4. Smalltalk Coding Idioms

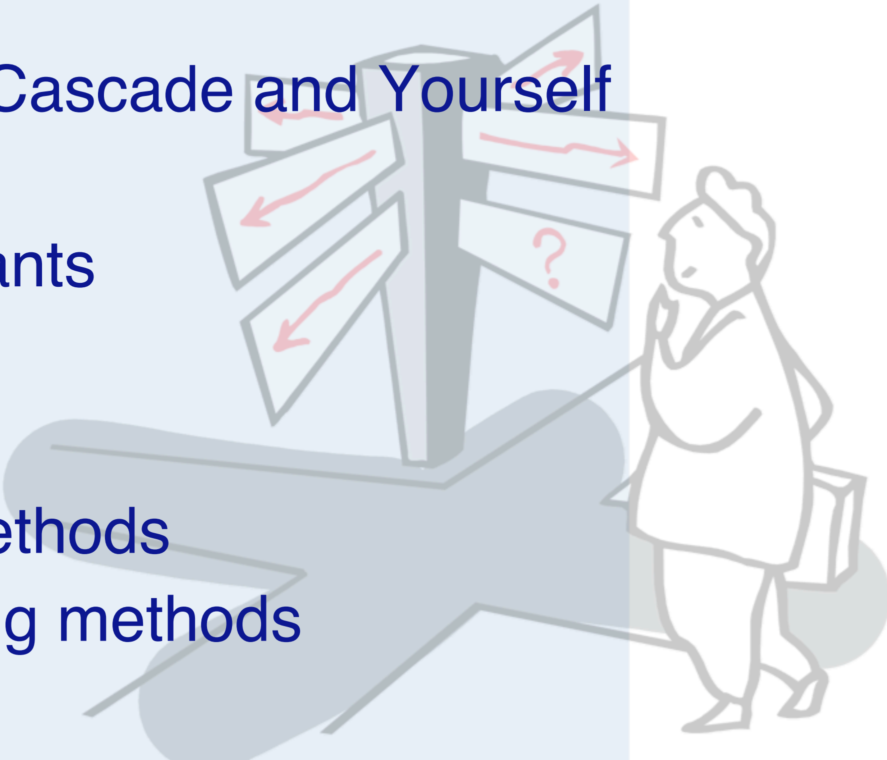
UNIVERSITÄT BERN

Birds-eye view

Distribute responsibility — in a well-designed object-oriented system you will typically find many, small, carefully named methods.
This promotes fluent interfaces, reuse, and maintainability.

Roadmap

- Snakes and Ladders — Cascade and Yourself
- Lots of Little Methods
- Establishing class invariants
- Printing state
- Self and super
- Accessors and Query methods
- Decomposing and naming methods

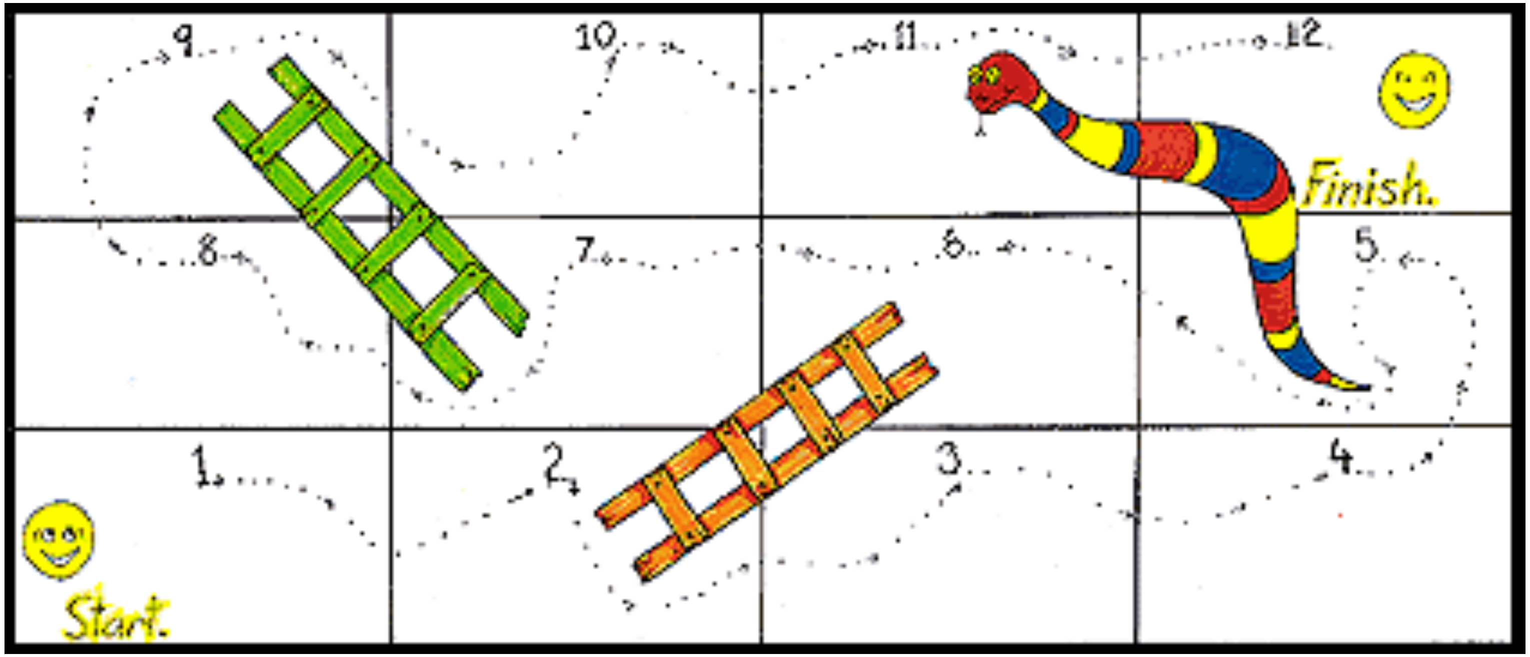


Selected material based on: Kent Beck, *Smalltalk Best Practice Patterns*, Prentice-Hall, 1997.
Selected material courtesy Stéphane Ducasse

Roadmap

- **Snakes and Ladders — Cascade and Yourself**
- Lots of Little Methods
- Establishing class invariants
- Printing state
- Self and super
- Accessors and Query methods
- Decomposing and naming methods

Snakes and Ladders



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

Scripting a use case

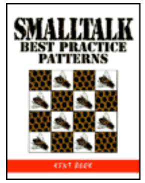
We need a way to:

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

The example script helps us to identify responsibilities, classes and needed interfaces

```
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
```

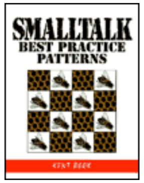
Cascade



How do you format multiple messages to the same receiver?

> Use a Cascade. Separate the messages with a semicolon. Put each message on its own line and indent one tab. Only use Cascades for messages with zero or one argument.

Yourself



How can you use the value of a Cascade if the last message doesn't return the receiver of the message?

> Append the message `yourself` to the Cascade.

