practice Patterns” (1997).

A draft version can be found online:

- http://stephane.ducasse.free.fr/FreeBooks/BestSmalltalkPractices/Draft-Smalltalk%20Best%20Practice%20Patterns%20Kent%20Beck.pdf
- http://scgresources.unibe.ch/Literature/Books/Beck97aDraftSmalltalkBestPracticePatterns.pdf

Composed Method



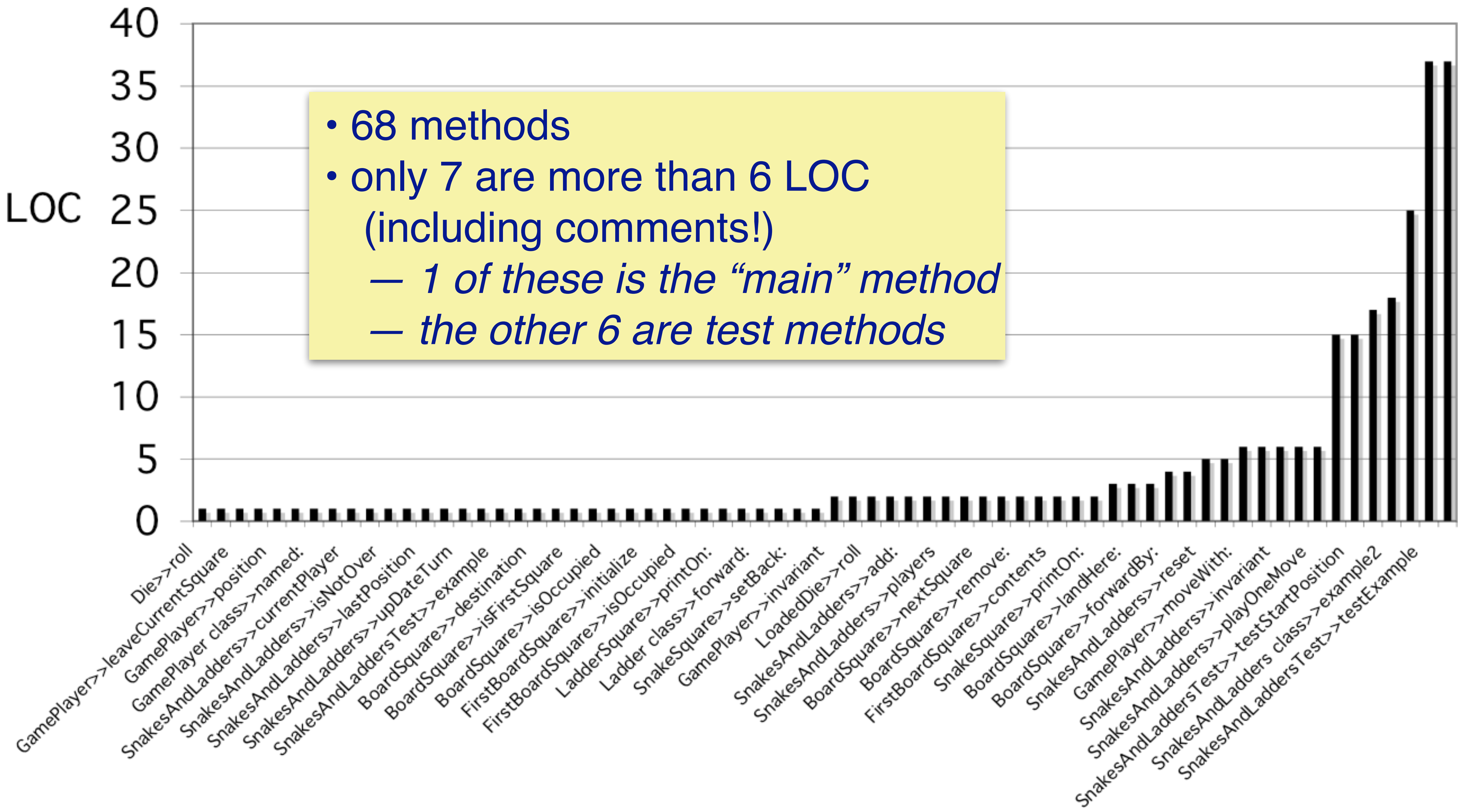
How do you divide a program into methods?

> 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 Laddders methods



- 68 methods
- only 7 are more than 6 LOC (including comments!)
  - *1 of these is the “main” method*
  - *the other 6 are test methods*

Our implementation of Snakes and Ladders follows the "Lots of Little Pieces" idiom. This makes it easy to understand and to implement correctly each method.

The longest methods are test methods (scripts) and one algorithm (the "main" method).

`SnakesAndLadders>>playToEnd` is still just 15 LOC, including comment and blank lines.

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)?

Smalltalk differs from most OO languages in that the abstraction boundary is the object, not the class. Since *a class is an object*, it is *distinct* from its instances has *has no access to their private state*. On the other hand, a class exists both to provide behaviour to its instances and to provide a way to create them. So how can a class initialise its instances?

The answer is that it does not. It can create an instance, but then must use instance methods to complete the initialisation process. This is very different from the way objects are constructed in languages like Java.

Explicit Initialization

How do you initialize instance variables to their default values?

> Implement a method `initialize` that sets all the values explicitly.

> This will be called *automatically* (in Pharo) when new instances are created

```
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`!

Actually Beck’s book says “Override the class message `new` to invoke it on new instances”. In some versions of Smalltalk this is necessary, but in Pharo this is done automatically by the default implementation of `new` in the class `Behavior`.

You are free to reimplement `new` for your own classes, but you are strongly advised to never override `basicNew` or any method starting with “`basic`”.

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.

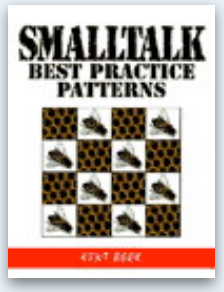
```
LadderSquare class>>forward: number
	^ self new initializeForward: number
```

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

Note how these class side “constructors” send `new` to themselves to create an instance and then use ordinary instance methods to complete the initialisation. This is (almost) the only way that the class can initialize state of new instances.

Aside: why send “self new” instead of “`LadderSquare new`*” or “*`SnakeSquare new`*”?*

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.

Better yet, use “`initialize`” as the prefix

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

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

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

“`set`” as a prefix can be confused nowadays as a Java-style setter method. By using “`initialize`” as the prefix we make the intent clear, and communicate that the methods should only be used for initialization, and never as an accessor.

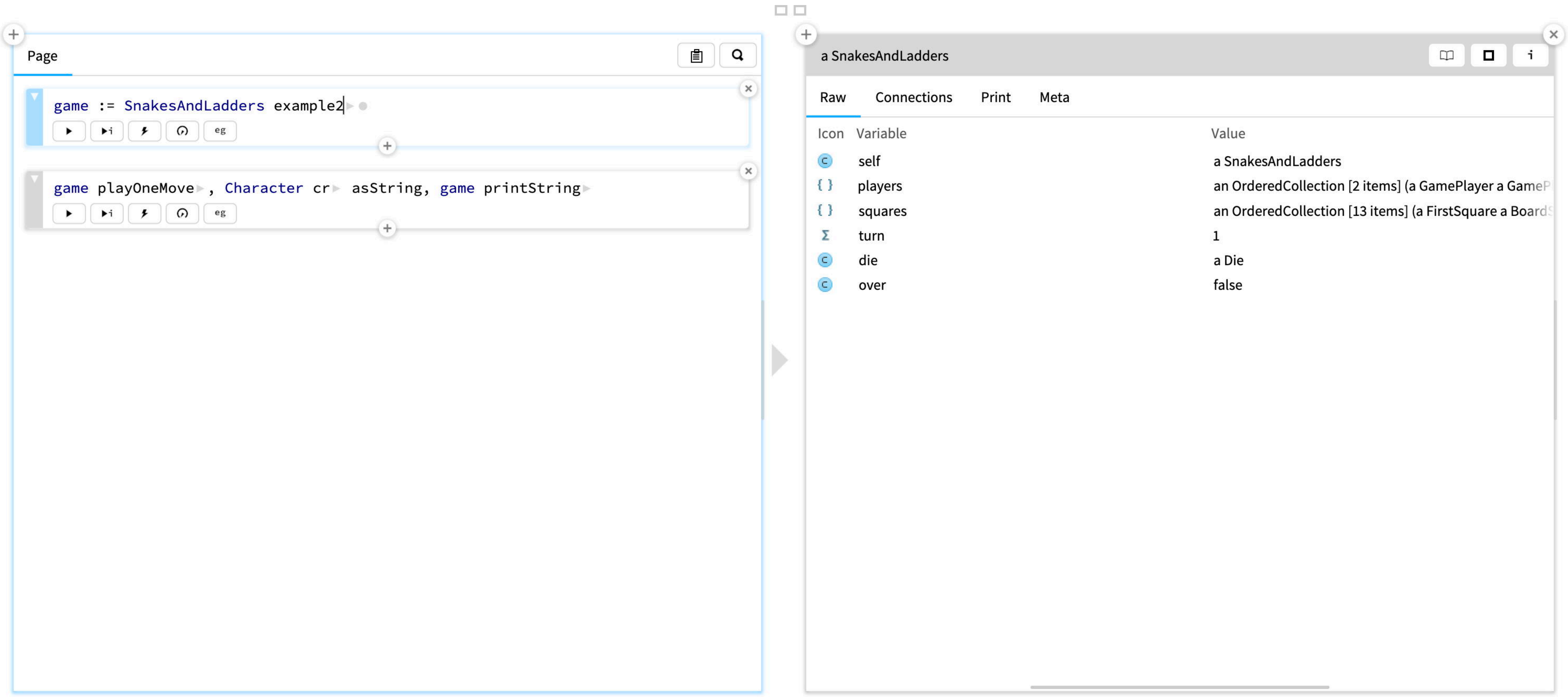
Debug Printing Method

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



Page

```
game := SnakesAndLadders example2
```

```
game playOneMove , Character cr asString, game printString
```

a SnakesAndLadders

Icon	Variable	Value
C	self	a SnakesAndLadders
{ }	players	an OrderedCollection [2 items] (a GamePlayer a GameP
{ }	squares	an OrderedCollection [13 items] (a FirstSquare a Board
Σ	turn	1
C	die	a Die
C	over	false

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

gt

The default view given by the Inspector gives no insight into the actual state of the game. To fix this we should make each object responsible for providing a textual representation of its state.

Whereas in Java we would implement `toString()`, in Smalltalk we implement `#printOn:`

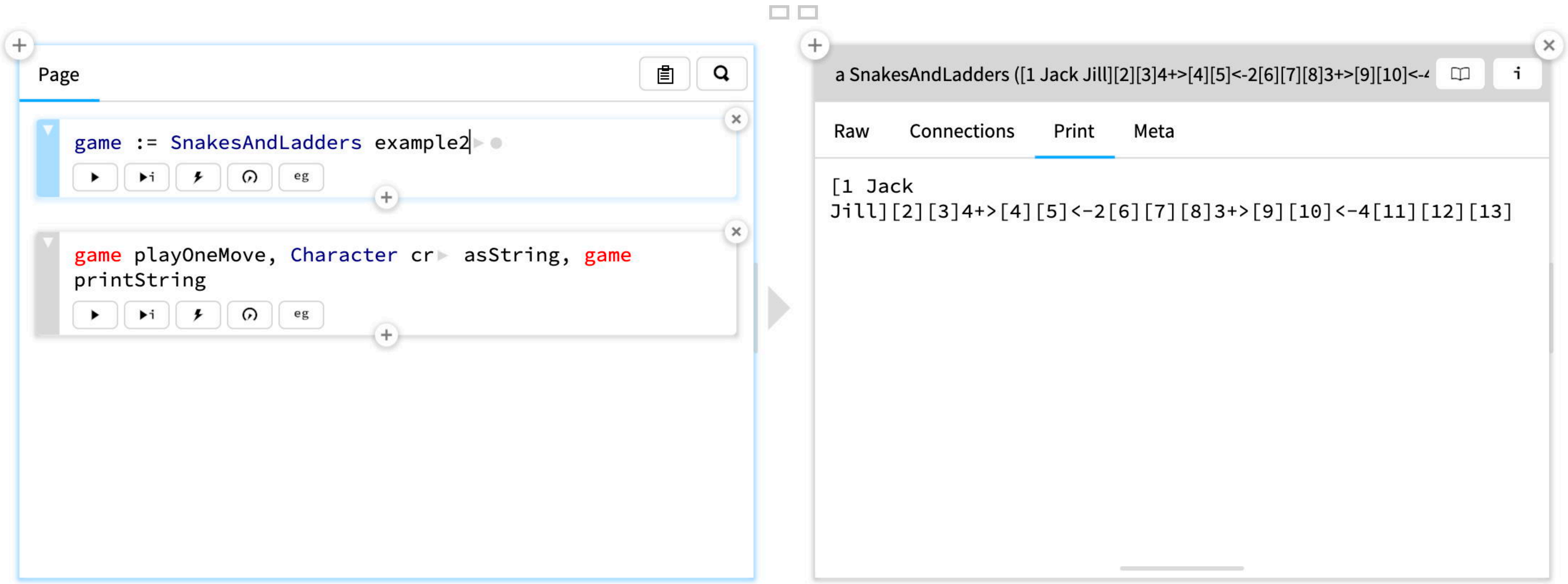
Implementing printOn:

