# Appendix 4 - Classes, Metaclasses, and Metaprogramming

## Overview

This chapter examines the ultimate question - the nature of classes. As we know, everything in Smalltalk is an instance of a class and this applies even to classes themselves. We will see that every class is an instance of a metaclass and that the hierarchy of metaclasses parallels the hierarchy of classes. Metaclasses are, of course, also objects and therefore instances of other, perhaps even more abstract classes. It turns out that the reality is simpler and that each metaclass is an instance of one special class called, rather predictably, Metaclass. Since Metaclass is a class, it is itself an instance of the metaclass Metaclass - according to the above general rule. The metaclass of Metaclass is, naturally, an instance of Metaclass like all metaclasses. Since the metaclass hierarchy parallels the corresponding class hierarchy, all metaclasses are subclasses of the metaclass of class Object. A few additional classes that define the shared behavior of classes complete the picture.

Classes and metaclasses are not only an elegant solution to the problem of making everything an instance of a class. The structure also has some very useful and important implications that are essential for class creation, editing, and gathering of information about classes. Programming based on the class-metaclass structure is called metaprogramming and we will give several examples of its use.

## A 4.1 Classes and Metaclasses

All Smalltalk objects are instances of classes: A Smalltalk character is an instance of Character, an array is an instance of Array, a browser is an instance of Browser, and so on. To be fully consistent with this principle, all classes should themselves be instances of other classes - and indeed they are. These higher level classes are called *metaclasses* and their names are identical to the names of the classes themselves. Class Array is thus an instance of metaclass Array, class Character is an instance of metaclass Character, and so on. To avoid proliferation of names and confusion, metaclasses don't have separate names and can only be accessed by sending the class message to the class. As an example, to access the metaclass of Array execute expression

```
Array class
```

When executed with *inspect*, this opens the inspector in Figure A 4.1.

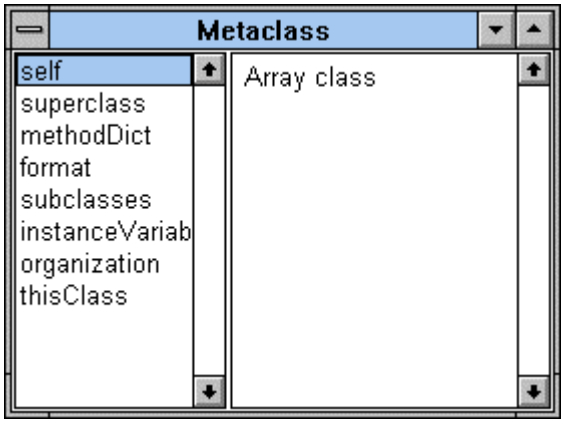


Figure A 4.1. Inspector on Array class - the metaclass of Array.

The hierarchy of metaclasses parallels the hierarchy of classes, and when you create a new class, its metaclass is automatically created with it (Figure A 4.2).

