Welcome!

Reflective Programming in Smalltalk

by Bob Hinkle

OOPSLA '93

Tutorial Outline

1. Introduction
2. Reflecting on Contexts
3. Reflecting on the Compiler
4. Reflecting on Classes
5. Reflecting on Message Lookup
6. Conclusion

Reflection—An Academic Definition

What is reflection?

Reflection is the ability of a programming system to inspect and change a model of itself. Any changes to the model will alter the system's future computation, and vice versa.

- Introspection is the ability to inspect a self-model.

- Reflection is the ability to change the self-model.

- Causal connection is the requirement that changes to the self-model cause changes to the computation.

- Reification is the act of bringing previously unavailable features up to the reflective level.

The Joy of Being "Meta"

In the jargon, the prefix "meta" is used to indicate program behavior at the reflective level. This has unfortunate connotations of very coarse-grained reflection—that computation lurches from the normal (non-meta) level to the meta-level for some period of time, until (presumably) it lurches back again. As an alternative, we view reflection as occurring in a fine-grained, case-by-case way.

As an example of meta-ness, consider these two sentences:

Corn grows in Illinois.

"Corn" has one syllable.

In the first sentence the word "corn" denotes a yellow cylindrical vegetable. But in the second, quotation marks are used to shift to the meta-level, to discuss properties ot the word itself.

Reflection — A Working Definition

What is reflection?

A language (or more generally system) is reflective if it has access to the entities that define its computation. That is, if the elements that describe how things are done are as accessible as the elements that describe what is done. This access may be complete (ie, pervasive throughout the system) or partial.

In OOPL, we have objects to represent elements of the application domain: these might include customers, accounts, shipments, invoices, etc.

For reflection, we must also have objects to represent elements of the computation: these might include processes and contexts, messages, variables, parsers and compilers, etc.

Why Reflect?

Reflection gives programmers the ability to adapt and extend a language from within the language itself.

Thus, the language can develop and grow without having to wait for language implementors.

Programmers can add behavior that was unanticipated by the language designer.

Programs during execution can use reflection to dynamically configure themselves to their environment.

Why Not Reflect?

Reflection can add to system complexity, making program harder to understand, maintain, and extend

- Imagine programs that change variable access policies from lexical scoping to dynamic scoping on the fly, or that alter multiple-inheritance resolution rules. Does anyone want that???

Reflection adds new efficiency problems

- Extra message overhead to provide reification
- Difficulty in optimizing in the face of reflection

Language divergence

- Will reflective changes result in essentially different Smalltalks?

Goodness Criteria

We look for localized system changes that have a practical pay-off, either in terms of developer productivity or language expressiveness.

Changes should leave "normal" (ie, unreflecting) programmers unaffected—they should be able to program as usual without being aware that reflection is occurring or could occur.

Changes should have negligible effect on the system when reflection isn't being used. "You don't pay if you don't play."

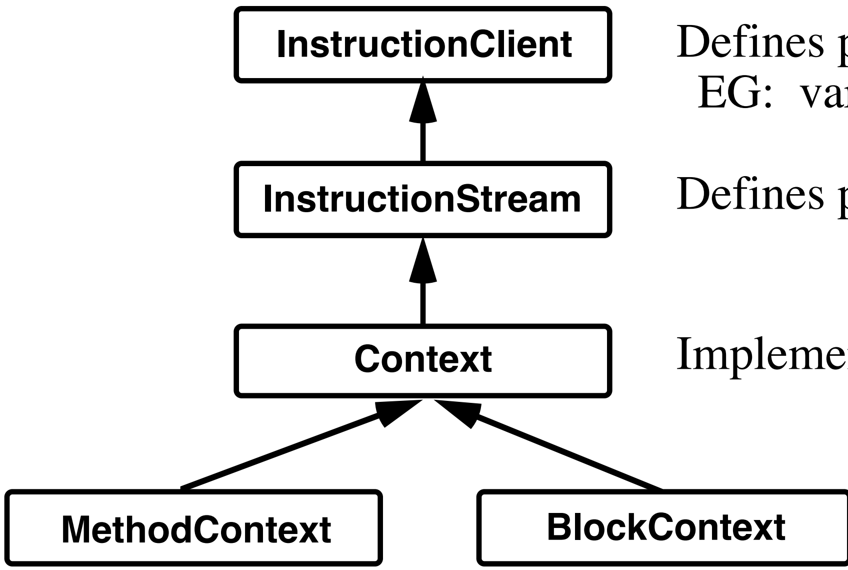
Reflecting on Contexts

With access to contexts, we have access to the interpretation process, and thus to debugging. In addition, contexts reify the execution structure and so allow us to affect control flow.

Two examples of reflection on contexts:

- Monitoring variable accesses
- Adding backtracking

The Interpretation Hierarchy



Defines protocol for things that interpret
EG: variables, jumps, primitives, sends