```
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
```

The `#printOn:` method expects a `Stream` object as its argument. Send `#nextPutAll:` to print a `String` to a `Stream`. (Browse the class for other streaming methods.)

Note the use of `super` sends to compose the `#printOn:` methods of `BoardSquare` and its subclasses.

Viewing the game state



```
game := SnakesAndLadders example2
```

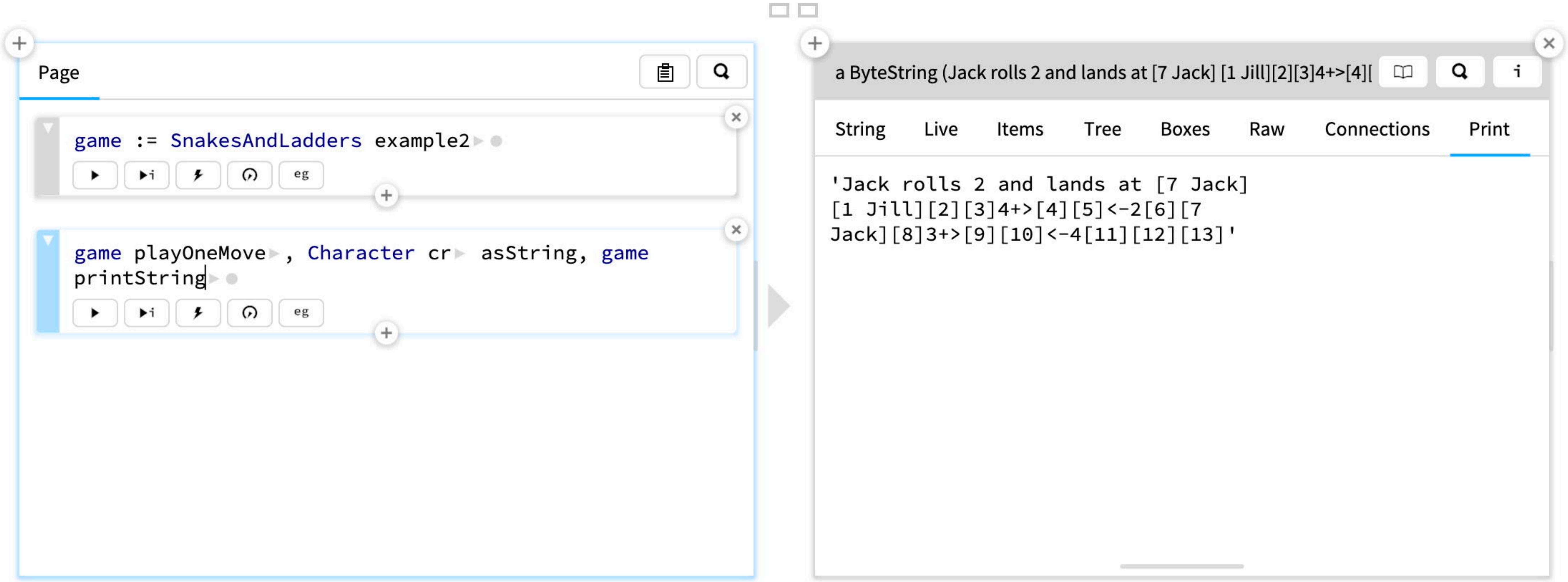
```
game playOneMove, Character cr asString, game
printString
```

a SnakesAndLadders ([1 Jack Jill][2][3]4+>[4][5]<-2[6][7][8]3+>[9][10]<-4
Raw
Connections
Print
Meta

```
[1 Jack
Jill][2][3]4+>[4][5]<-2[6][7][8]3+>[9][10]<-4[11][12][13]
```

It may not be beautiful, but now we get a useful Inspector view of the game state, showing the layout of the board and the position of the players.

Interacting with the game



```
game := SnakesAndLadders example2
```

```
game playOneMove , Character cr asString, game
printString
```

a ByteString (Jack rolls 2 and lands at [7 Jack] [1 Jill][2][3]4+>[4][

```
'Jack rolls 2 and lands at [7 Jack]
[1 Jill][2][3]4+>[4][5]<-2[6][7
Jack][8]3+>[9][10]<-4[11][12][13]'
```

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

NB: In order to refresh the textual view of “self”, you need to click on the “refresh” icon.

With bit more effort, a dedicated “moldable” view can be generated in the Playground, but that falls out of the scope of this simple example …

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.

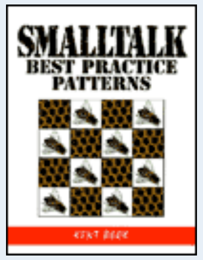
Some query methods

```
SnakesAndLadders>>isNotOver
   ^ self isOver not

BoardSquare>>isFirstSquare
   ^ position = 1

BoardSquare>>isLastSquare
   ^ position = board lastPosition

BoardSquare>>isOccupied
   ^ player notNil

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

Roadmap

> OO modeling idioms
> **Understanding `self` and `super`**
> Metaclasses in 7 points

Super



How can you invoke superclass behaviour?

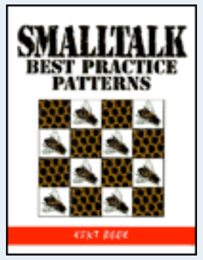
> Invoke code in a superclass explicitly by sending a message to `super` instead of `self`.