About yourself

- The effect of a cascade is to send all messages to the receiver of the first message in the cascade
  - `self new add: FirstSquare new; …`
- But the value of the cascade is the value returned by the last message sent

```
(OrderedCollection with: 1) add: 25; add: 35
```

```
35
```

- To get the *receiver* as a result we must send the additional message `yourself`

```
(OrderedCollection with: 1) add: 25; add: 35; yourself
```

```
an OrderedCollection(1 25 35)
```

Yourself implementation

> The implementation of `yourself` is trivial, and occurs just once in the system:

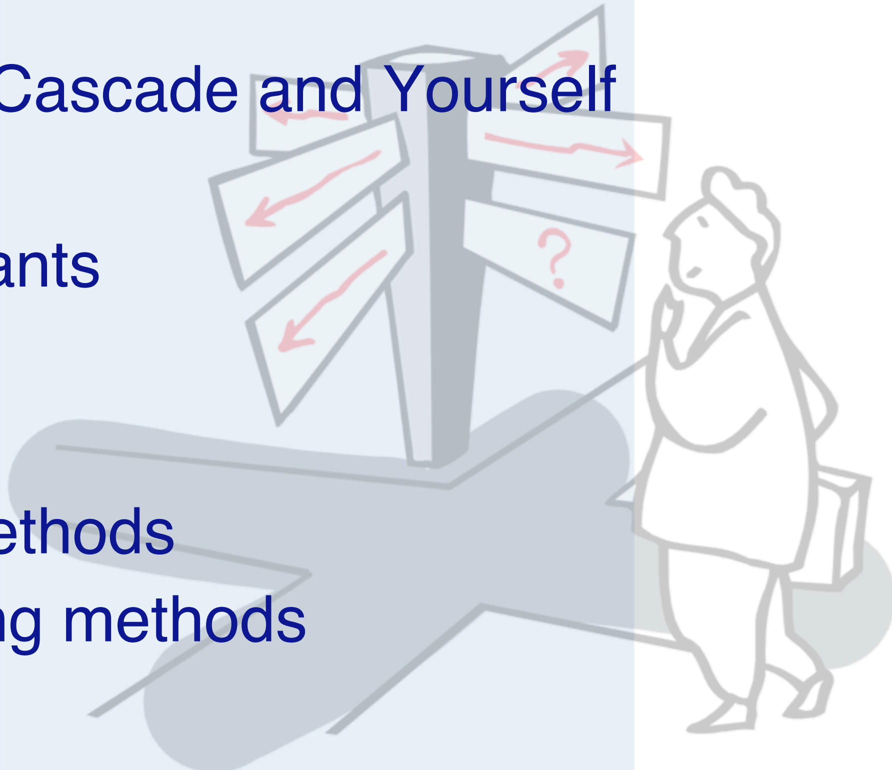
```
Object>>yourself
   ^ self
```

Do we need yourself here?

```
SnakesAndLadders class>>example
   "self example playToEnd"
   ^ self new
     add: FirstSquare new;
     …
     join: (GamePlayer named: 'Jill');
     yourself
```

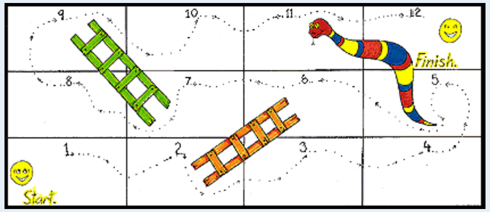
Could be. Don't really know yet …

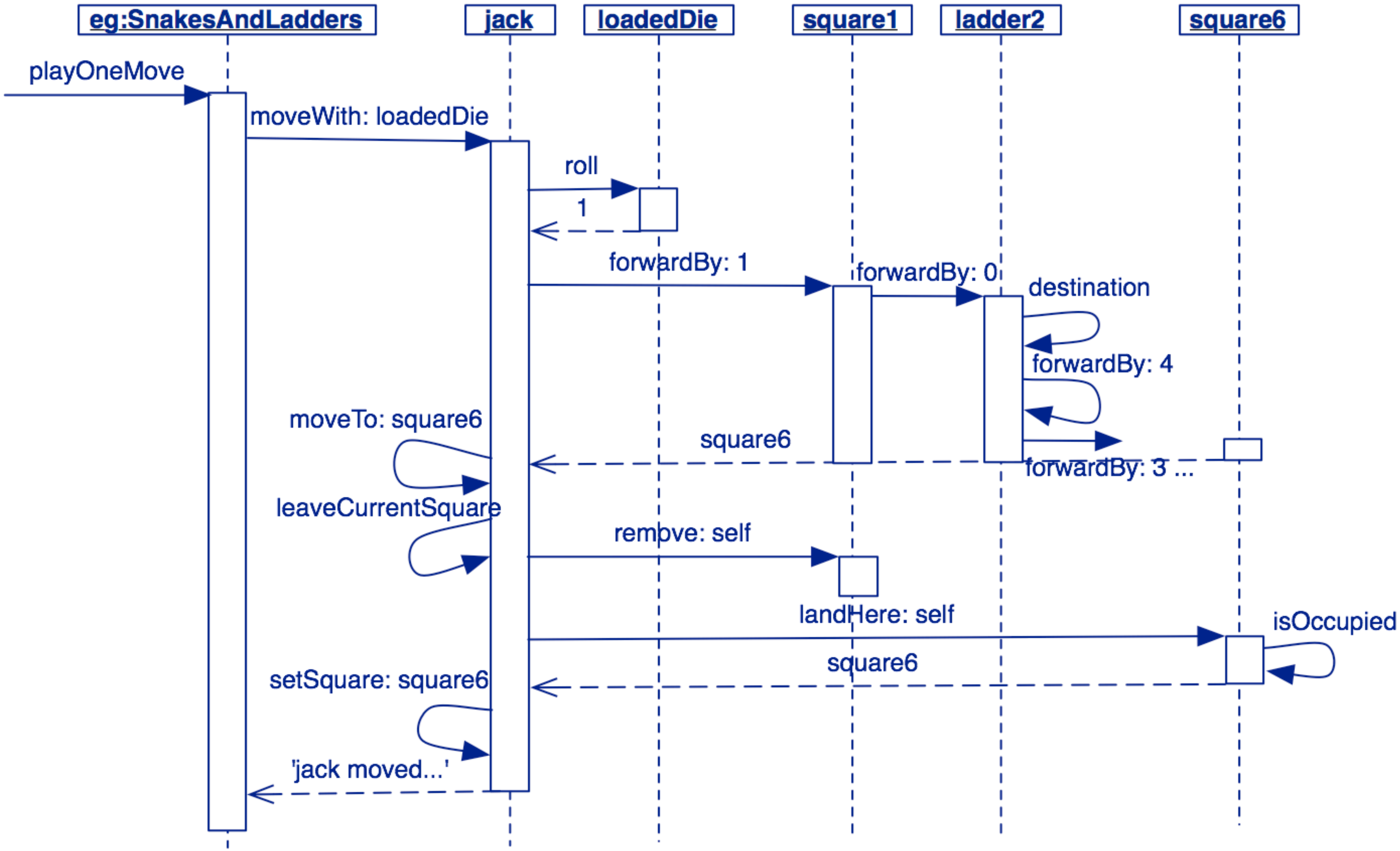
Roadmap



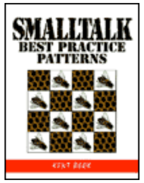
- > Snakes and Ladders — Cascade and Yourself
- > **Lots of Little Methods**
- > Establishing class invariants
- > Printing state
- > Self and super
- > Accessors and Query methods
- > Decomposing and naming methods