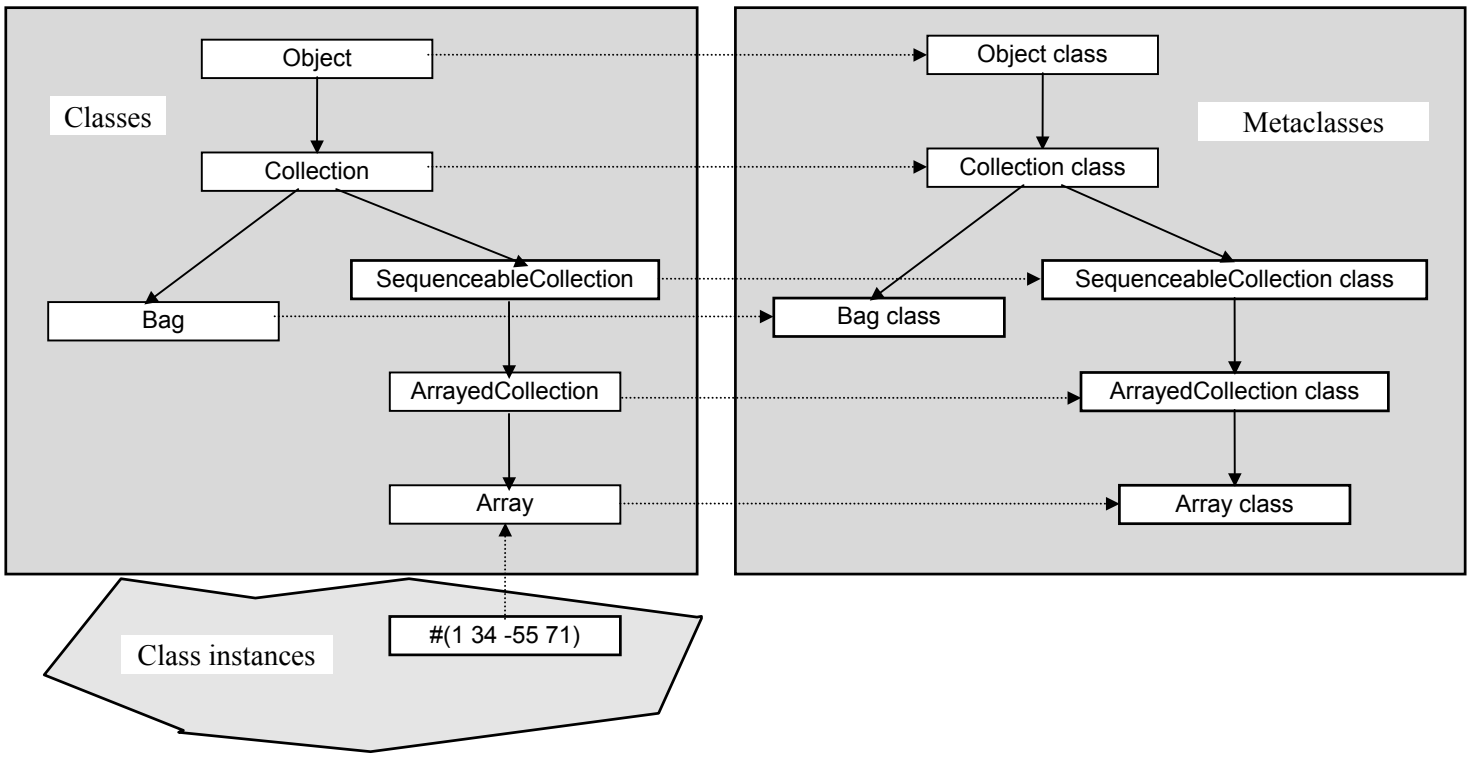


Figure A 4.2. Class hierarchy is paralleled by metaclass hierarchy. The ⋯▸ relationship means 'is instance of' whereas ⟶ represent subclassing.

These principles immediately invite several questions such as - what is the class of a metaclass, how are metaclasses different from classes, how does the existence of a parallel hierarchy tree relate to the fact that the superclass of every class is class Object, do metaclasses have any methods and instance or class variables, and finally - what is the use of this elaborate structure.

Main lessons learned:

- Every class is a single instance of its metaclass.
- A class and its metaclass have the same name.
- A class is accessed by its name but its metaclass is accessed by sending message class to the class.
- Metaclasses are not accessible through the browser but can be accessed by an inspector.
- When a class is created, its metaclass is automatically created with it.
- Metaclasses are arranged in an inheritance tree that parallels the inheritance tree of classes.

Exercises

1. Inspect the components of the metaclass of Character and the metaclass of Array. Make preliminary comments on the variables.

**A 4.2. What is the complete class hierarchy?**

If an 'ordinary' class is an instance of its metaclass, what is a metaclass? The answer is easy to find - we only need to ask each metaclass what is its class as in

```
Integer class class      "Returns Metaclass"
Array class class        "Returns Metaclass"
```

We conclude that all metaclasses are instances of class Metaclass (Figure A 4.3).