— 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

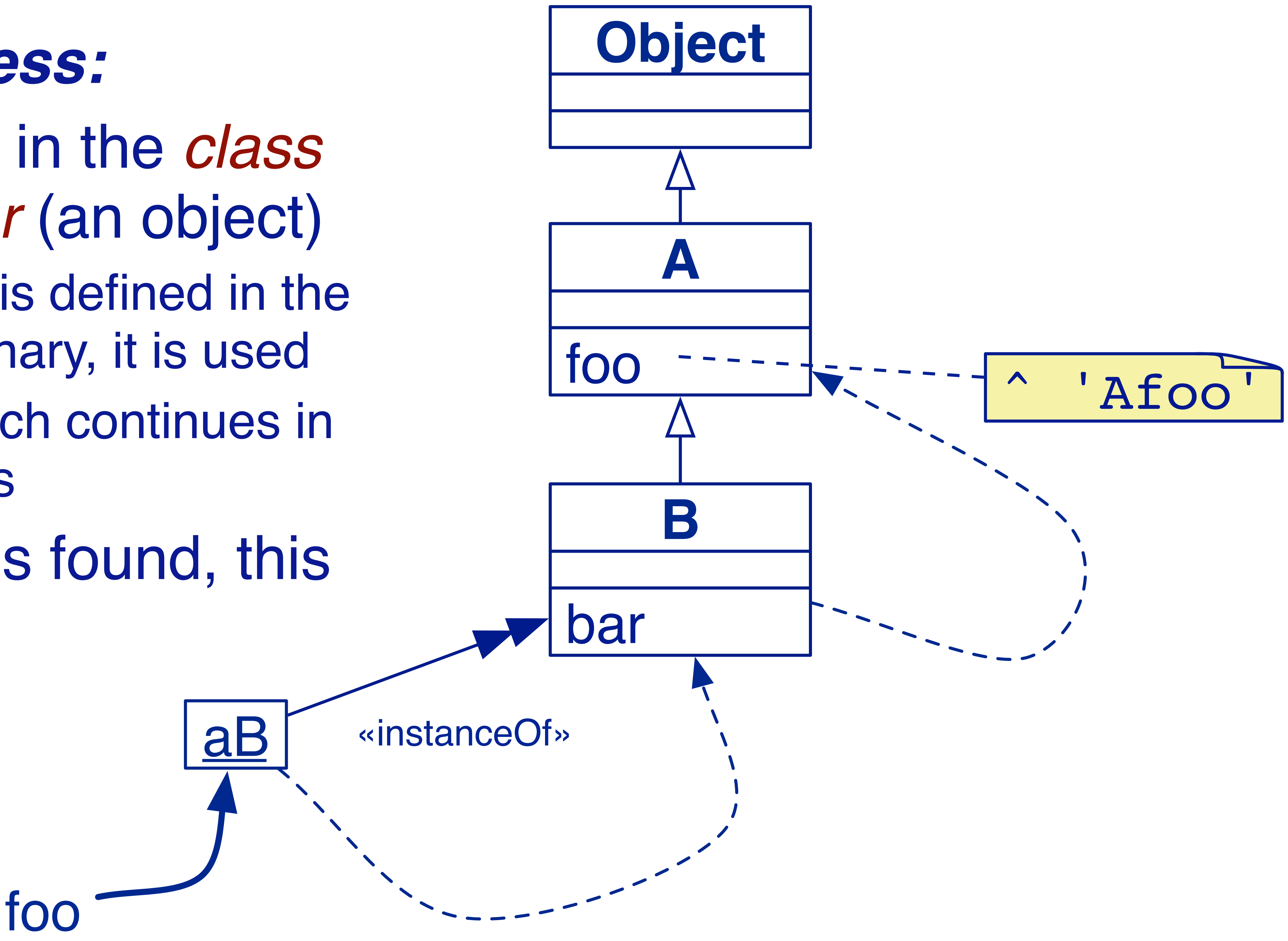
```
BoardSquare>>printOn: aStream
   aStream nextPutAll:
      '[', position printString, self contents, ']'

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* …



A common confusion is to think that `super` refers to the superclass of the receiver. *This is completely wrong*.

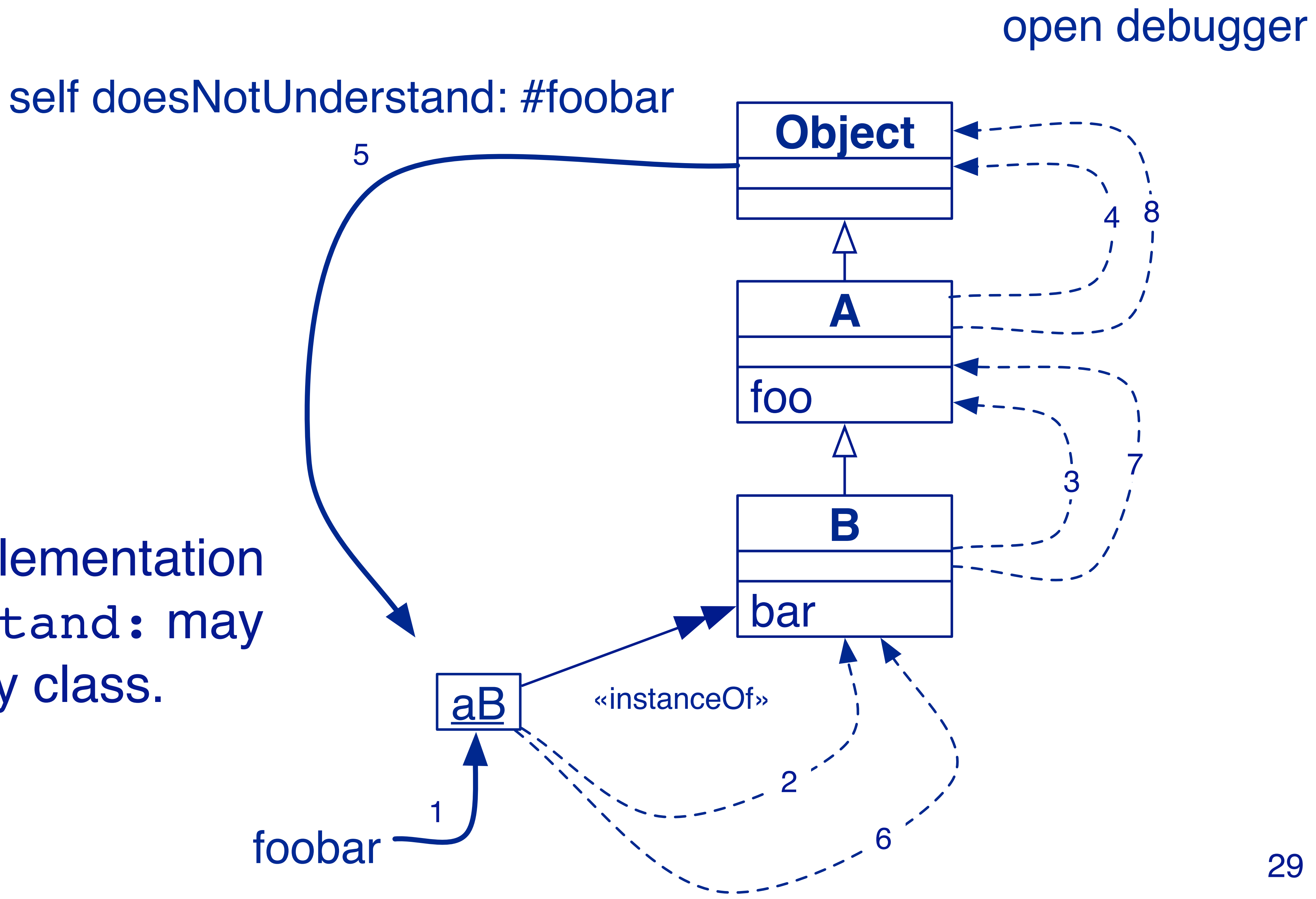
`super` and `self` *both refer to the receiver*, i.e., an *object*, not a *class*. The difference is in the way the method lookup is performed.

With a normal send (and also with `self` sends), lookup starts in the class of the receiver.

Here object `aB` is an instance of the class `B`. When it receives the message `#foo`, lookup starts in the class `B` and, if necessary, proceeds up the hierarchy until the method `foo` is found in the class `A`.

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.

What happens if no method is found to respond to the given message? The default behaviour in Smalltalk is to send a new message `#doesNotUnderstand:` to the original receiver with the original message as its argument.

Again the lookup proceeds as before. Typically the method for `#doesNotUnderstand:` will be found in the class `Object`, causing the Debugger to start up.

However it is also possible to implement `#doesNotUnderstand:` lower in the hierarchy to do something more interesting, as we will see later in the lecture on metaprogramming.

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!

Like `self`, `super` represents the *receiver*, but changes the lookup algorithm. Instead of starting in the class of the receiver, *lookup starts in the superclass of the method containing the* `super` *send.*

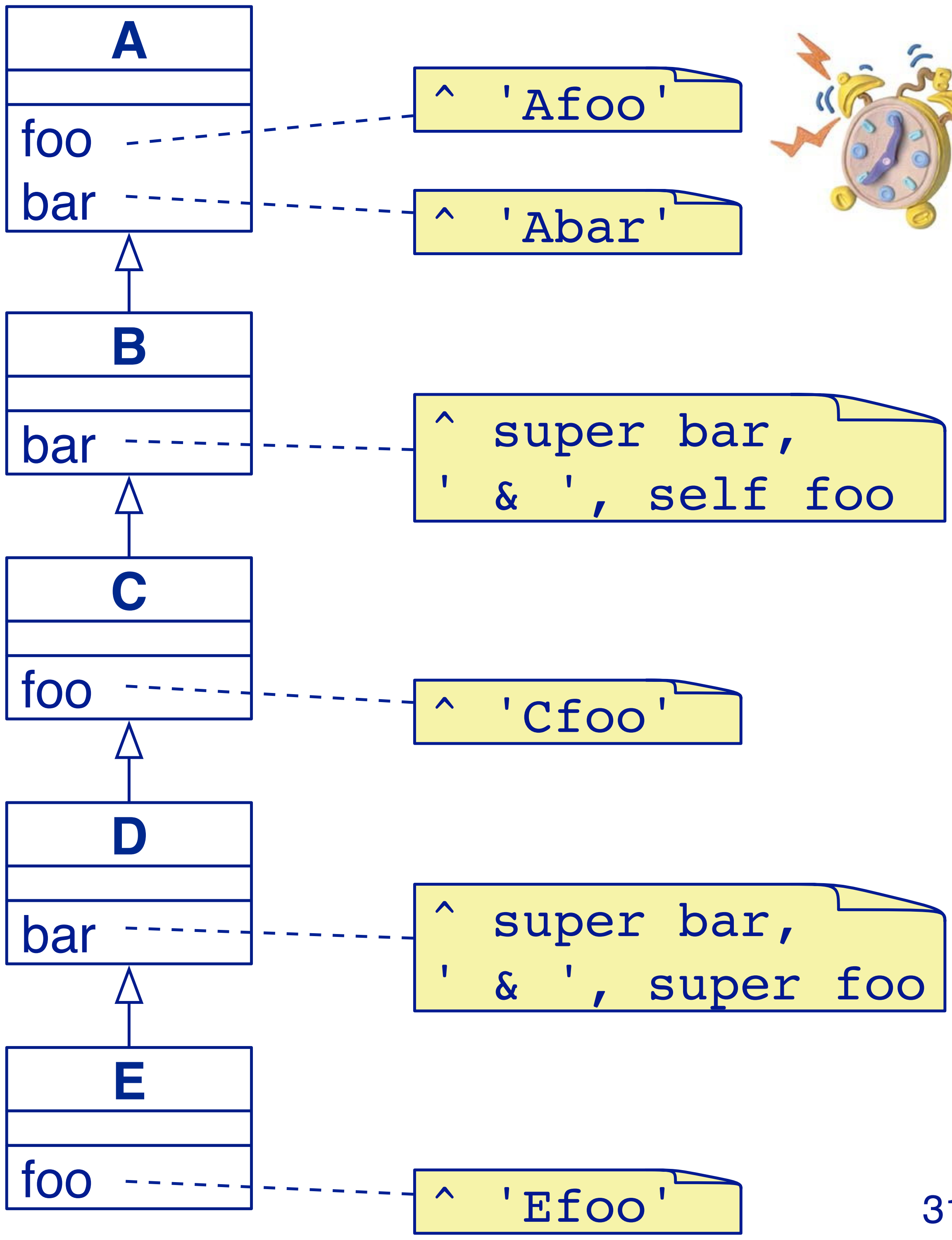
NB: Again there is a common confusion, which is to think that lookup starts in the superclass of the receiver’s class, but this is also completely wrong, as we shall see.

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`!



This toy example illustrates how `self` and super interact.

`A new bar` → `Abar`

`B new bar` → `Abar & Afoo`

`B>>#bar` uses `super` to extend `A>>#bar`.

`C new bar` → `Abar & Cfoo`

`C` inherits the extended `bar` method and reimplements `foo`.

`D new bar` → `Abar & Cfoo & Cfoo`

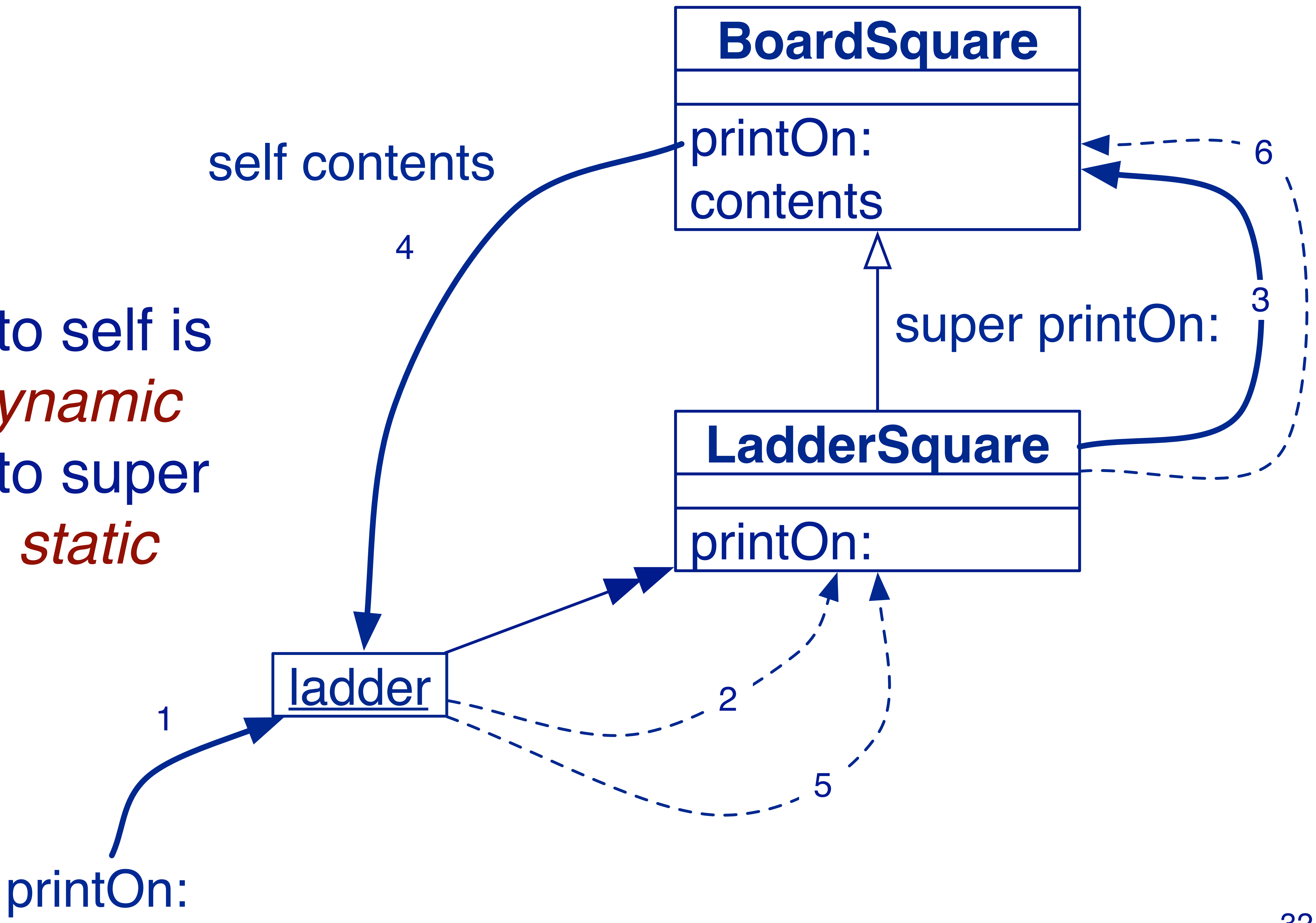
`D` does a strange thing, using `super` not to extend `foo` but to *hardwire* its lookup within `D>>#bar`.

`E new bar` → `Abar & Efoo & Cfoo`

Here we see that the *wrong* `foo` is looked up by `D>>#bar` but the correct one is found by `B>>#bar`. *Never use* `super` *to short-circuit lookup but only to extend an overridden method.* Use a `self` send instead.

Aside: Suppose lookup started in the superclass of the receiver? Then `C new bar` would loop infinitely, since `super bar` would cause lookup to start again in `B` (`C`'s superclass) rather than in `A`.

Self and super



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

The take-home message is that `self` *is dynamic while* `super` *is static.*

A `self` send is looked-up dynamically depending on the class of the receiver, while a `super` send statically refers to the nearest method of that name looked up in the (static) class of the method doing the `super` send. *The receiver's class is ignored.*

Note that this holds for all single-inheritance object-oriented languages, whether they are dynamically or statically typed. Also in Java, a method call on `this` will be looked up dynamically, while a call to `super` will be statically resolved.

Did you really understand self and super?

What is the result of evaluating this code?

