A Layer-based Approach to Hierarchical Dynamically-scoped Open Classes

Matthias Springer[†], ○Hidehiko Masuhara[†], Robert Hirschfeld[§]

[†] Department of Mathematical and Computing Sciences / School of Computing, Tokyo Institute of Technology

[§] Hasso Plattner Institute, University of Potsdam

August 10, 2016

Introduction

- ”Open Classes” in Ruby
  - Modify existing classes or modules
  - Add or overwrite methods
- Why Open Classes?
  - Object-oriented auxiliary methods
    e.g.: **`5.minutes + 9.hours`**
  - Multi-dimensional separation of concerns [Tarr99]
  - Bug fixing (*monkey patching*)
- Support in programming languages
  - Ruby: open classes
  - Smalltalk: extension methods
  - Python: modifiable method dictionary

Introduction

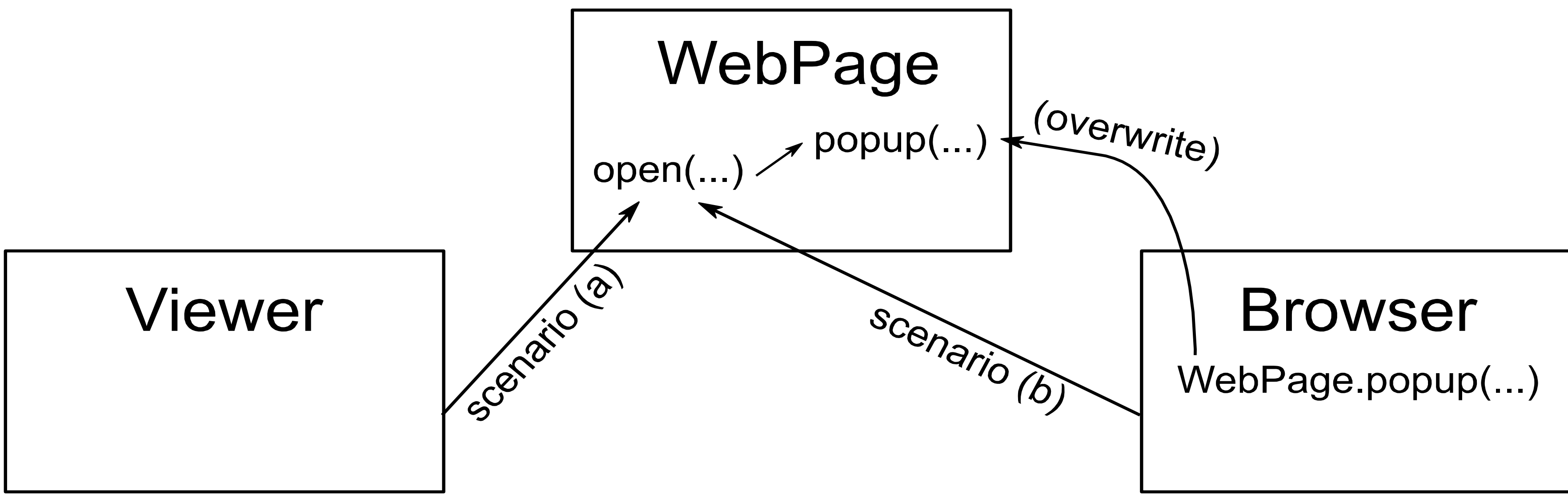
Open Classes in Ruby by example

```
# in standard library
class Fixnum
    ...
end
```

```
# in a different component
class Fixnum
      def minutes
            return self * 60
      end

      def hours
            return self * 3600
      end
end
```

Example 1: WebPage Library [Takeshita13]

- A library: WebPage
  renders HTML and might show popups
- Two applications: using WebPage
  - Browser: should ***not*** show popups
  - Viewer: should show popup



The Problem: Global Visibility

```
class WebPage
      def open(url)
            # ...
            if popup_requested
                  popup(...)
            end
      end

      def popup(text)
            # show popup window
      end
end
```

WebPage library

```
class Browser; end
```

overwrite

Browser app

```
class Viewer; end
```

(should show popups)

Viewer app

OK NG!

The Problem: Global Visibility

```
class WebPage
    def op
    end

    def popup(text)
        # show popup window
    end
end
```

Modifications are visible everywhere

↓

Other components can break

```
class Browser; end
```

overwrite

Browser app

```
class Viewer; end
```

(should show popups)

Viewer app

NG!

Example 1: With Open Classes

```
class WebPage
      def popup(text); end
end

class Viewer
      def check(file)
            # ...
            if file.is_confidential?
                  page.popup("<b>confidential</b>")
            end
      end
end
```

overwrite with No-Op

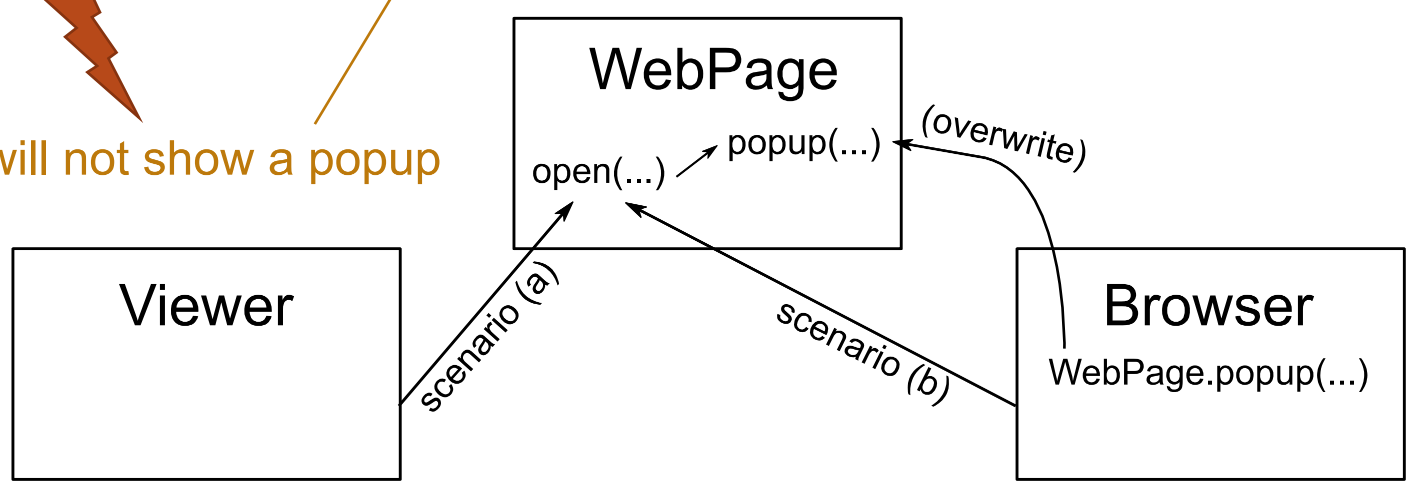
Open Classes are globally visible in Ruby
→`Viewer` broken

Example 1: With Open Classes

```
require "webpage"
require "viewer"
require "browser"

class Application
      def main
            Browser.new.open("http://www.titech.ac.jp")
            Viewer.new.check("secret.html")
      end
end
```

overwrites `WebPage.popup`

will not show a popup



The Problem with Open Classes

- Global Visibility
  - Modifications are visible everywhere
  - Other components (e.g., `Viewer`) can break
    → "*Destructive Modifications*"
- Solution: *Locality of Changes*

Our Solution: ***Extension Classes***

Idea: scope control of modifications

- *Using only classes* (vs. classboxes, method shells etc.)
- Reusability through Ruby *modules* (or mixins) (vs. new syntax for refinements)
- Consistent with Ruby's language features: take into account *class nesting hierarchy*
- Amenable to other programming languages with
  - object-based, class-based
  - unit of reuse (e.g., mixins/modules, traits, …)
  - (class nesting hierarchy)

Extension Classes

- Modifications are defined as "inner classes"
- only visible from "enclosing classes" (details follow)

```
class Browser
      def open(url)
            WebPage.new.open(url)
      end

      partial

      class ::WebPage
            def popup; end
      end
end
```

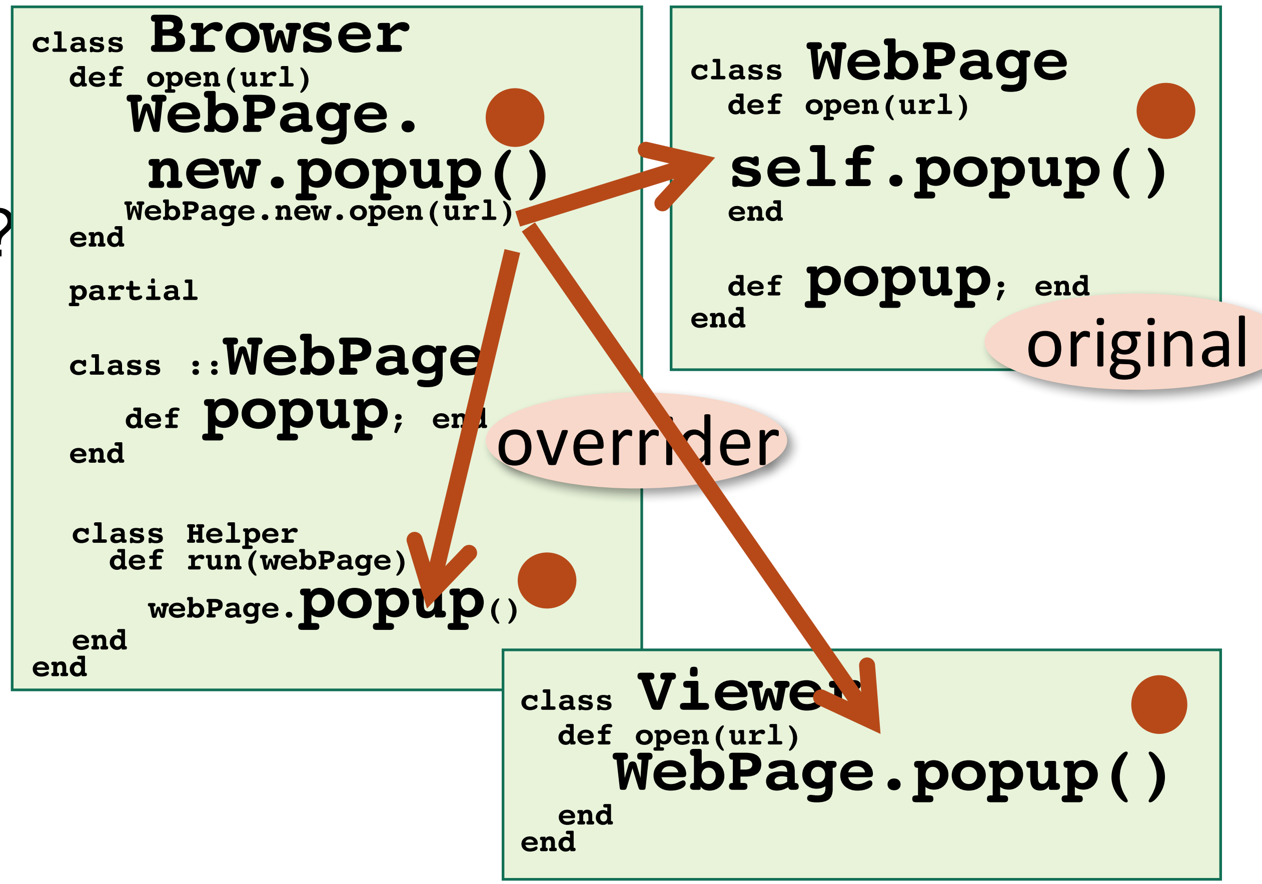
Browser app

modifies (global) WebPage but only visible from Browser

Subtleties of visibility

should modification visible when it is called:

- directly?
- from a different class?
- via a different class?
- via another method in the modified class?
- via a sibling inner class?
- via a superclass?
- via a subclass?