Distributing responsibilities





Lots of Little Methods

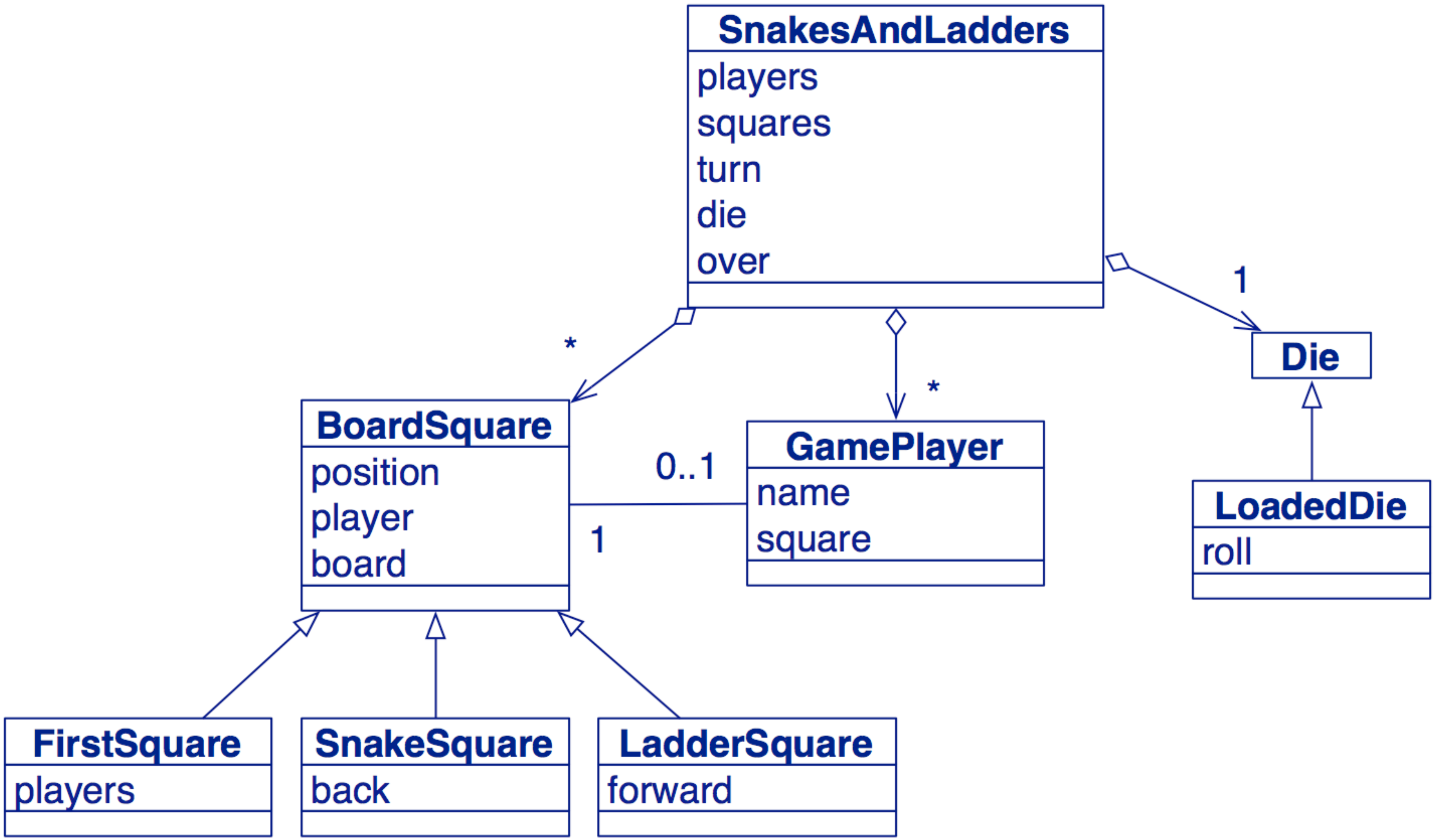


> ***Once and only once***
>
> — "In a program written with good style, everything is said once and only once."

> ***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."

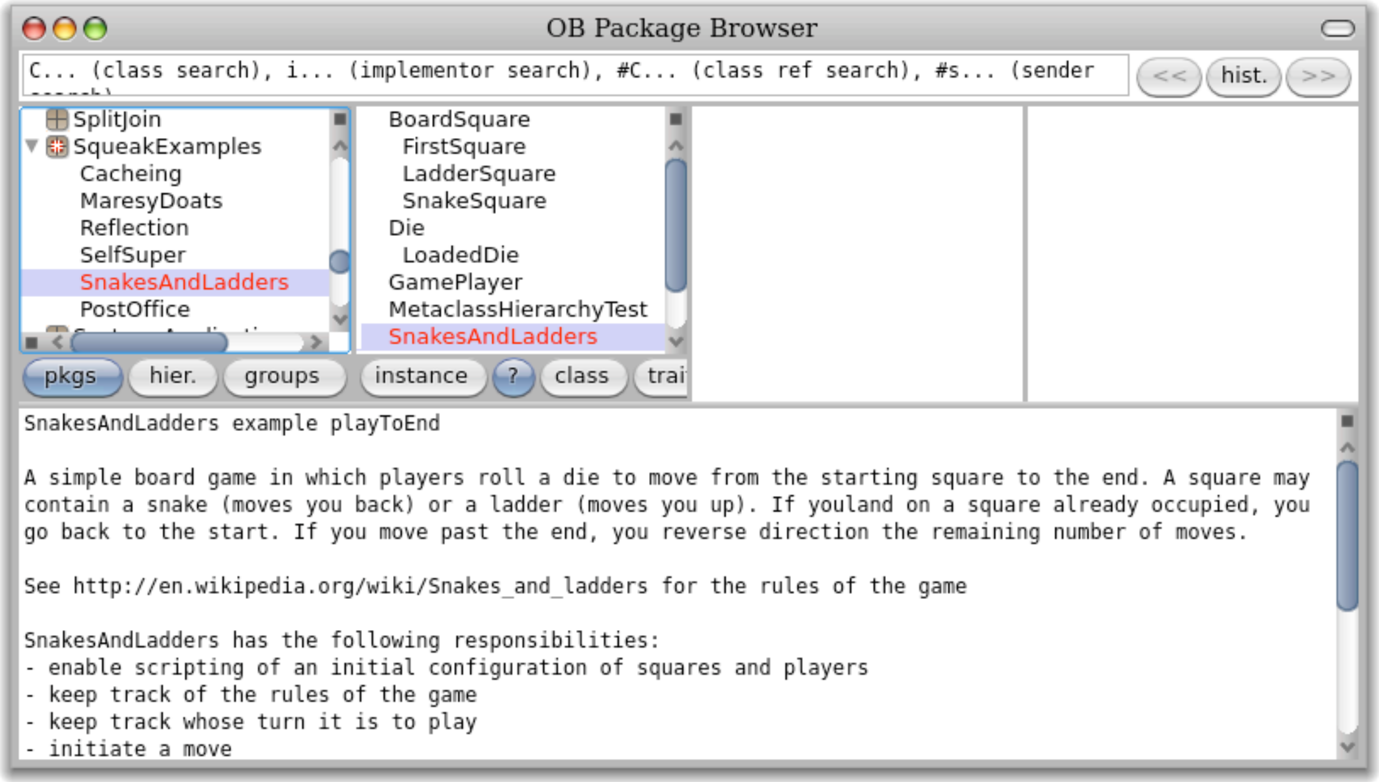
– *Kent Beck, Smalltalk Best Practice Patterns*

Class responsibilities and collaborations



Class Comments

> Add a comment to each class indicating its responsibilities
>
> — Optionally include sample code to run an example



Inheritance in Smalltalk

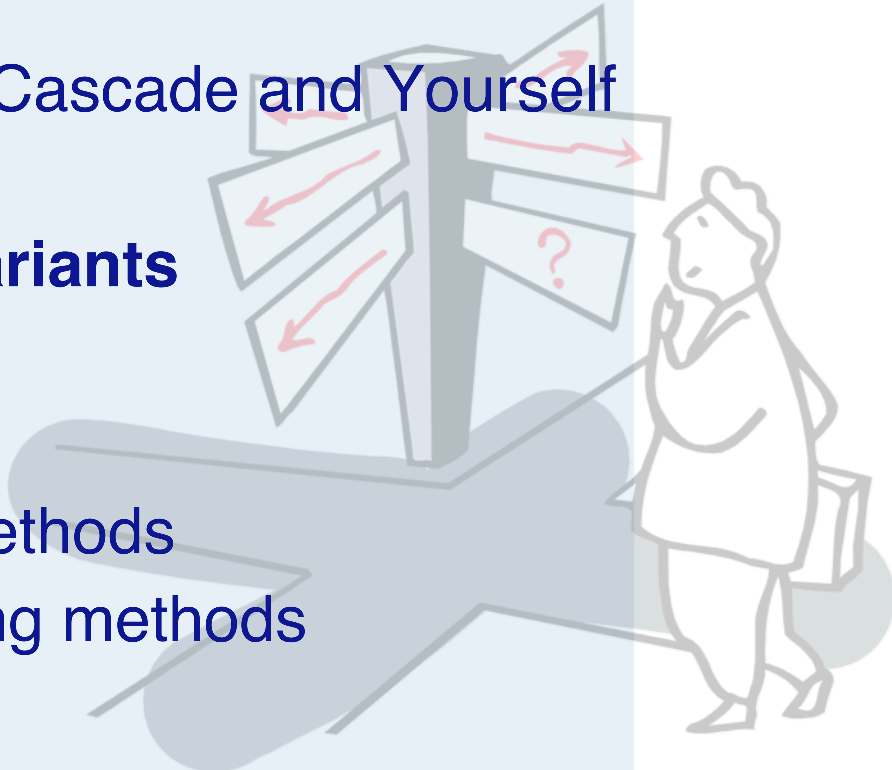
- Single inheritance

- Static for the instance variables
  - Instance variables are collected from the class and its direct and indirect superclasses.