```
self == super
```

You can put this code in an method `foo` of an arbitrary class `X` and try to evaluate `X new foo`, or you can simply evaluate it in a Playground, where self will be bound to `nil` (an instance of `UndefinedObject`).

In any case, the message `== super` will be sent to the instance.

What should normally be the result?

Under what circumstances would the result be different?

How to use super (and how not to)

- Proper usage: method extension
  - Extending constructors/initializers (establishing superclass invariant)
  - Extending superclass methods
- Improper usage
  - Composing arbitrary methods (use self instead)
    - *How would you find all methods that improperly use super?*

The super construct has one proper usage: *extending a superclass method with new behaviour*. There are two classical situations:

1. A *constructor* in a subclass should always call the superclass constructor as its first statement. This will establish the class invariant for all inherited state. In Smalltalk the method `initialize` should always first evaluate `super initialize`.
2. An inherited and overridden method often *extends* rather than replaces inherited behaviour. The super method may be invoked before, after or in the middle of the new behaviour.

Frameworks often use *template methods* to avoid requiring subclasses to invoke `super`. The template method contains the behaviour to be reused, and subclasses just override or define the hook method.

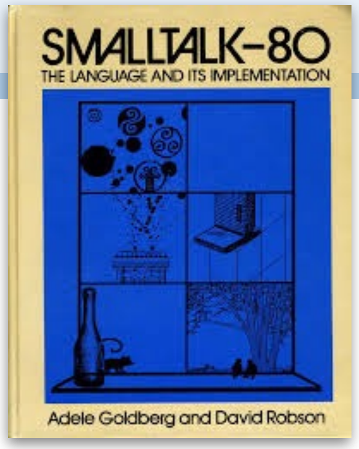
Incorrect usage is using super to invoke different methods. This short-circuits the dynamic lookup of the invoked method.

(Next week we will see how to query the system for such improper usage.)

Roadmap

> OO modeling idioms
> Understanding `self` and `super`
> **Metaclasses in 7 points**

Metaclasses in 7 points


SMALLTALK-80
THE LANGUAGE AND ITS IMPLEMENTATION
Adele Goldberg and David Robson

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

There are many possible metamodels for OO languages, but they are all very similar. Smalltalk's metamodel is especially interesting as it supports not only introspection but intercession: we can change the system at run time.

The seven rules here are freely adapted from the book by Adele Goldberg and David Robson, *Smalltalk 80: the Language and its Implementation*, (Addison Wesley, 1983)

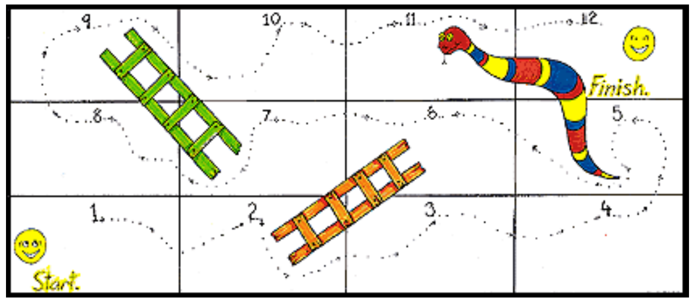
- http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf
- http://scgresources.unibe.ch/Literature/Books/Gold83aBluebook.pdf

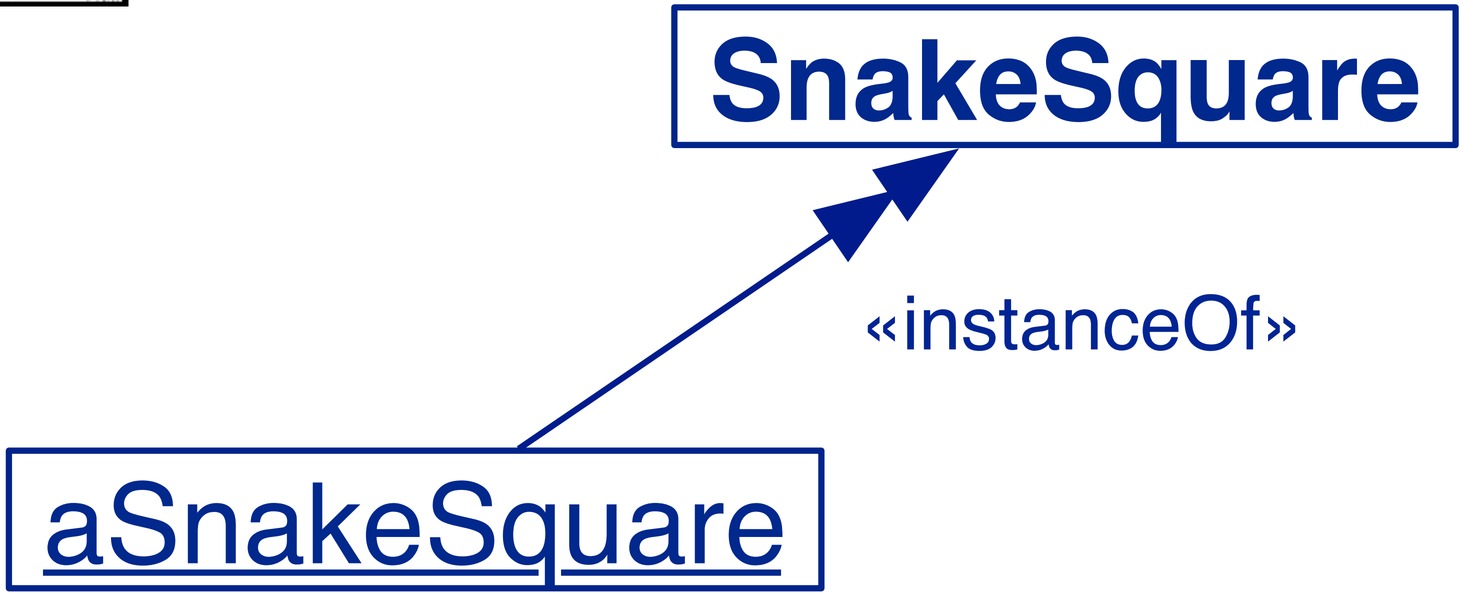
We will now illustrate these rules with the Snakes and Ladders example.