Our principle of visibility

Modifications are visible

- in the context of an enclosing class, and
- as long as the context remains within enclosing/ sibling classes

(cf. COP)

```
class Browser
  def open(url)
    WebPage.
     new.popup()
    WebPage.new.open(url)
  end

  partial

  class ::WebPage
    def popup; end
  end


  class Helper
    def run(webPage)
      webPage.popup()
  end
end
```

overrider

```
class WebPage
  def open(url)
    self.popup()
  end

  def popup; end
end
```

original

```
class Viewer
  def open(url)
    WebPage.popup()
  end
end
```

A workaround to affect external classes: empty modifications

Activation Rule

- Set of active classes S = { }
- When calling a method **`C.foo`**: Add C to S

```
browser.open(...)        # S += Browser
viewer.check(...)        # S += Viewer
```

Deactivation Rule

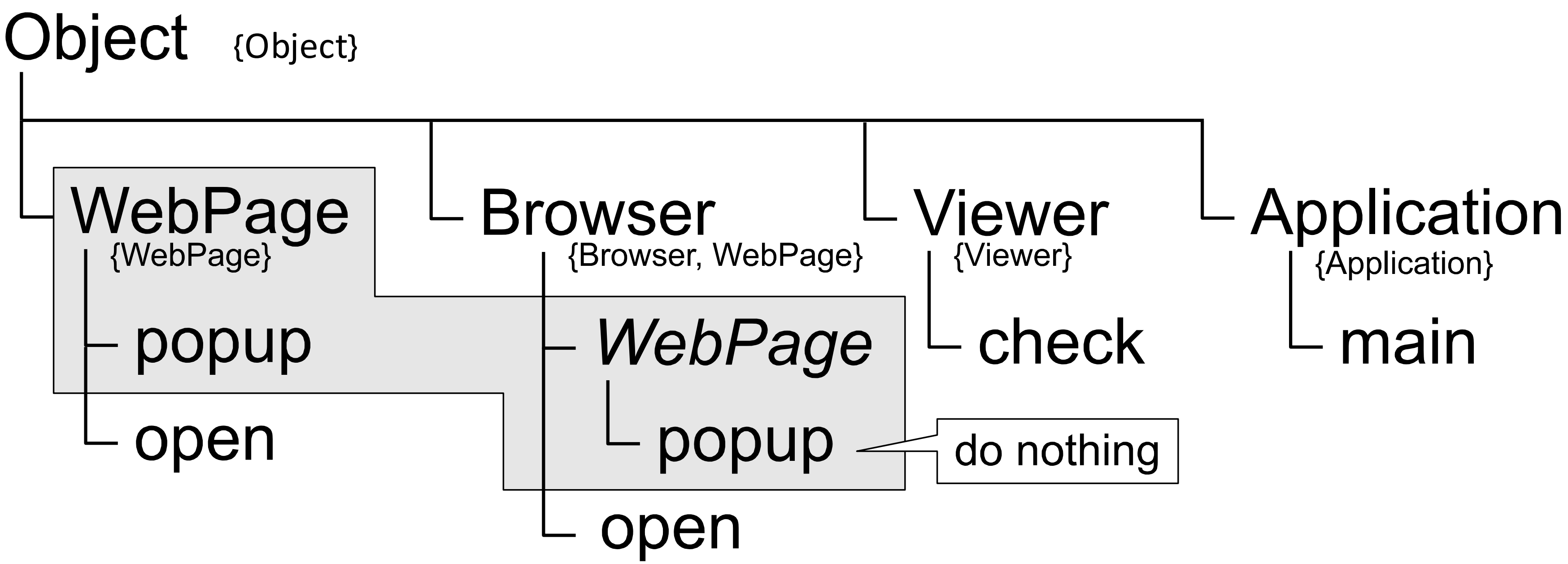
- Restore original S when returning from a method call
- When calling `C.foo`, deactivate all classes a $\in$ S, where C $\notin$ *scope(a)*
  - Intuitively: scope(C) is a set of classes that are compatible with the modifications defined by C
  - Mathematically: scope(C) = { C } $\cup$ *all target classes*
  - Definition will be extended later

```
class Browser
      partial

      class ::WebPage; end
end
```

scope(Browser) =
{ Browser, WebPage }

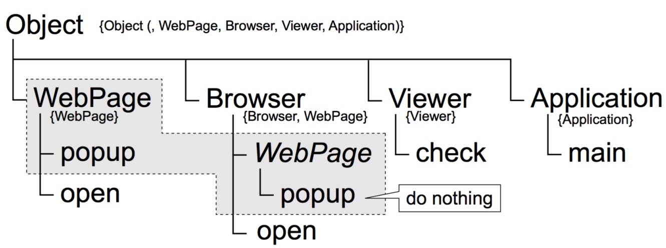
- `Browser` is compatible with `Browser`'s modifications
- `WebPage` is compatible with `Browser`'s modifications

Example 1: Overview



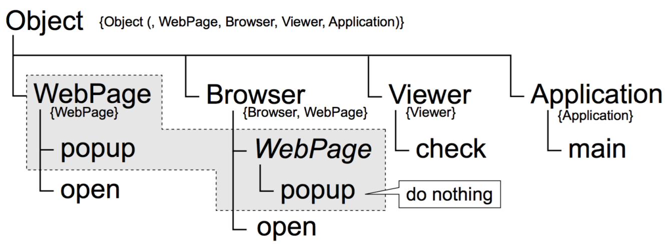

```
class Application
      def main
            Browser.new.open("http://www.titech.ac.jp")
            Viewer.new.check("secret.html")
      end
end
```

Example 1: Step by Step

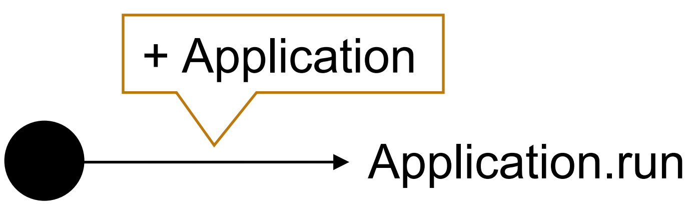


S = { }

Example 1: Step by Step

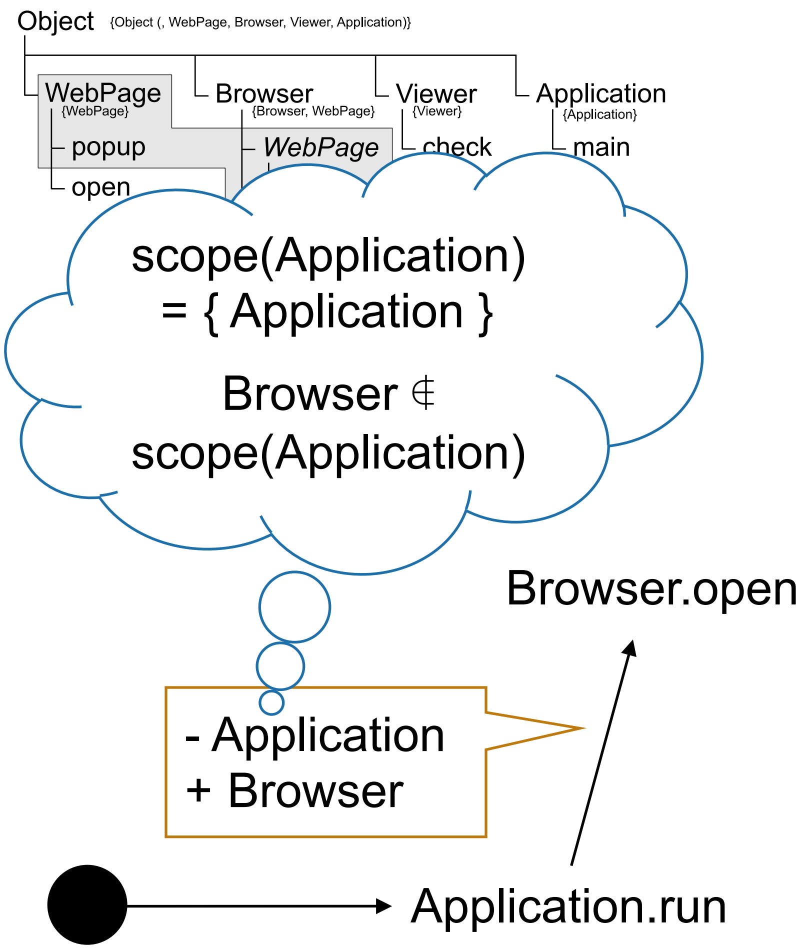


S = { Application }

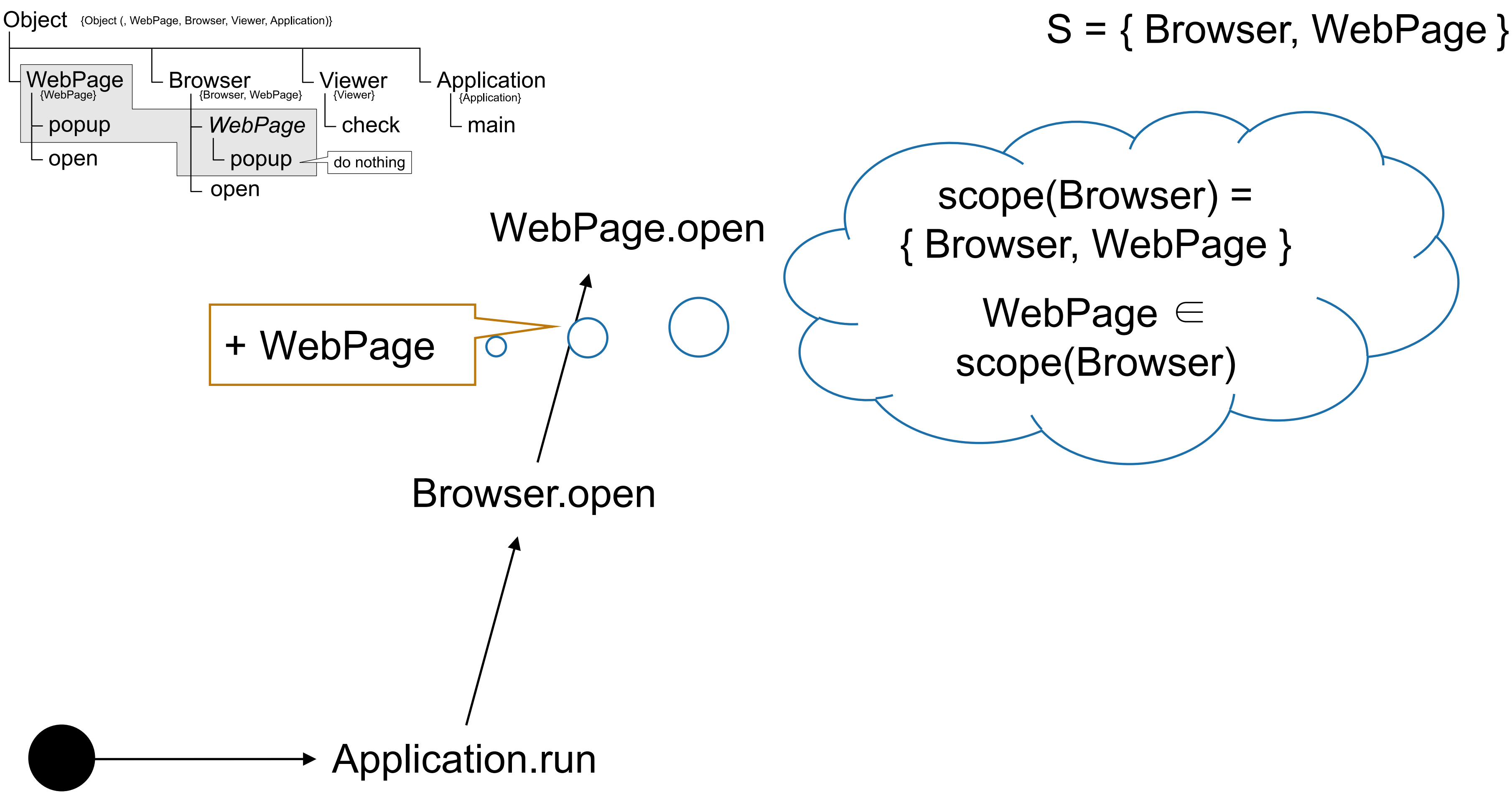


Example 1: Step by Step

S = { Browser }

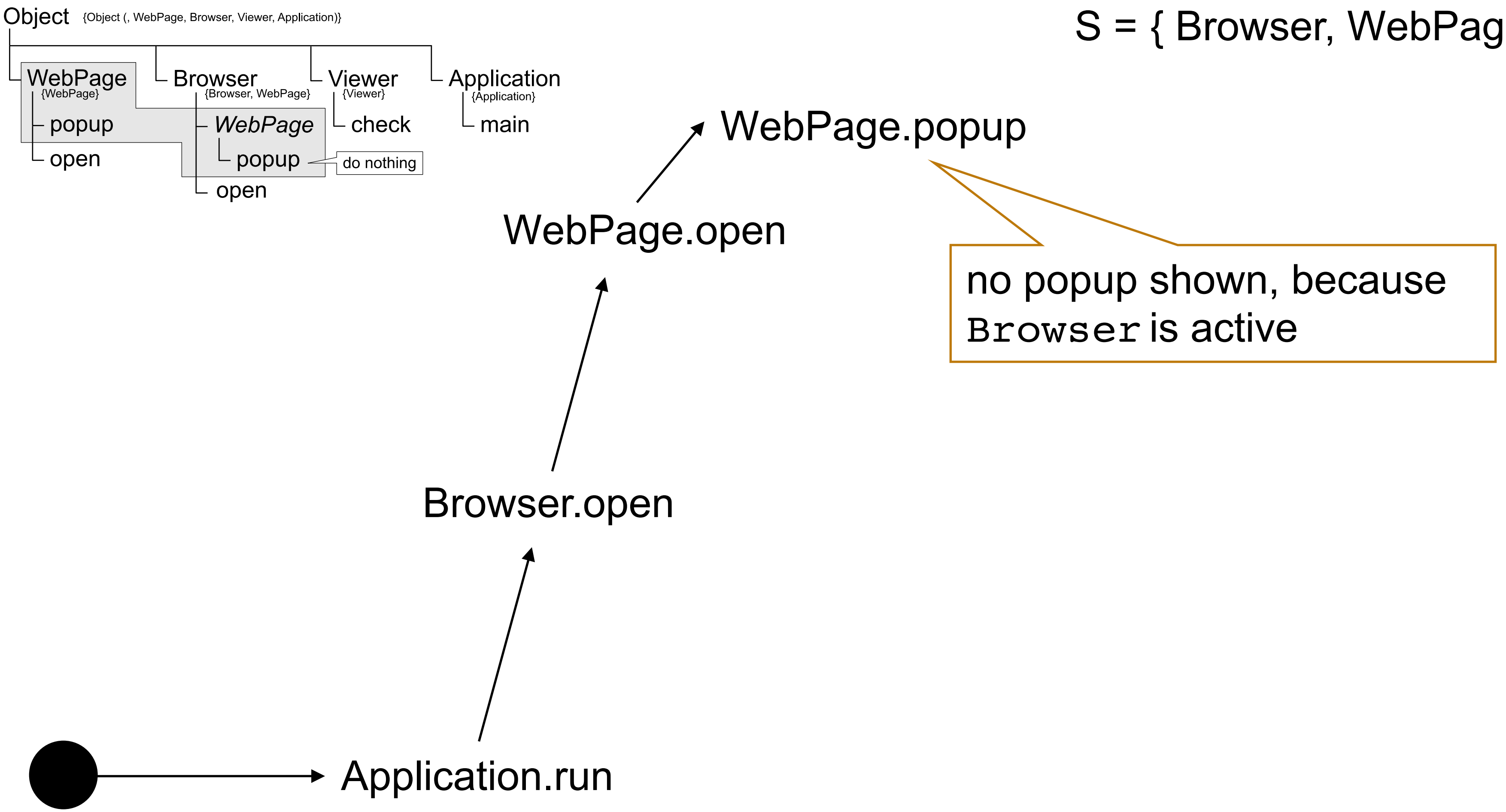


Example 1: Step by Step



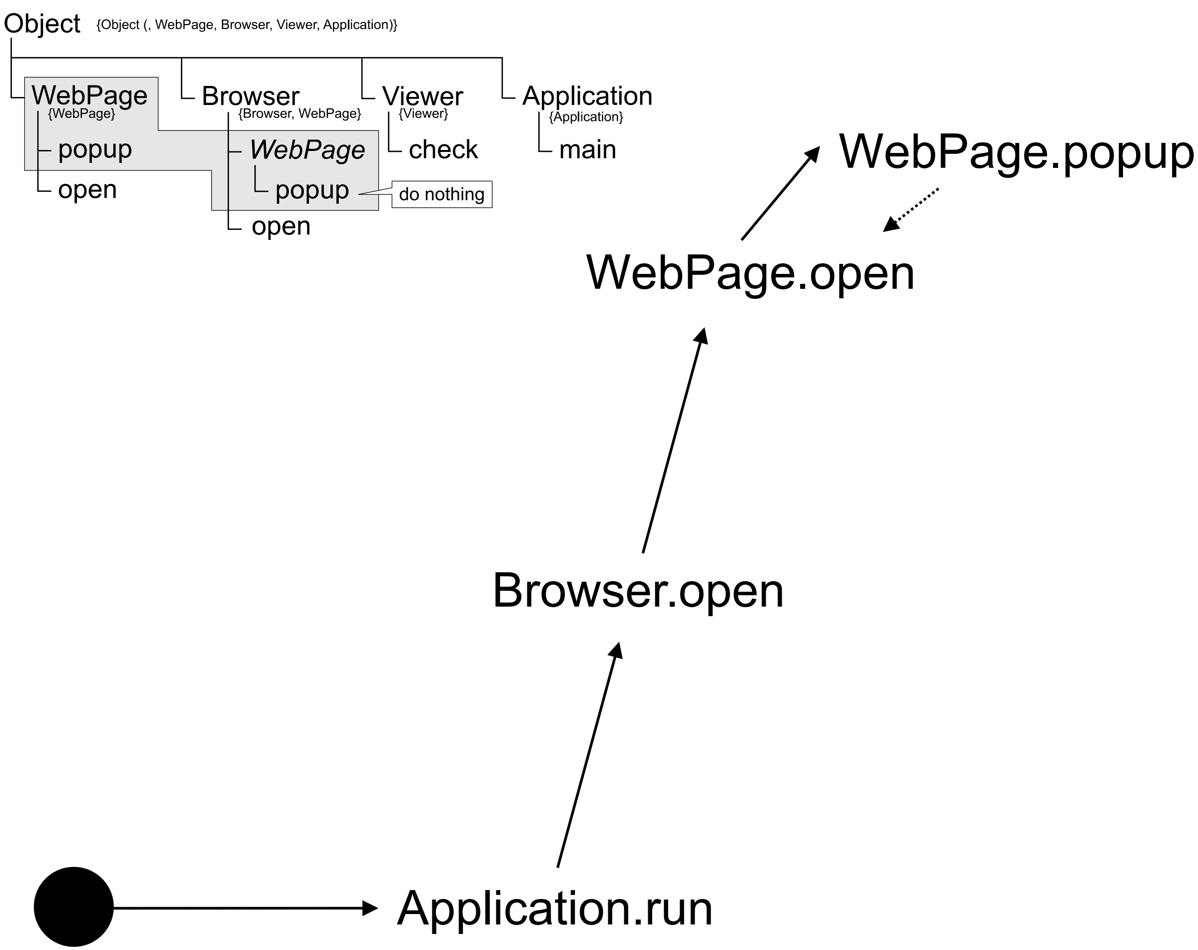
Example 1: Step by Step

S = { Browser, WebPage }



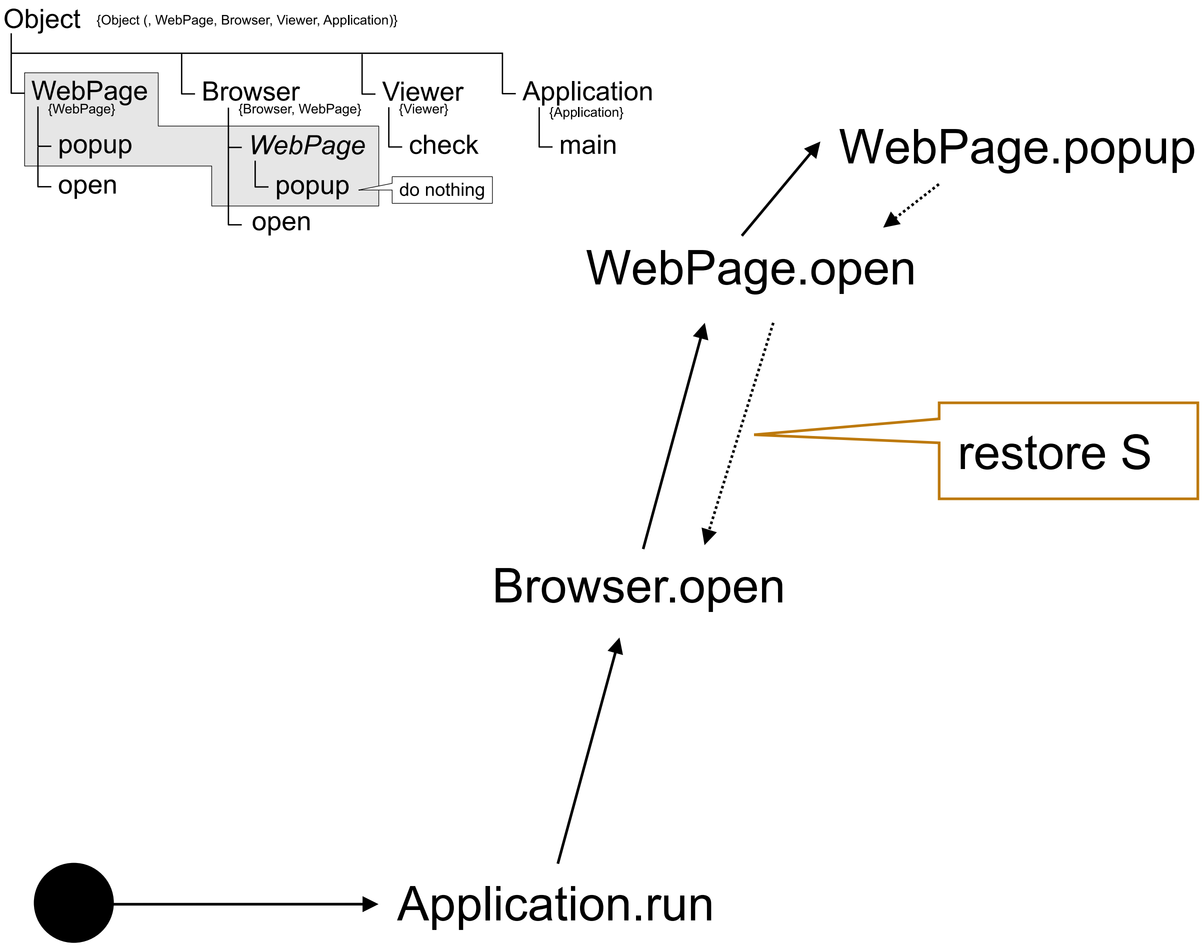
Example 1: Step by Step

S = { Browser, WebPage }



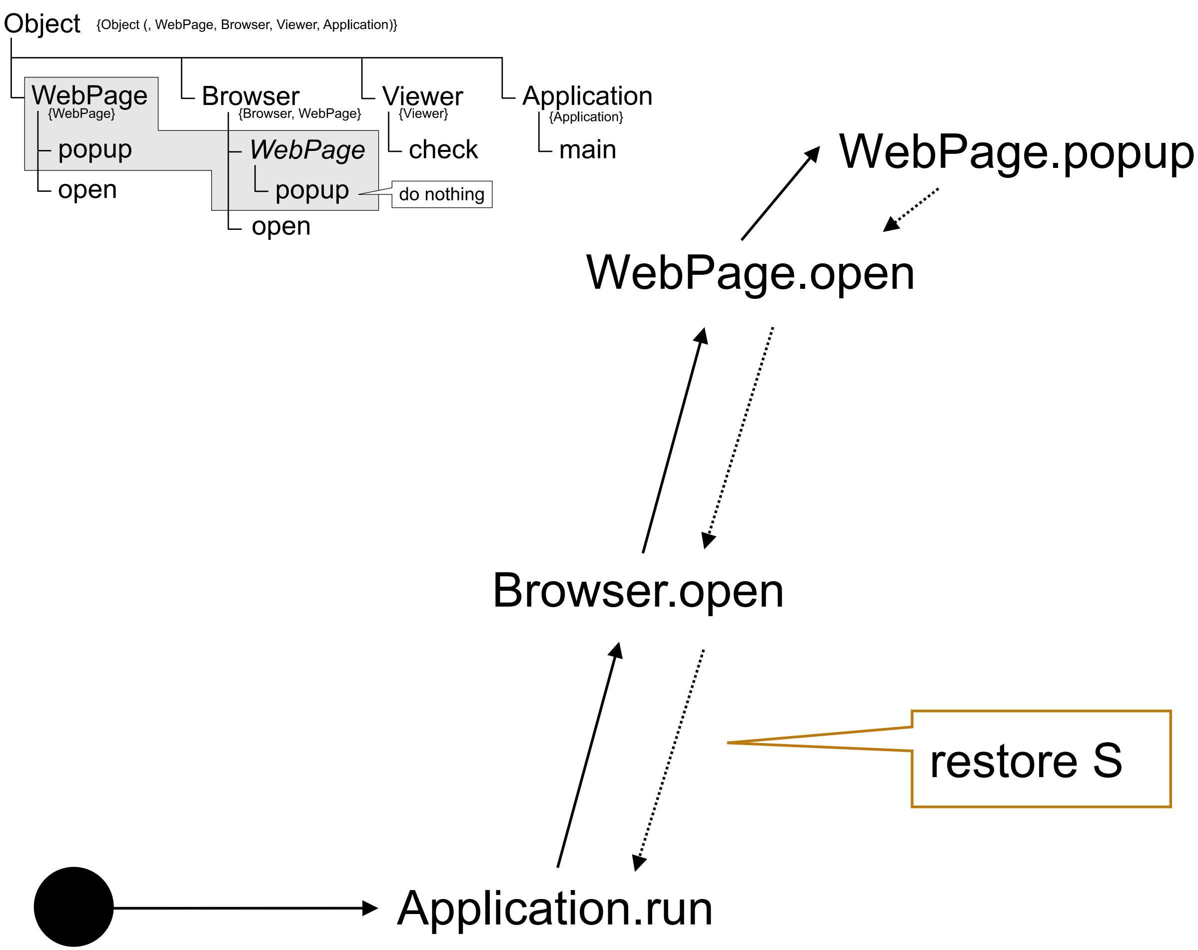
Example 1: Step by Step

S = { Browser }



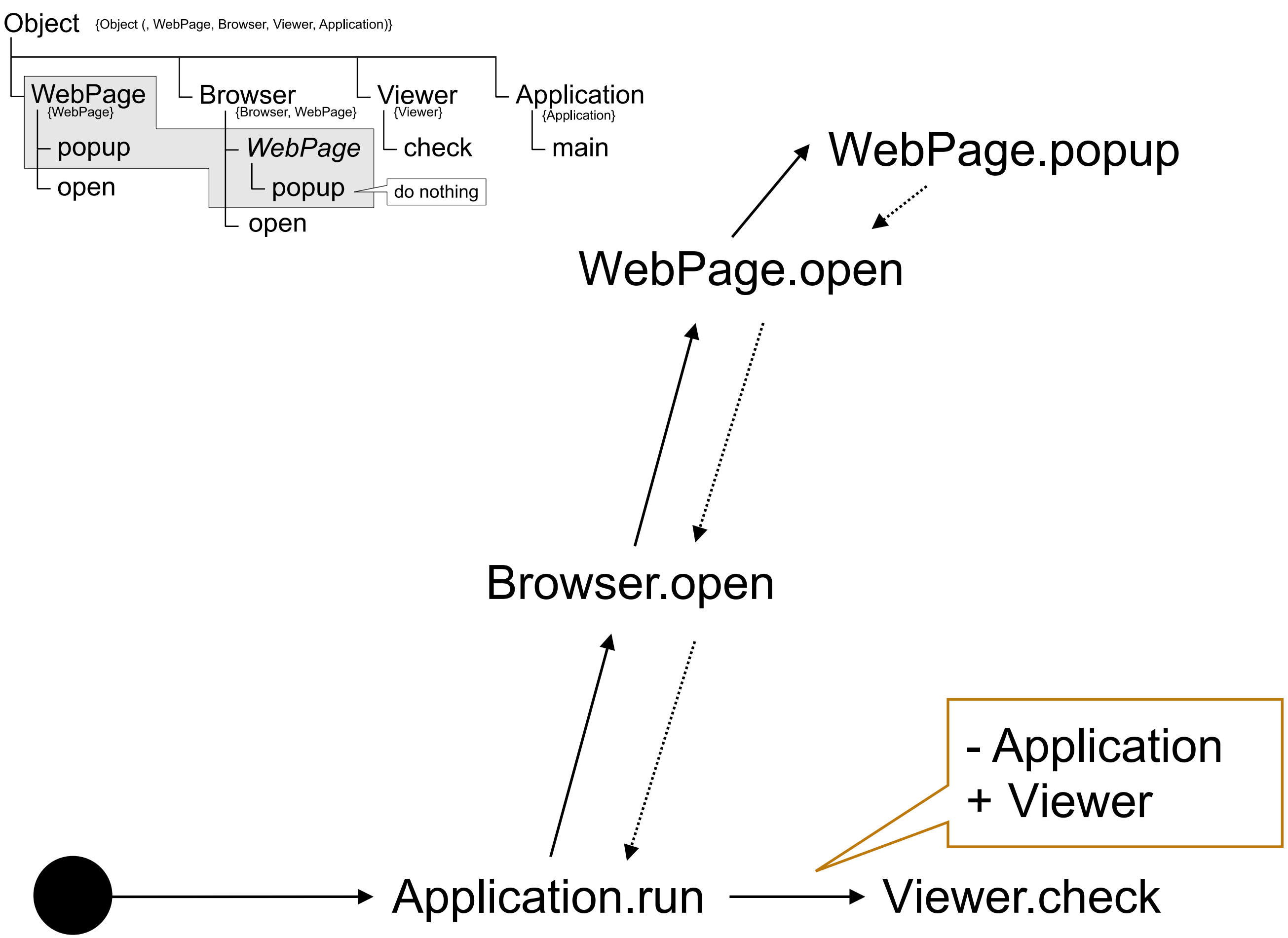
Example 1: Step by Step

S = { Application }



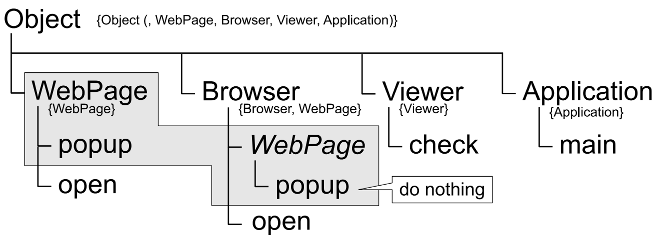
Example 1: Step by Step