- Dynamic for the methods
  - Methods are looked up at run-time depending on the dynamic type of the receiver.

Roadmap

> Snakes and Ladders — Cascade and Yourself
> Lots of Little Methods
> **Establishing class invariants**
> Printing state
> Self and super
> Accessors and Query methods
> Decomposing and naming methods



Creating classes

> A class is created by sending a message to its superclass

```
Object subclass: #SnakesAndLadders
   instanceVariableNames: 'players squares turn die over'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'SnakesAndLadders'
```

Named Instance Variables

> Instance variables:
- Begin with a *lowercase letter*
- Must be explicitly declared: a list of instance variables
- Name should be *unique* in the inheritance chain
- Default value of instance variable is `nil`
- *Private to the instance*, not the class (in contrast to Java)
- Can be accessed by *all the methods of the class and its subclasses*
- Instance variables *cannot be accessed by class methods*.
- The clients must use *accessors* to access an instance variable.

Design Hint:
- Do not directly access instance variables of a superclass from subclass methods. This way classes are not strongly linked.

Problem — how to initialize objects?

Problem

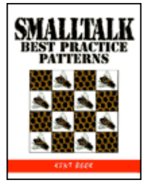
- To create a new instance of a class, the message `new` must be sent to the class
  - But the class (an object) cannot access the instance variables of the new object (!)
  - So how can the class establish the invariant of the new object?

Solution

- Provide *instance-side initialization methods* in the protocol initialize-release that can be used to create a valid instance

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.
```

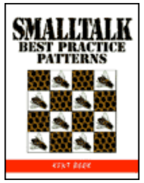
Who calls initialize?

> 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`!
> Every metaclass ultimately inherits from `Behavior`
  – *More on this later …*

Ordered Collection



How do you code Collections whose size can’t be determined when they are created?

> Use an `OrderedCollection` as your default dynamically sized Collection.

Invariants

> If your objects have non-trivial invariants, or if they can only be initialized incrementally, consider explicitly implementing an invariant-checking method:

```
SnakesAndLadders>>invariant
	"Should also check that snakes and ladders
	lead to ordinary squares, and do not bounce
	past the beginning or end of the board."
	^ squares size > 1
		and: [players size > 1]
		and: [turn >= 1]
		and: [turn <= players size]
```

Contracts

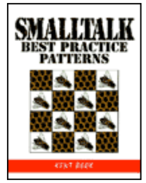
> Apply Design by Contract
  - Aid debugging by checking
    - *Pre-conditions to public methods*
    - *Non-trivial invariants*
    - *Non-trivial post-conditions*

```
BoardSquare>>nextSquare
   self assert: self isLastSquare not.
   ^ board at: position + 1
```

```
SnakesAndLadders>>reset
   die := Die new.
   turn := 1.
   over := false.
   players do: [:each | each moveTo: self firstSquare ].
   self assert: self invariant.
```

Constructor Method

How do you represent instance creation?



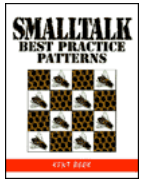
> Provide methods in the class side “instance creation” protocol that create well-formed instances. Pass all required parameters to them.

```
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?



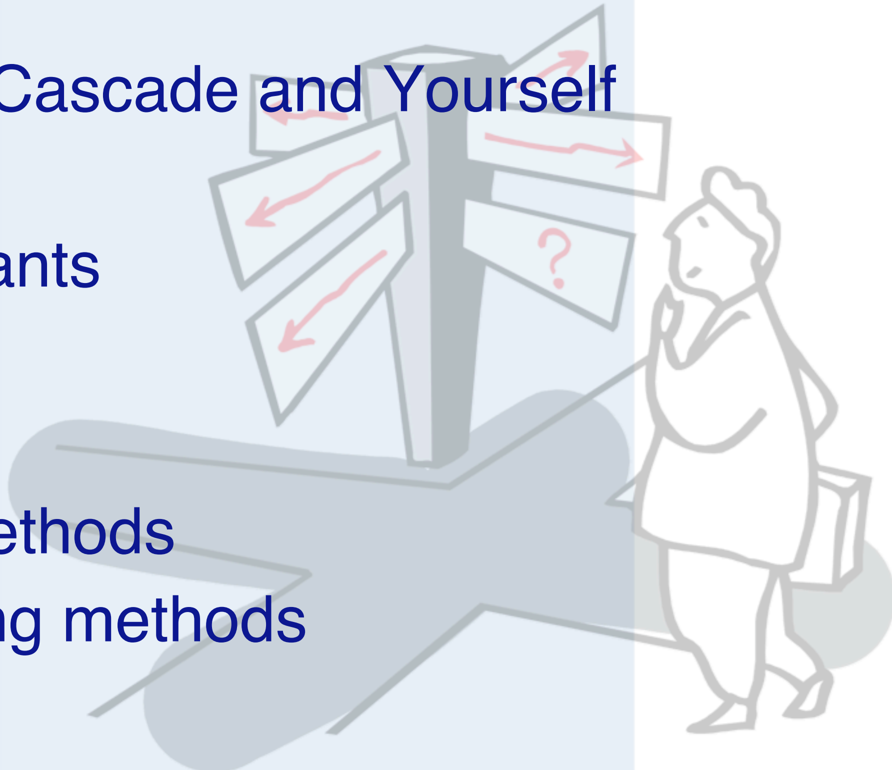
> Code a single method that sets all the variables. Preface its name with “set”, then the names of the variables.

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

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

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

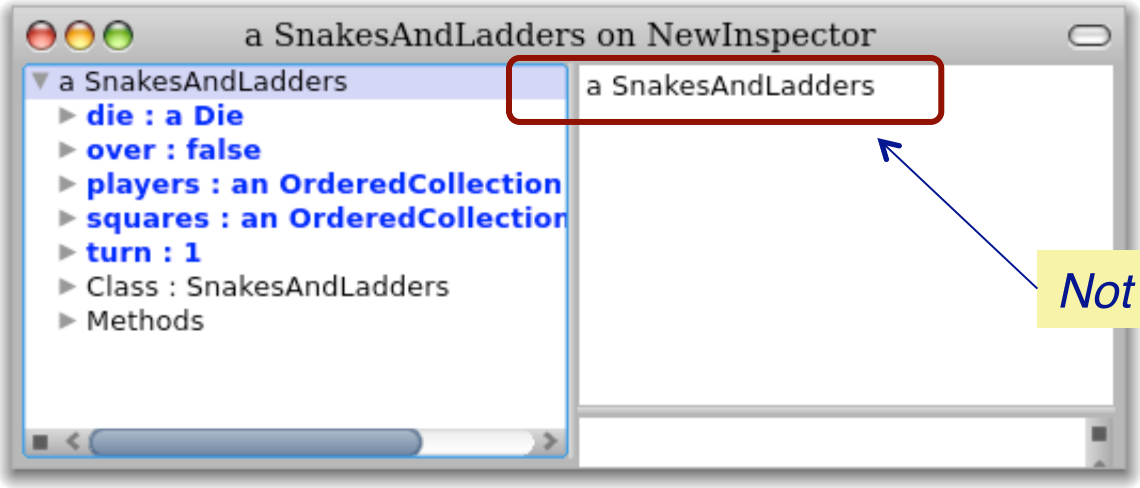
Roadmap



> Snakes and Ladders — Cascade and Yourself
> Lots of Little Methods
> Establishing class invariants
> **Printing state**
> Self and super
> Accessors and Query methods
> Decomposing and naming methods

Viewing the game state

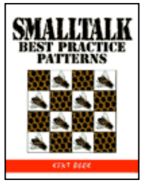
```
SnakesAndLadders example inspect
```



Not very informative

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

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

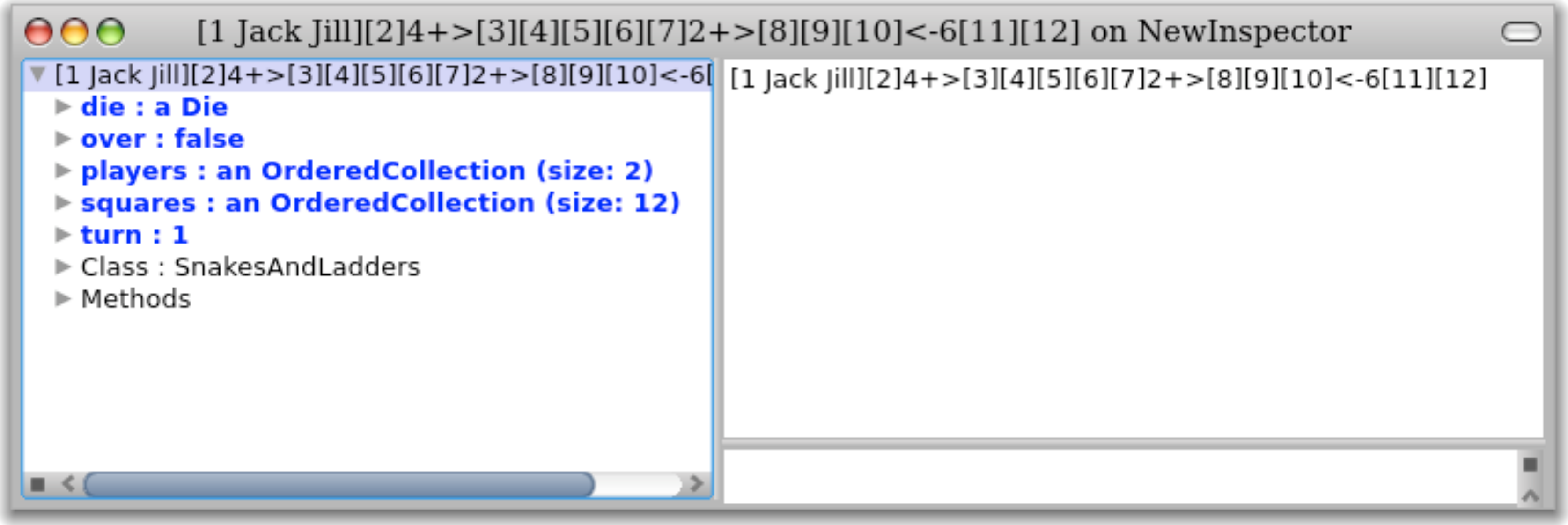
— Put printing methods in the "printing" protocol

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

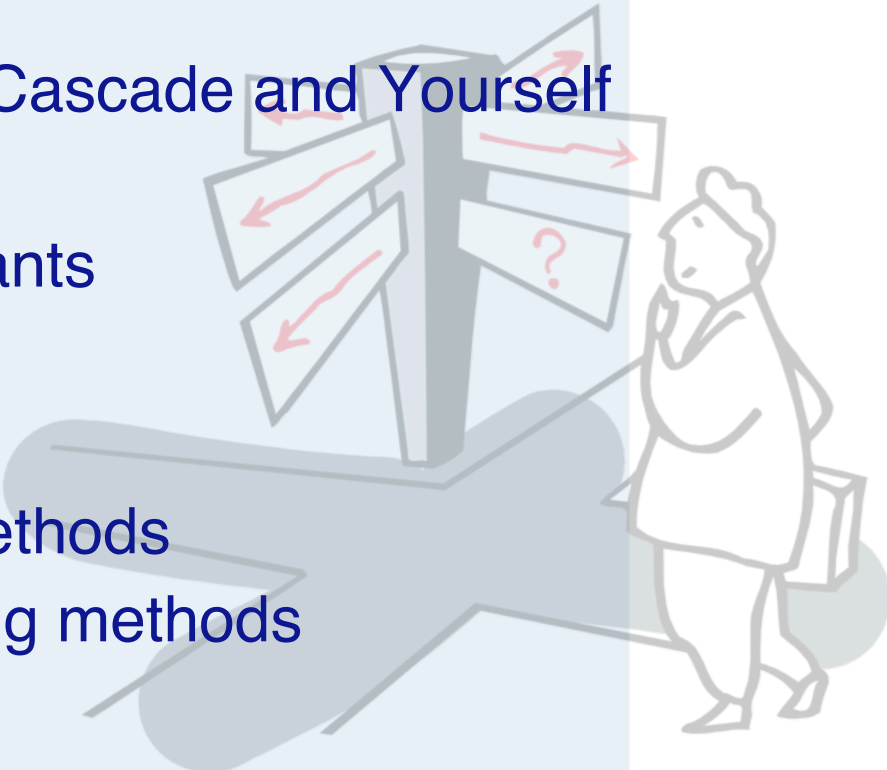
```
SnakesAndLadders example inspect
```



Interacting with the game

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

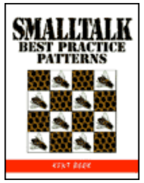
Roadmap



> Snakes and Ladders — Cascade and Yourself
> Lots of Little Methods
> Establishing class invariants
> Printing state
> **Self and super**
> Accessors and Query methods
> Decomposing and naming methods

Super

How can you invoke superclass behaviour?

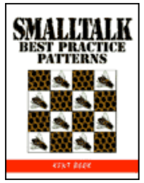


> Invoke code in a superclass explicitly by sending a message to `super` instead of `self`.

- 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

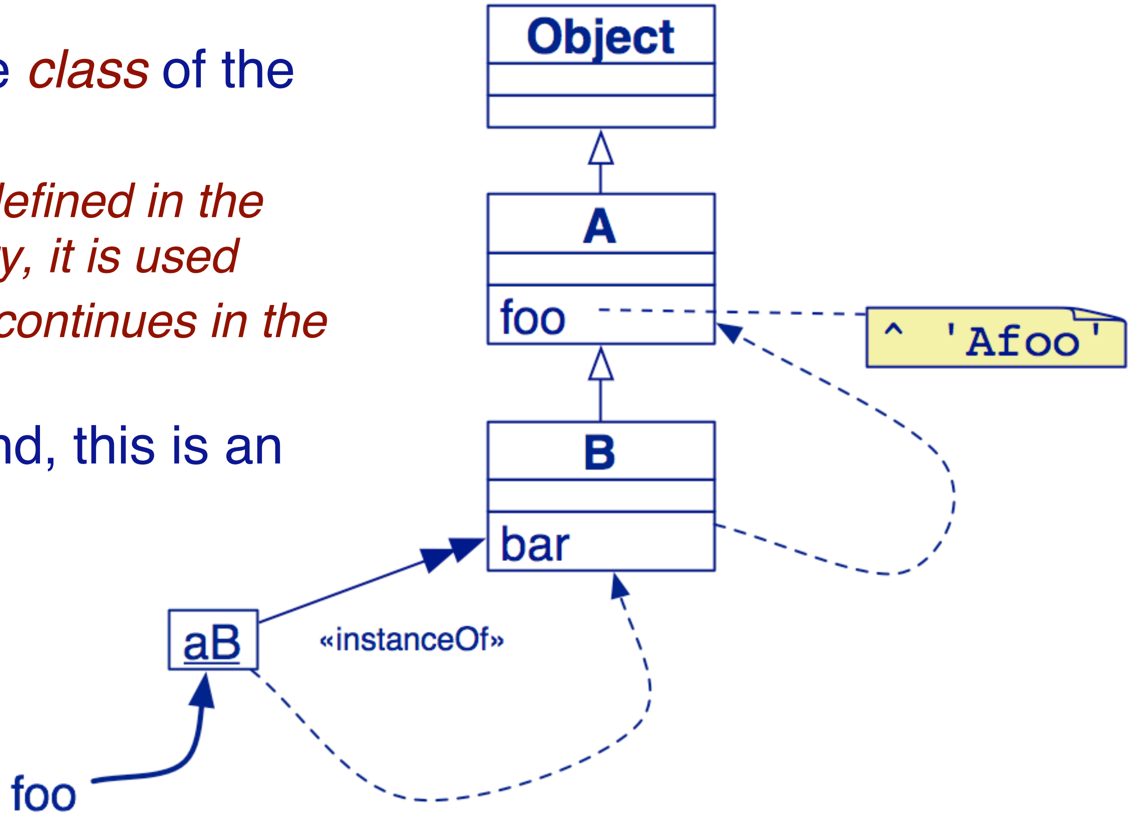
> Snake and Ladder both extend the printOn: method of their superclass

```
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

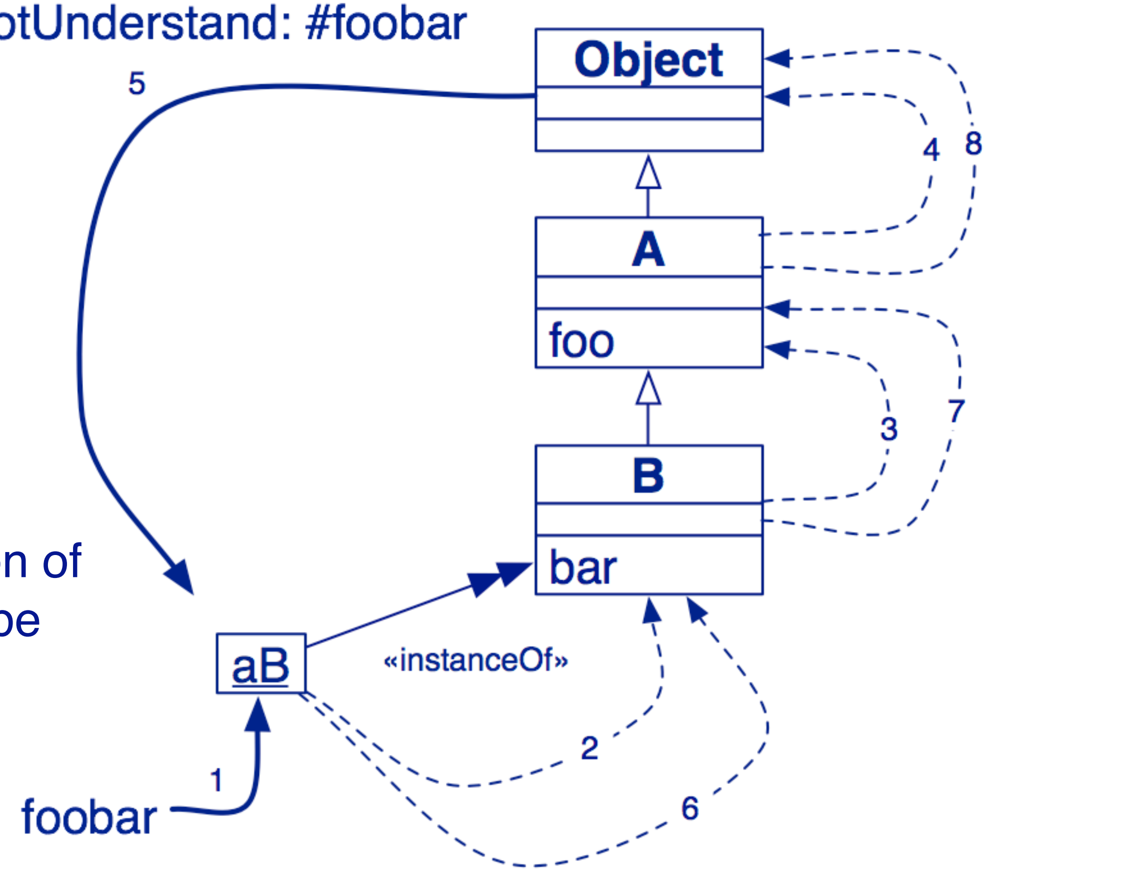
Two step process:

- Lookup starts in the *class* of the *receiver* (an object)
  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.



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

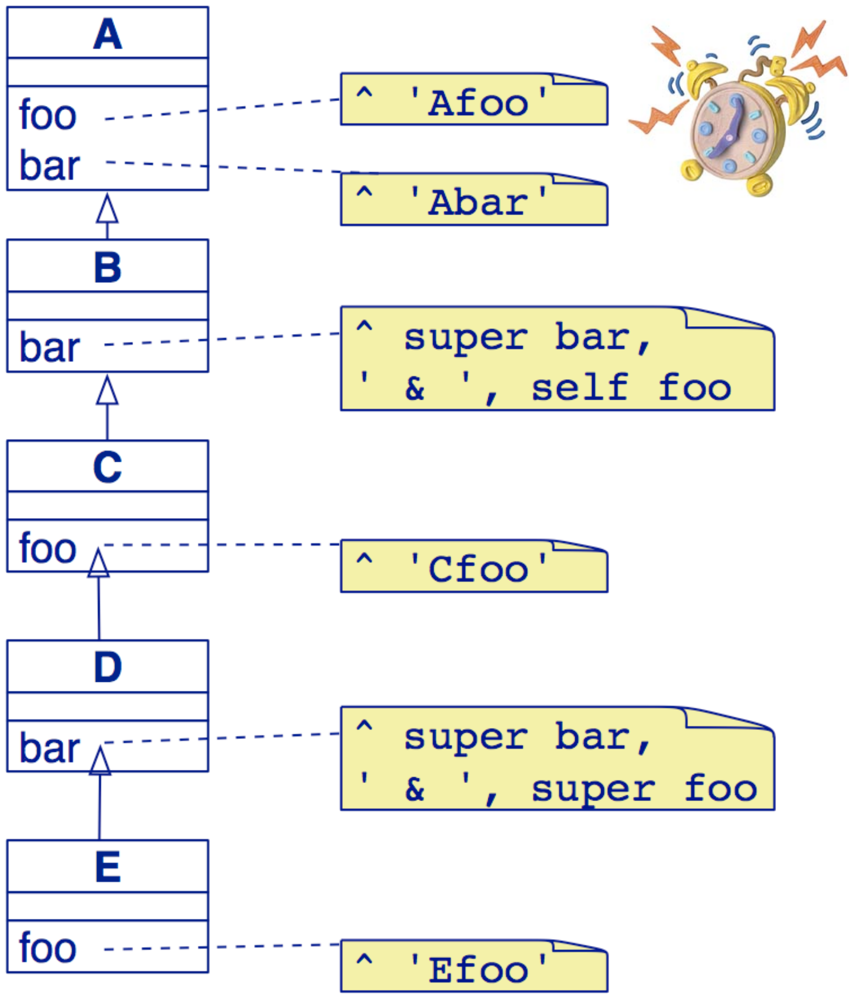
Super

> 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

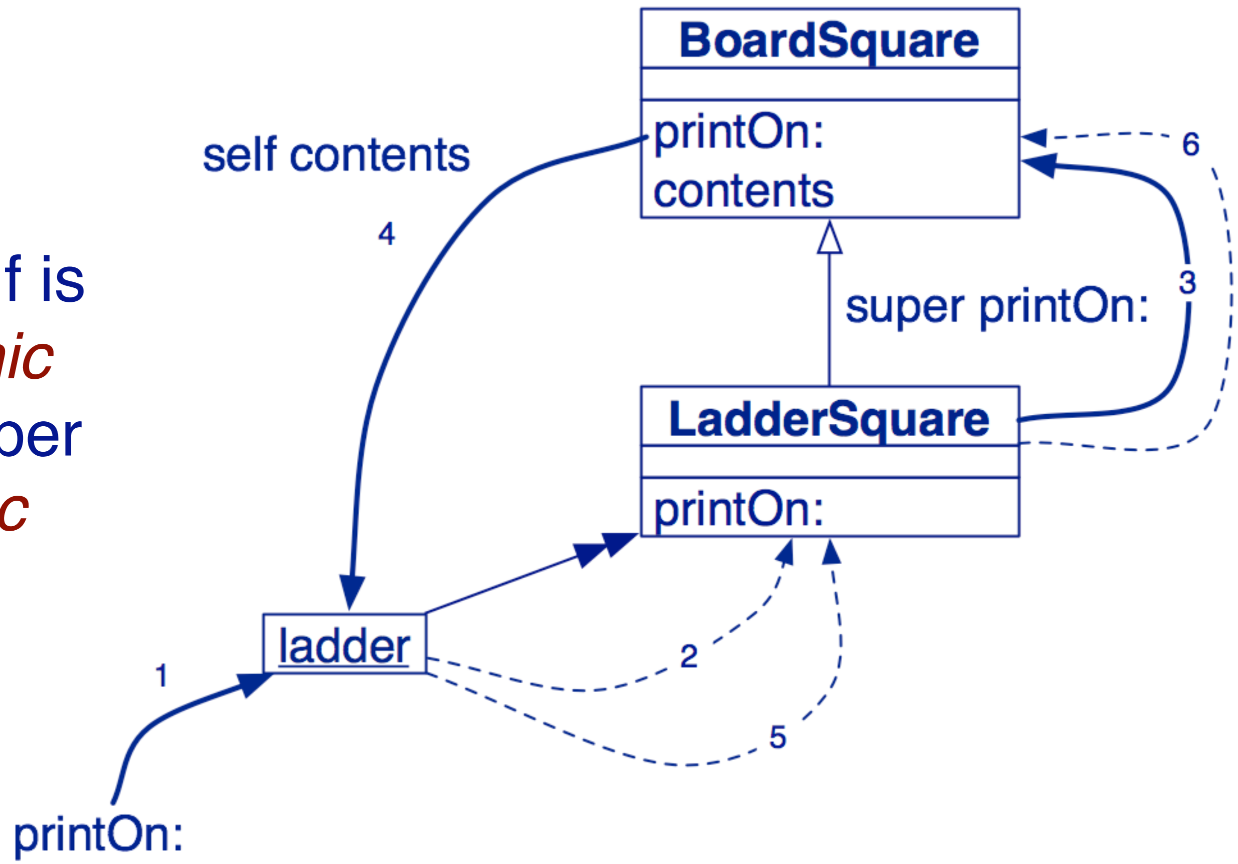
`A new bar`	**`'Abar'`**
`B new bar`	**`'Abar & Afoo'`**
`C new bar`	**`'Abar & Cfoo'`**
`D new bar`	**`'Abar & Cfoo & Cfoo'`**
`E new bar`	**`'Abar & Efoo & Cfoo'`**



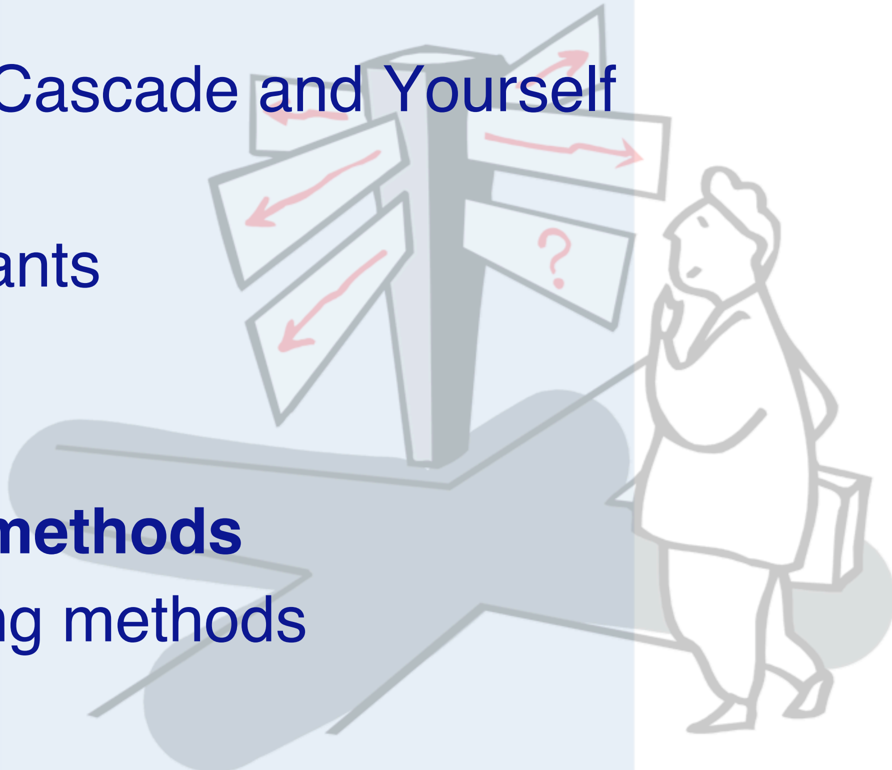
NB: It is usually a *mistake* to super-send to a different method. *D*>>`bar` should probably do `self foo`, not `super foo`!

Self and super

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



Roadmap



- Snakes and Ladders — Cascade and Yourself
- Lots of Little Methods
- Establishing class invariants
- Printing state
- Self and super
- **Accessors and Query methods**
- Decomposing and naming methods

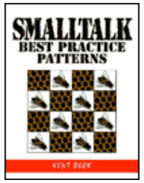
Testing

> In order to enable deterministic test scenarios, we need to fix the game with a loaded die!
  - *The loaded die will turn up the numbers we tell it to.*

```
SnakesAndLaddersTest>>setUp
   eg := self example.
   loadedDie := LoadedDie new.
   eg setDie: loadedDie.
   jack := eg players first.
   jill := eg players last.
```

Getting Method

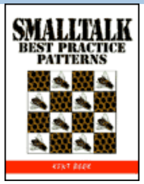
How do you provide access to an instance variable?



> Provide a method that returns the value of the variable.
  - Give it the same name as the variable.
    - *NB: not called "get…"*

```
LoadedDie>>roll
    self assert: roll notNil.
    ^ roll
```

Setting Method



How do you change the value of an instance variable?

- Provide a method with the same name as the variable.
  - Have it take a single parameter, the value to be set.
    - *NB: not called "set..."*

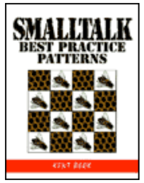
```
LoadedDie>>roll: aNumber
   self assert: ((1 to: 6) includes: aNumber).
   roll := aNumber.
```

Testing the state of objects

> To enable tests, we will need to implement various *query methods*

```
SnakesAndLaddersTest>>testStartPosition
    self assert: eg lastPosition = 12.
    self assert: eg isNotOver.
    self assert: eg currentPlayer = jack.

    self assert: eg firstSquare isFirstSquare.
    self assert: eg firstSquare isLastSquare not.
    self assert: eg firstSquare position = 1.
    self assert: eg firstSquare isOccupied.
    self assert: (eg at: eg lastPosition) isFirstSquare not.
    self assert: (eg at: eg lastPosition) isLastSquare.
    self assert: (eg at: eg lastPosition) position = 12.
    self assert: (eg at: eg lastPosition) isOccupied not.

    self assert: jack name = 'Jack'.
    self assert: jill name = 'Jill'.
    self assert: jack position = 1.
    self assert: jill position = 1.
```

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
```

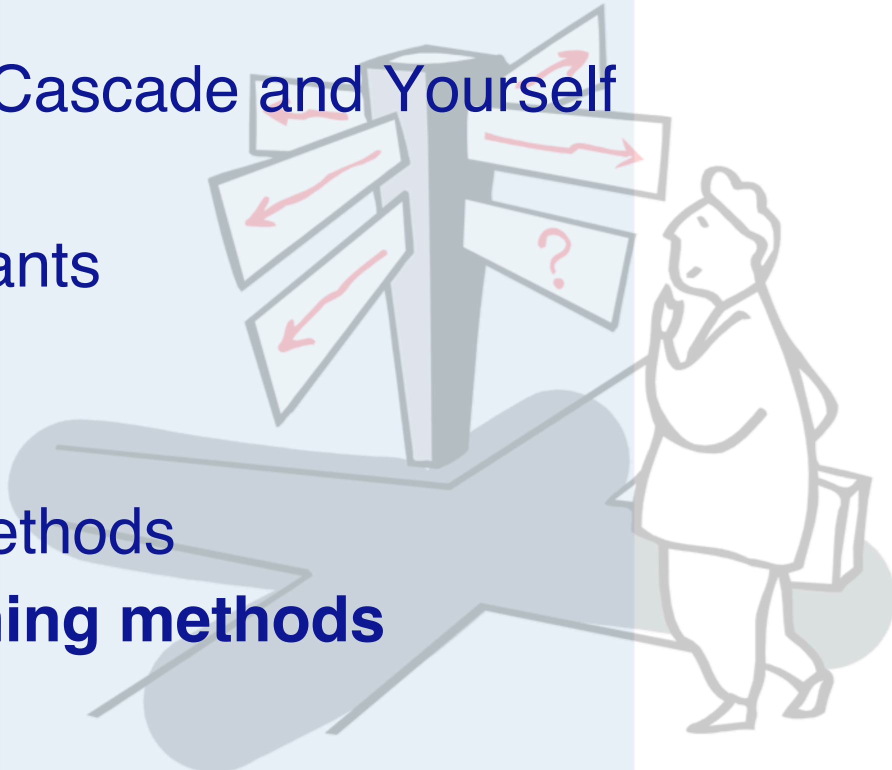
A Test Scenario

> To carry out a test scenario, we need to *play a fixed game* instead of a random one.

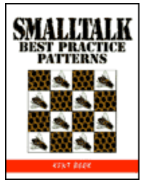
```
SnakesAndLaddersTest>>testExample
   self assert: eg currentPlayer = jack.
   loadedDie roll: 1.
   eg playOneMove.
   self assert: jack position = 6.

   …

   self assert: eg currentPlayer = jack.
   loadedDie roll: 2.
   eg playOneMove.
   self assert: jack position = 12.

   self assert: eg isOver.
```

Roadmap

> Snakes and Ladders — Cascade and Yourself
> Lots of Little Methods
> Establishing class invariants
> Printing state
> Self and super
> Accessors and Query methods
> **Decomposing and naming methods**



Composed Method



How do you divide a program into methods?

> Divide your program into methods that perform one identifiable task.

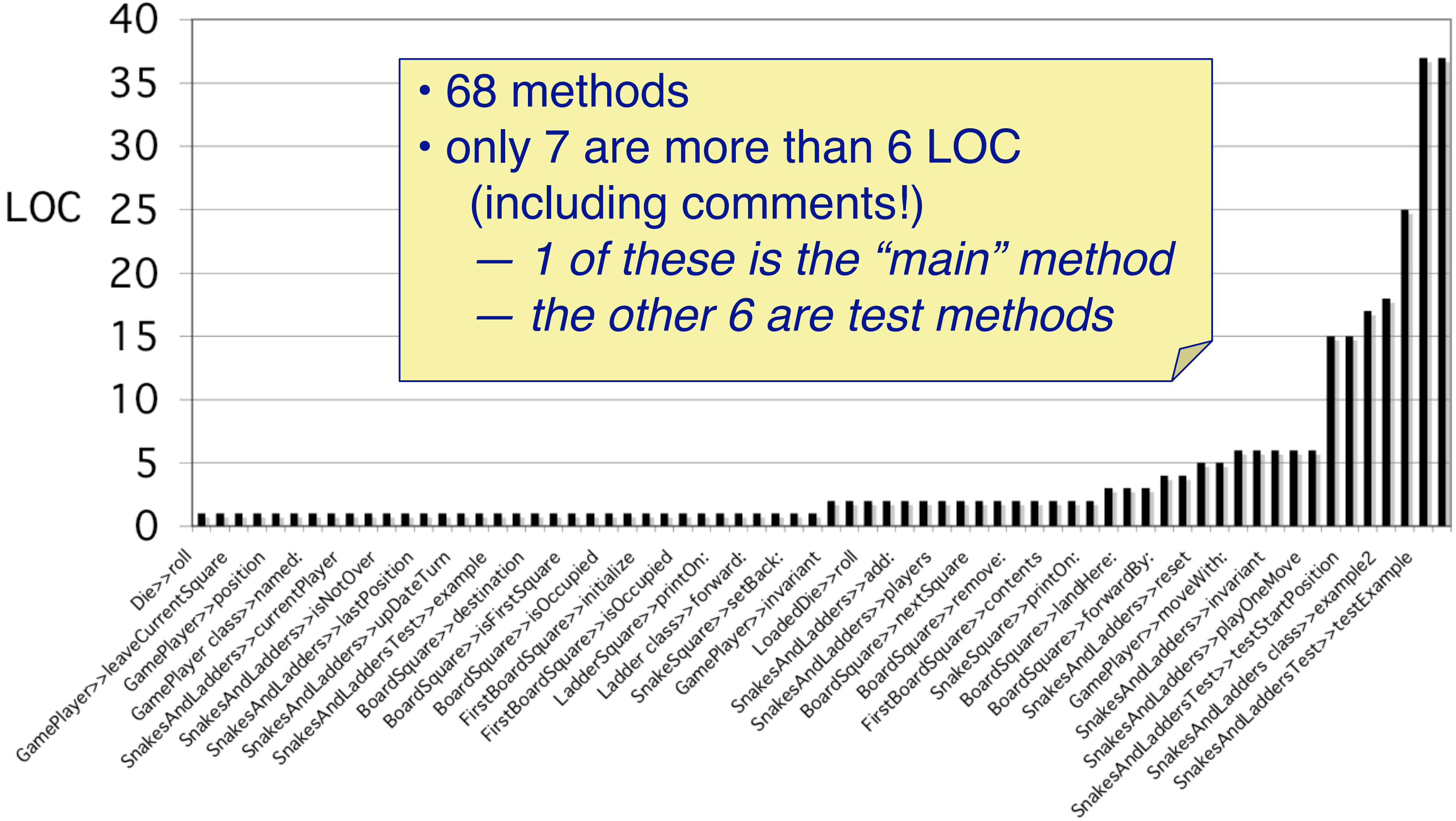
- — Keep all of the operations in a method at the *same level of abstraction*.
- — This will naturally result in programs with *many small methods, each a few lines long*.

Method size

- Most methods will be small and self-documenting
  - Few exceptions:
    - *Complex algorithms*
    - *Scripts (configurations)*
    - *Tests*

```
SnakesAndLadders>>playOneMove
    | result |
    self assert: self invariant.
    ^ self isOver
      ifTrue: ['The game is over']
      ifFalse: [
          result :=
              (self currentPlayer moveWith: die),
              self checkResult.
          self upDateTurn.
          result ]
```

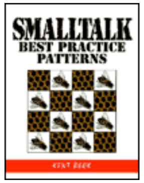
Snakes and Ladders methods



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

Intention Revealing Message

How do you communicate your intent when the implementation is simple?



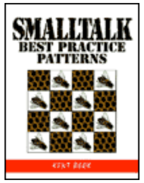
> Send a message to self.

- Name the message so it communicates what is to be done rather than how it is to be done.
- Code a simple method for the message

```
SnakesAndLadders>>currentPlayer
   ^ players at: turn
```

Intention Revealing Selector

What do you name a method?



- Name methods after what they accomplish.
  - Well-named methods can eliminate the need for most comments

```
SnakesAndLadders>>upDateTurn
   turn := 1 + (turn\\players size).
```

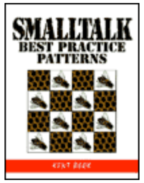
Some Naming Conventions

> Use imperative verbs for methods performing an action
  — `moveTo:`, `leaveCurrentSquare`, `playOneMove`

> Prefix testing methods (i.e., that return a boolean) with "is" or "has"
  — `isNil`, `isNotOver`, `isOccupied`

> Prefix converting methods with "as"
  — `asString`

Message Comment



How do you comment methods?

> Communicate important information that is not obvious from the code in a comment at the beginning of the method.
  - Method dependencies
  - To-do items
  - Sample code to execute

```
SnakesAndLadders>>playToEnd
    "SnakesAndLadders example playToEnd"
    …
```

Hint: comments may be code smells in disguise!
— Try to refactor code and rename methods to get rid of comments!

What you should know!

- *What does* `yourself` *return? Why is it needed?*
- *How is a new instance of a class initialized?*
- *When should you implement invariants and preconditions?*
- *What happens when we evaluate an expression with "print it"?*
- *Why should a method never send* `super` *a different message?*
- *How is* `super` *static and* `self` *dynamic?*
- *How do you make your code self-documenting?*

Can you answer these questions?

- *When should you override* `new`*?*
- *If instance variables are really private, why can we see them with an inspector?*
- *When does* `self = super`*?*
- *When does* `super = self`*?*
- *Which classes implement* `assert:` *?*
- *What does* `self` *refer to in the method* `SnakesAndLadders class>>example`*?*

License

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