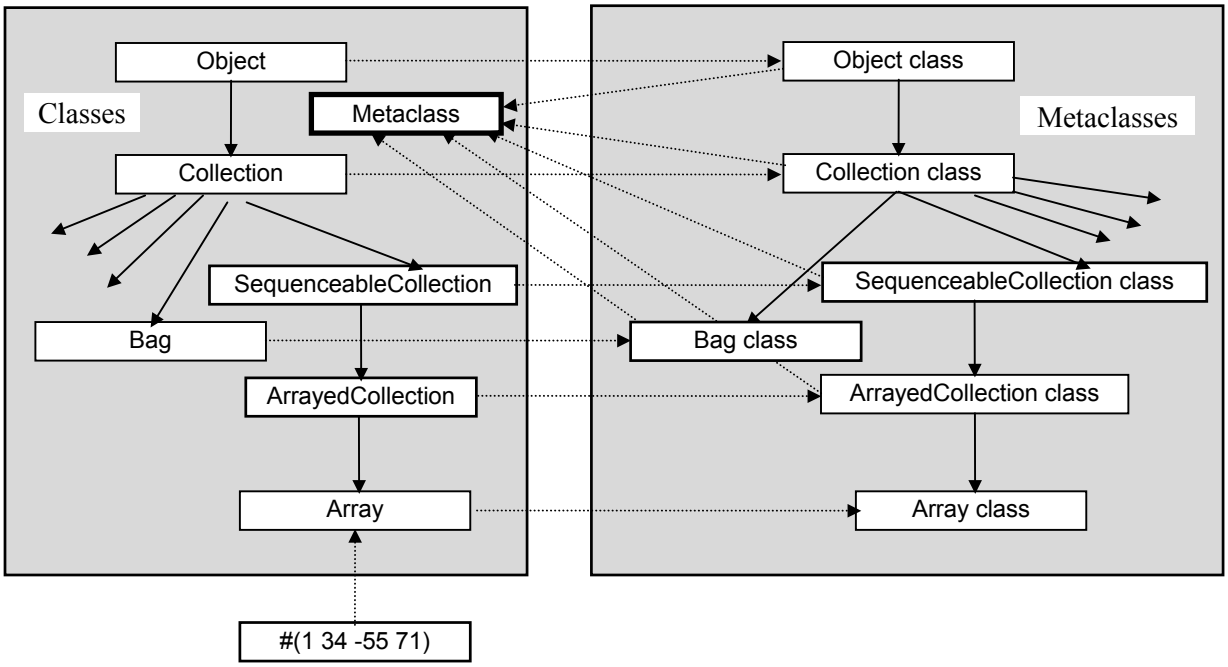


Figure A 4.3. Every metaclass is an instance of class Metaclass.

But if this is so, then what is the class of Metaclass? Since Metaclass is a class just like Array or Integer, the answer, of course, is that Metaclass is an instance of Metaclass class. And what is the metaclass of Metaclass class? The answer is again simple - since Metaclass class is a metaclass and since the metaclass of every class is Metaclass, the metaclass of Metaclass class is Metaclass (Figure A 4.4). This is very natural when you realize that everything is governed by the following two rules:

1. The class of class X is its metaclass X class.
2. Every metaclass is an instance of class Metaclass.

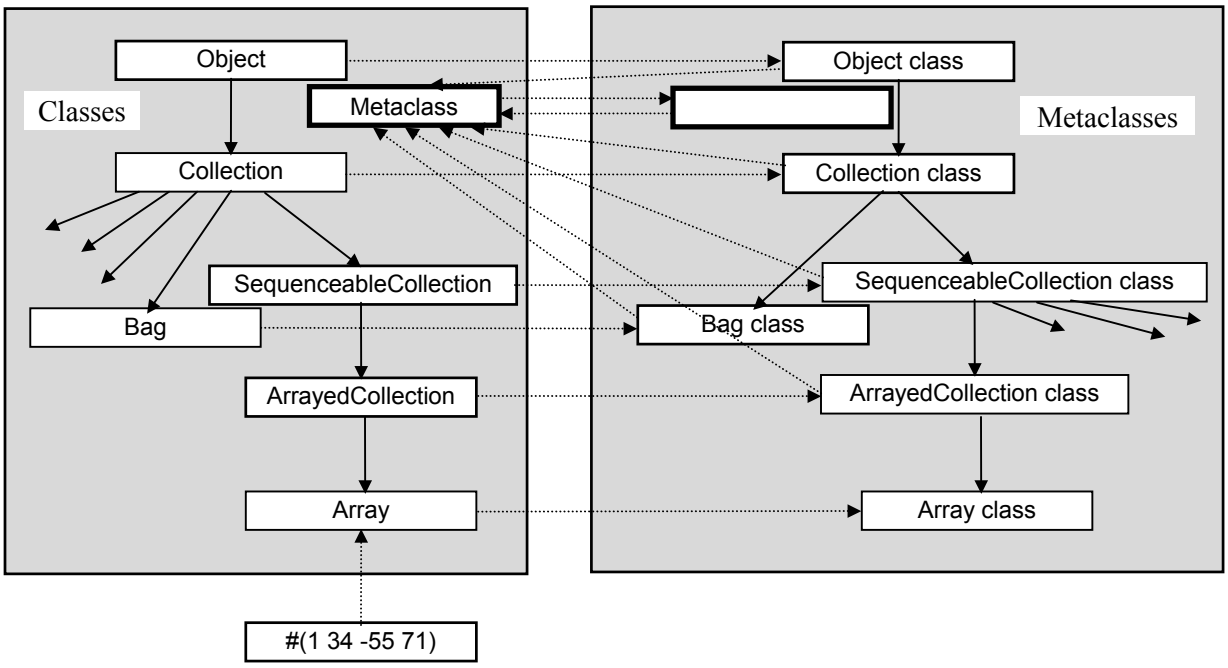


Figure A 4.4. The metaclass of Metaclass is Metaclass class. Metaclass class is an instance of Metaclass.