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





UML class diagrams don’t include objects or an *instance-of* relationship, so we will add a special double arrow to with an *«instance-of»* stereotype to represent this concept.

Every object is an instance of a class, in particular a “snake square” in the board game is an instance of the class `SnakeSquare`.

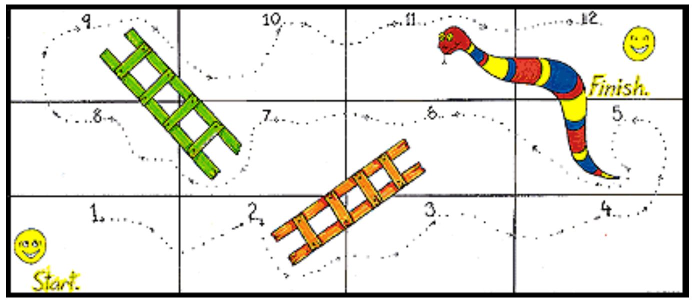
Metaclasses in 7 points

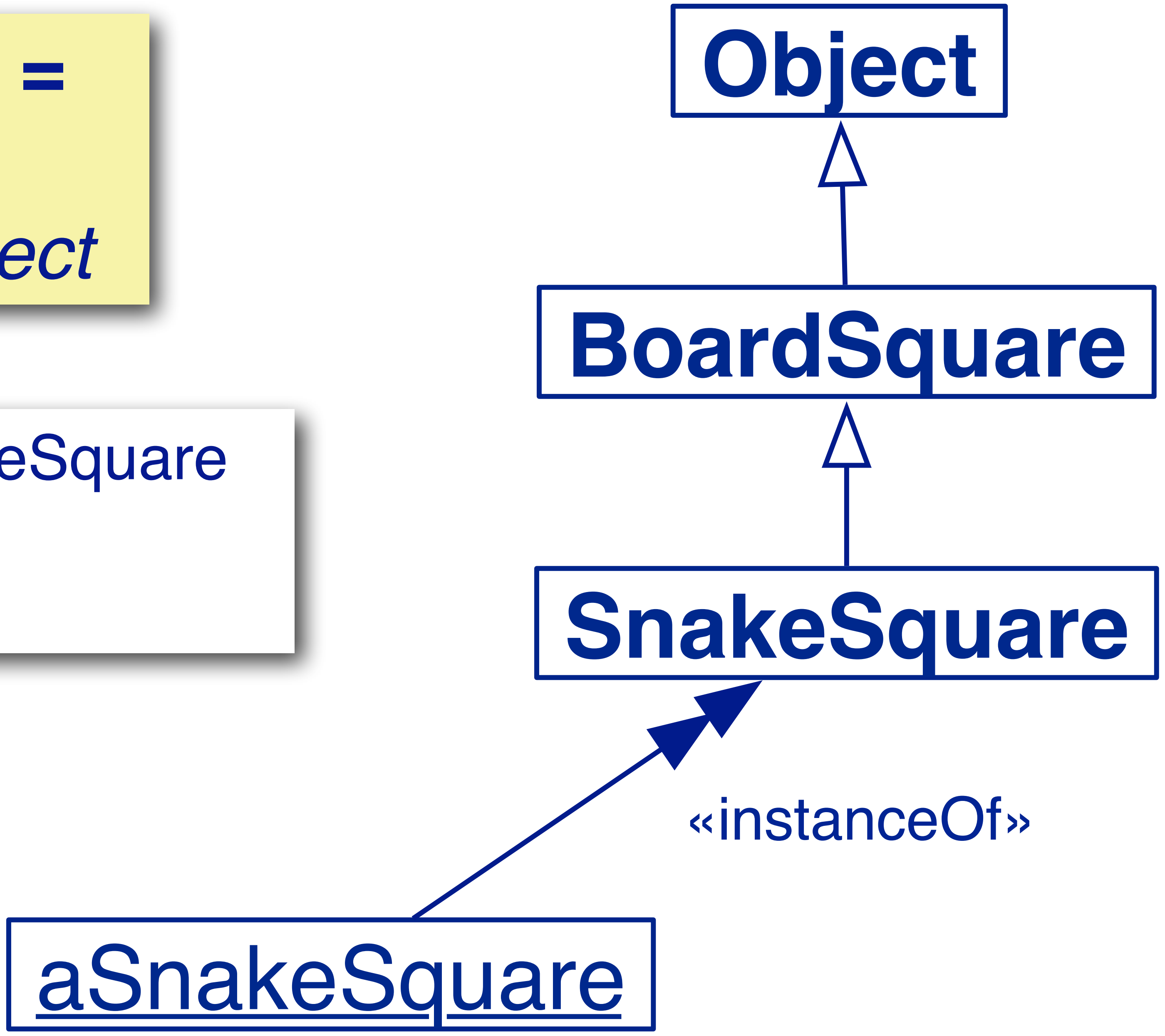
1. Every object is an instance of a class
2. **Every class eventually inherits from Object**
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass
7. The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

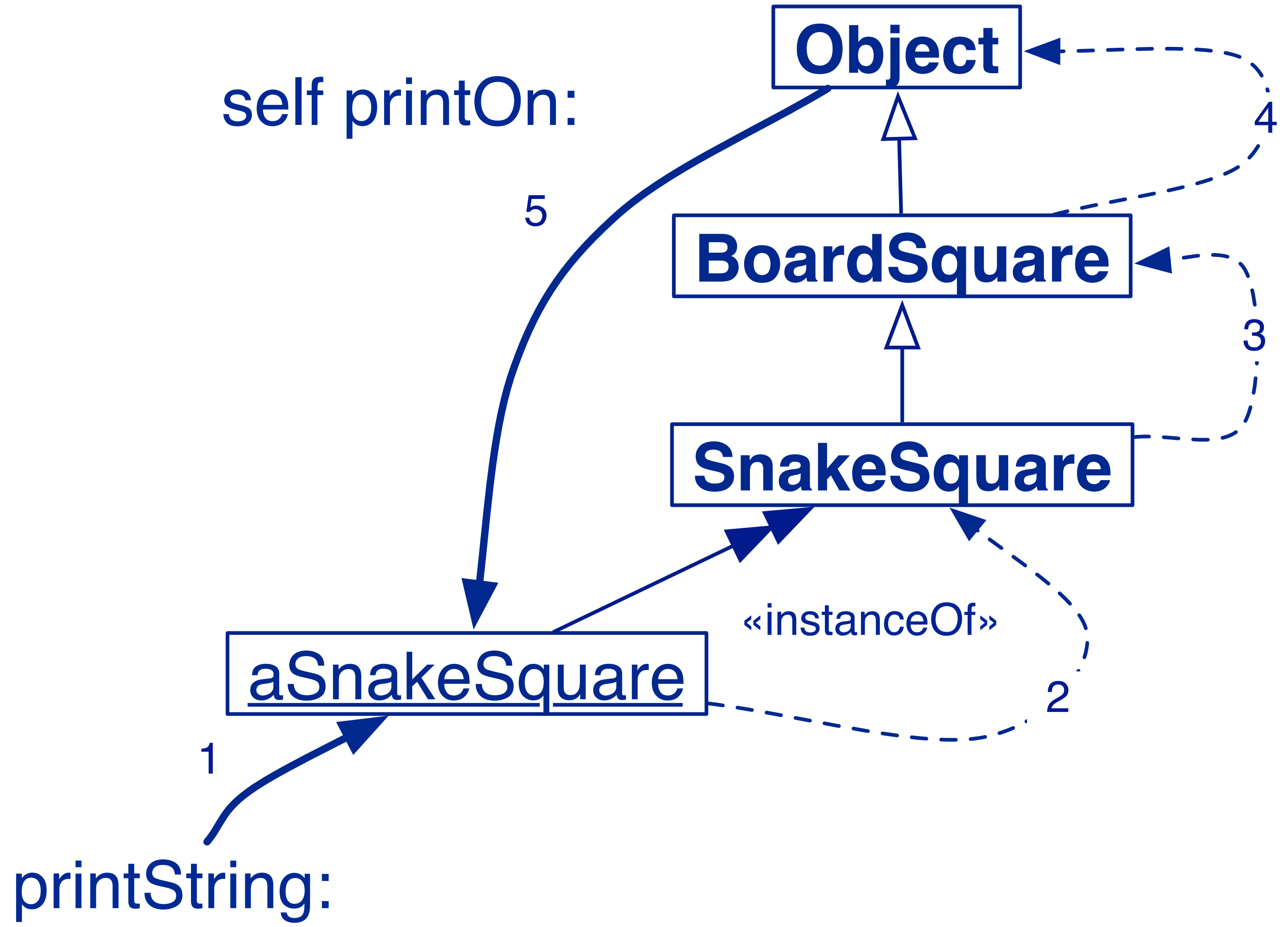
aSnakeSquare *is-a* SnakeSquare
and *is-a* BoardSquare
and *is-an* Object





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



The *is-a* relationship expresses that an object x satisfies all the requirements of some class Y. In effect, an object x *is-a* Y for all classes Y starting from its own class, all the way up the superclass chain. In particular this rule simply states the obvious, which is that *every object is-an Object.*

Pretty much every OO language has a class hierarchy with `Object` as its root class.

Responsibilities of Object

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

Caveat: in Pharo, Object has a superclass called ProtoObject

In Pharo, `Object` contains roughly 400 methods. In other OO languages, like Java, the root class is much leaner.

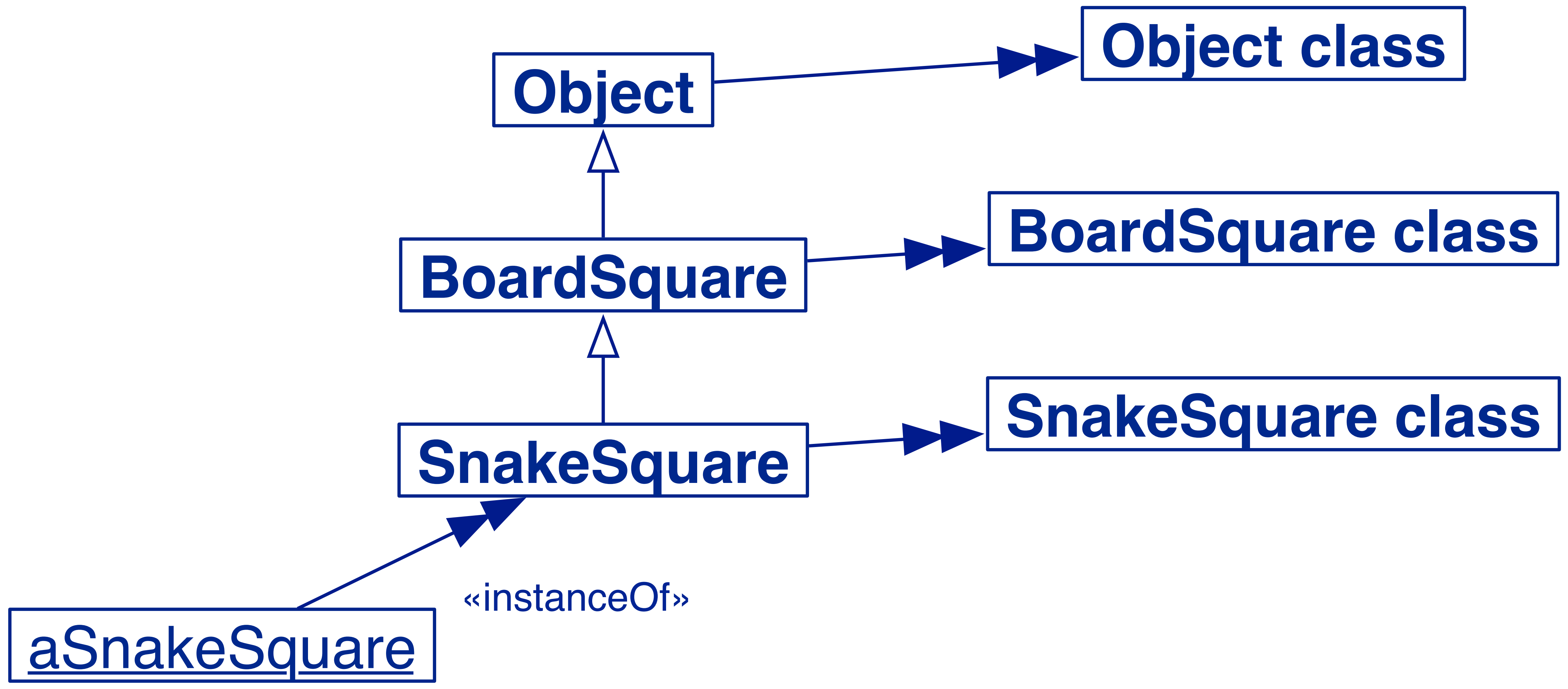
Caveat: In Pharo, there is a lean class that is a superclass of Object, called `ProtoObject`. It can be mostly ignored, but we will find it useful later for metaprogramming. (`ProtoObject` has only about 40 methods.)

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



Since everything is an object in Smalltalk, classes are objects too. This means that they are instances of special classes known as *metaclasses*.

Here different OO languages take different design decisions. In Java, classes are not objects, so there is no need for metaclasses. You may ask an object for a representation of its class, but this is not the actual class. In particular, you cannot modify a “class object” in Java to change the behaviour of its instances.

Metaclasses are implicit

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

Another important difference is that in some languages, like CLOS (Common Lisp Object System), metaclasses can be explicitly programmed. In classic Smalltalk, *metaclasses are implicit*. Every class `X` automatically is assigned a unique metaclass called `X class`.

Metaclasses by Example

```
BoardSquare allSubclasses
```

```
an OrderedCollection(FirstSquare LadderSquare SnakeSquare)
```

```
SnakeSquare allInstances
SnakeSquare instVarNames
```

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

```
SnakeSquare back: 5
```

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

```
SnakeSquare selectors
```

```
#(#destination #initializeBack: #printOn:)
```

`SnakeSquare class` `selectors`

```
#(#back:)
```

```
SnakeSquare canUnderstand: #initializeBack:
SnakeSquare canUnderstand: #new
```
`SnakeSquare class` `canUnderstand: #new`

```
true
false
```
`true`

Here are several examples of class-side methods, i.e., methods implemented in the metaclass of the given class as receiver.

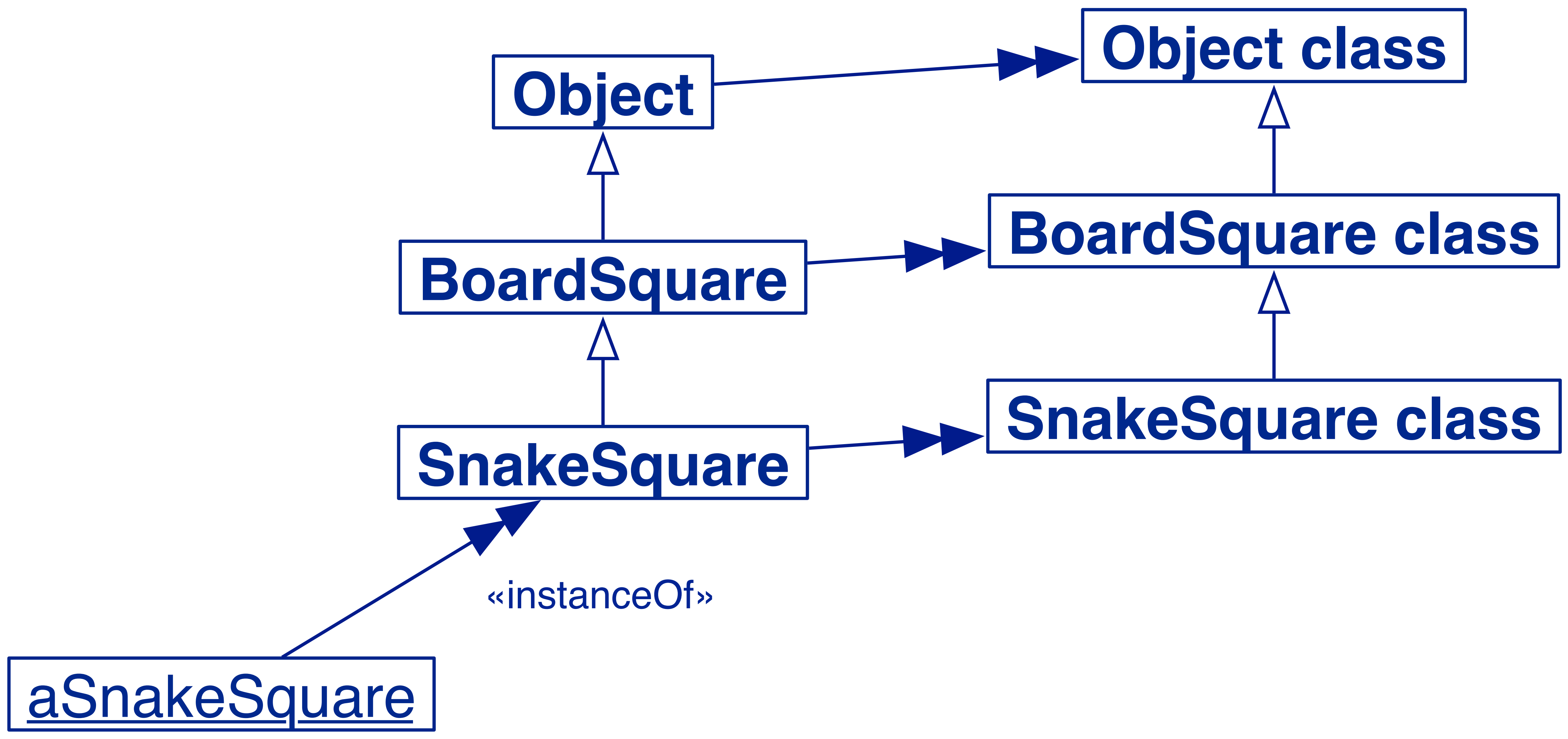
A class serves as *a repository for the behaviour of its instances*. So the selectors (method signatures) implemented by `SnakeSquare` are those understood by its instances, e.g., `#destination`. On the other hand, the messages understood by the class `SnakeSquare`, such as `#back:`, are implemented by the metaclass, i.e., `SnakeSquare class`

NB: `#canUnderstand:` tells you which messages *instances* can understand. This is more than just the selectors (messages) that it implements. A `SnakeSquare` does not understand `new`, but its class does!

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

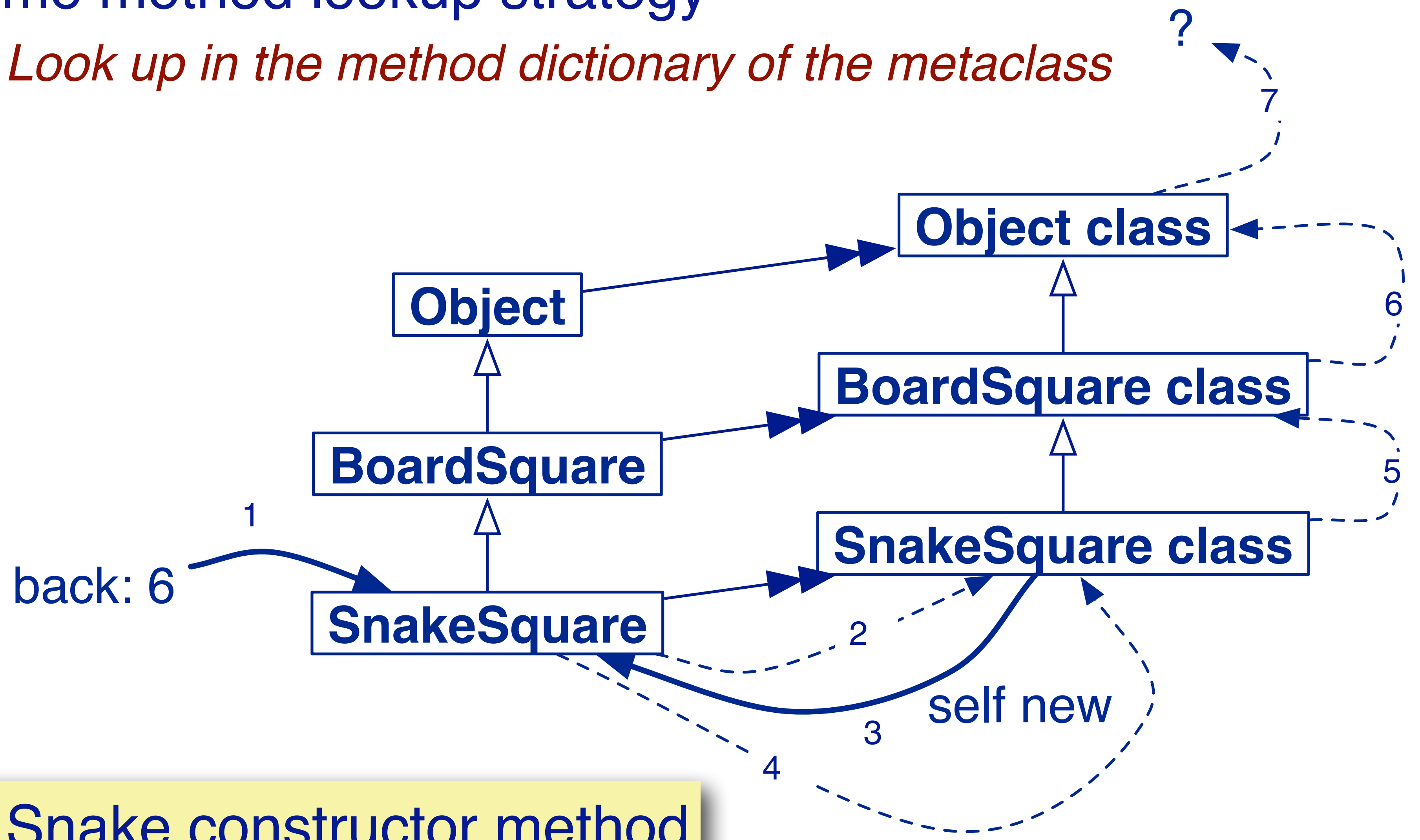
4. The metaclass hierarchy parallels the class hierarchy



This is a pragmatic design choice in the Smalltalk-80 system. By making the metaclass hierarchy parallel the class hierarchy, we ensure that every class also inherits its class-side methods from its metaclass.

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*



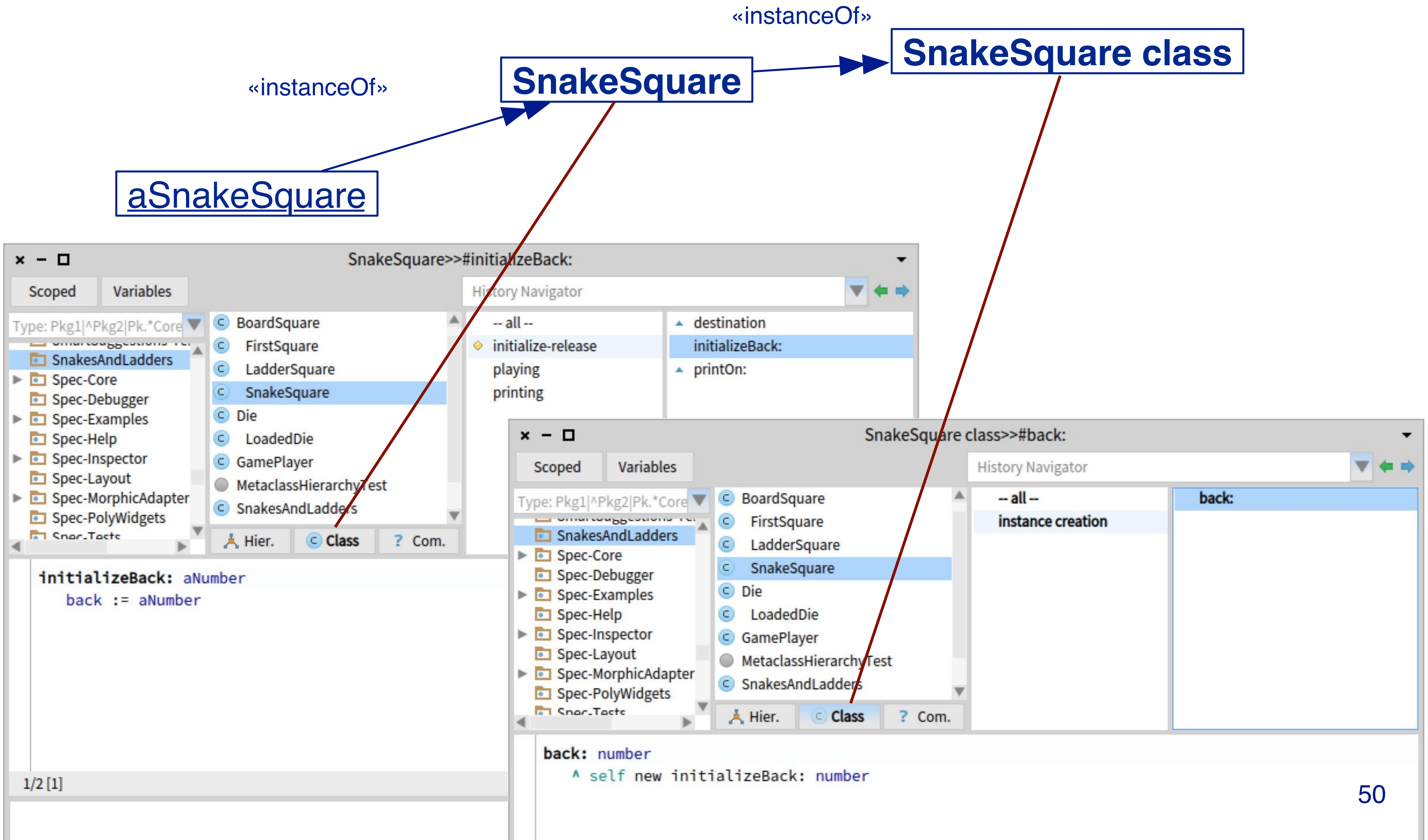
back: is a Snake constructor method

Since *classes are objects too*, we can suppose that method lookup works exactly as it does for ordinary objects. The only difference is that methods are looked up in the metaclass hierarchy.

When we evaluate `SnakeSquare back: 5` we look up `#back:` in `SnakeSquare class`. That method sends `self new`, which starts the lookup again. Since neither `SnakeSquare class` nor `BoardSquare class` implement `#new`, the lookup continues up the hierarchy.

The question is, where is `#new` found?

About the Buttons (in Pharo)



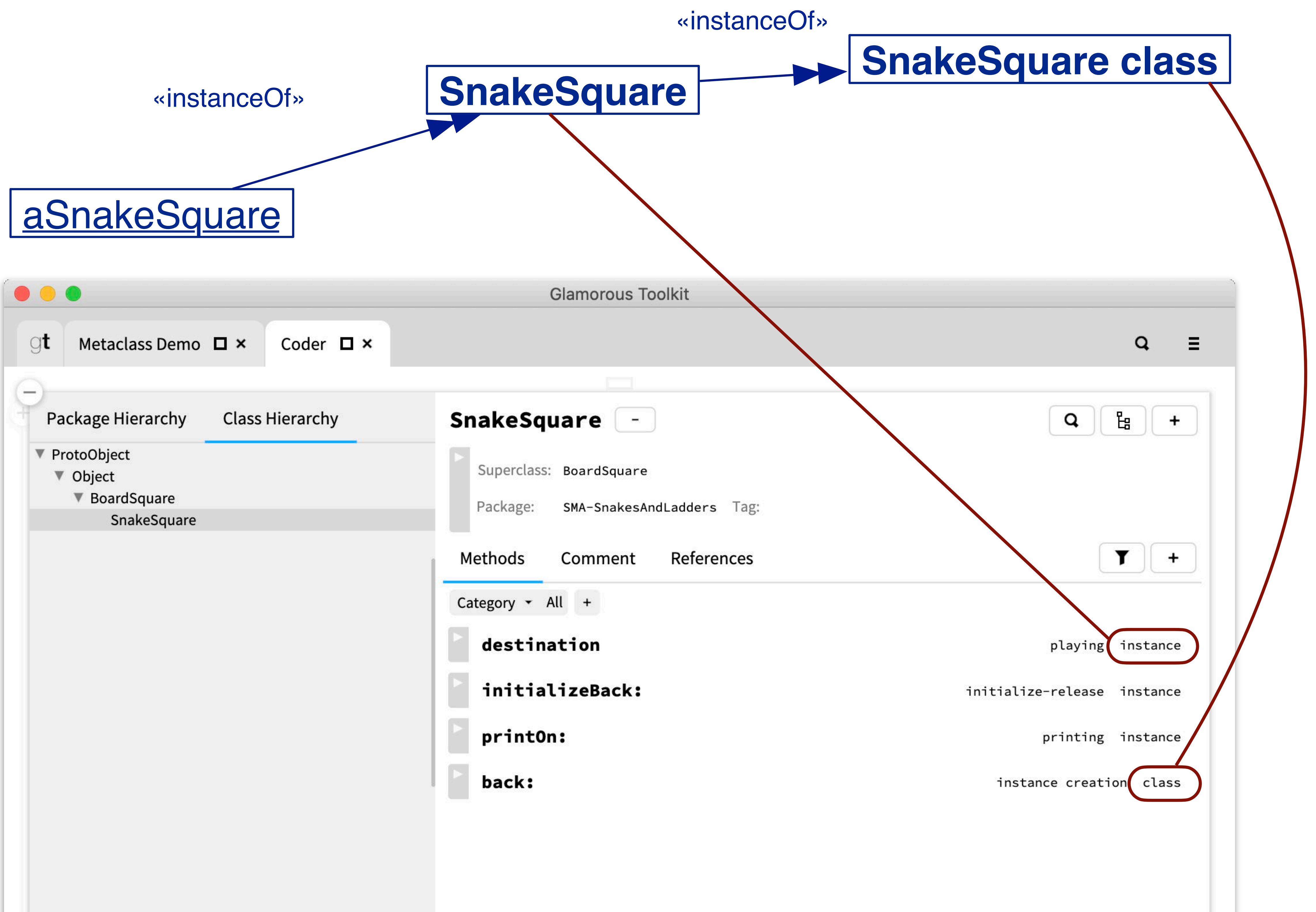
```
initializeBack: aNumber
    back := aNumber
```

```
back: number
    ^ self new initializeBack: number
```

In order to switch between the methods of a class and its metaclass you must toggle the “Class” button in the browser.

Note that in the class-side view all the methods and protocols are in **bold**.

Metaclasses in Gt



SnakeSquare

Superclass: BoardSquare

Package: SMA-SnakesAndLadders Tag:

destination	playing	instance
initializeBack:	initialize-release	instance
printOn:	printing	instance
back:	instance creation	class

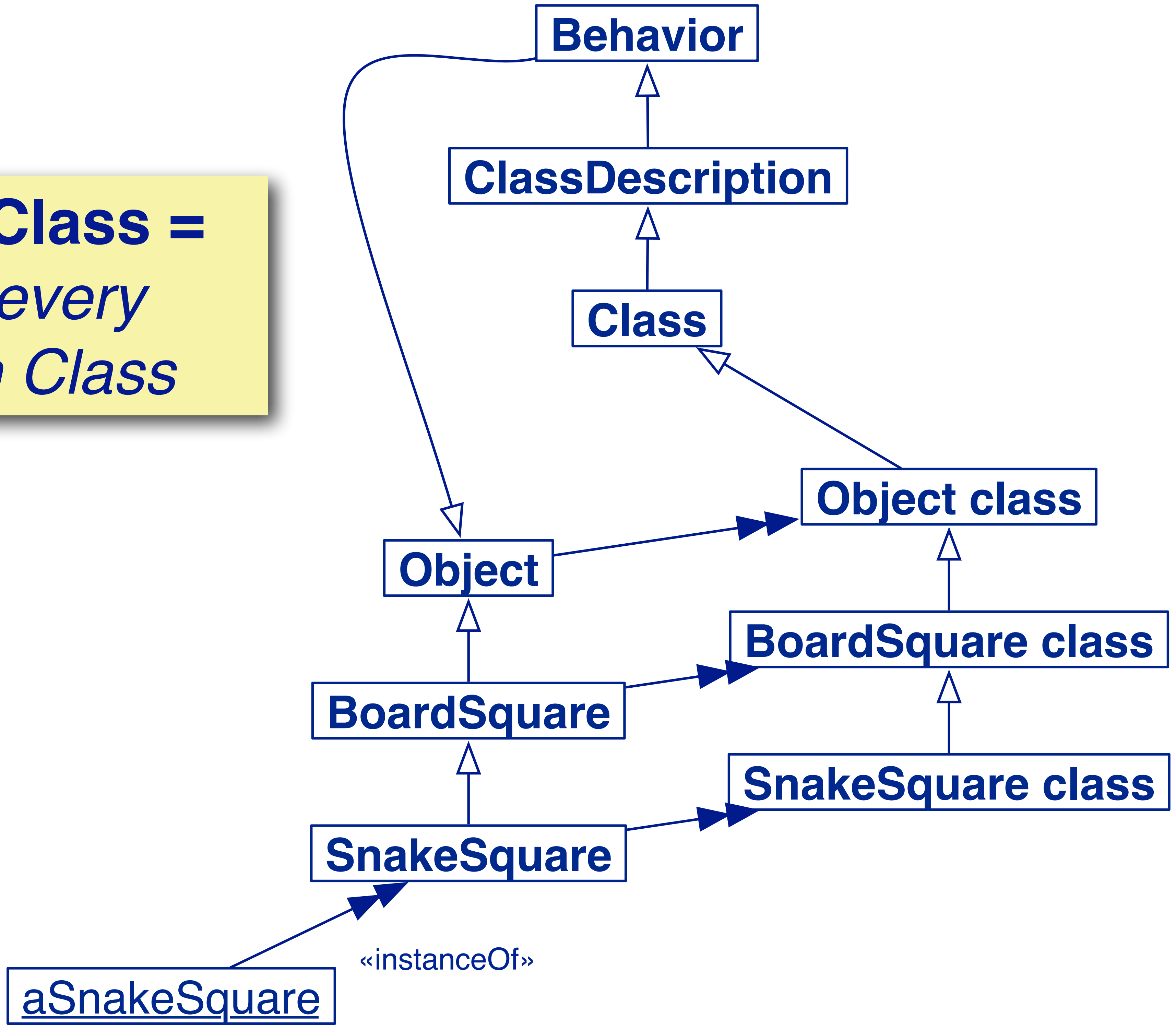
In Gt, on the other hand, instance and class methods are displayed together, but tagged to indicate which kind of method they are.

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

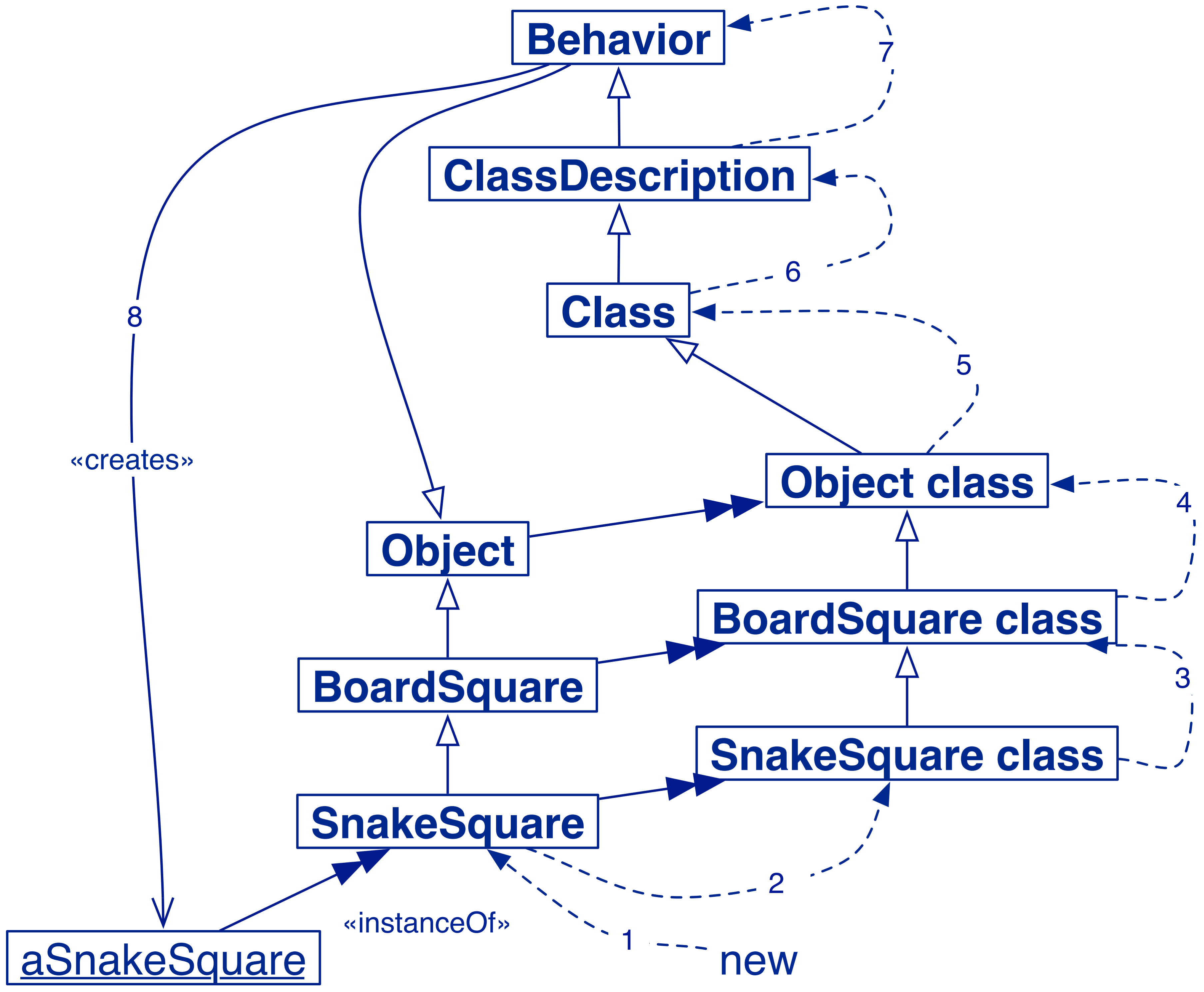
Every class is-a Class =
The metaclass of every class inherits from Class



Since rule 2 tells us that all classes eventually inherit from `Object`, we should infer that the same holds for metaclasses, which are also classes. So the metaclass hierarchy does not stop with `Object class`, but rather with `Object`.

In between, however, we have the special system classes `Class`, `ClassDescription` and `Behavior`.

Where is new defined?



So, `#new` is not defined in `Object`, or even in `Object class`, but in `Behavior`.

Note however that it is possible to override `#new` in any metaclass, for example to track the creation of instances. In Pharo `#new` is rarely overridden since most special initialization behavior can be invoked automatically in the instance-side `#initialize` method.

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.

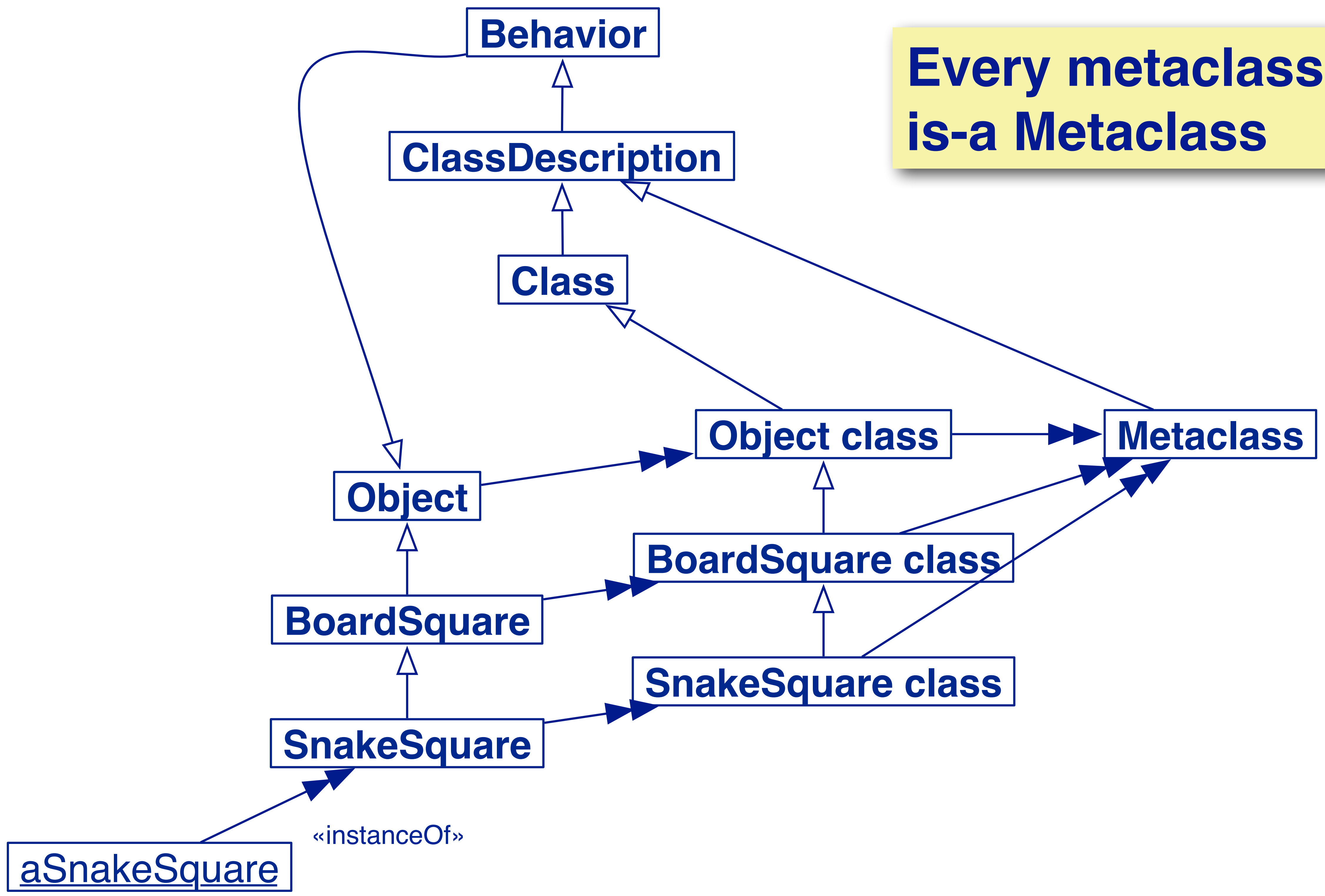
Responsibilities of Class