Defines protocol for things that are interpreted

Implements **InstructionClient** messages

Concrete subclasses of **Context**

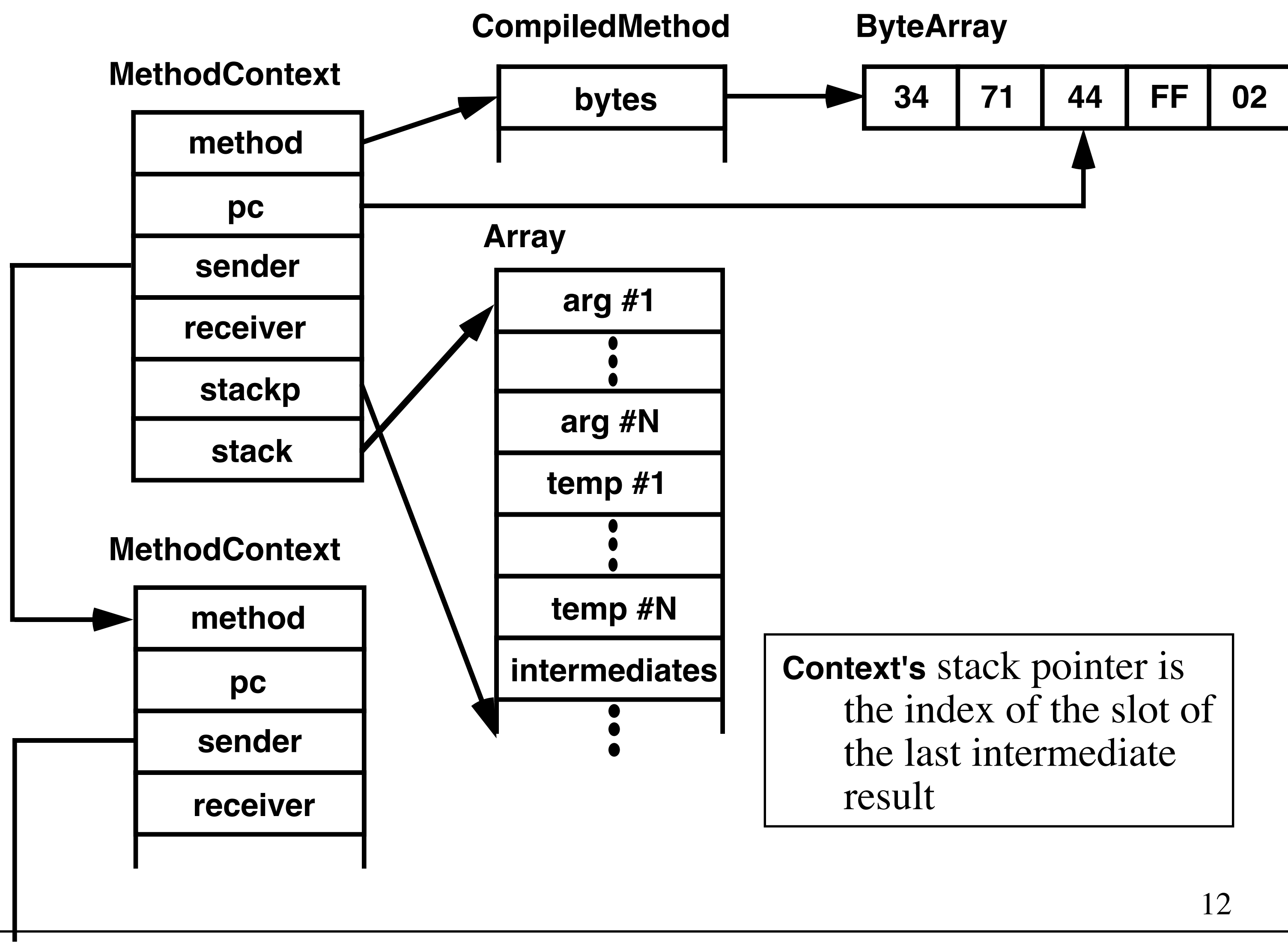
How Interpretation Works

InstructionStream>>interpretNextInstructionFor: reads and decodes the next byte code in the **CompiledCode** being interpreted. Depending on the byte code, an **InstructionClient** method will be called to (for example) **#pushInst:**, **#storeStatic:**, or **#send:numArgs:**.

However, the methods in **InstructionClient** are (almost) all empty: they must be over-ridden in a subclass for something concrete to happen during interpretation.

That's what **Context** does—it re-defines each of the methods in **IntructionClient** to operate on the layout of **Contexts**. And it provides other methods to make debugging easier and to facilitate manipulation of a process' context stack.

The Layout of Context



Context's stack pointer is the index of the slot of the last intermediate result

Context's Instance Variables

method — a **CompiledCode** (or subclass), which has an array of byte codes

pc — a **SmallInteger** indexing the next byte code to be executed

sender — the **Context** that initiated this **Context's** message send

receiver — any **Object**, the recipient of the message implemented by method

stackp — a **SmallInteger** indexing the last result on the **Context's** stack. (This is counted starting from the beginning of the stack, including arguments and temporaries.)

stack — an **Array** containing the arguments, the temporaries, message receivers, message parameters, and results of intermediate calculations

A Few Context Methods

```
pushLocal: index
  self push: (self localAt: index + 1)
```

```
storeInst: index
  receiver instVarAt: index + 1 put: self top
```

```
returnTop
  ^self return: self top
```

```
send: selector numArgs: na
  | rcvr |
  rcvr := self localAt: stackp - na.
  ^self
    send: selector
    receiver: rcvr
    class: rcvr dispatchingClass
    super: false
    numArgs: na
```

```
push: value
  stackp := stackp + 1.
  self localAt: stackp put: value
```

```
return: value
  | caller |
  caller := sender.
  caller isNil
    ifTrue: [^self cannotReturn: value]
    ifFalse:
      [sender := nil.
      ^caller push: value]
```

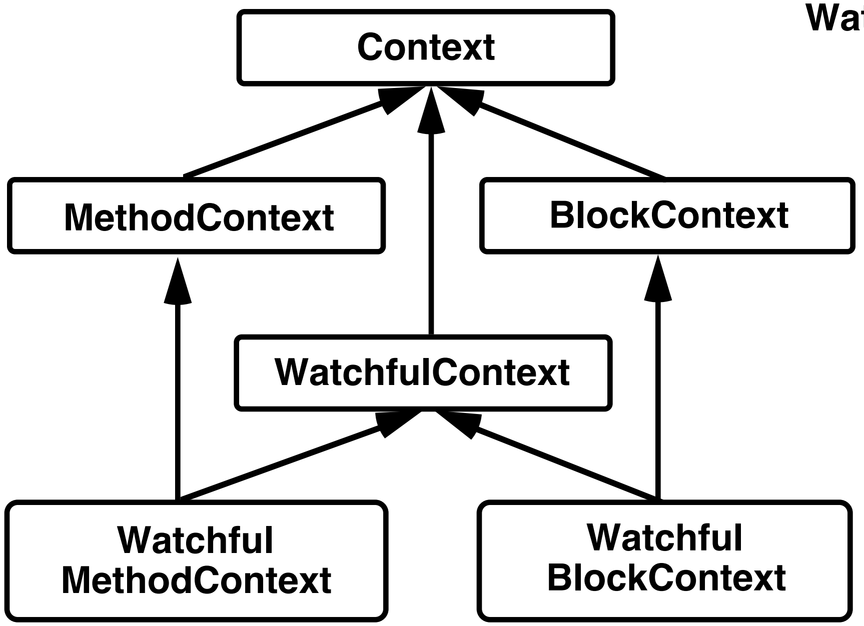

Example #1: Variable Accesses

Objective: We want to know which variables are accessed during the execution of a particular method or code fragment.

Solution: Implement a new kind of **Context**. (Well, three, actually.) These new **Contexts** will behave like normal **Contexts**, except that when instance variables are read—which happens when the **#pushInst:** and **#pushInstAny:** messages are sent—notify some client of which variable is being read from which object.

Key: Though the VM will always create normal **Contexts**, we can start execution with our new **Context**. That's our hook into the interpretation process.

Variable Accesses—2



WatchfulContext implements:

```
pushInst: index
  self describeInstAccess: index + 1.
  self push: (receiver instVarAt: index + 1)

describeInstAccess: index
  self client
    context: self
    accessingInstanceVariable: index
    for: receiver.
```

It's also necessary to re-factor **Context** so that any new **Contexts** created during interpretation are from the right class. This involves replacing hard-coded references to class names by message sends. (A common modification for reflective programming, it seems . . . .)

Variable Accesses—3

Assuming we have a **CompiledMethod** to interpret (called **aMethod**, say), we can use **WatchfulContexts** like this:

```
ctx := WatchfulMethodContext
  sender: root receiver: nil method: aMethod arguments: #(nil).
ctx client: anObject.
Send := true.
[ctx == root] whileFalse: [ctx := ctx step].
return := ctx top
```

With a client that prints to the **Transcript**, monitoring the method generated from **'Date today asDays'** results in:

#Date->Date==>'value' #Time->Time==>'value' #Integer->Integer==>'value' #LargePositiveInteger->LargePositiveInteger==>'value' #SmallInteger->SmallInteger==>'value' SmallInteger==>'maxBytes' . . .

a TimeZone==>'secondsForDST' #Date->Date==>'value' 23 July 1993==>'year' 23 July 1993==>'day' #Array->Array==>'value' #Time->Time==>'value' 23 July 1993==>'year' 23 July 1993==>'day' . . . .

Example #2: Control Flow

Objective: We want to modify control flow in Smalltalk.

Solution: Use a designated message send to create a "placeholder" context on the stack. Later, execution can be backed up to any placeholders when a failure occurs.

Key: The pseudovariable **thisContext** and the **sender** message allow us to inspect the entire context stack. The context stack can be manipulated with messages like **#restart**, **#unwindUpTo:**, and **#resumeWith:**.

Context Messages for Control Flow

restart — reinitializes stack pointer and nils out temporaries

resumeWith: — will continue execution in the receiving context, with the message's parameter used as the result of the active message send

sender: — changes the receiver's sender to be the parameter

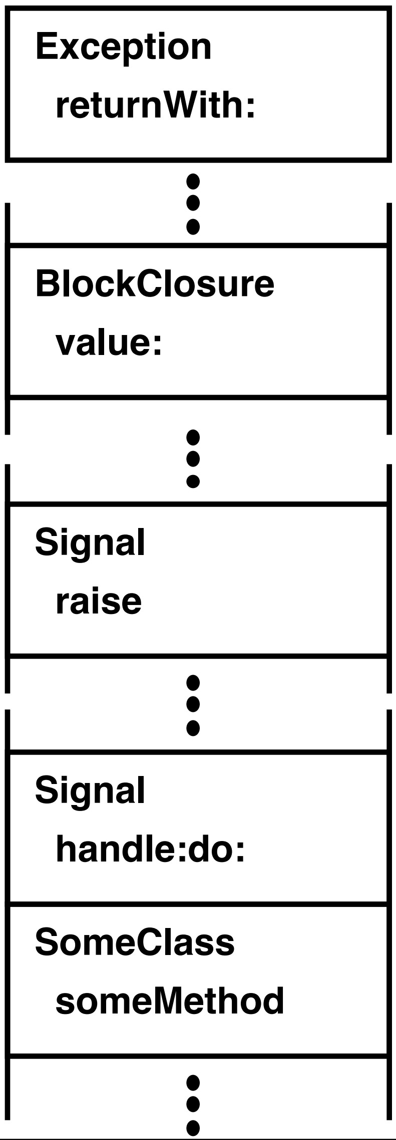
tempAt: — accesses a temporary variable

tempAt:put: — changes the value of a temporary variable

terminateTo: — terminates all contexts between the receiver and the message's parameter, and ends with the receiver's sender being the parameter. (Thus, when the receiver returns, execution will continue in the parameter—whether it was part of the receiver's original call chain or not.)

unwindUpTo: — runs any unwind blocks between the receiver and the parameter

Contexts as Placeholders

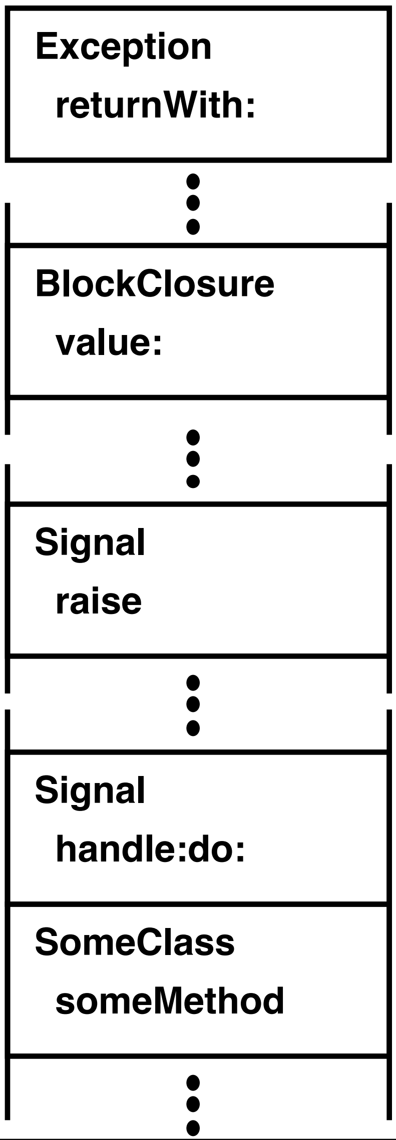



```
Signal>>handle: handlerBlock do: doBlock
  ^doBlock value
```

```
Exception>>returnWith: aValue
      "Unwind the stack and return aValue
      as the value of the handle:do: expression."

      self unwind.
      thisContext terminateTo: handlerContext.
      ^aValue
```

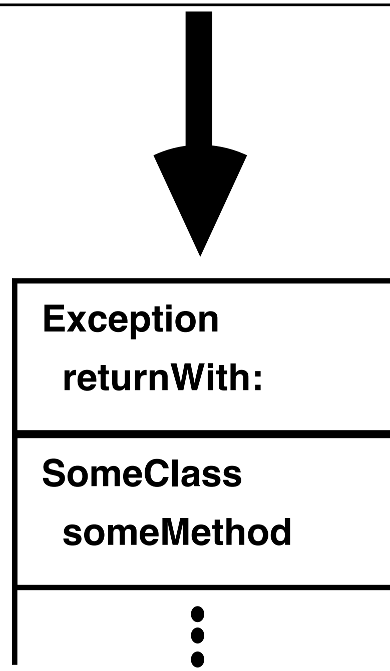
The **#handle:do:** context is a convenient marker on the stack. Execution can be unwound up to it, and then control terminated back into the **#handle:do:'s** sending context, with an arbitrary value returned as its result.

Contexts as Placeholders—2



When **Exception>>returnWith:** is executed, the stack is unwound from **thisContext** to the **#handle:do:** context.

Then, #**terminateTo:** removes contexts from **#handle:do:** to the one below **#returnWith:** from the call chain, leaving a truncated stack:



Finally, when **#returnWith:** returns, execution resumes in the **#someMethod** context, with **aValue** (the parameter from **#returnWith:**) used as the return value for the **#handle:do:** send.

Reflecting on the Compiler

With access to the compiler, we can intervene in the process of translating source text into byte codes. We can alter parsing to modify the language, or we can manipulate parse trees directly to make program transformations, or we can alter code generation to give new meanings to existing language elements.

Two examples of reflection on the compiler:

- Support for program refactoring
- Active variables

The Compiler's Interface

Most of the compiler's function is defined in the abstract class **SmalltalkCompiler**. Its subclass **Compiler** defines a few methods to specify classes to be used while compiling. The interface for **SmalltalkCompiler** includes:

- **compile:in:notifying:ifFail:** — compiles the source string and returns a **MethodNodeHolder**

- **evaluate:in:receiver:notifying:ifFail:** — generates a method for the source string and causes the receiver parameter to perform the method. Returns the result of that evaluation.

- **format:in:notifying:** — standardizes the source string by generating a method for it and then printing the parse tree

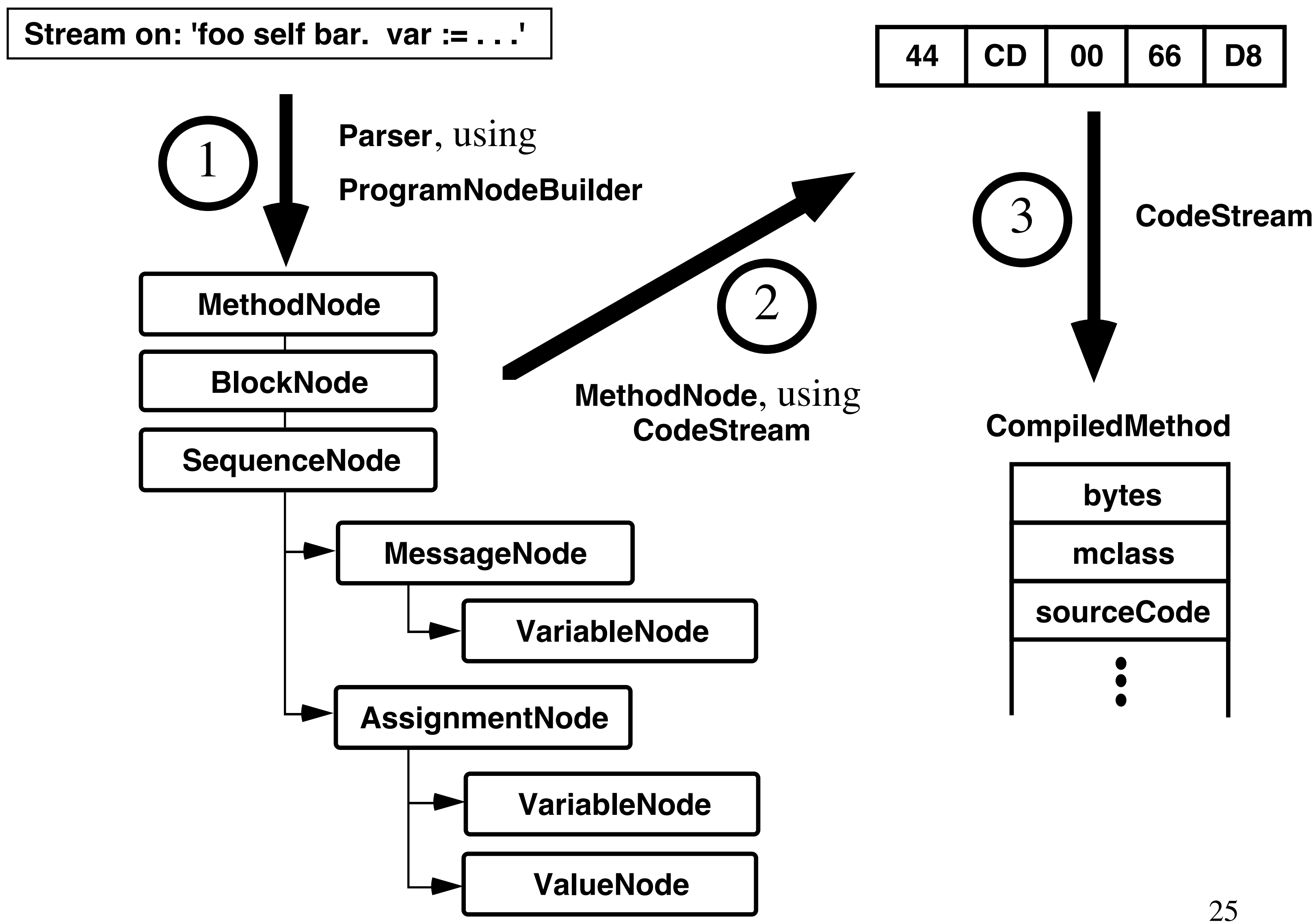
- **parse:in:notifying:** — compiles the source string and returns a **MethodNodeHolder**. Also saves the source map for debugging.

The Real Work in Compiling

These four methods all call on a common worker function:

SmalltalkCompiler
translate: <a ReadStream>
noPattern: <a Boolean: true for evaluation, false for compilation>
ifFail: <aBlockClosure, evaluated if parsing fails>
needSourceMap: <a Boolean, true if the source map (which maps between source position and program counters) should be saved in the MethodNodeHolder>
handler: <a CompilerErrorHandler, notified of errors in parsing and code generation>

The Real Work in Compiling—2



Reflecting on Compilation

We can intervene at various points of the compilation process to achieve different effects:

- By changing the parser, we can make syntactic changes to the language, to support new constant types or constructors, for example.

- We can also intervene between parsing and code generation to manipulate the parse tree directly, as with the refactoring example coming next.

- By changing the code stream, we can make semantic changes to the language, interpreting members of the **ProgramNode** family as we wish, as with the active variables example.

Example #3: Refactoring

Objective: We want to support common design refactoring in the browsers, so users don't have to do them by hand. In this example, we'll look at instance variable renaming.

Solution: Use a subclass of **ProgramNodeEnumerator** to manipulate parse trees. After modifying the parse tree, we can generate both new source and new byte codes with the instance variable renamed as desired.

Key: The compiler is first class, so we can add to its interface to support parse tree changes. And parse trees are first class, so we can use their operations to easily find and replace variable nodes.

Refactoring—2

ProgramNodeEnumerator is used by asking it to **#doNode:** on the root of some parse tree. It uses double-dispatching by asking each node in turn to **#nodeDo:** itself on the **ProgramNodeEnumerator**. The messages sent in response to this, such as **#doAssignment:variable:value:** and **#doMessage:receiver:selector:arguments:**, are left for subclasses to implement.

Our subclass, **ProgramNodeInstantiator**, walks a parse tree and replaces each node in succession. In most cases, nothing is done, as in

```
doAssignment: aNode variable: var value: val
    ^aNode
        variable: (self doNode: var)
        value:  (self doNode: value)
```

Refactoring—3

However, when instance variables are referenced, we check to see if the one we're interested in is being accessed, and, if so, we rename it:

```
doVariable: aNode name: nameString
    ^nameString = variableToRename
        ifTrue: [aNode name: newVariableName]
        ifFalse: [aNode]
```

After transforming the parse tree, we can generate byte codes by using **CodeStream** as normal, and we can generate new, corrected source for the method by printing the parse tree (as in **SmalltalkCompiler>>format:in:notifying:**).

Example #4: Active Variables

Objective: We want to hook into variable accesses for a particular object, which will support different kinds of tracing and profiling, as well as the addition of watchpoints to the debugger.

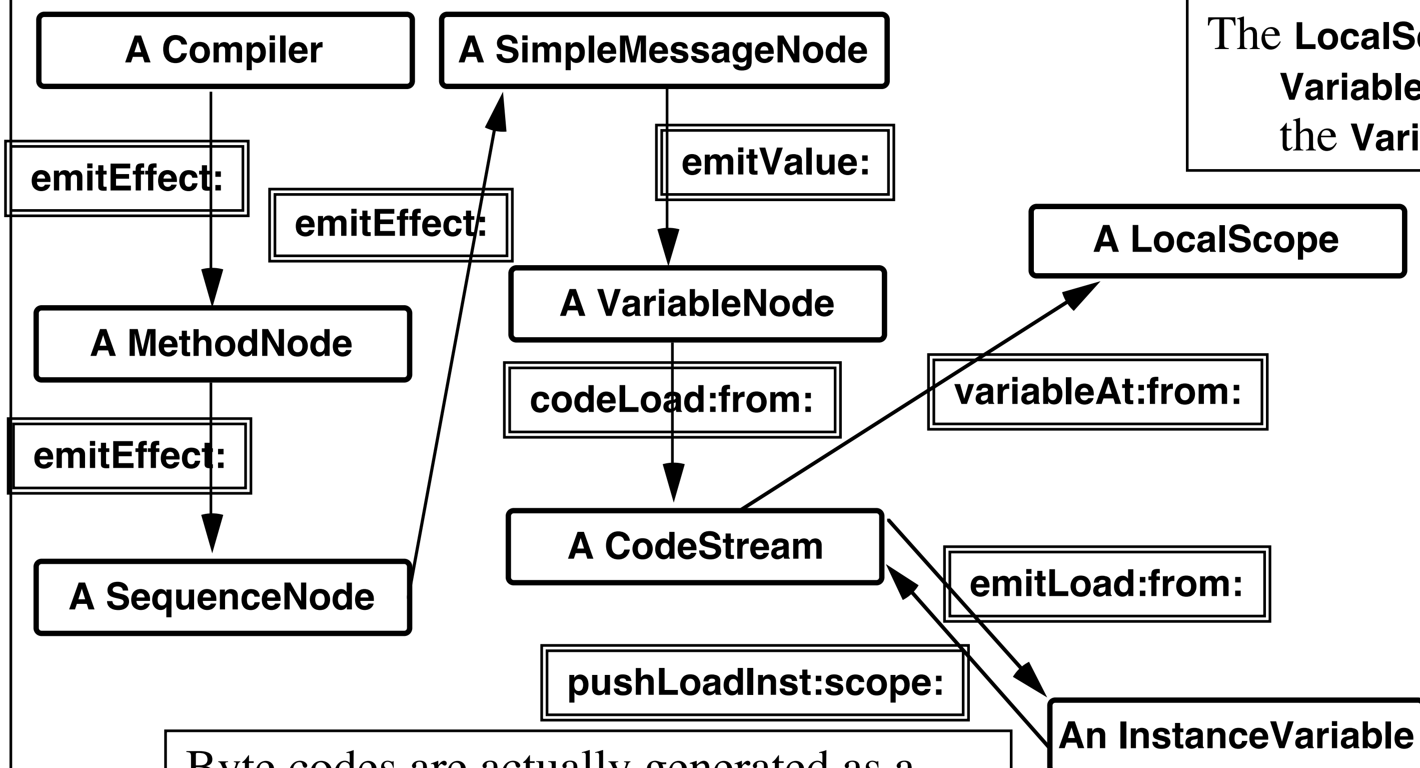
Solution: Create new classes to be used when variables emit their effect onto the **CodeStream**. These new classes will ensure that active variables are accessed via message sends rather than directly.

Key: The code generation process is distributed between **CodeStream** and the **ProgramNode** objects. By slightly changing the way **VariableNodes** get mapped to **VariableDefinitions**, we can modify a small part of code generation without affecting anything else.

Code Generation

Code is generated by an interaction between subinstances of **ProgramNode** and a **CodeStream** object. It's an interwoven process:

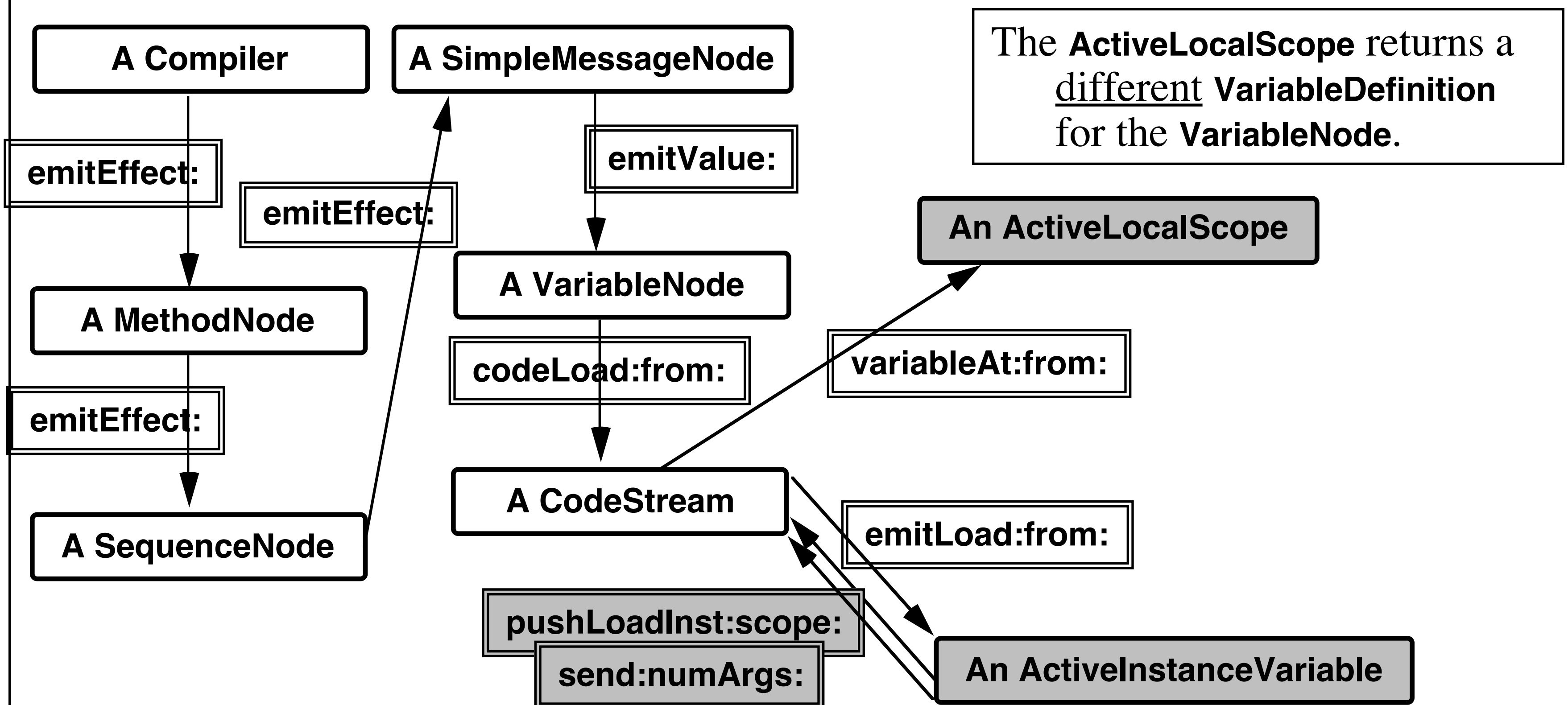
The **LocalScope** returns a **VariableDefinition** for the **VariableNode**.



Byte codes are actually generated as a result of **#pushLoadInst:scope:**.

Code Generation for Active Variables

Since an **InstanceVariable** determines the protocol for accessing itself, we can alter code generation simply by creating an **ActiveInstanceVariable** as a replacement. Voila!



Reflecting on Classes

With access to classes, we have access to the facilities by which objects are created and their behavior defined. Since classes are first-class objects, we can extend the existing tools or create new ones to query classes and to change them at development time or at (user) run time.

Two examples of reflection on classes:

- Object debugging
- Run-time method definition

The Organization of Classes

Four classes participate in defining classes in Smalltalk. They are **Behavior**, **Class**, **ClassDescription**, and **Metaclass**.

- **Behavior** — Provides minimum functions to support method definition and compilation. Most of **Behavior's** methods have to do with compiling code or with accessing method dictionaries, the superclass, and the subclass list.

- **ClassDescription** — Adds additional function to **Behavior** to support named instance variables, category organization for classes and for methods, change file logging, and filing out.

- **Class** — Adds class variables, pool variables, and subclass creation.

- **Metaclass** — Specialized for classes of classes, designed to have unique instances

Run-time Method Definition

Using the compilation methods in **Behavior**, we can add new methods to a class on the fly. This supports dynamic specialization of object behavior (as in, for example, the Alternate Reality Kit), or the construction of programming support environments.

```
aClass compile: 'foo ^3' classified: 'temporary methods'
```

compiles the source string, adds the resulting method to **aClass'** method dictionary, updates **aClass'** class organization, and adds the source string to the changes file.

```
aClass compile: 'foo ^3' notifying: nil ifFail: []
```

will compile and install the method without changing the class organization or changes file.

Run-time Method Definition—2

Alternatively, we can define and run a method without involving a class, using the same technique as **ParagraphEditor's** text evaluation. That method calls **SmalltalkCompiler>>evaluate:in:receiver:notifying:ifFail**, which is defined as:

```
evaluate: textOrStream in: aContext receiver: receiver notifying: aRequestor ifFail: failBlock
 | methodNode method |
 class := (aContext == nil
  ifTrue: [receiver]
  ifFalse: [aContext homeReceiver]) class.
 self from: textOrStream
  class: class
  context: aContext
  notifying: aRequestor.
 methodNode := self translate: sourceStream noPattern: true ifFail: [^failBlock value].
 method := methodNode generate.
 ^context == nil
  ifTrue: [receiver performMethod: method]
  ifFalse: [receiver performMethod: method with: context]
```

The message **#from:class:context:notifying:** initializes the compiler.

#translate:noPattern:ifFail: forwards to the method that parses the source and generates byte codes.

Example #5: Lightweight Classes

Objective: We want to give unique behavior to individual instances of the same class. This provides powerful debugging possibilities as well as support for per-object specialization.

Solution: Implement a new kind of **Behavior**, to be called **LightweightClass**. This new class will add a method dictionary between an object and its creation class, allowing object-unique behavior to be specified. **LightweightClasses** should be lightweight in terms of space and time to create, initialize, and destroy.

Key: Classes are first-class objects, so they can be created at run-time. Also, the **#changeClassToThatOf**: method supports changing (most) objects' class at run-time.

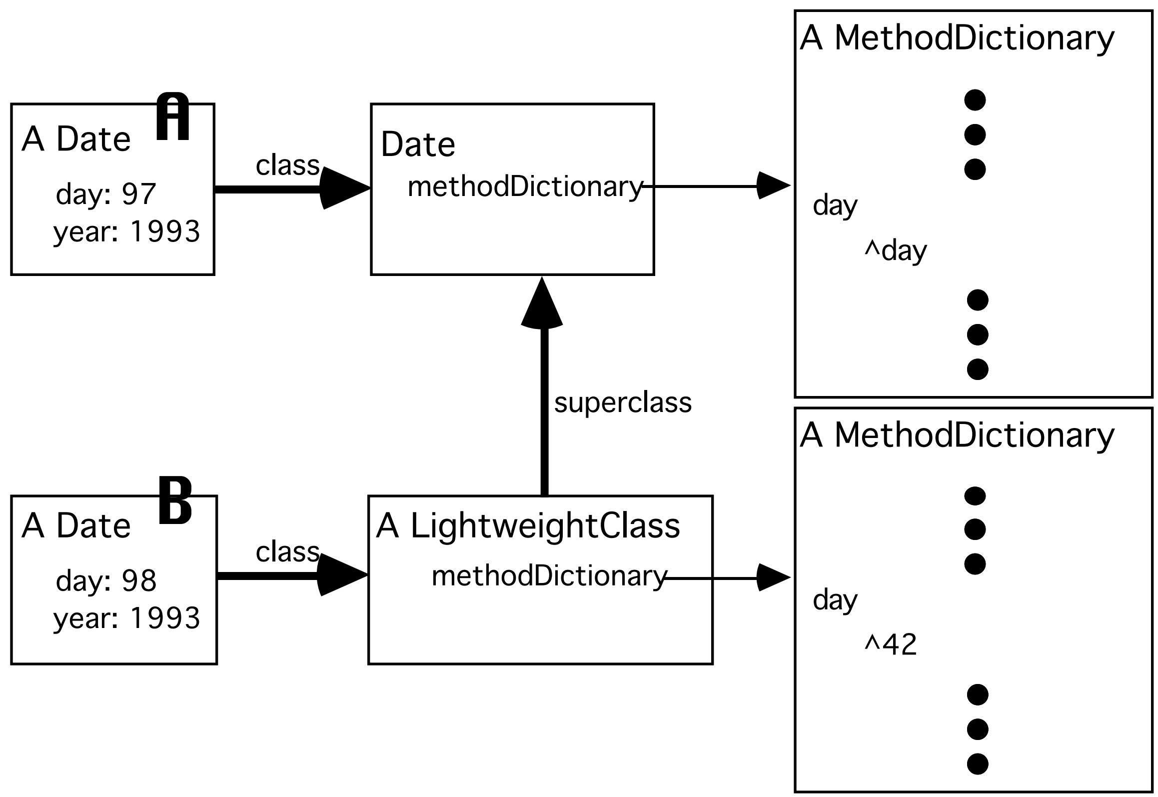
Creating Lightweight Objects

```
becomeLightweight
 | lwc |
 (self dispatchingClass isKindOf: LightweightClass)
  ifFalse: [
   lwc := LightweightClass newWithSuper: self class.
   self changeClassToThatOf: lwc basicNew]
```

If the receiver already has a **LightweightClass**, we don't need to do anything. Otherwise, we create a **LightweightClass** whose superclass is the receiver's creation class. (This will make inherited messages work correctly—see next page.)

Then we change the receiver's class (as stored in its memory structure) to the **LightweightClass**. We have to instantiate the **LightweightClass** to do this, but use **#basicNew** for efficiency.

Messages to Lightweight Objects



A day returns **97**, as expected. But **B day** returns **42**, since the method to implement the **#day** message is found first in the **LightweightClass'** method dictionary.

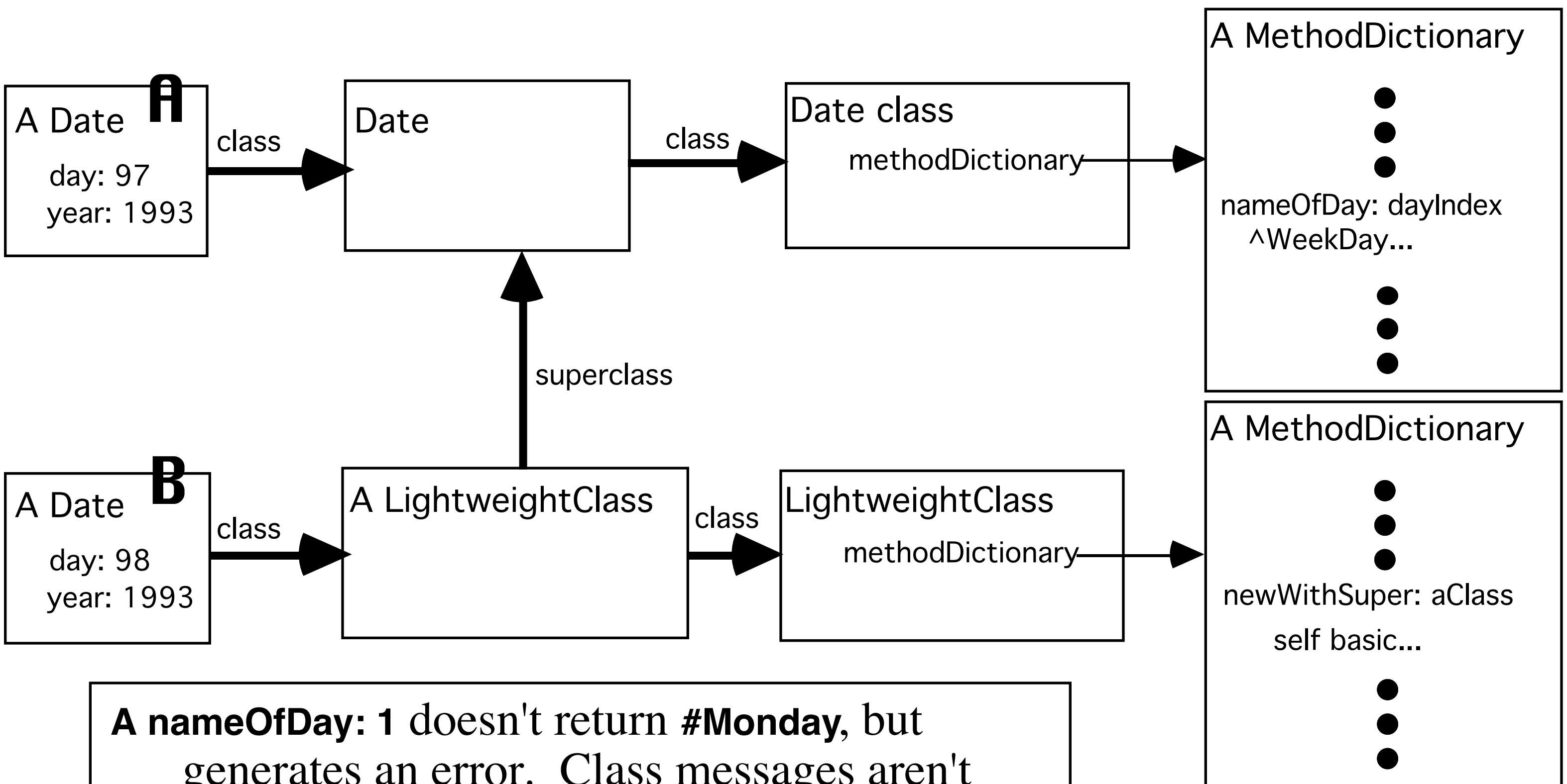
Creating Lightweight Classes

```
Behavior subclass: #LightweightClass
    instanceVariableNames: 'name'
    classVariableNames: 'ClassMethod '
    poolDictionaries: ''
    category: 'Object Debugging'

newWithSuper: aClass
 ^self basicNew initializeWithSuper: aClass

initializeWithSuper: aClass
 | md |
 self superclass: aClass.
 md := MethodDictionary new.
 md at: #class
  put: (ClassMethod copy mclass: self).
 self methodDictionary: md.
 format := (aClass instSize bitAnd: 255) bitOr: -4096.
 self name: '{', aClass name, '}'.
```

We need to initialize the method dictionary to solve a problem with class messages.

Messages to Lightweight Objects Gone Astray

What happens to class messages???



A nameOfDay: 1 doesn't return **#Monday**, but generates an error. Class messages aren't handled correctly yet . . .

Fixing Lightweight Classes

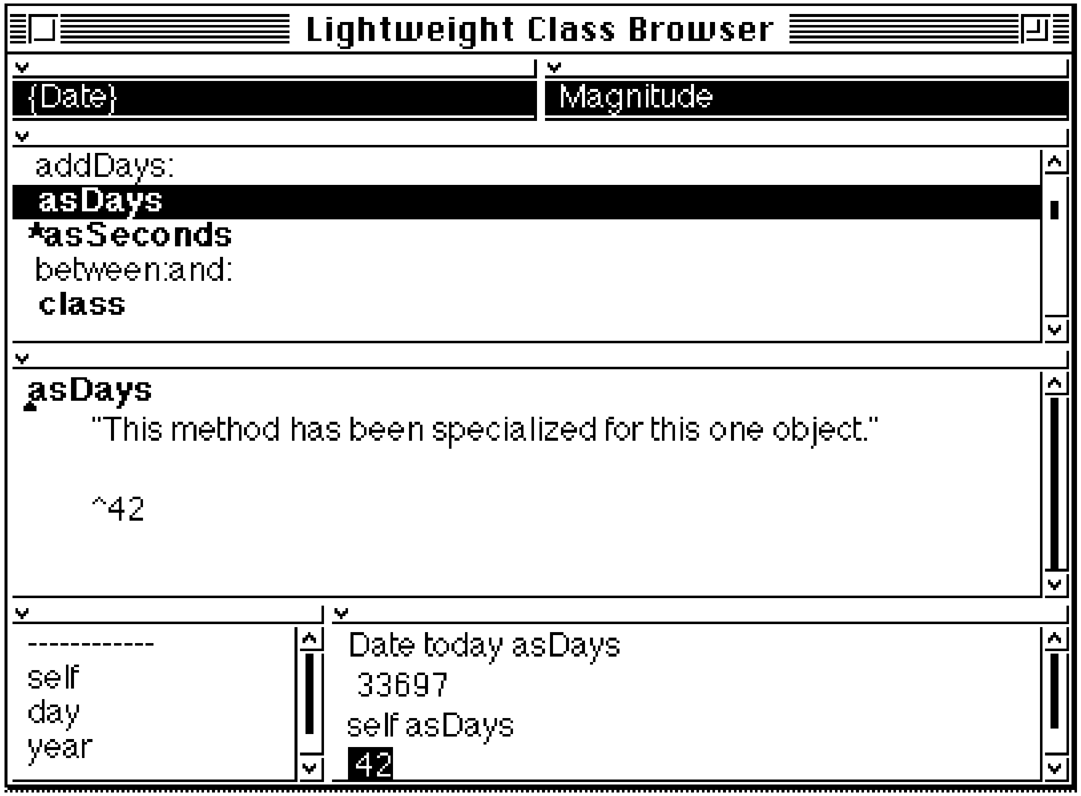
```
class
 <primitive: 111>
 self primitiveFailed

dispatchingClass
 <primitive: 111>
 self primitiveFailed

initialize
 | source |
 source := 'class
  ^self dispatchingClass superclass'.
 ClassMethod := (LightweightCompiler new
  compile: source
  in: Object
  notifying: nil
  ifFail: []) generate.
 ClassMethod sourceCode: source
```

In instances of **LightweightClass**, the **#class** method is overridden to return the **LightweightClass**' superclass. This is set in **#initializeWithSuper:** to be the lightweight object's creation class.

The message **#dispatchingClass** will always return the first class in the lookup chain for the receiver.

Methods throughout the image may have to be changed if they send the **#class** message.

Object Debugging



This browser allows users to add new methods or modify existing methods on a per-object basis, useful for debugging. The browser can be opened by a menu selection from all inspectors.

Reflecting on Message Lookup

Messaging is one of the places in Smalltalk that's hardest to access. It's possible to do some reflecting by altering the compilation process and inserting code at the beginning of methods or whenever a send is done. However, there is one hook built in to Smalltalk that lets us into message sending in a less heavy-handed way: it's the infamous **#doesNotUnderstand:**. Using **#doesNotUnderstand:** allows us to do some filtering of messages or forwarding them in intelligent ways.

As an example of reflection using the **#doesNotUnderstand:** message, we'll look at creating proxies, which might be used to support persistent objects.

How Message Lookup Works

When a message is sent, the receiver and arguments will be stored on top of the context's stack, and the selector to be used will be referenced by an index into the literals of the context's **CompiledCode** or by an index into the list of special selectors.

A method implementing the given selector is searched for in the method dictionary of the receiver's class. If a method is found, it is run with the receiver and given arguments.

If no method is found, the search continues in the receiver's class' superclass, and it continues up the superclass chain one method dictionary at a time, until eventually a method for the selector is found.

However, . . .

How Message Lookup Doesn't Work

Sometimes the message lookup process reaches a class with no superclass (usually **Object**). If that class' method dictionary doesn't contain the given selector, no implementing method is available for it.

The specified behavior for the virtual machine in this case is to send the **#doesNotUnderstand:** message to the receiver of the unimplemented message. The parameter to **#doesNotUnderstand** is a **Message** object, which wraps up the selector and arguments in a tidy bundle.

```
doesNotUnderstand: aMessage
 | selectorString |
 selectorString := Object errorSignal
  handle: [:ex | ex returnWith: '** unprintable selector **']
  do: [aMessage selector printString].
 Object messageNotUnderstoodSignal
  raiseRequestWith: aMessage
  errorString: 'Message not understood: ' , selectorString.
 ^self perform: aMessage selector withArguments: aMessage arguments
```

This default response raises a signal for the message that's not understood. This leads to the "bomb box" if the signal doesn't have a handler.

Hooking into **#doesNotUnderstand:**

With this design, we can override **#doesNotUnderstand:** to build proxy objects to load in file-based objects. First, we need a **Proxy** class that doesn't understand <u>any</u> messages:

```
nil subclass: #Proxy
    instanceVariableNames: 'client'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Kernel-Objects'
```

We can write an **#on:** method for **Proxy class** that will create a new **Proxy** with the parameter as its client.

```
on: anObject
    ^self new client: anObject
```

Hooking into **#doesNotUnderstand:**—2

As it is, **Proxies** don't understand any messages except **#client** (which we could name something more obscure, or even use **#instVarAt:put:** (!!!) to avoid).

The only other thing to do is to implement **#doesNotUnderstand:**. A starting point is:

```
doesNotUnderstand: aMessage
    "We could do some pre-processing here."
    ^client perform: aMessage selector withArguments: aMessage arguments
    "We could capture the result and post-process here."
```

To add protection to objects, we could check an access list before forwarding the message. Or we could write some trace information to the screen or to a file.

One-time Proxy Usage

As we will see, there are problems with this implementation, but it does support a kind of one-time usage. For example, we could use this kind of proxy to reduce memory usage by either compressing or caching on disk a large and infrequently used object. After the Proxy replaces the object throughout the image (by way of become:), it can store a pointer to the compressed representation or file as its client, and the original object can be garbage collected. To restore the object if and when it was ever referenced again, we would implement

```
doesNotUnderstand: aMessage
    | anObject |
    anObject := self restoreObjectFromFile: client.
    self become: anObject.
    ^self perform: aMessage selector withArguments: aMessage arguments
```

The Self Problem

Unfortunately, this naive implementation has a major problem: it leaks!!

After the **#perform:withArguments:**, control is out of the **Proxy's** hands until the return. During this time, the **Proxy's client** can pass itself (using the pseudovariable **self**) as an argument in any messages it sends, breaking out of the **Proxy's** protection.

There are possible solutions to this problem in Smalltalk, but none (yet) are cheap. One way that at least localizes the execution cost is to use a new kind of **MethodContext** to run the **client's** method. In this **MethodContext**, any references to **self** would be re-interpreted to mean the **Proxy**. But this would require interpretation as long as the program is running **client's** code.

Concluding Thoughts

- Reflection is good! It gives programmers the power to extend the language and environment to suit their purposes.
- Even though the theoretical underpinnings are still being formulated, there are currently practical and useful things to do with reflection.
- We need to take advantage of existing reflective facilities to increase programmer effectiveness and to understand what reflective support will be needed from next-generation languages.
- Reflection is not free, but with controlled use its costs can be managed.

Reflection in Smalltalk

Changes to interpretation

- Debugging, tracing, profiling, testing

Extension of the environment

- Change management, new and improved browsers, dependency systems, multiple namespaces

New execution types

- Backtracking, exception handling

Reflection in Smalltalk—2

Intervention in message sends

- Proxies, capabilities, tracers, filtering & transformation

Changes to the architecture of classes

- Method dictionary flattening, multiple inheritance, mix-ins, refactoring of Behavior, lightweight classes

New compiling functionality

- Optimization, parse-tree manipulation, refactoring

Reflection in Smalltalk—3

Language extension

- New constant types, constructors, uninitialized variables, macros

Changes to methods

- First-class interfaces, first-class comments, method types, delegation, multi-methods

Concluding Quotation

The
unreflected life
is not worth
living.

— Socrates