Up to this point, we have been concentrating on metaclasses and neglected the subclass relationship. As an example, we have not said what is the superclass of Object class - the head of our metaclass hierarchy. To find out, execute

```
Object class superclass
```

The answer is that the superclass of Object class is Class (Figure A 4.5). Class Class itself is, of course a subclass of Object - like every other class. And the metaclass of Class is Class class.

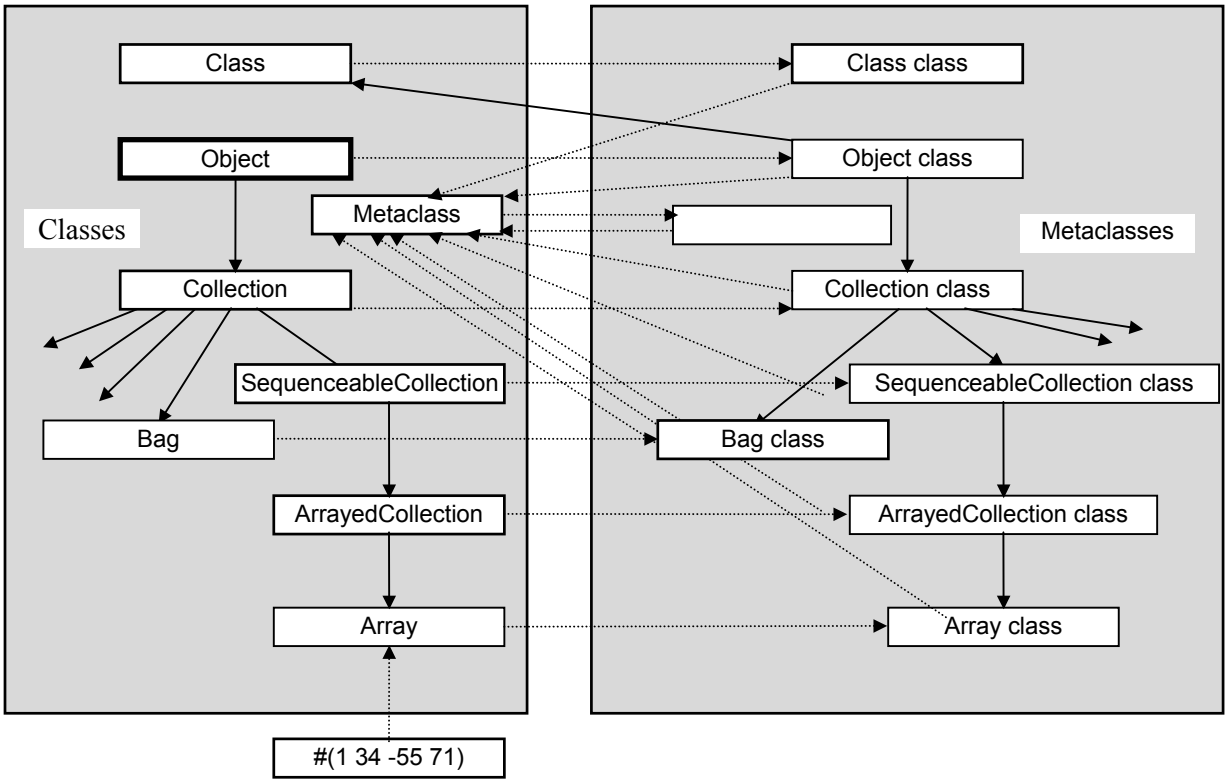


Figure A 4.5. The superclass of metaclass Object class is class Class.

However, our picture is incomplete because there are two additional classes called Behavior and ClassDescription between Class and Object. This last refinement is shown in Figure A 4.6.

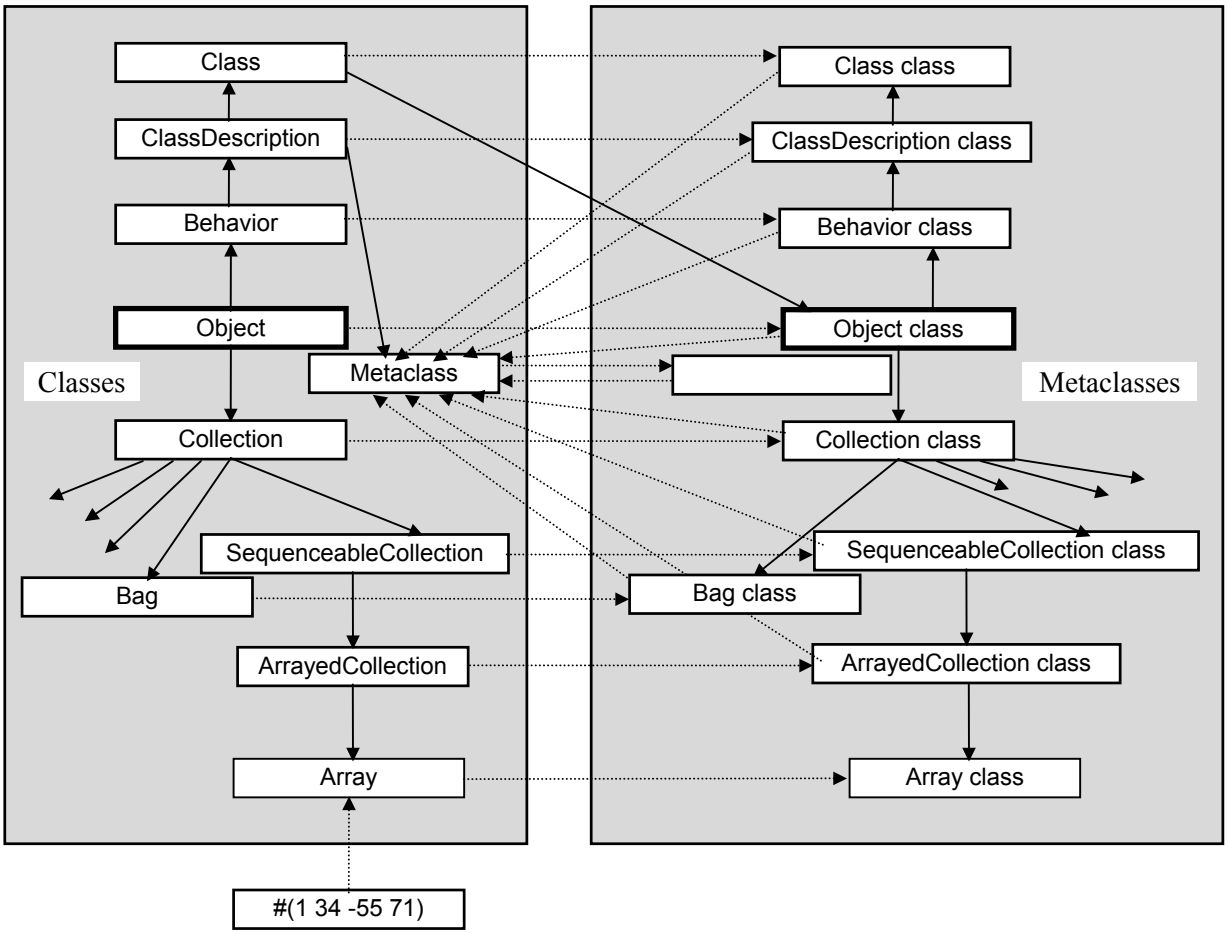


Figure A 4.6. Complete hierarchy of Smalltalk classes. Object is the superclass of all classes and all metaclasses are instances of Metaclass.

Main lessons learned:

- Every class is an instance of its metaclass and every metaclass is an instance of class Metaclass.
- The superclasses of Metaclass are Class, ClassDescription, Behavior, and Object, in this order.
- Object remains at the top of the class hierarchy, including metaclasses.

Exercises

1. Use the System Browser to find all superclasses of metaclass Date class.

### A 4.3 What are the main properties of metaclasses?

Unlike most 'ordinary' classes, each metaclass has only one instance - its class. As an example, the only instance of metaclass Array class is class Array. Also, messages that we have so far classified as class messages are, in fact, instance messages of the metaclass. As an example, all class messages of class Array are instance messages of metaclass Array class., and all class messages of Object are instance messages of Object class:

```
Object class selectors asSortedCollection      "Returns selectors of all class messages of Object."
Object selectors asSortedCollection "Returns selectors of all instance messages of Object."
```

This arrangement is logically related to the general rule of message execution:

*When a message is sent to an object, the search for its definition begins in the class of the receiver.*

As an example, when we send 13 factorial, the search for the declaration of factorial starts in class SmallInteger, the class of object A 4. It is thus quite consistent that when we send, for example, String new the search for the declaration of new starts in the class of String, in other words, in metaclass String class. And if the declaration of the message is not found there, the search continues in the superclass of String class and so on.

Besides being responsible for class methods, metaclasses inherit all their behavior from their superclasses Class, ClassDescription, Behavior, and Metaclass, All grouped in category Kernel-Classes. We will examine the most important of these shared behaviors starting with the simplest and most general of these classes - Behavior - and then explore the functionality that its subclasses add to it. First, however, here is again the relevant part of the hierarchy:

```
Behavior ('superclass' 'methodDict' 'format' 'subclasses')
        ClassDescription ('instanceVariables' 'organization')
                Class ('name' 'classPool' 'sharedPools')
                        ... all metaclasses ...
                Metaclass ('thisClass')
```

and its graphical representation in Figure A 4.7.

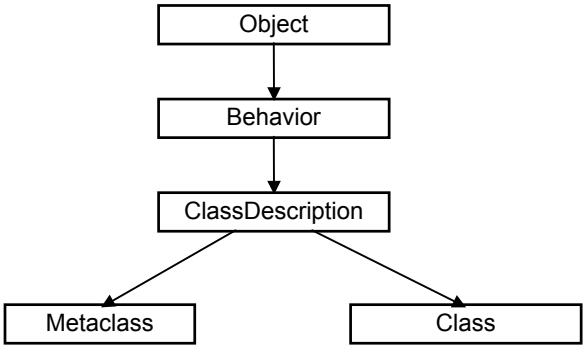


Figure A 4.7. Metaclass-defining classes.

Main lessons learned:

- Class methods of a class are instance methods of its metaclass.

Exercises

1. We have seen that class messages of a class are instance messages of its metaclass. What is the relationship of a class, its metaclass, and instance, class, and class instance variables?

### A 4.4 Class Behavior

Class Behavior defines most of the functionality shared by classes and metaclasses. It is responsible for the basic instance creation protocol, the protocol that finds the subclasses and superclasses of a class, access to methods defined in a class, and the kind of the class (more on this later). It also provides the mechanism for finding all instances of a class. Class Behavior does *not* have information that would allow a class to know about the names of its instance variables and message protocols, its class comment, and other class-related naming information; this information is the responsibility of Behavior's subclass ClassDescription.

To be able to implement its functionality, Behavior has instance variables superclass, methodDict, format, and subclasses. Variable format contains information about the storage layout of the class. methodDict stores associations *selector -> compiled method*. As an example, when you inspect String (a class, and thus subclass of Behavior), you will find the inherited instance variable methodDict among its instance variables (Figure A 4.8) and when you inspect it, you will find that its keys are selectors of all methods defined in String. When you inspect the value of one of these selectors, you will find that it is the compiled method of the corresponding selector - an instance of CompiledMethod.

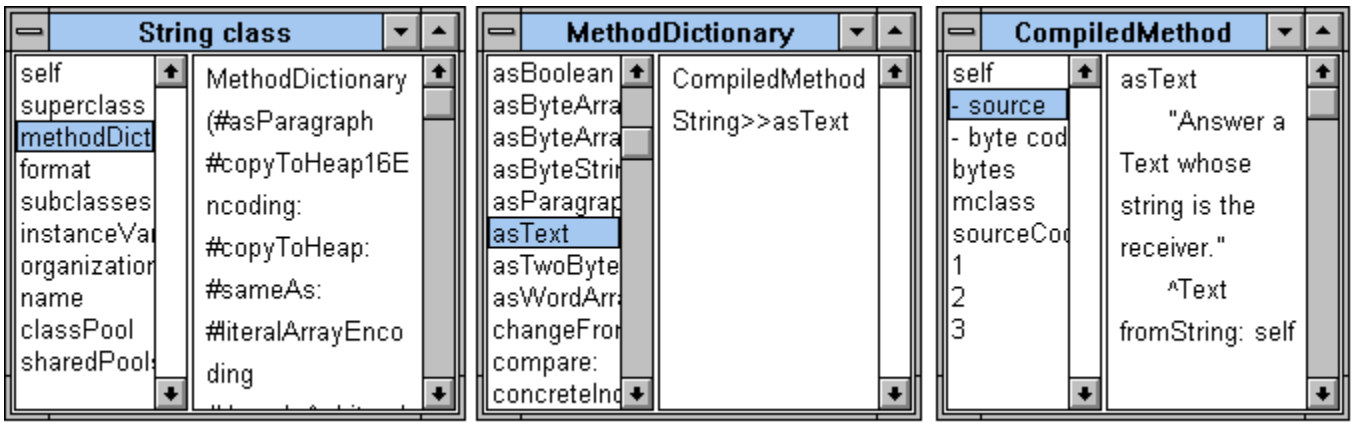


Figure A 4.8. Inspectors revealing instance variable methodDictionary inherited from class Behavior, and its contents.

Class Behavior does not have any class methods and its instance methods are understood by all classes. The following are examples of the most interesting Behavior responsibilities and we encourage you to browse the class for more details.

Instance creation

The most common messages originating in class Behavior are new and new:. These are the default creation messages inherited (and sometimes redefined) throughout the class hierarchy. Methods basicNew and basicNew: are their duplicates that should not be redeclared in any class.

Instances, superclasses, subclasses, and enumeration over them

Numerous behaviors are available in this category and we will illustrate a few of them on examples.
To find how many instances of Array currently exist in the environment, execute

```
Array allInstances size
```

To find all subclasses of String execute

```
String subclasses          "Returns #(ByteEncodedString TwoByteString Symbol GapString)."
```

To find which of all existing arrays has the largest size, evaluate

```
| size |
size := 0.
Array allInstancesDo: [:anArray| size := size max: anArray size].
size
```

or more simply

```
Array allInstances inject: 0 into: [:max :anArray| max max: anArray size]
```

To find all superclasses of a class, execute allSuperclasses as in

```
Set allSuperclasses          "OrderedCollection (Collection Object)."
```

Instance variable format contains a code that describes what kind of class this is. The protocol based on format makes it possible to ask a class whether it has fixed size (classes that have only named variables) or variable size (collections represented internally as indexed elements) and, in the case of a variable size class, whether its variables are stored as eight-bit or 16-bit quantities. As an example, all variable size classes can be obtained by

```
Object withAllSubclasses select: [:aClass| aClass isVariable]
```

which returns

```
OrderedCollection (AnnotatedMethod Array BinaryStorageBytes BOSSBytes BOSSReaderMap ByteArray
ByteEncodedString ByteString ByteSymbol CCompositeType CCompoundType CEnumerationType
CEnvironment CompiledBlock CompiledCode CompiledMethod CProcedureType DependentsCollection
Dictionary Double ExternalDictionary ExternalMethod ExternalRegistry Float FontDescriptionBundle
GraphicsHandle HandleRegistry HandlerList IdentityDictionary IdentitySet ISO8859L1String LargeArray
LargeInteger LargeNegativeInteger LargePositiveInteger LargeWordArray LensLinkedDictionary
LensObjectRegistry LensProtectedLinkedDictionary LensRegistry LensWeakCollection LensWeakRegistry
LinkedOrderedCollection LinkedWeakAssociationDictionary MacString MarkedMethod MethodDictionary
MultiValueDictionary ObjectMemory ObjectRegistry OrderedCollection OS2String PoolDictionary
PropertyListDictionary ScannerTable SegmentedCollection Set SignalCollection SortedCollection
SortedCollectionWithPolicy SPActiveLines SPSortedLines SystemDictionary TwoByteString TwoByteSymbol
UninterpretedBytes WeakArray WeakAssociationDictionary WeakDictionary WordArray)
```

As another example of Behavior-defined behavior, statement

```
(Smalltalk select: [:id| id isBehavior and: [id isBits]])
```

returns all elements of Smalltalk that are classes (message isBehavior) and whose instances are stored directly as bits rather than accessed by pointers. It returns

```
SystemDictionary keys (#ByteString #ByteSymbol #GraphicsHandle #ISO8859L1String #Character #Double
#Float #OS2String #LargeInteger #LargePositiveInteger #TwoByteString #ByteArray #BOSSBytes
#WordArray #LargeNegativeInteger #UninterpretedBytes #ByteEncodedString #TwoByteSymbol
#SmallInteger #BinaryStorageBytes #MacString)
```

Expression

```
Object withAllSubclasses select: [:aClass| aClass isBits]
```

would produce the same result. Access to instances of all other classes is by pointers.

Functionality based on access to class hierarchy and method dictionary

Behavior provides several methods for adding new selectors to the method dictionary or removing existing ones, finding method selectors, printing hierarchies, and so on. As an example,

```
Array printHierarchy
```

returns a string containing the hierarchy of class Array, the same as provided by the browser. Expression

```
Point allSelectors
```

returns a set containing selectors of all messages understood by instances of Point, in other words, names of all instance methods declared in class Point or its superclasses. To find all *class* messages understood by Point, we must ask *metaclass* Point class as in

```
Point class allSelectors
```

Behavior also provides access to method source code. As an example,

```
Collection sourceCodeAt: #contains:
```

returns the text of the definition of instance method contains:

Similarly, the following expression returns the definition of the *class* method with:

```
Collection class sourceCodeAt: #with: 'with: anObject
```

Other protocols

Among other interesting messages, class Behavior provides methods for compiling source code and inserting it into the method dictionary, and a method for decompiling byte codes to produce code with artificially created temporary variables; this method is used by the Debugger when it does not have access to the source code of the selected message. All these methods use classes Compiler and Decompiler to do the actual work. The implementation is very flexible and uses a compiler and decompiler retrieved by an accessing methods so that a different compiler or decompiler may be written and used if desired.

Main lessons learned:

- Behavior is an abstract class that factors out much but not all of the behavior shared by classes and metaclasses.
- Behavior introduces information about a class's superclass and subclasses, its methods, and its format. It does not provide information about instance variables, comment, and protocols.
- The functionality of Behavior is inherited by all classes.

Exercises

1. The all-important new and new: methods are defined in the instance protocol of Behavior and redefined in several other classes. However, new and new: in Behavior are instance methods whereas their use obviously is as a class methods. As an example, we use Array new: 5, sending the new: message to class Array. Moreover, the new definitions in those classes that redefine new and new: are on the class side of the protocol. This seems to violate the fact that instance methods are inherited on the instance side and class methods on the class side. Explain.
2. Write a code fragment to find all classes that implement hash or = but not both.
3. Write method find: aString to ask the user for a string and find all its occurrences in the source code of all methods declared in the receiver class. As an example, executing Object find: 'assign' should return an ordered collection with selectors of all Object instance methods whose source code contains the string 'assign'.
4. Write a code fragment to find all methods in category Collection-Streams that disable inherited methods (in other words, all methods in category Collection-Streams that use shouldNotImplement). Display the names of the methods and their classes in the Transcript.
5. Add a new <operate> command called *debug* to open the Debugger and start execution of the selected code. In other words, executing *debug* should work just like adding self halt at the beginning of the selection and executing it with *do it*.
6. How many class methods are unary, binary, keyword methods?

### A 4.5 Class ClassDescription

Class ClassDescription is the only immediate subclass of Behavior. It provides facilities for naming classes, method protocols, and instance variables, and for class comments. The essence of its comment is as follows:

```
ClassDescription adds a number of facilities to basic Behavior:
                - named instance variables
                - category organization for methods
                - most of the mechanism for fileOut
ClassDescription is an abstract class: its facilities are intended for inheritance by the two subclasses, Class
and Metaclass.
Instance Variables:
                instanceVariables <Array of: String> names of instance fields
                organization                <ClassOrganizer> organization of message protocol
```

As the comment states, ClassDescription gathers all behavior shared by classes and metaclasses that is not defined in Behavior. Its definition contains many methods:

```
ClassDescription allSelectors size
```

returns 343, meaning that 343 methods are defined in ClassDescription. We leave it to you to discover how to find that there are 12 instance method protocols. The most interesting of them are briefly described below.

Accessing

This protocol provides access to the comment and the standard comment template via messages such as comment, comment:, and commentTemplate.

Copying

This protocol makes it possible to copy a method, a protocol, or all protocols from one place to another. As an example,

```
Test copyCategory: #accessing from: Collection
```

copies the accessing protocol from class Collection to class Test. ClassDescription uses the term *category* to refer both to class categories and to method protocols.

Printing

The printing protocol contains methods that return strings with instance variable names, the class definition, and so on. As an example,

```
Explainer instanceVariablesString
```

returns the string

```
'class selector instance context methodText '
```

Instance variables

This protocol makes it possible to add new instance variables and remove existing ones, and find in which position a named instance variable is stored in the sequence representing a class's instance variables, taking into considerations inherited instance variables. This protocol is needed to understand the byte codes produced by method compilation. As an example,

```
Window instVarIndexFor: 'sensor'
```

returns 7 meaning that references to the 7th variable in messages whose receiver is an instance of Window refer to instance variable sensor.

Organization

This protocol contains methods dealing with information about method protocols stored in instance variable organization, an instance of ClassOrganizer. As an example,

```
Array category
```

returns #'Collections-Arrayed', the name of the category containing class Array and

```
View organization categories
```

returns an array containing the names of the three protocols defined in class View:

```
#(#'display box accessing' #'controller accessing' #private)
```

Since ClassDescription has access to organization, it can also answer questions such as 'Which category (protocol) contains the method with a given selector?' As an example, the following expression returns the name of the protocol containing the class method with:

```
Collection class whichCategoryIncludesSelector: #with:          "Returns #'instance creation'."
```

File-out

This protocol supports filing out of the source code of the class.

ClassDescription has two subclasses - Class and Metaclass - which define specialized behavior of classes and metaclasses respectively. We will outline their functionality in the next section.

Main lessons learned:

- ClassDescription is a subclass of Behavior whose most visible responsibilities include access to instance variable names, class comment, and organization of instance methods into protocols.
- The combination of Object, Behavior and ClassDescription defines all behaviors shared by classes and metaclasses.

Exercises

1. When you are creating a new test version of a class, you might want to copy all protocols and methods from an existing class to the new class. