A Browser for Incremental Programming

Nathanael Schärli (SCG, University of Bern)
and
Andrew P. Black (CSE, Oregon Health & Science University)

A Browser for Incremental Programming

Nathanael Schärli (SCG, University of Bern)
and
Andrew P. Black (CSE, Oregon Health & Science University)

A Browser for Incremental Programming

Nathanael Schärli (SCG, University of Bern)
and
Andrew P. Black (CSE, Oregon Health & Science University)

me

What *is* Incremental Programming?

- What we do in Smalltalk!
- One of the Extreme Programming (XP) practices
- Characterized by some patterns of work that should be familiar to you:

Incremental work patterns

- Programming with limited knowledge
- Working in multiple contexts
- Refactoring
- Testing
- Understanding how classes collaborate
- Understanding what is still missing

Incremental work patterns

- Programming with limited knowledge
  - Generic protocols, absence of declarations
- Working in multiple contexts
- Refactoring
- Testing
- Understanding how classes collaborate
- Understanding what is still missing

Incremental work patterns

- Programming with limited knowledge
  - Generic protocols, absence of declarations
- Working in multiple contexts
  - Multiple windows, tiling browsers
- Refactoring
- Testing
- Understanding how classes collaborate
- Understanding what is still missing

Incremental work patterns

- Programming with limited knowledge
  - Generic protocols, absence of declarations
- Working in multiple contexts
  - Multiple windows, tiling browsers
- Refactoring
  - Refactoring Browser
- Testing
- Understanding how classes collaborate
- Understanding what is still missing

Incremental work patterns

- Programming with limited knowledge
  - Generic protocols, absence of declarations
- Working in multiple contexts
  - Multiple windows, tiling browsers
- Refactoring
  - Refactoring Browser
- Testing
  - SUnit
- Understanding how classes collaborate
- Understanding what is still missing

Incremental work patterns

- Programming with limited knowledge
  - Generic protocols, absence of declarations
- Working in multiple contexts
  - Multiple windows, tiling browsers
- Refactoring
  - Refactoring Browser
- Testing
  - SUnit
- Understanding how classes collaborate
  - ?
- Understanding what is still missing

Incremental work patterns

- Programming with limited knowledge
  - Generic protocols, absence of declarations
- Working in multiple contexts
  - Multiple windows, tiling browsers
- Refactoring
  - Refactoring Browser
- Testing
  - SUnit
- Understanding how classes collaborate
  - ?
- Understanding what is still missing
  - ?

How should we support Incremental Programming?

- Provide information about completeness of classes and collaborations between classes that is
  - statically computed,
  - always accessible, and
  - always up-to-date
- Why not? This information *is* in the code
  - My PowerBook is 50 times faster than a Dorado

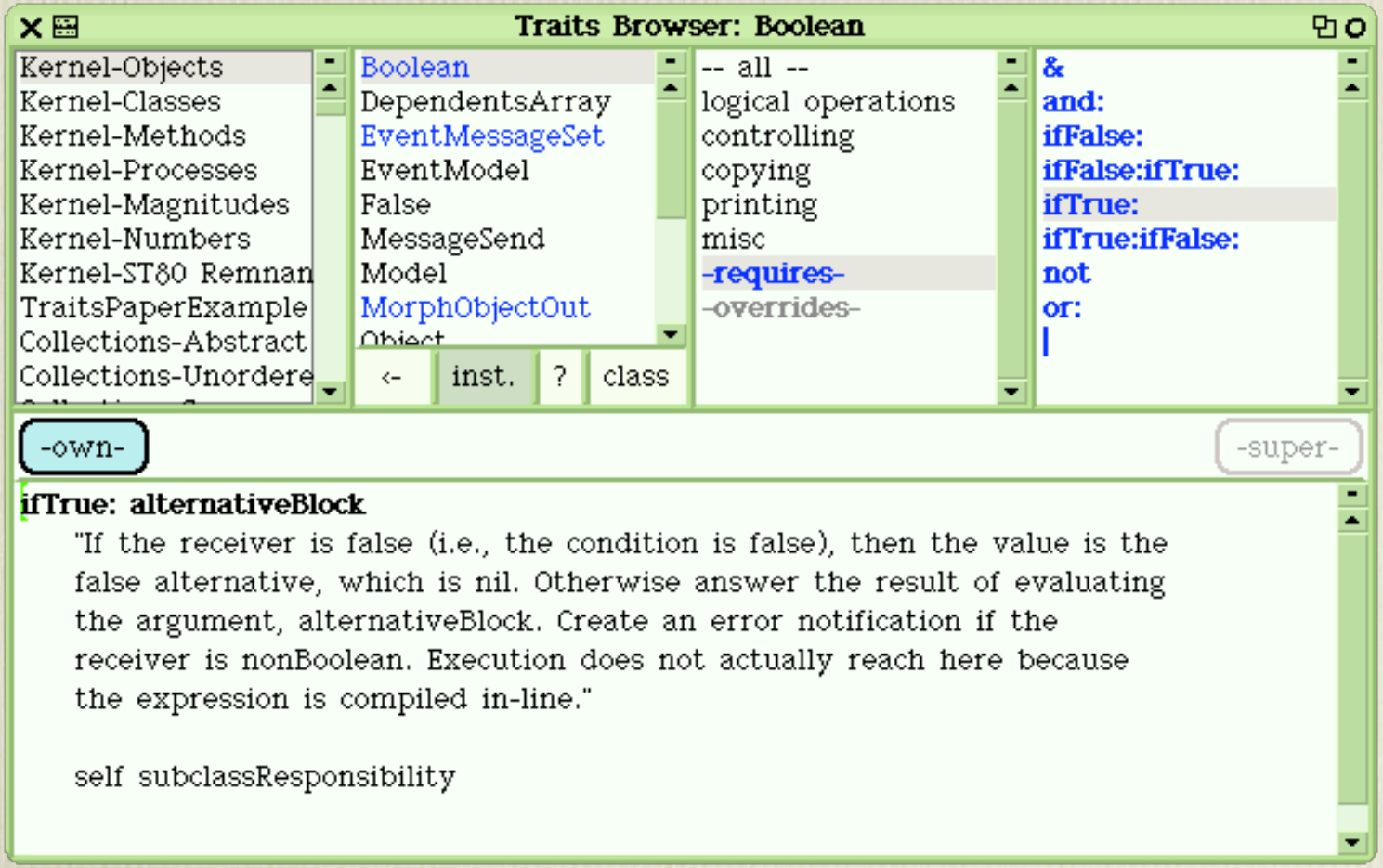
How to present it?

How to present it?

- Virtual Categories

How to present it?

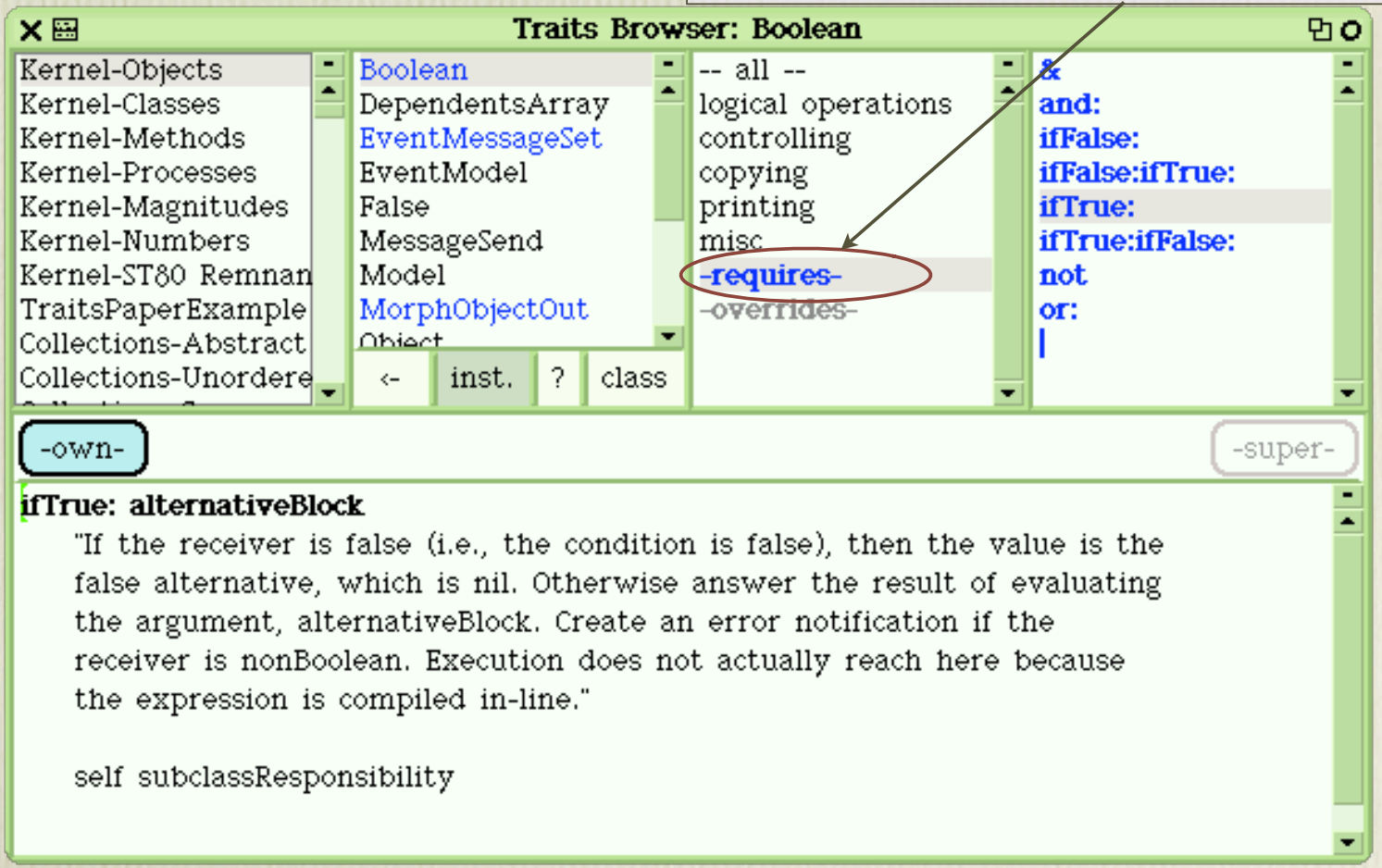
- Virtual Categories



How to present it?