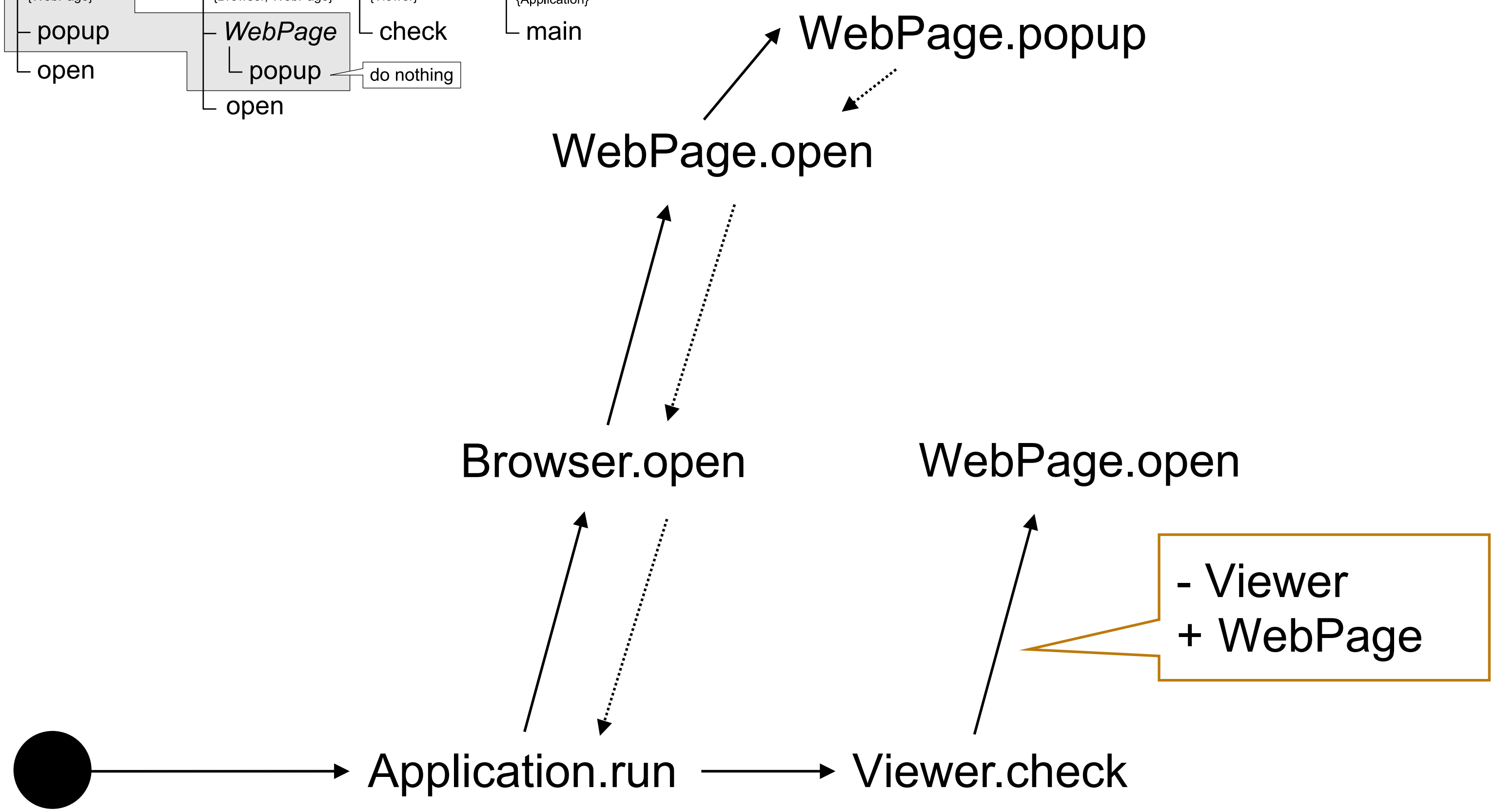
S = { Viewer }



Example 1: Step by Step

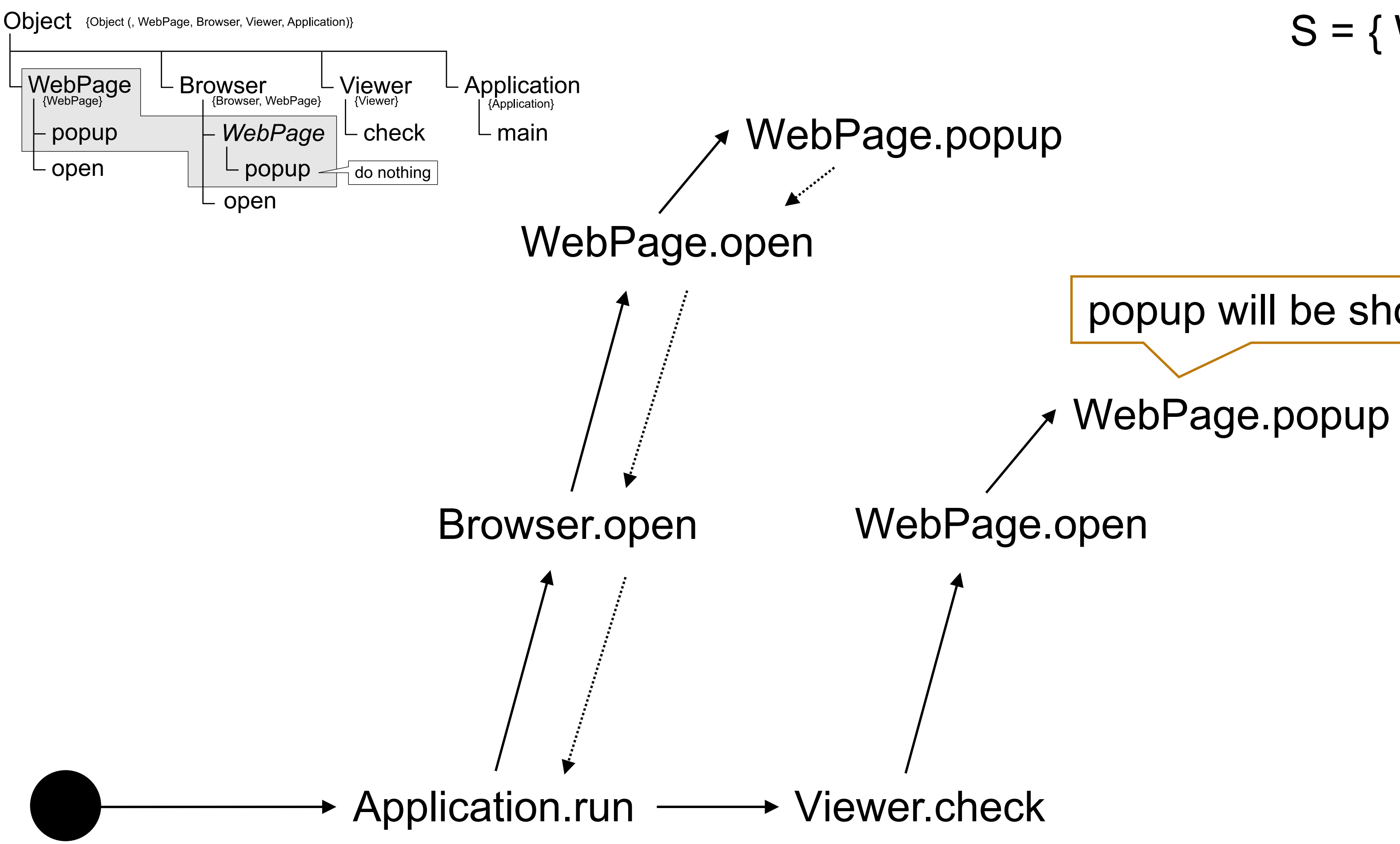
S = { WebPage }





Example 1: Step by Step

S = { WebPage }



Example 1: Variations

```
class Application
      def main
            # ...
      end

      partial

      class ::WebPage
            def popup
                  # ...
            end
      end

      class ::Browser; end
      class ::Viewer; end
end
```

scope(Application) =
{ Application, WebPage,
Browser, Viewer}

Application will remain
active in a call sequence
Application → Viewer →
Webpage

Reusability with Modules

- Classes and modules can define partial classes
- Modifications are active in including classes (as if they were defined there directly)