- **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



Every metaclass is-a Metaclass

If everything is an object and classes are objects, it follows that `metaclasses are objects too`. What then is the class of a metaclass?

Here we have another language design choice. Rather than having implicit meta-metaclasses (and so on), we have a single (explicit) class, called `Metaclass`, of which all metaclasses are instances. This class serves as the shared repository of behavior for all metaclasses (just as `Class` is the shared repository of behavior for all normal classes).

In other words, *"every metaclass is-a Metaclass"*.

Metaclass Responsibilities

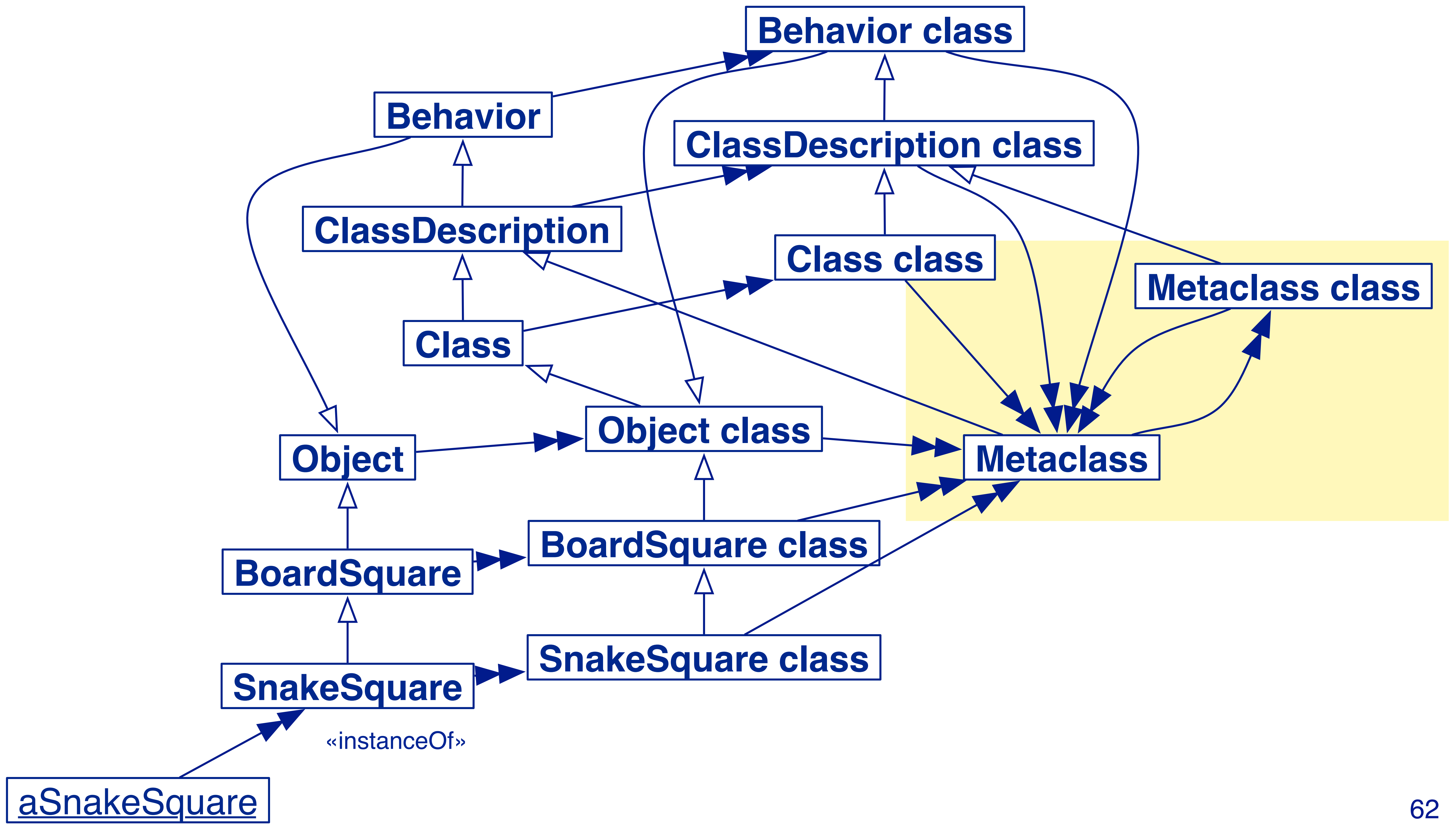
> **Metaclass**

- 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

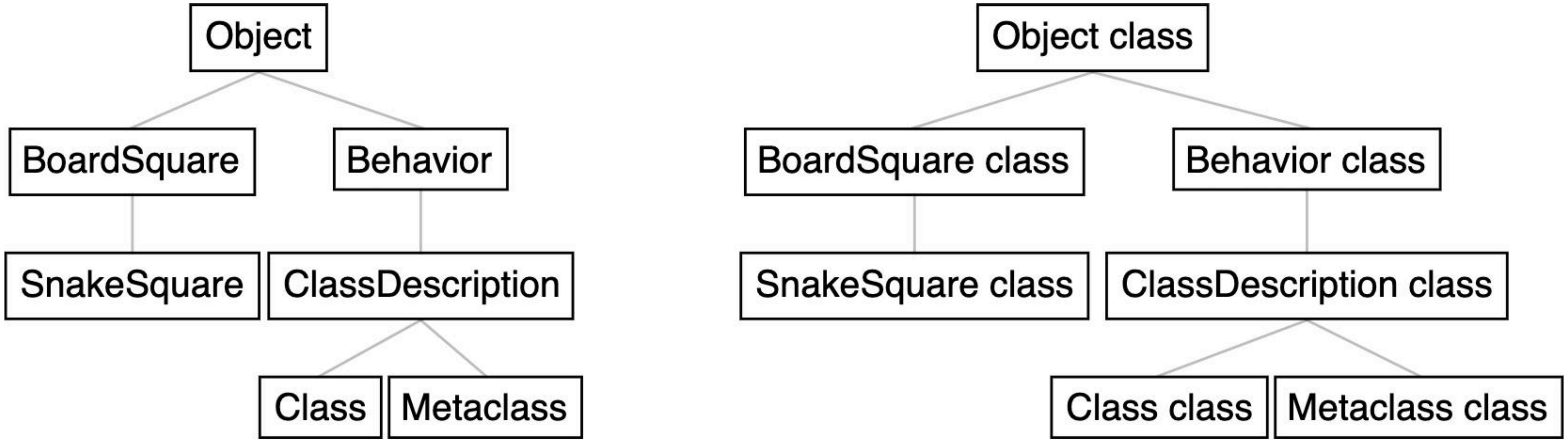


Finally we arrive at the fixpoint. `Metaclass` is a regular class. By rule 2 it must be an instance of its metaclass, `Metaclass class`. By rule 6 however that metaclass must be an instance of `Metaclass`, thus yielding rule 7.

In other words:

```
Metaclass class class = Metaclass → true
```

Parallel hierarchies



Viewed another way, we can clearly see two parallel hierarchies. Every class in the left (class) hierarchy is an instance of its metaclass in the right hierarchy. Each of the metaclasses in the right (metaclass) hierarchy is an instance of Metaclass.

Navigating the metaclass hierarchy

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

What you should know!

- How is a new instance of a class initialized?
- Why 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 do metaclasses inherit from `Class` and not from `Metaclass`?
> 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?