- Virtual Categories

categorization of methods by the browser, based on their characteristics; always up-to-date



Four Categories:

-requires-

- all messages sent to this class for which there is no method defined or inherited

-supplies-

- all messages required by some other class for which methods are provided in this class

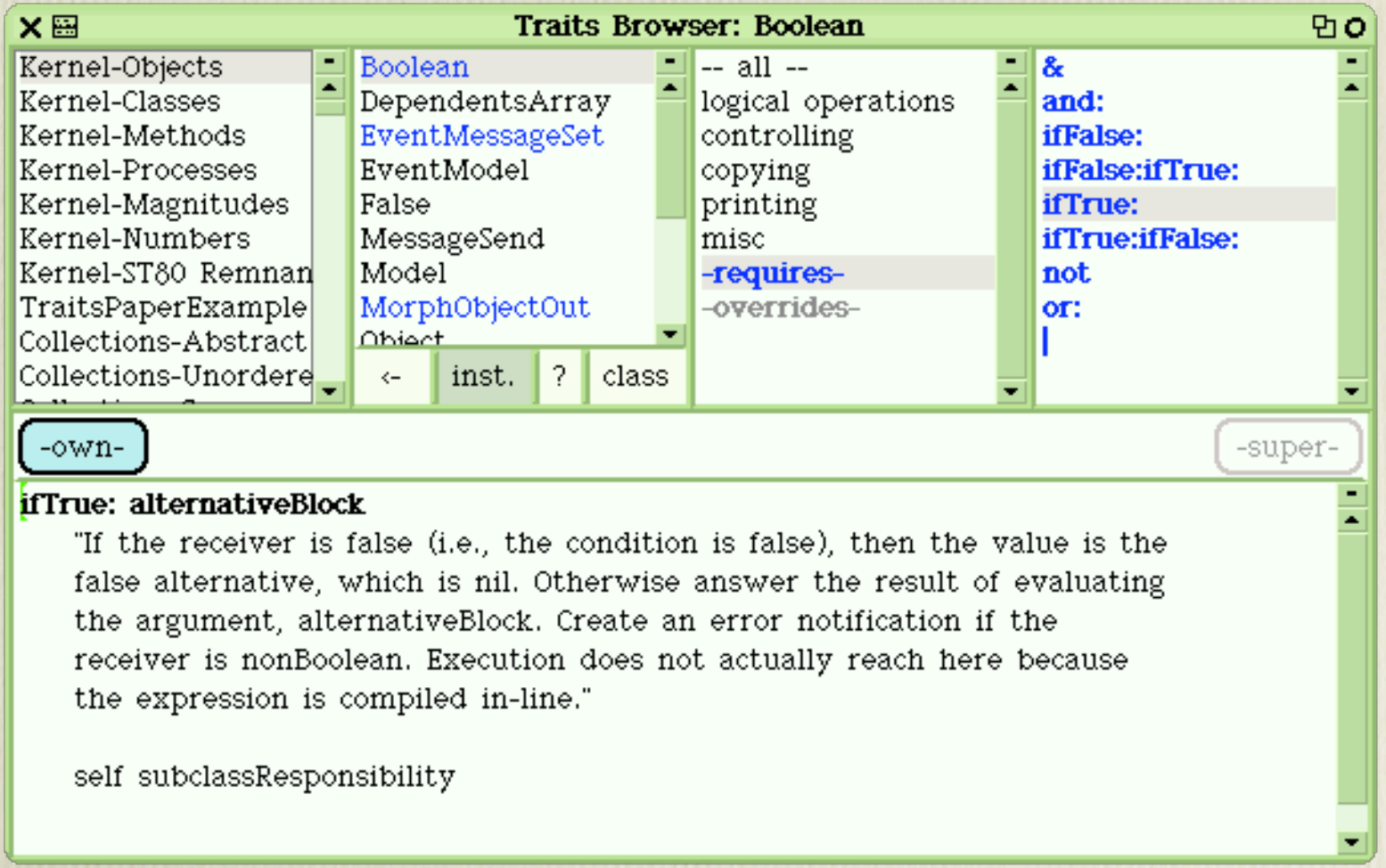
-overrides-

- methods defined in this class that override inherited methods

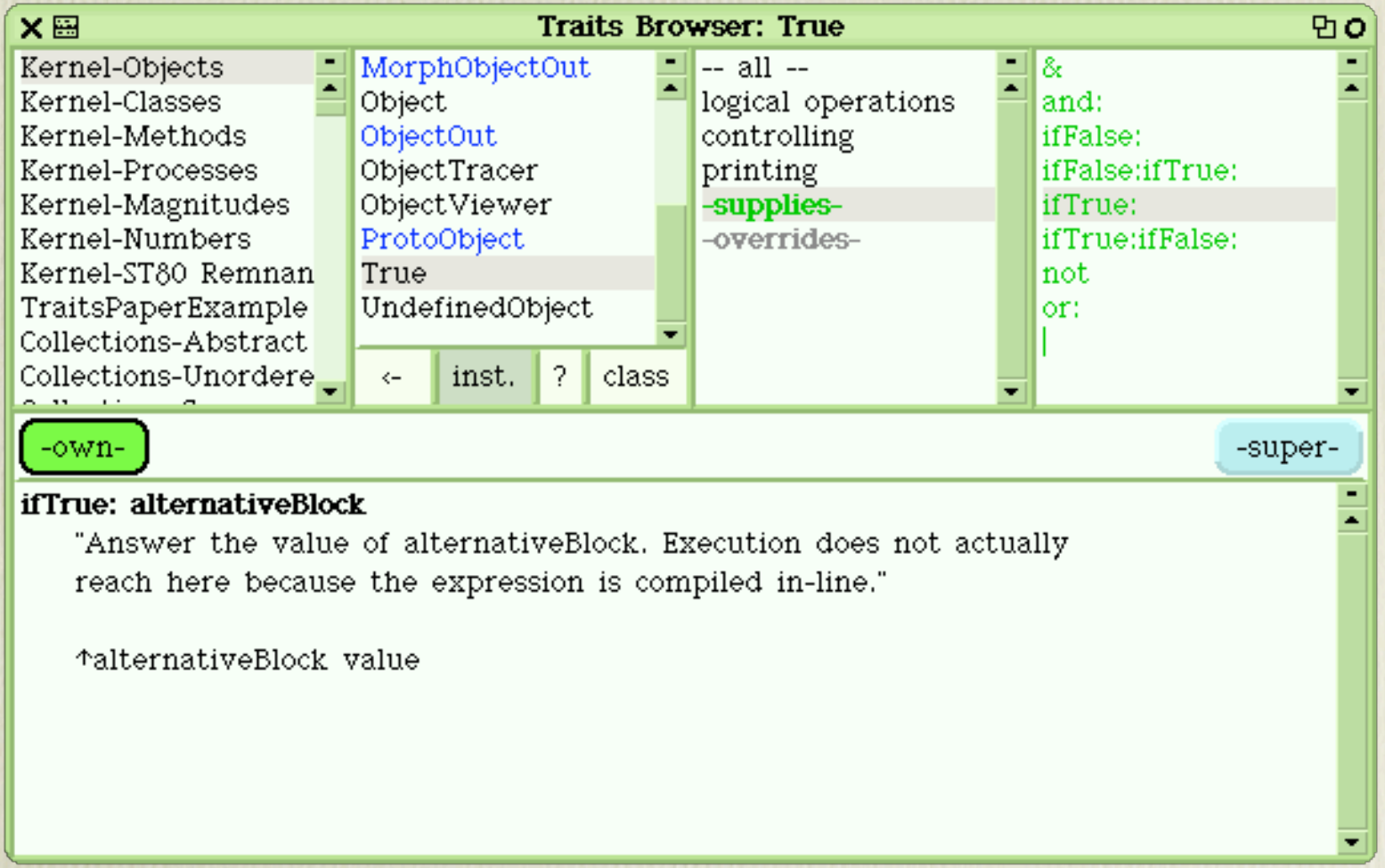
-sending super-

- methods that perform *super sends*

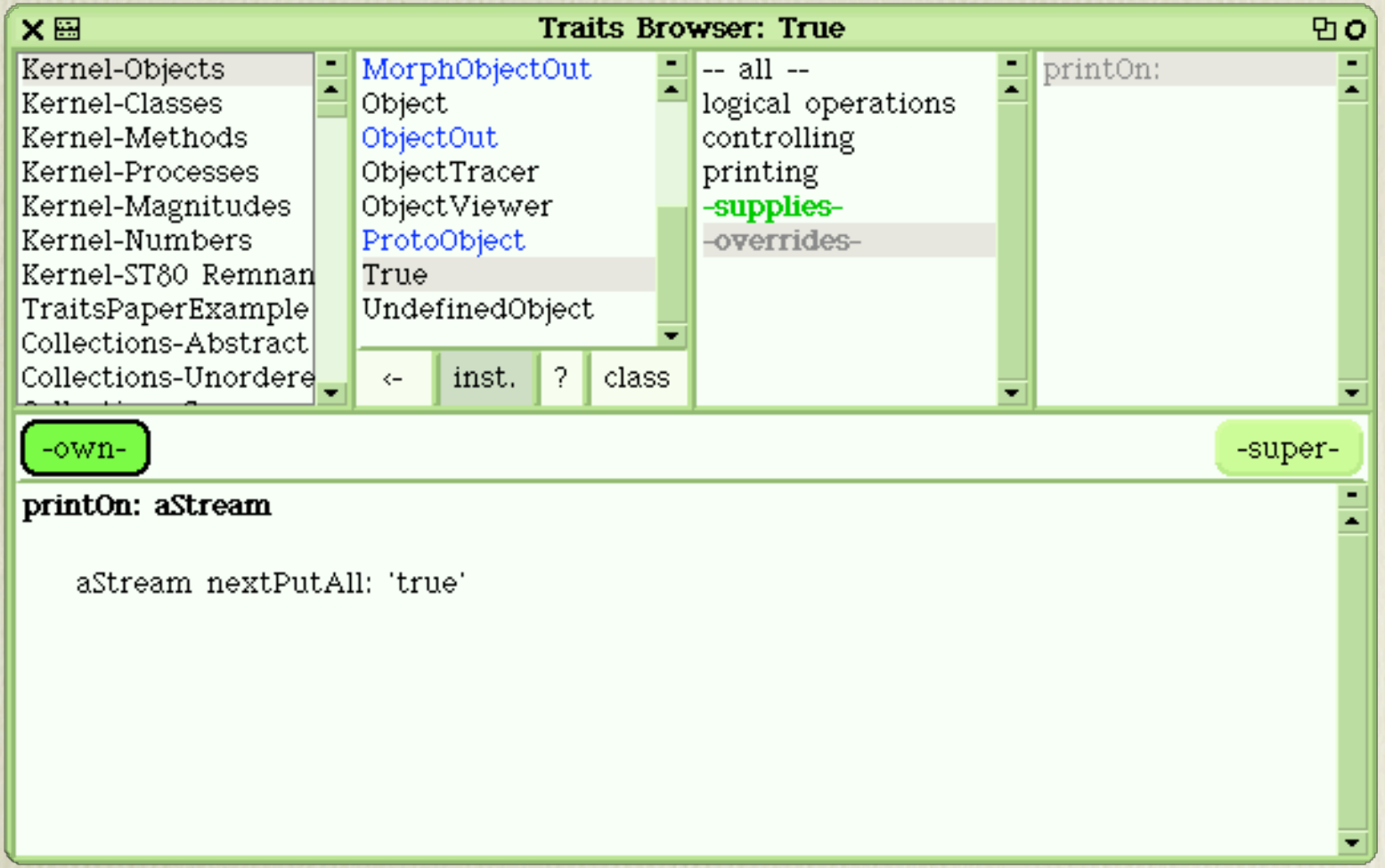
supplies & overrides



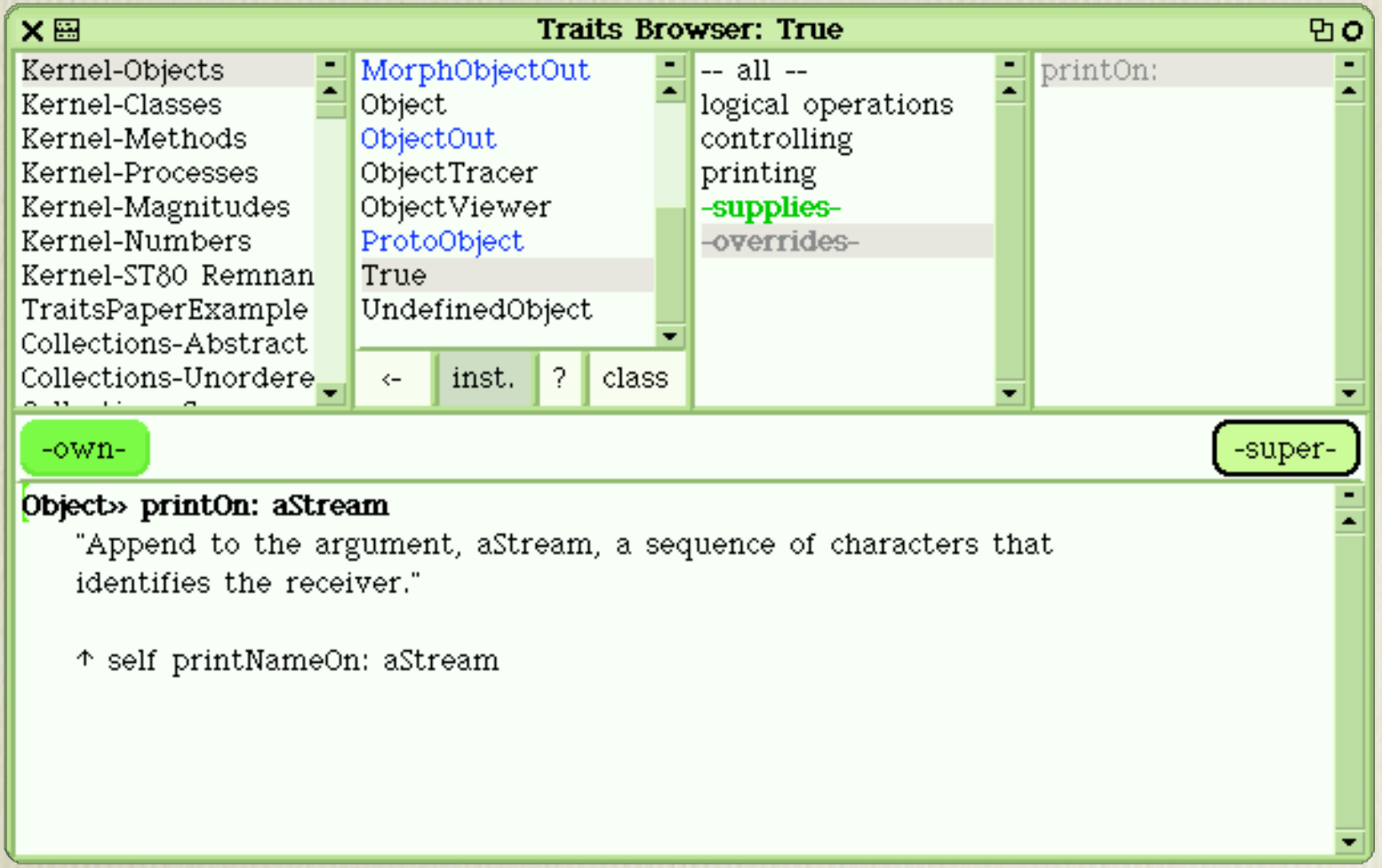
supplies & overrides



supplies & overrides

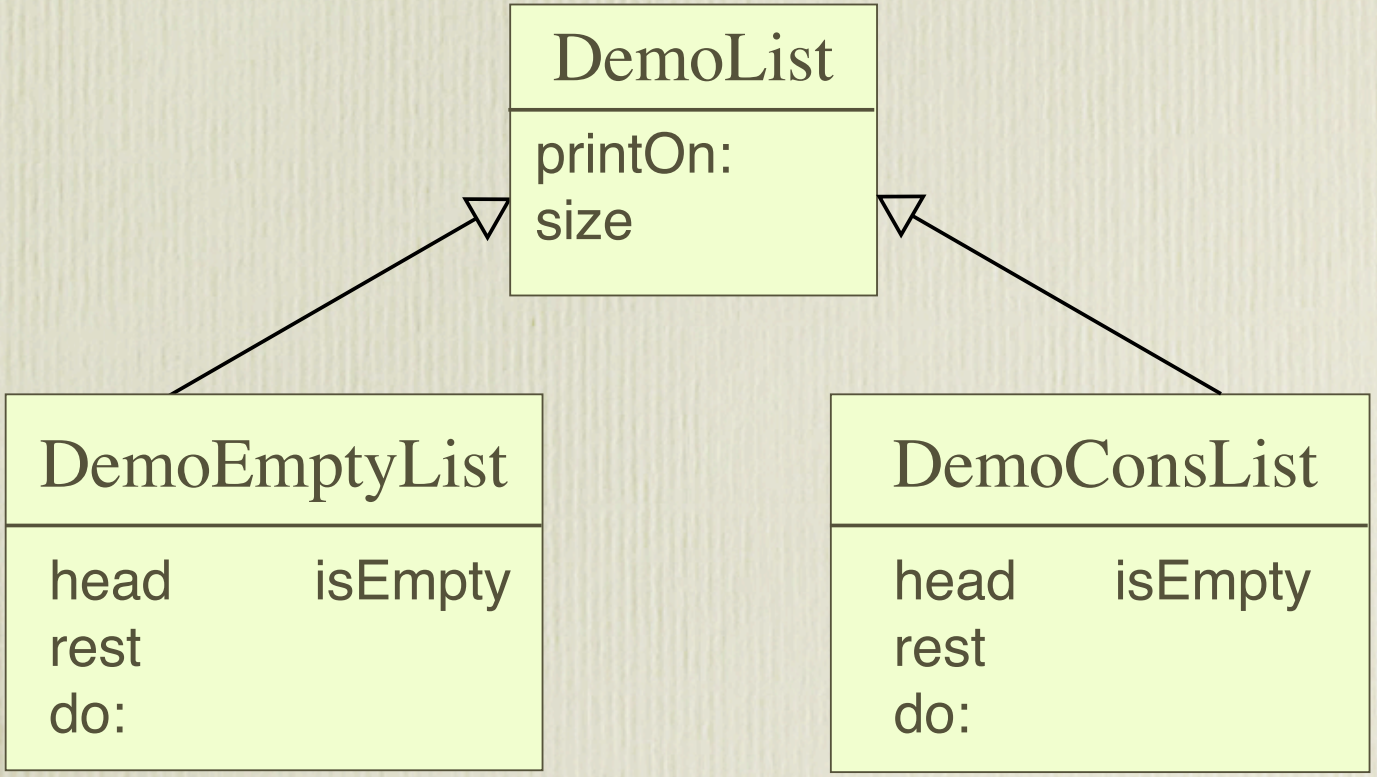


supplies & overrides



Extended Example/Demonstration

- Creating a new sub-tree of classes



Understanding and Modifying Existing Hierarchies

- Feedback from the browser helps us find:
  - which methods are “core” and which are “support”
  - how the sub- and superclasses depend on each other
- When extending code, the browser helps avoid:
  - introducing inter-level errors
  - accidentally incomplete classes

The Core/Support Split

- A common pattern used to increase reuse in a data type implementation [Black ECOOP inh wk 2002]
  - *e.g.*, the abstract superclass `Collection` defines 110 *support* methods
    - they don't access the state of any collection directly
    - instead, they depend on 4 *core* methods
      - `add:`, `atRandom:`, `do:` and `remove:ifAbsent:`
        - 3 are defined as `self subclassResponsibility`
        - `atRandom:` is not defined at all
    - The browser finds these 4 required methods amongst the 110!

The Core/Support Split (2)

- When we look at a subclass of `Collection`, *e.g.,* `Bag`, we can distinguish:
  - the 4 supplied methods,
  - 10 methods that override the inherited methods, either to disable them or to improve their efficiency, and
  - 7 additional methods that widen the interface of `Bag` beyond that of `Collection`

Accidentally Abstract Classes

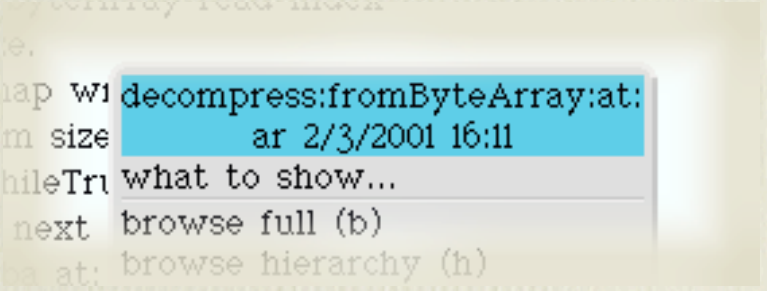
- The browser tells us some surprising things about Squeak's core classes:
  - Fraction is abstract
    - it implements the support method printOn:, whereas it should implement the core method printOn:Base:
  - Bitmap is abstract
    - the programmer sends an error message primitiveFail, which he forgot to define
  - Debugger, CharacterSet, Morph (and nearly all of its subclasses): all are abstract

Accidentally Abstract Classes (2)

- *Why* are these errors present in a code base that has been used by thousands of users for many years?
  - It is *not* because a bad programmer wrote BitMap
- It is because even good programmers will make mistakes unless they have good tools

Accidentally Abstract Classes (2)

- *Why* are these errors present in a code base that has been used by thousands of users for many years?
  - It is *not* because a bad programmer wrote BitMap



- It is because even good programmers will make mistakes unless they have good tools

Implementation

-sending super- is easy

- look for the bytecode for *super sends*

-overrides- is easy

- compare this class's selectors with its superclass's protocol

-supplies- is easy once one knows requires

- compare this class's selectors with the requires set of other classes

-requires- is most definitely *not* easy

- implementing requires in real-time required a lot of careful thinking and more careful programming!

What's in the requires category?

- Pseudocode:

```
Behavior >> requires
    self reachableMethods selfMessages
        difference:
            (self allReallyImplementedSelectors)
```

What's in the requires category?

We had to invent and formalize a definition of reachability

- Pseudocode:

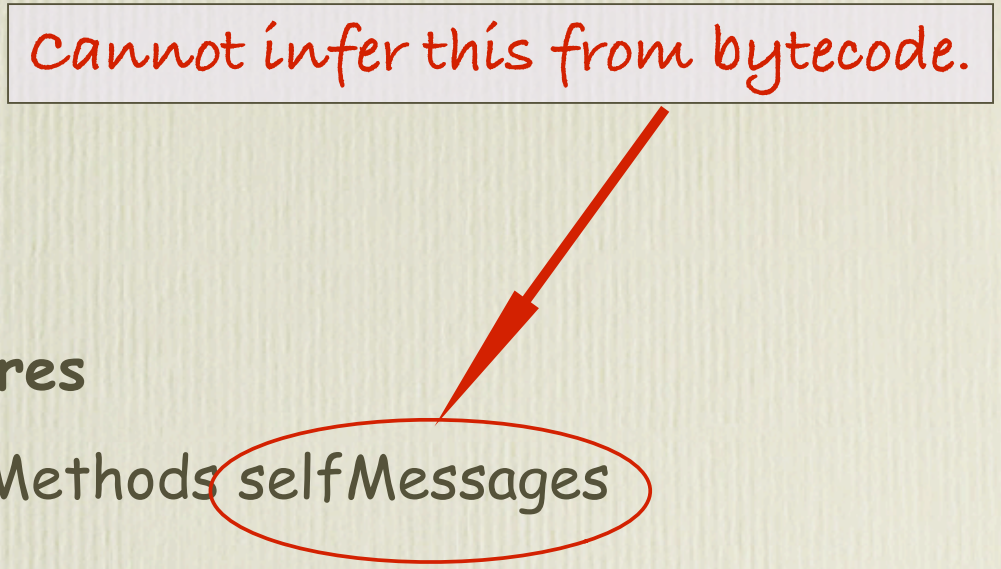
```
Behavior >> requires
    self reachableMethods selfMessages
        difference:
            (self allReallyImplementedSelectors)
```

What's in the requires category?

- Pseudocode:

```
Behavior >> requires
    self reachableMethods selfMessages
        difference:
            (self allReallyImplementedSelectors)
```

What's in the requires category?



- Pseudocode:

```
Behavior >> requires
    self reachableMethods selfMessages
        difference:
            (self allReallyImplementedSelectors)
```

What's in the requires category?

- Pseudocode:

```
Behavior >> requires
    self reachableMethods selfMessages
        difference:
            (self allReallyImplementedSelectors)
```

What's in the requires category?

- Pseudocode:

```
Behavior >> requires
    self reachableMethods selfMessages
        difference:
            (self allReallyImplementedSelectors)
```

allSelectors \ those not really implemented (subclassResponsibility, shouldNotImplement, requirement, etc.)

Recognizing self-sends

- Recognizing self-sends requires a full parse of the method text
- A change in, say, Object, might change the required methods of every class in the system!
- Squeak images contain > 60 000 methods
- We decided that we needed to cache the self-sends for every method when it is compiled

A problem of scale

- Even with these caches for self- and super sends, the first implementation took over 3 minutes to ascertain the required methods of a class!

Two key insights

- The caches should be arranged "backwards"
  - for each *message*, cache the methods that self-send it
- We don't need to know the requires set, all we need to know is whether it is empty
  - Does a subclass override *all* of the methods that self-send a message required by the superclass?
    - if not, we *immediately* know that it is also required in the subclass

The Complete Algorithm...

- is far too complex to put on a slide
  - that's what the paper is for!
- Computing the required set now takes less than 100 ms — fast enough to provide "real-time" feedback

The Complete Algorithm...

- is far too complex to put on a slide
  - that's what the paper is for!
- Computing the required set now takes less than 100 ms — fast enough to provide "real-time" feedback

A Browser for Incremental Programming

Nathanael Schärli and Andrew P. Black

Software Composition Group, University of Bern, Switzerland

OGI School of Science & Engineering, Oregon Health and Science University, USA

Abstract

Much of the elegance and power of Smalltalk comes from its programming environment and tools. First introduced more than 20 years ago, the Smalltalk browser enables programmers to "home in" on particular methods using a hierarchy of manually defined classifications. By its nature, this classification scheme says a lot about the *desired* state of the code, but nothing at all about the actual state of the code as it is being developed. We have extended the Smalltalk browser with dynamically computed virtual categories that dramatically improve the browser's support for incremental programming. We illustrate these improvements by example, and summarize the algorithms used to compute the virtual categories efficiently.

Key words: Smalltalk browser, incremental programming, intentional programming, method reachability, requires set.

1 Introduction

The most important of the Smalltalk programming tools is the *Browser*, which allows the programmer to examine, modify and extend the code of applications and of the system itself. The Browser was revolutionary when it was first introduced, and over the intervening years it has been improved in several ways. For example, semi-automated refactoring has been added, leading to a tool known as the Refactoring Browser [10], and in many Smalltalk dialects some form of package construct has subsumed the original primitive categorization of classes. However, the way in which today's browser organizes the methods of a class is essentially the same as it was in 1980: a hierarchy of manually assigned "protocols".

In the meantime, the concept of the "Integrated Development Environment"—for that is what the Smalltalk toolset would now be called—has proved to be so successful that similar environments have been created for other programming languages. For example, IBM's VisualAge for Java [7] was essentially a re-targeting of

Email address: schaerli@iam.unibe.ch, blackcse.ogi.edu (Nathanael Schärli and Andrew P. Black).

12 August 2003

Preprint submitted to ESUG 2003 Academic Track

Related Work

- *To do* lists
  - Trellis's "grass catcher" was also the product of changing a single method
  - More commonly, as with Eclipse's "Tasks" window, to do lists are updated only on global recompilation.
- Browser extensions
  - decoration of names to indicate *local* properties such as overrides or sends to super, *e.g.*, in VisualWorks
  - Star Browser allows the definition of intentional classifications that are recomputed when necessary

Future Work

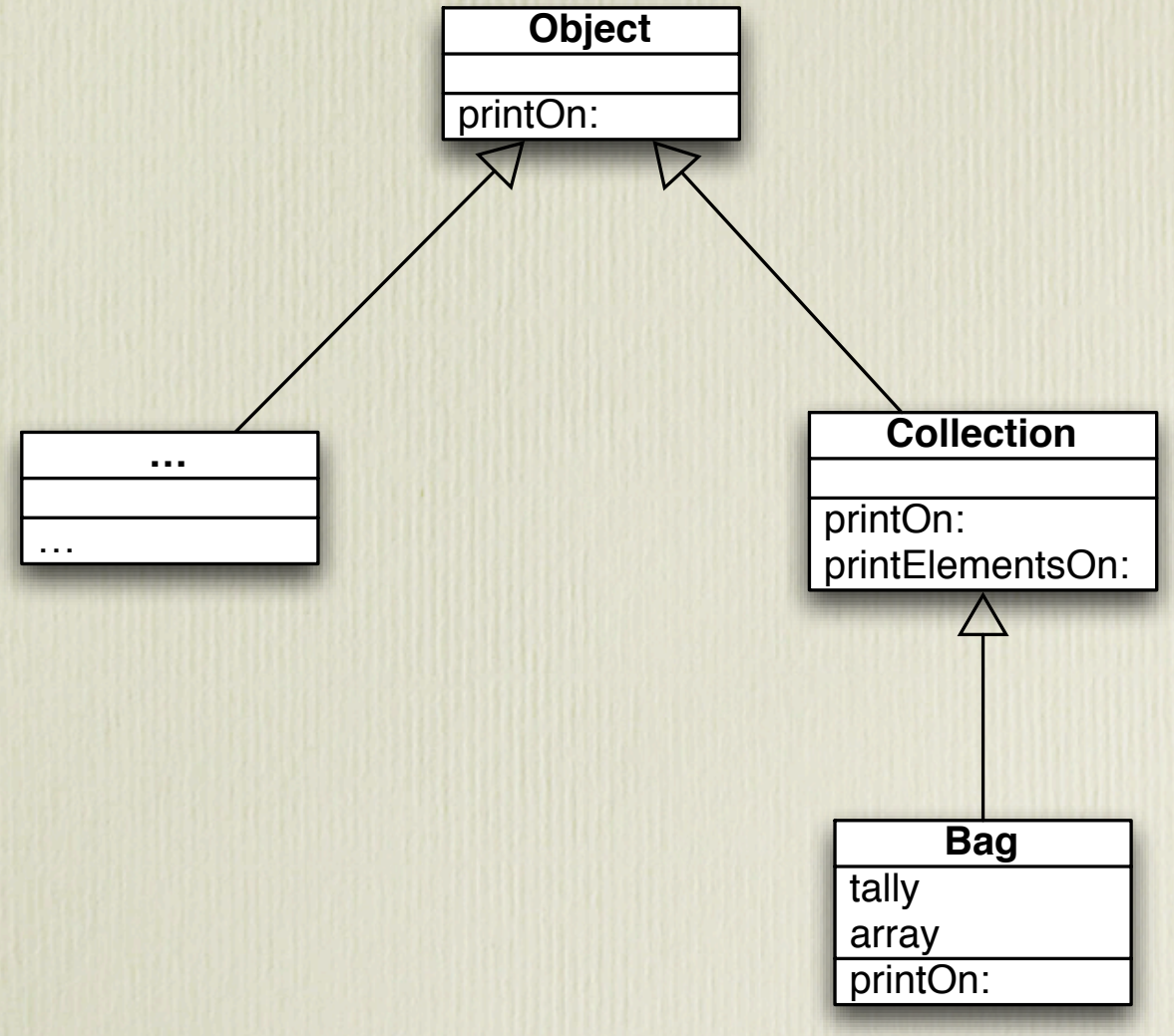
- Other visualizations of the *self-send* information
  - *e.g.*, Blueprint-like diagrams
- Two directions for extension:
  - Help in understanding other kinds of collaboration
    - *e.g.*, delegation, aggregation, Mudpie's package dependencies
  - A pluggable browser framework
    - what are the key features?

Conclusion

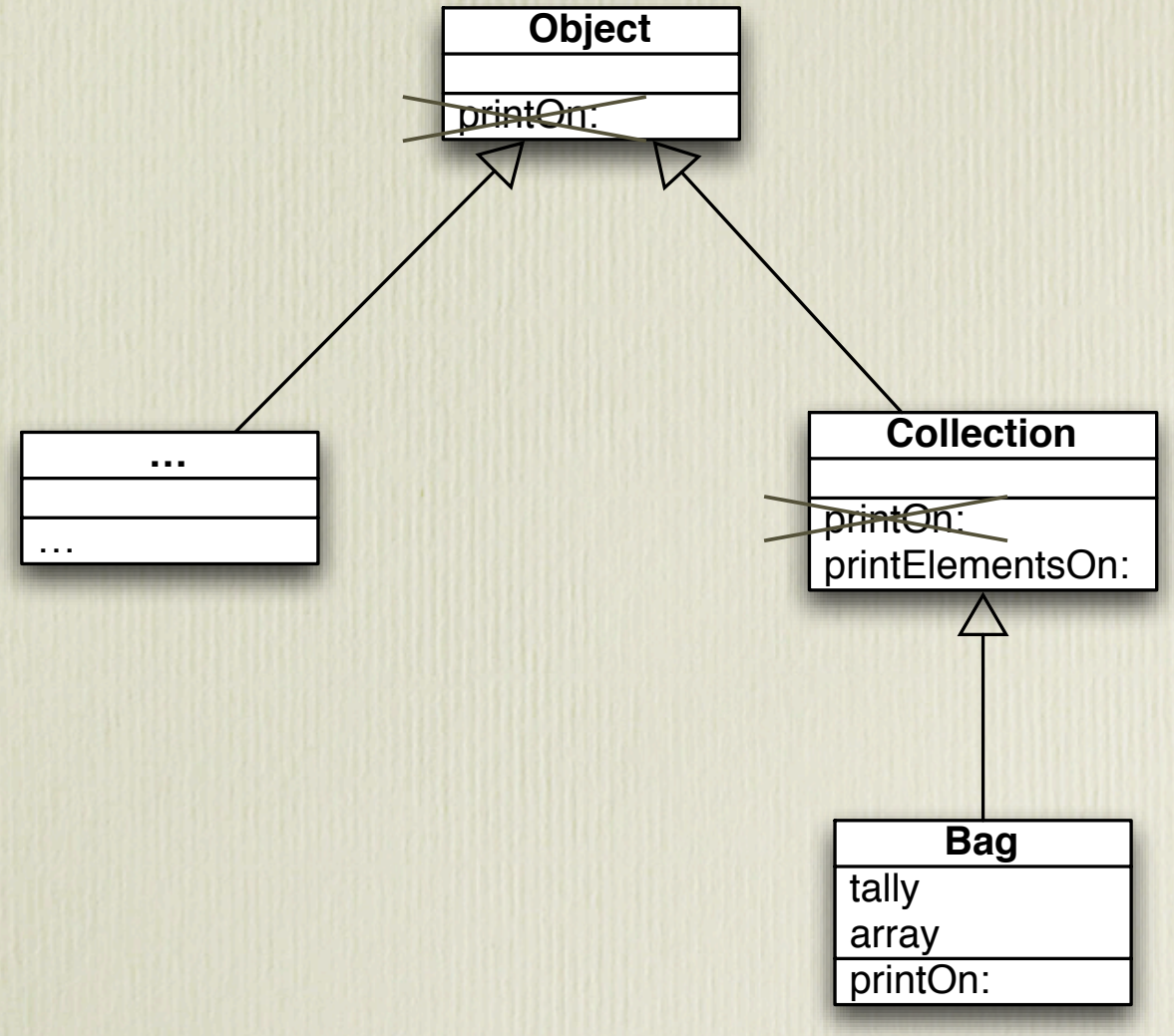
- The Browser is *Feasible*
  - with careful design and implementation, it *is* feasible to provide real-time feedback even for global properties such as required methods
- The Browser is *Useful*
  - Simplifies Intentional Programming
  - Makes it easier to understanding existing classes
  - Clarifies the relationship between sub- and superclasses
  - Exposes many bugs in existing code

Questions!

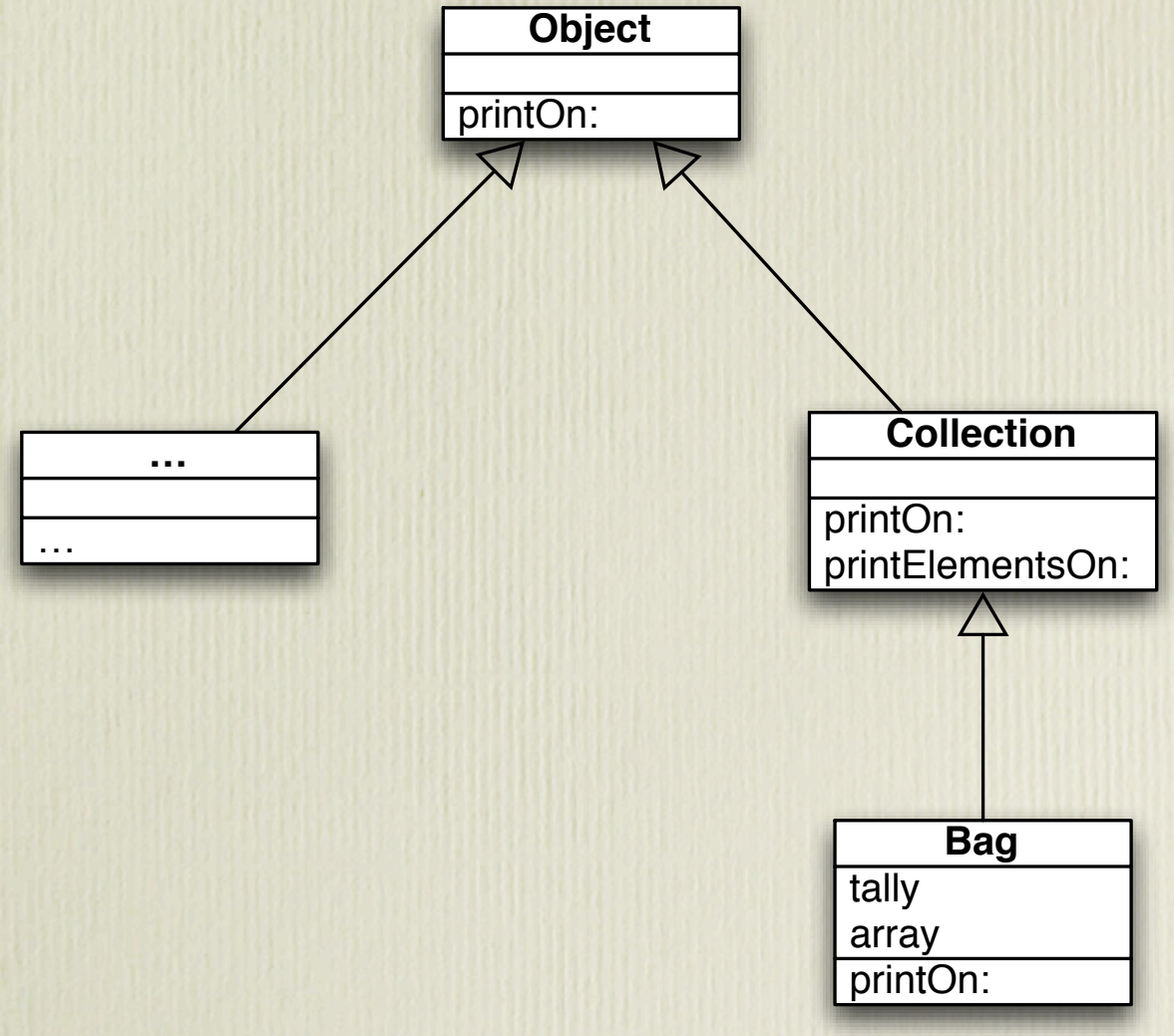
Which methods are reachable?



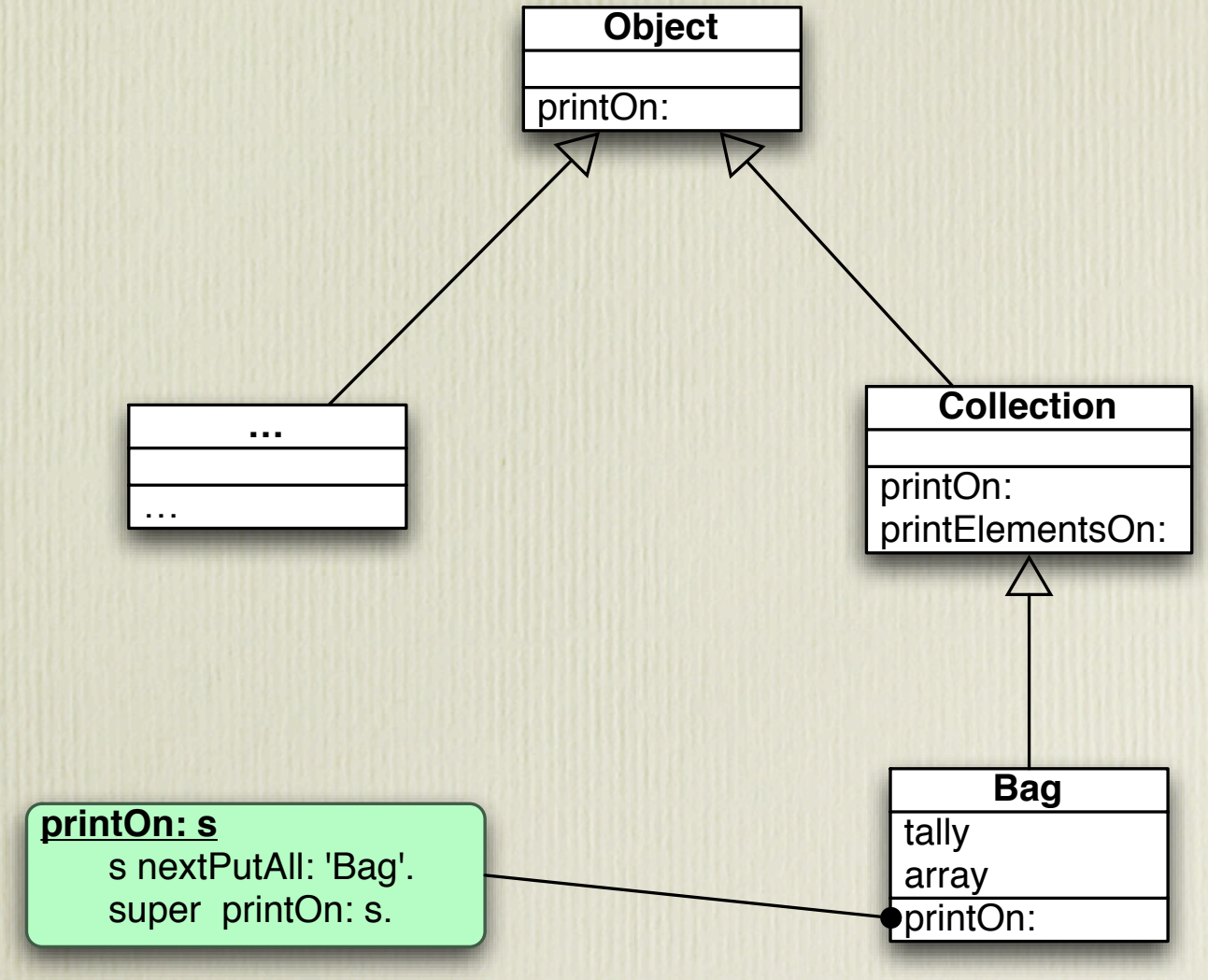
Which methods are reachable?



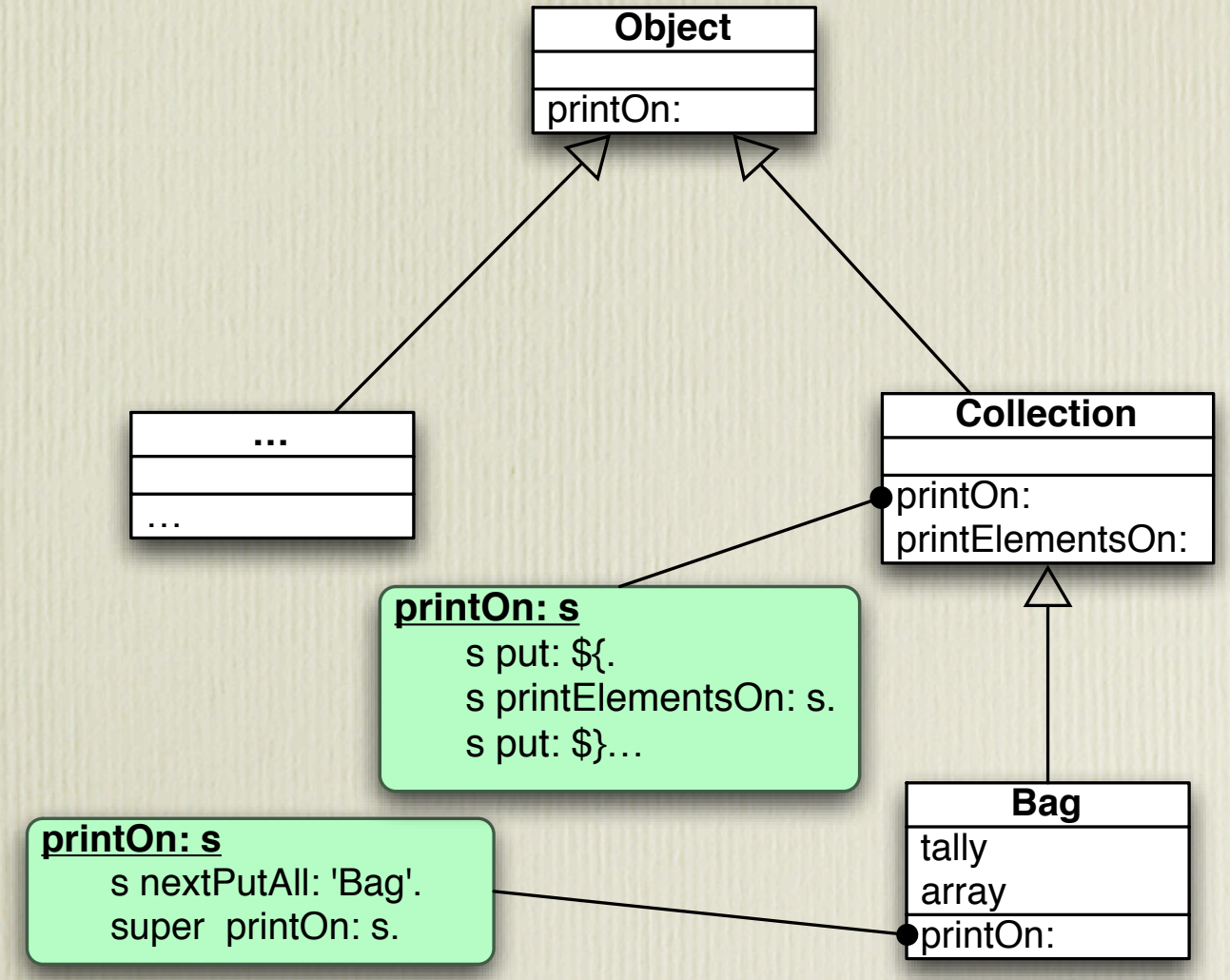
Which methods are reachable?



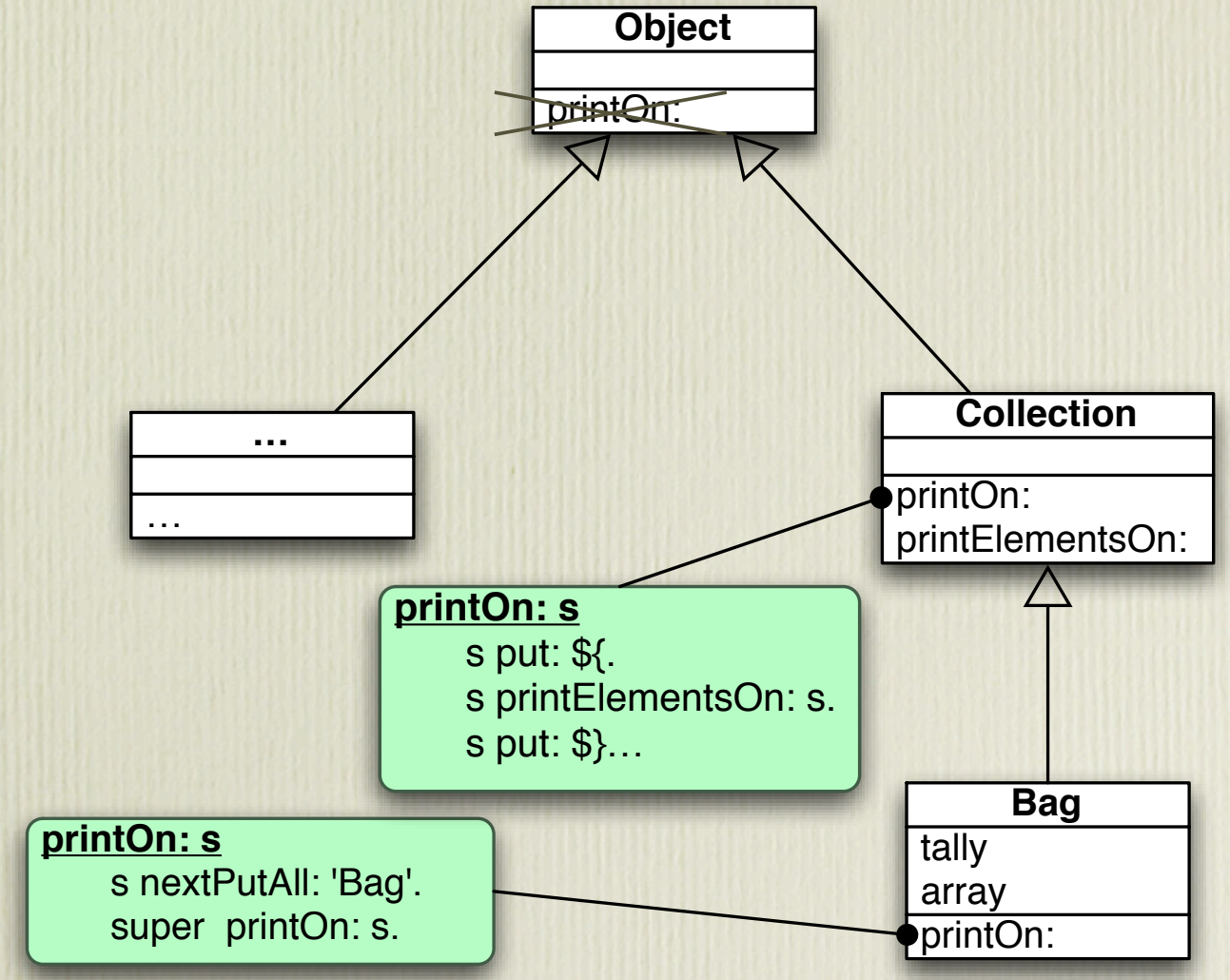
Which methods are reachable?



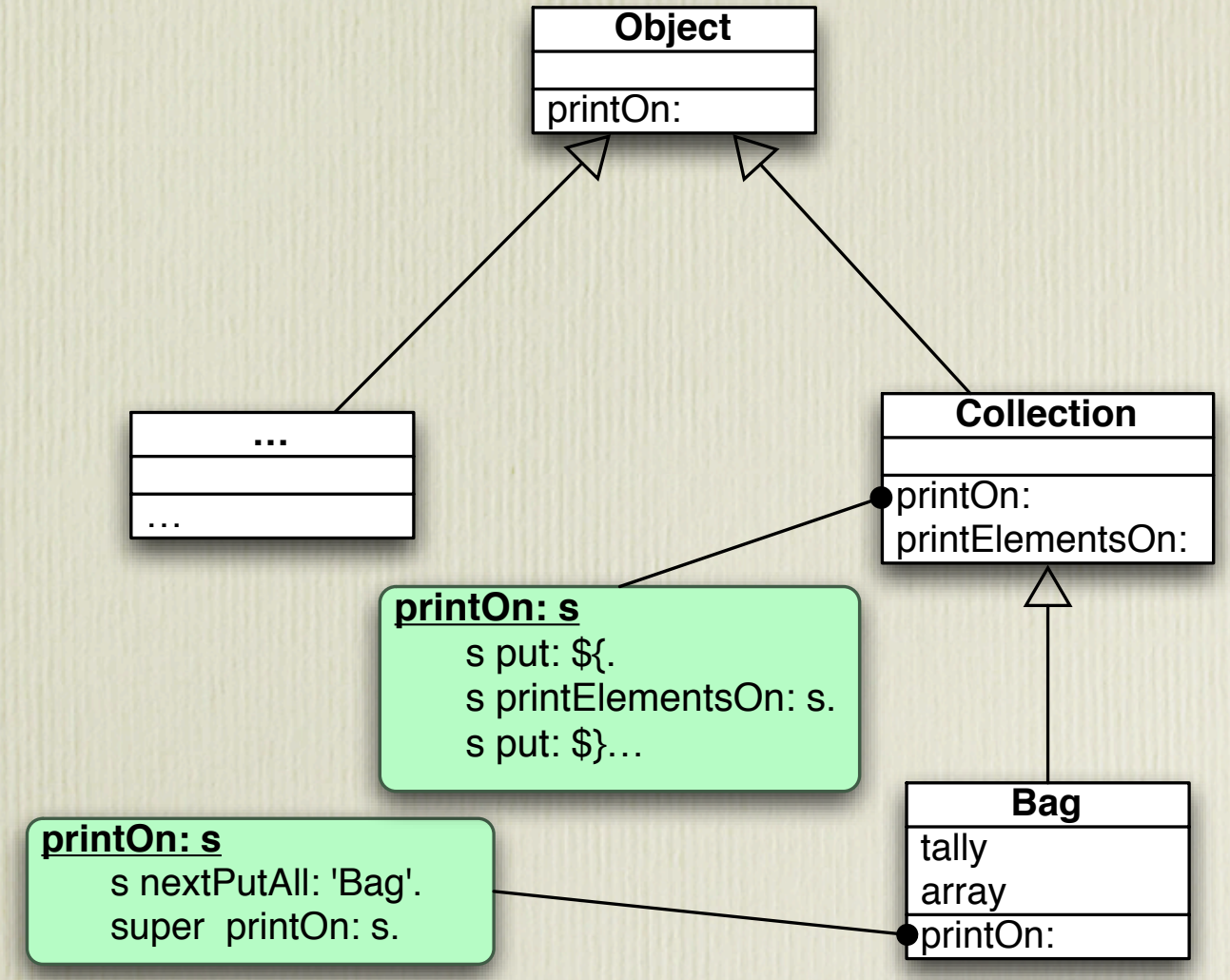
Which methods are reachable?



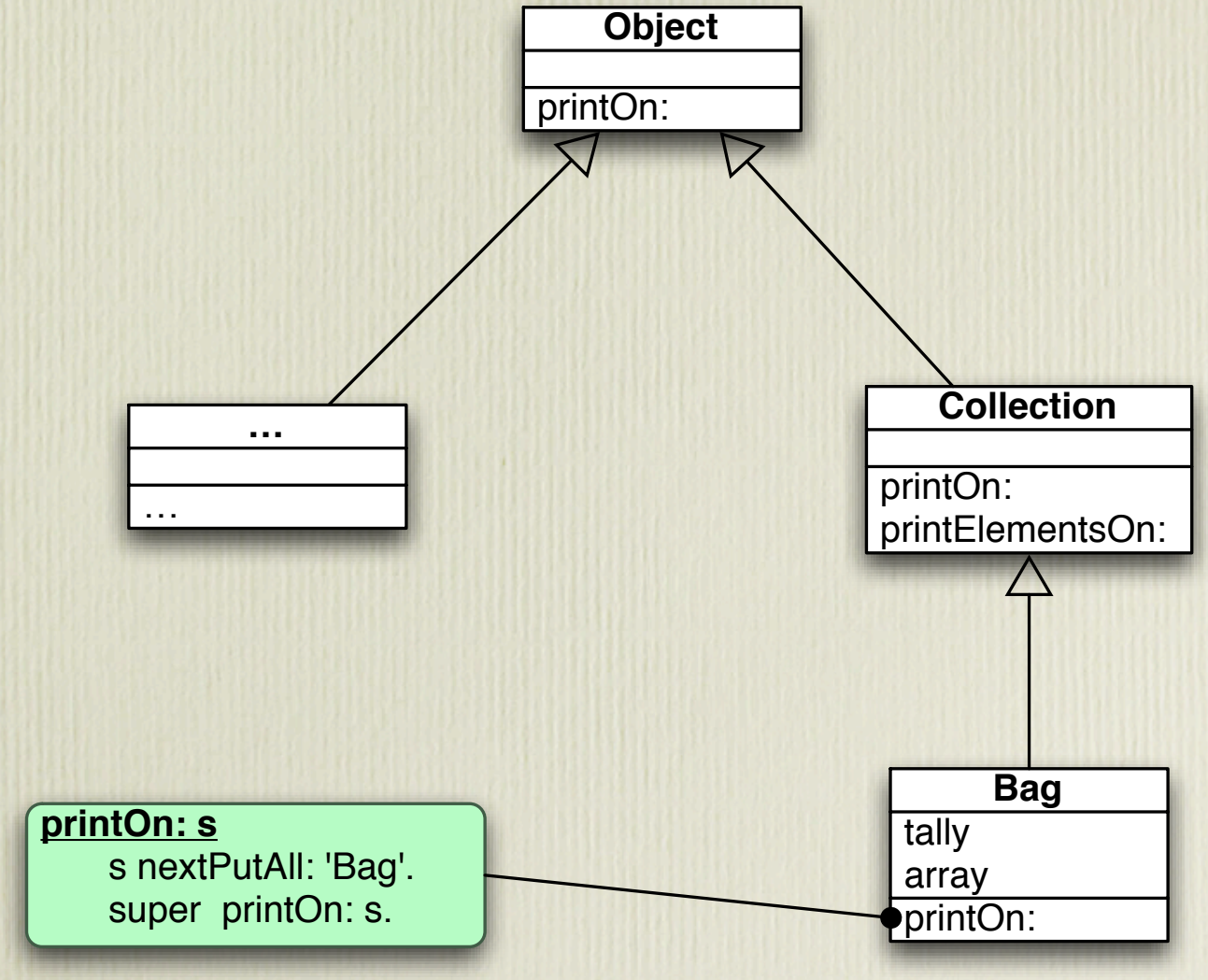
Which methods are reachable?



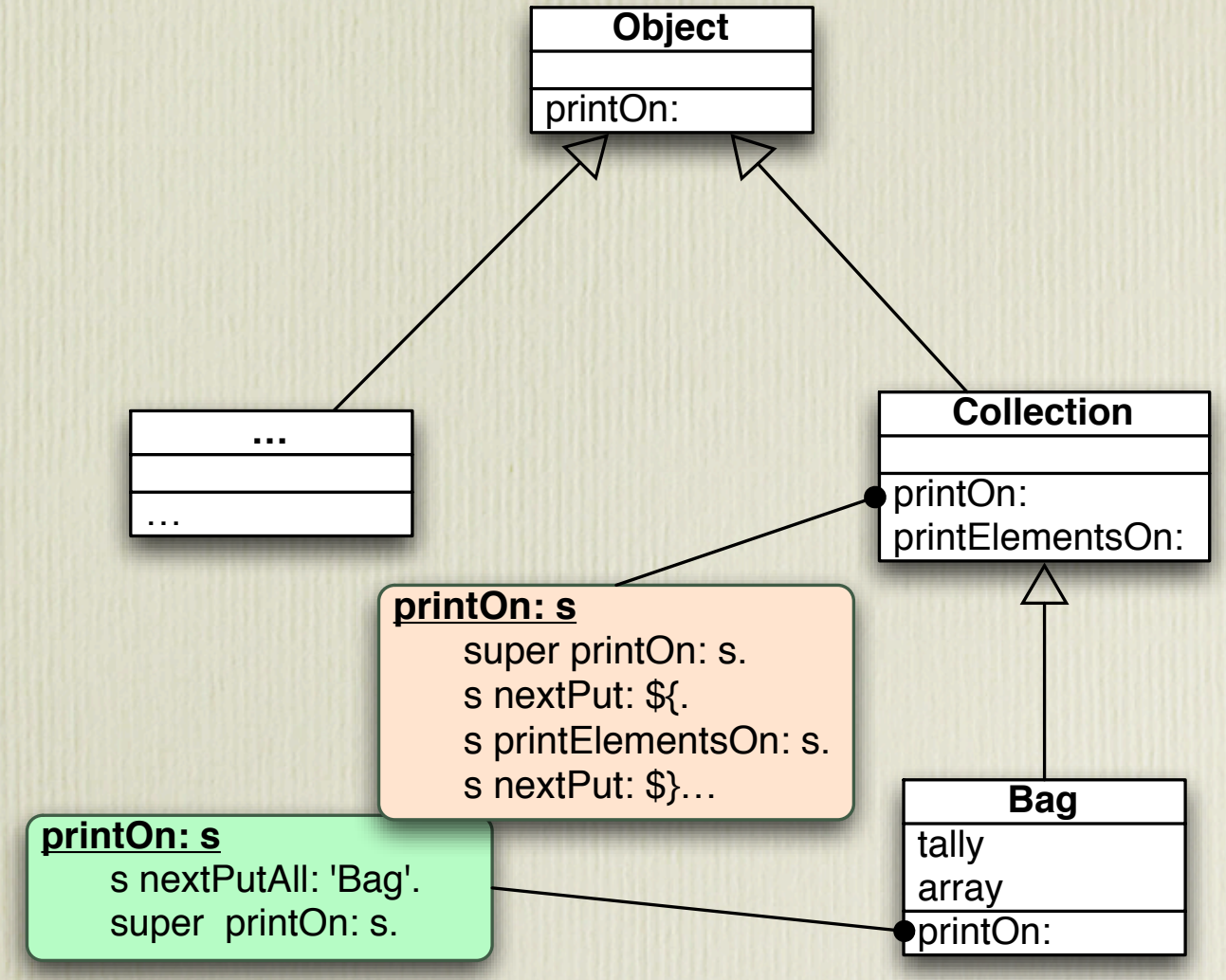
Which methods are reachable?



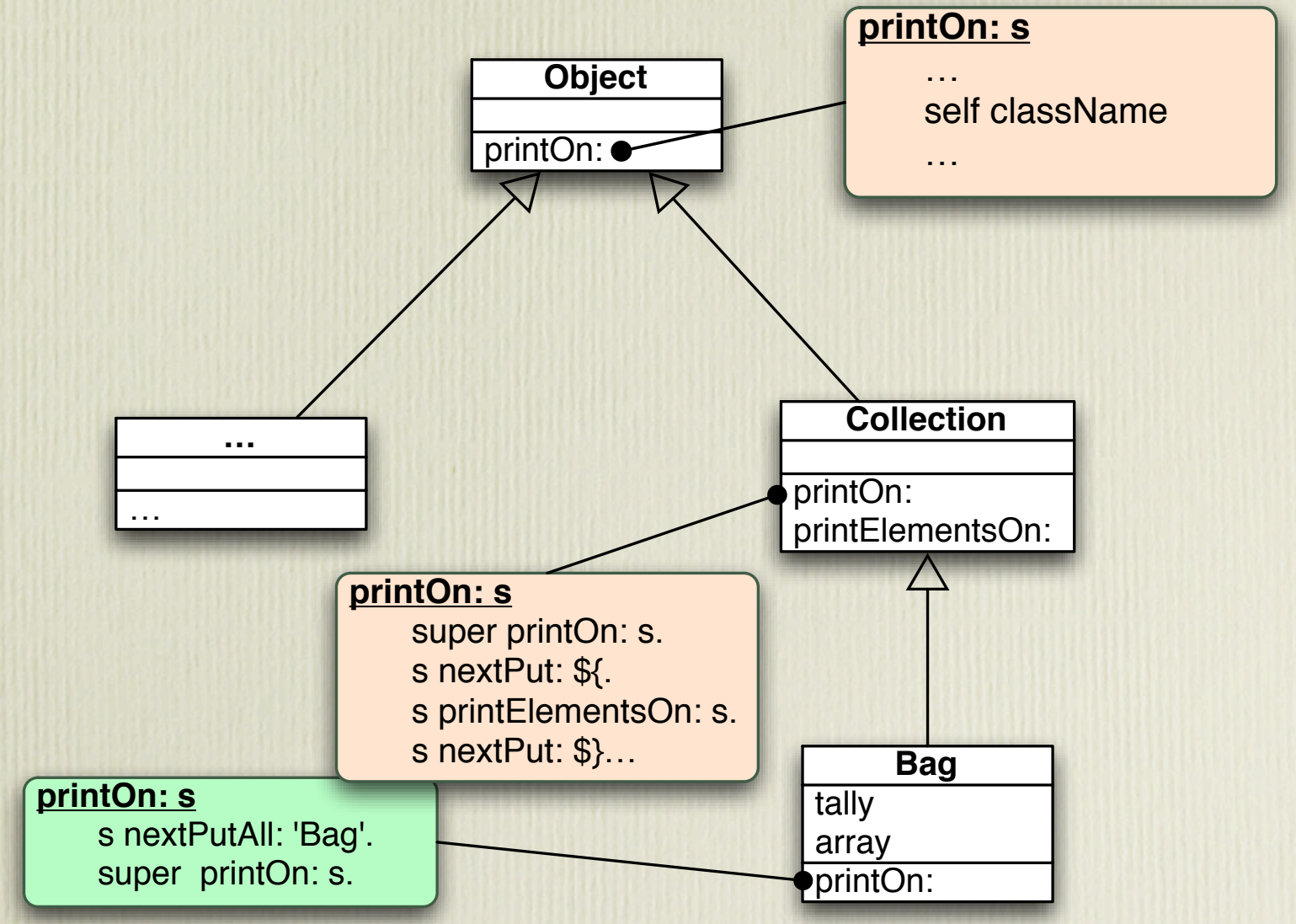
Which methods are reachable?



Which methods are reachable?



Which methods are reachable?



Which messages are self-sent?

```
fastenVerySecurely
    | temp |
    self hook.
    temp := self
    temp button.
    self class new clipTo: self
```

Which messages are self-sent?

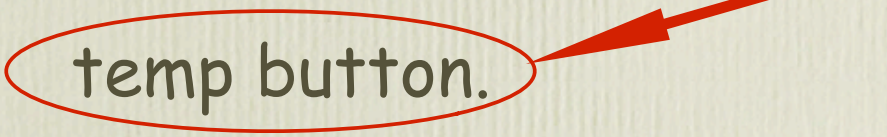
```
fastenVerySecurely
    | temp |
    self hook.
    temp := self
    temp button.
    self class new clipTo: self
```

Which messages are self-sent?

```
fastenVerySecurely
    | temp |
    self hook.
    temp := self
    temp button.
    self class new clipTo: self
```

Which messages are self-sent?

```
fastenVerySecurely
    | temp |
    self hook.
    temp := self
    temp button.
    self class new clipTo: self
```



This is a self-send, but we *don't* recognize it

Which messages are self-sent?

```
fastenVerySecurely
    | temp |
    self hook.
    temp := self
    temp button.
    self class new clipTo: self
```

This is a self-send, but we *don't* recognize it