```
module NoPopup
  partial

  class ::WebPage
    def popup
      # ...
    end
  end
end
```

```
class Browser
  include NoPopup
  # ...
end

class Viewer
  include NoPopup
  #...
end
```

Class Activation Schemes

- How can we ensure that a class M is active when running code from class C?

Class-based Activation

- Control flow passes through class M
- For every class c that is visited on the way to C: c∈scope(M)
- M pushes modifications to C (cf. *local rebinding/ dynamic scoping*)

Mixin-based Activation

- M is a module/mixin
- C includes M
- C requests modifications from M

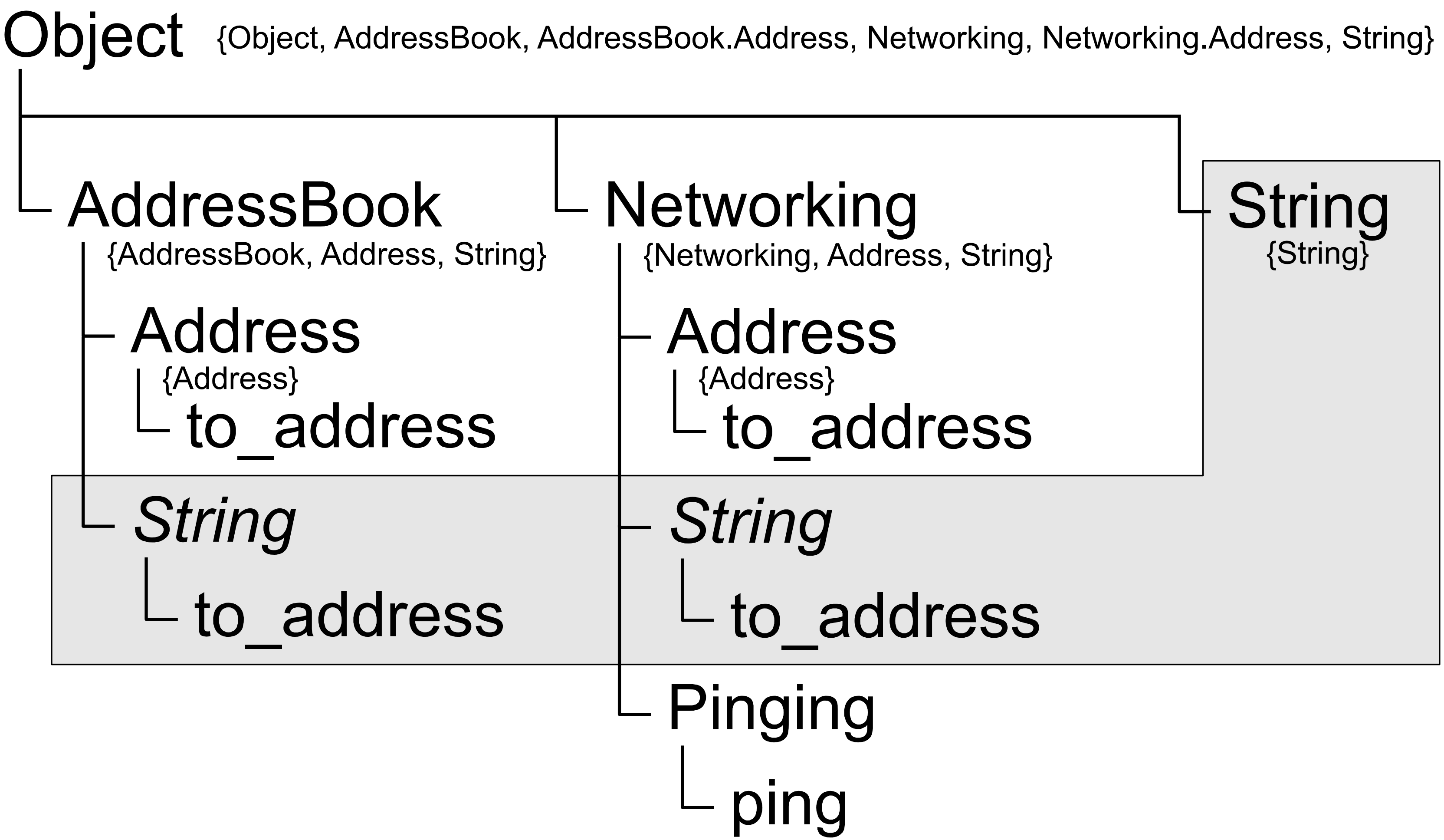
```
include NoPopup
```

```
class ::Viewer; end
class ::WebPage; ...;
end
```

Hierarchical Scoping

- How do we share modifications among an entire class nesting hierarchy?
- Modifications of class C should affect all classes that are nested inside C

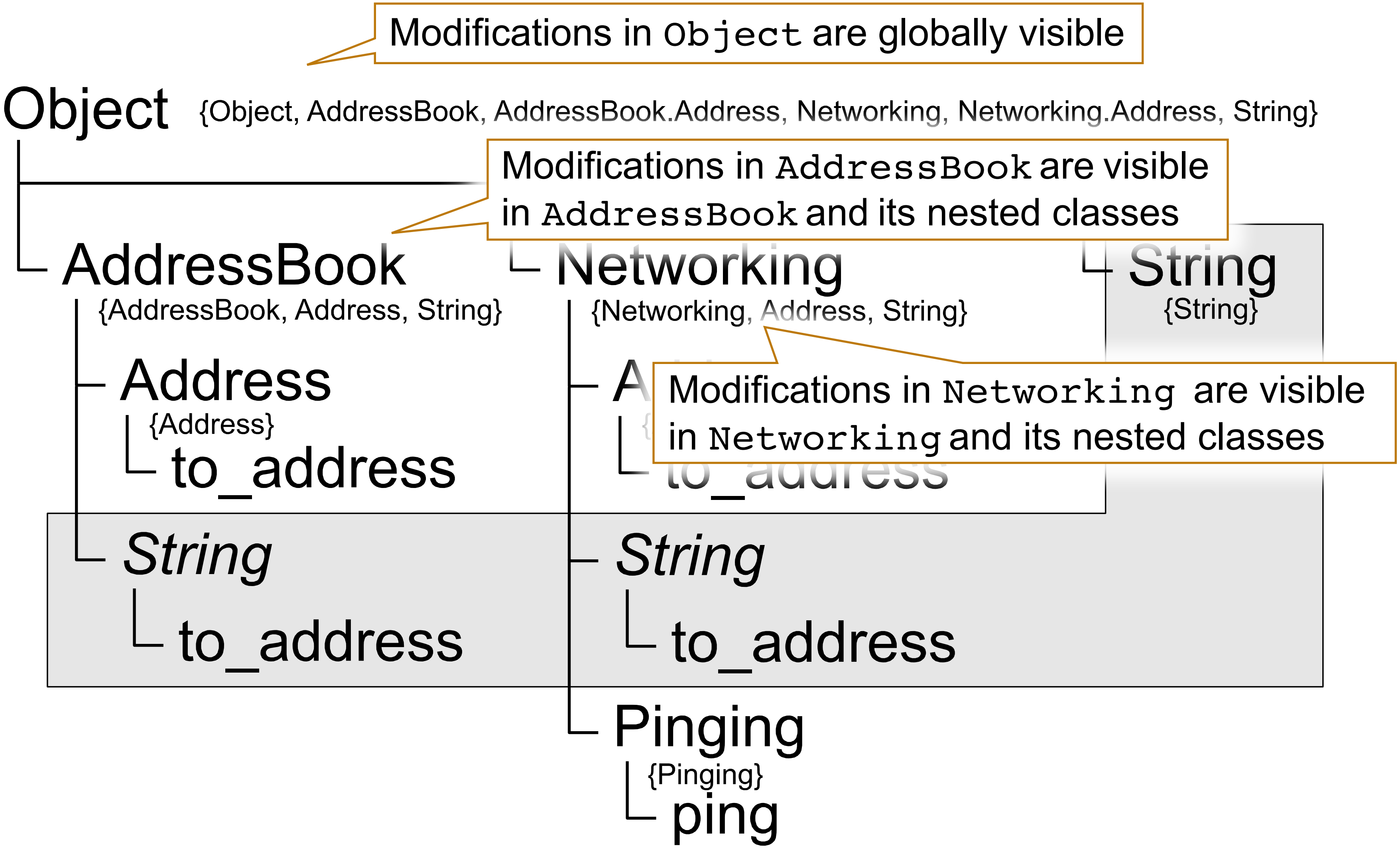
Example 2: Overview



Activation / Deactivation Rule

- Extend both rules
- *Activation:* When calling a method `C.foo`, activate C and all of its enclosing classes
- *Deactivation:* Extend scope(C) such that it also includes the scope of all nested classes of C

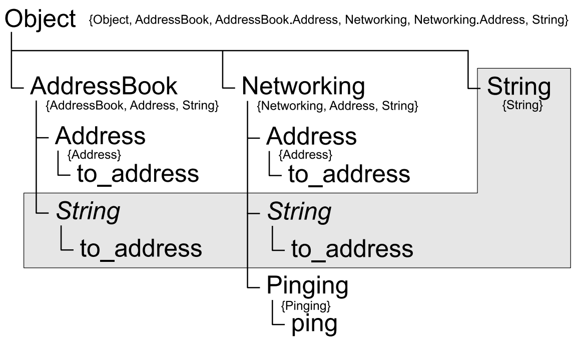
Example 2: Scope of Classes



Example 2: Invocation Code

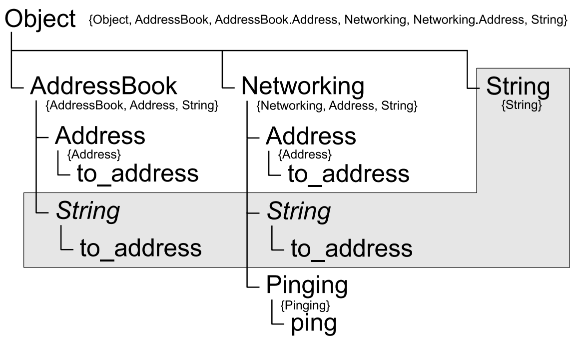
```
module Networking
	module Pinging
		def self.ping(addr)
			addr.to_address
		end

		# ...
	end

	# ...
end

class Application
	def run
		Networking::Pinging.ping(“127.0.0.1”)
	end
end
```

Example 2: Step by Step

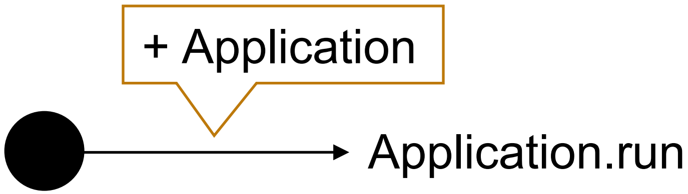


S = { Object }

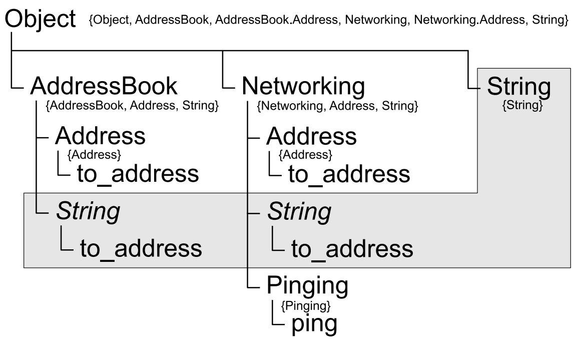
Example 2: Step by Step



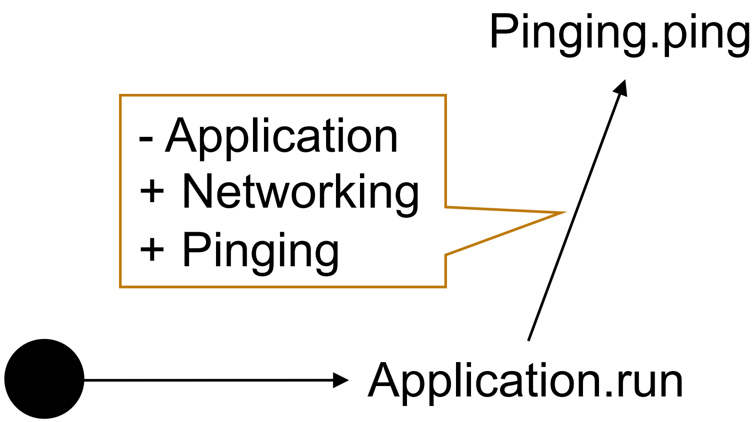
S = { Object, Application }



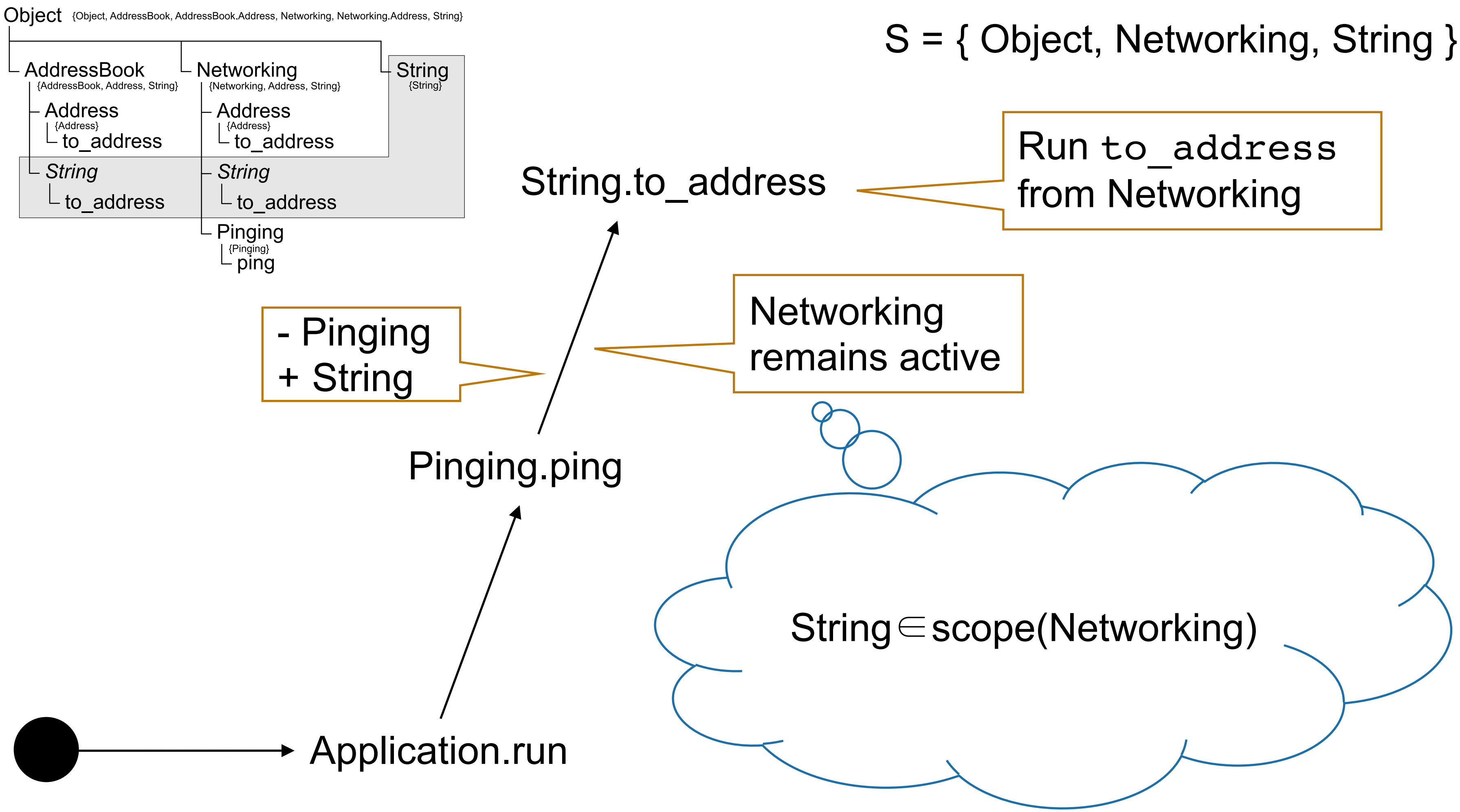
Example 2: Step by Step



S = { Object, Networking, Pinging }



Example 2: Step by Step



Implementation

- Prototypical implementation using metaprogramming
- Uses debug_inspector API for stack walking to implement customized method lookup in Ruby
- Give it a try (use Ruby 2.3): git@github.com:matthias-springer/ruby-class-ext.git

Related Work

- *Classboxes* [Bergel03]: Additional organizational unit (classbox), no support class nesting hierarchies
- *Ruby Refinements:*
  Pure lexical scoping (no local rebinding)
- *Context-oriented Programming (COP)* [Hirschfeld08]: Manual activation/deactivation necessary, difficult to control when modifications should be deactivated
- *Method Shells* [Takeshita13]: Additional organizational unit (method shell), new syntax for including/linking
- *MultiJava* [Clifton00], *Expanders* [Warth06]:
  No dynamic scoping, no new methods

Summary

"Extension Classes":
A new approach for open classes in Ruby

- Avoiding destructive modifications
- Reusable modifications (via modules)
- Scoped with respect to class nesting hierarchies
- Classes as only organizational unit

References

[Bergel03] A. Bergel, S. Ducasse, R. Wuyts. Classboxes: A Minimal Module Model Supporting Local Rebinding, Modular Programming Languages, 2003.

[Clifton00] C. Clifton, G. T. Leavens, C. Chambers, T. Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. OOPSLA, 2000.

[Hirschfeld08] R. Hirschfeld, P. Constanza, O. Nierstrasz. Context-oriented Programming. Journal of Object Technology, 2008.

[Takeshita13] W. Takeshita, S. Chiba. Method Shells: avoiding conflicts on destructive class extensions by implicit context switches. Software Composition, 2013.

[Tarr99] P. Tarr, H. Ossher, W. Harrison, S. M. Sutton. N degrees of separation: multi-dimensional separation of concerns. ICSE, 1999.

[Warth06] A. Warth, M. Stanojević, T. Millstein. Statically scoped object adaptation with expanders. OOPSLA, 2006.

Appendix

Ruby Refinements

```
module NoPopup
    refine WebPage do
        def popup; end
    end
end

class Browser
    using NoPopup

    def open(url)
        WebPage.new.open(url)
    end
end
```

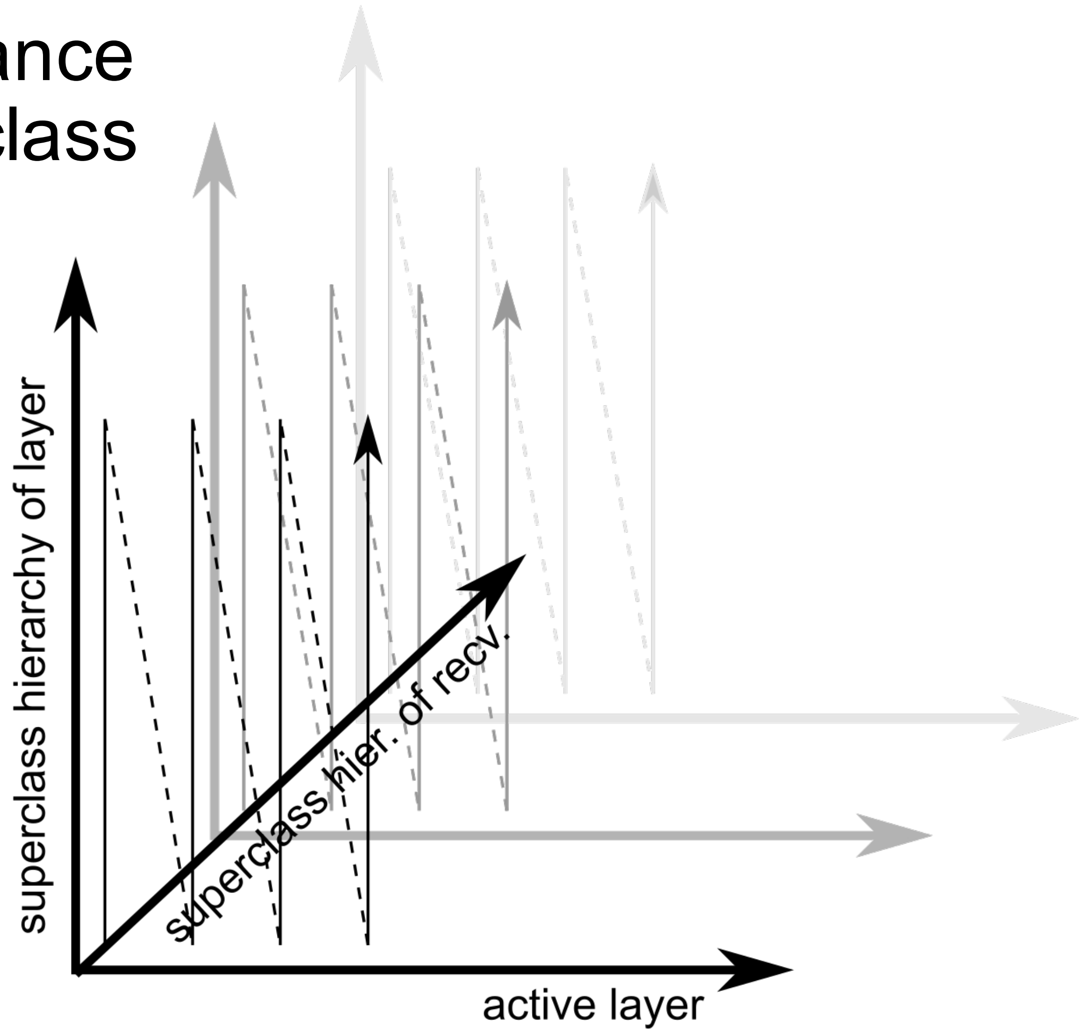
Pure lexical scoping: NoPopup will be deactivated after calling `WebPage.open`

Combining Modifications

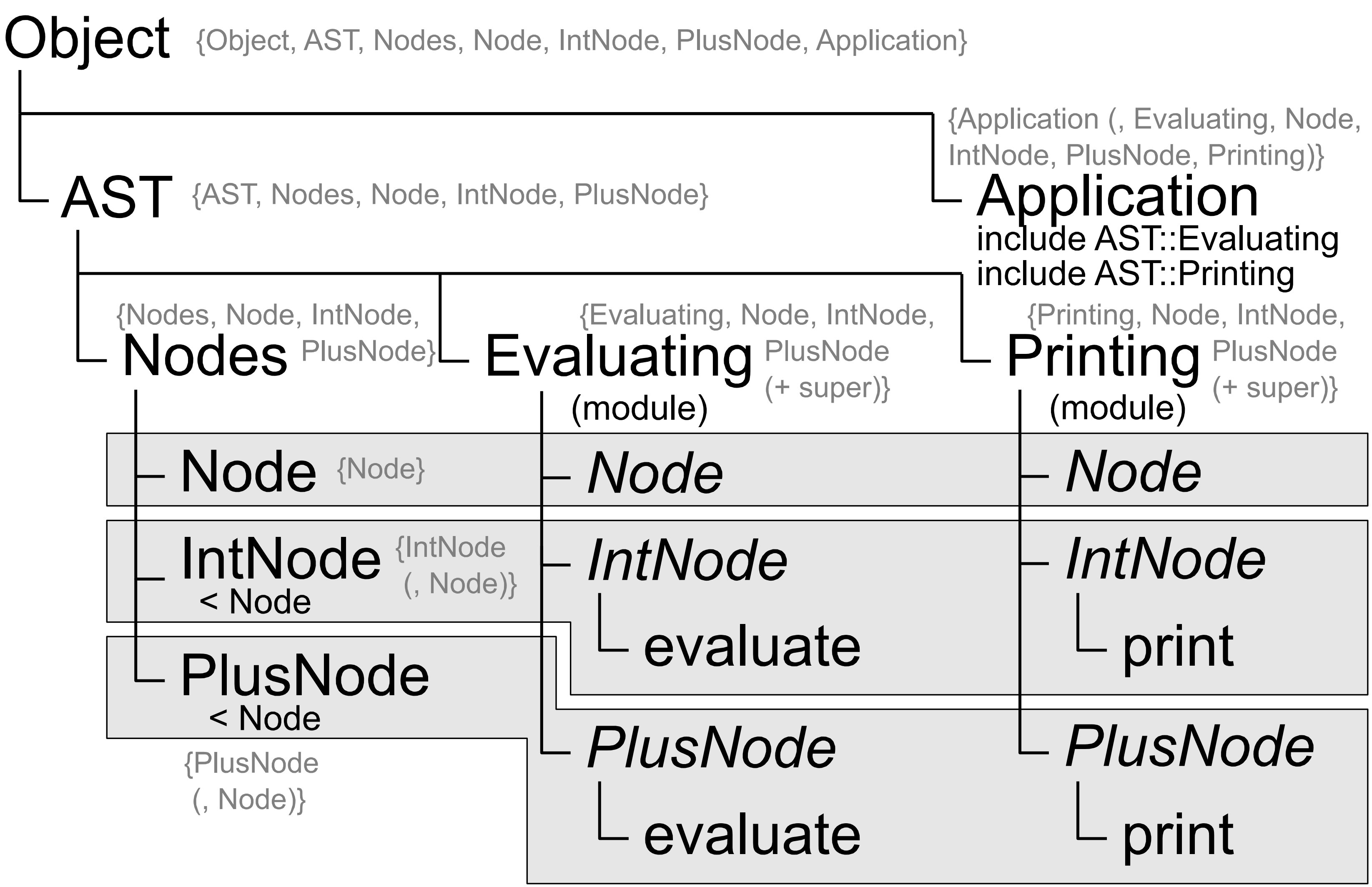
- What happens if multiple classes with variation points for the same method are active?
- S is actually not a set but a stack
  → *Class composition stack*
  (cf. layer composition stack in COP [Hirschfeld08])
- Last activated class takes precedence
- Modified super keyword to navigate 3 hierarchies
  - Inheritance hierarchy of layer class (i.e., of class containing modifications) → takes care of mixins
  - Class composition stack (`proceed` in COP, AOP)
  - Inheritance hierarchy of receiver class

Method Lookup

1. Superclass inheritance hierarchy of layer class
2. Layer composition stack
3. Receiver class inheritance hierarchy



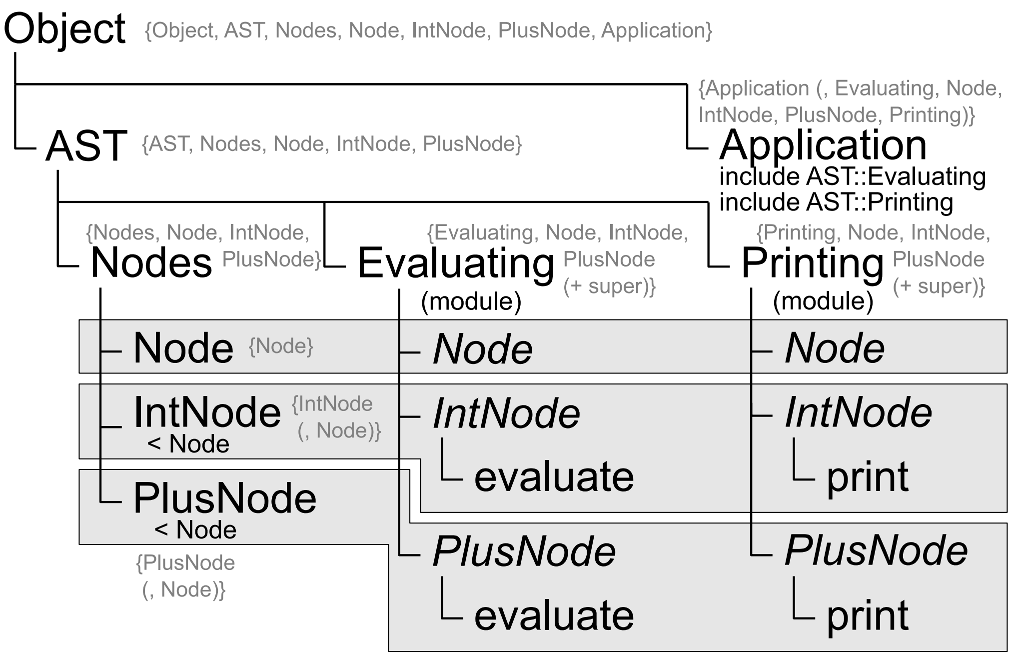
Example 3: Overview



Example 3: Mixins

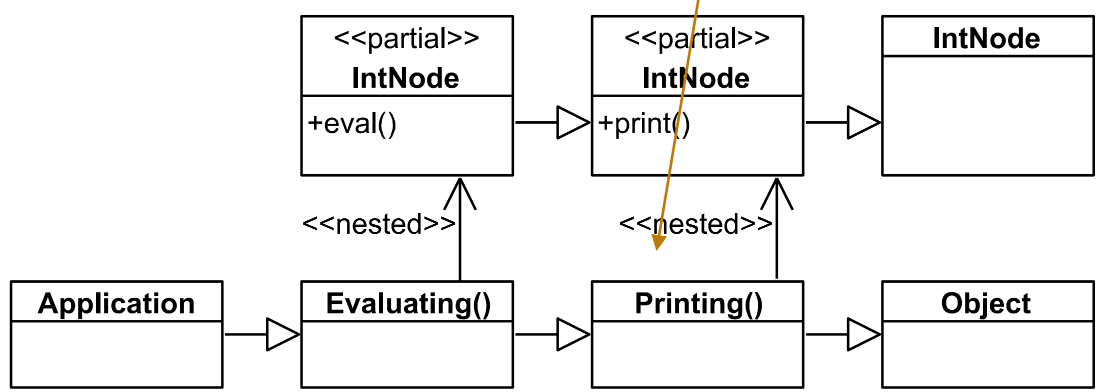
- Conceptually, module inclusion (mixin application) creates a new superclass
- Formalism does not have a special rule for mixin application, but only for superclasses
  → Assume that modules have been *desugared* to explicit superclasses from now on

Example 3: Inheritance

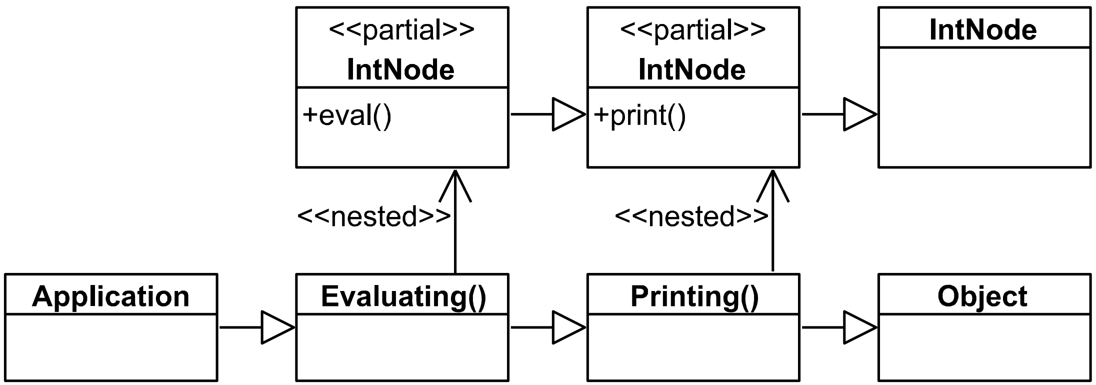


Mixins are desugared to explicit superclasses

```
class Application
    include AST::Evaluating
    include AST::Printing

    def evaluate(node)
        return node.evaluate
    end
end
```



Ex. 3: Method Lookup for `IntNode`



1. Application [IntNode]
2. Evaluating() [IntNode]
3. Printing() [IntNode]
4. Object [IntNode]
5. IntNode
6. Application [Node]
7. Evaluating() [Node]
8. Printing() [Node]
9. Object [Node]
10. Node
11. Application [Object]
12. Evaluating() [Object]
13. Printing() [Object]
14. Object [Object]
15. Object

Mixins are desugared to explicit superclasses

[...]: Partial class (conceptually)

Assuming single active class `Application`

- Activation/deactivation rules remain unchanged
- *Effective superclass hierarchy* determines method to be executed (and also guides the lookup for `proceed` (`super`))

Def.: Effective Superclass Hierarchy

Definition. *The effective superclass hierarchy of a class C is defined as $Effective(C)$, where S is the class composition stack ($S[1]$ is top of stack), $\#C$ is the number of superclasses of a class C, $super^i(C)$ is the i-th superclass of class C, $L[C]$ is the partial class targeting C defined in L (if there is one), $\langle\rangle$ brackets denote a (ordered) list, and summation is used for list concatenation.*

$$LayerHierarchy(L, C) = \sum_{i=0}^{\#L} \langle super^i(L)[C] \rangle$$

$$ClassLayers(C) = \Big(\sum_{i=1}^{|S|} LayerHierarchy(S[i], C) \Big) + \langle C \rangle$$

$$Effective(C) = \sum_{i=0}^{\#C} ClassLayers(super^i(C))$$

$LayerHierarchy(L, C)$ is the list of partial classes for class C defined in class L and its superclasses. $ClassLayers(C)$ is the list of partial classes of C (among all activated classes) and C itself.

In Example 3:

S = (Application)
(Object omitted)

LayerHierarchy(Application, IntNode) = (Application [IntNode], Evaluating() [IntNode], Printing() [IntNode], Object [IntNode])

Same as above plus IntNode
(only one layer/active class, i.e., `Application` in this example)

Account for superclasses of IntNode

Definition of “Scope of a Class”

Definition. *The scope of a class L is defined as the set containing L*

$$scope(L) = \boxed{\{L\}} \qquad \textit{(reflexivity)}$$

Direct method calls

Definition of “Scope of a Class”

Definition. *The scope of a class L is defined as the set containing L, all target classes corre- sponding to partial classes of L*

Indirect method calls

$$scope(L) = \{L\} \quad (reflexivity)$$

$$\cup \{ target(P) \mid P \in partials(L)\}$$

(dynamic scoping + local rebinding)

Definition of "Scope of a Class"

Definition. *The scope of a class L is defined as the set containing L, all target classes (and their reachable nested classes[*6]) corresponding to partial classes of L, all classes in the scope of all nested classes of L[*7]*

$$scope(L) = \{L\}$$ (reflexivity)

Visible in all nested classes of target classes

$$\cup \{C \mid C \in nested^*(target(P)) \wedge P \in partials(L)\}$$

(dynamic scoping + local rebinding (+ hierarch. scoping))

$$\cup \{C \mid C \in scope(N) \wedge N \in nested(L)\}$$ (hierarch. scoping)

Visible in all nested classes of defining class

Definition of “Scope of a Class”

Definition. *The scope of a class L is defined as the set containing L, all target classes (and their reachable nested classes*[*6]*) corresponding to partial classes of L, all classes in the scope of all nested classes of L*[*7]*, and all classes in the super-class of L*

$$scope(L) = \{L\} \qquad \textit{(reflexivity)}$$

$$\cup \{C \mid C \in nested^{*}(target(P)) \wedge P \in partials(L)\}$$

(dynamic scoping + local rebinding (+ hierarch. scoping))

Visible in nested classes + target classes of superclasses

$$\cup \{C \mid C \in scope(N) \wedge N \in nested(L)\} \qquad \textit{(hierarch. scoping)}$$

$$\cup\, scope(superclass(L)) \qquad \textit{(inheritance scoping)}$$

Definition of “Scope of a Class”

Definition. *The scope of a class L is defined as the set containing L, all target classes (and their reachable nested classes*[*6]*) corresponding to partial classes of L, all classes in the scope of all nested classes of L*[*7]*, and all classes in the scope of the superclass of L (if* $super(L) \neq$ `Object`*).*

Special rule because Object is the superclass of all classes. scope(Object) contains all classes.

$scope(L) =$ *flexivity)*

$\cup \{C \mid C \in nested^*(target(P)) \wedge P \in partials(L)\}$ *(dynamic scoping + local rebinding (+ hierarch. scoping))*

$\cup \{C \mid C \in scope(N) \wedge N \in nested(L)\}$ *(hierarch. scoping)*

$\cup\, scope(superclass(L))$ *(inheritance scoping)*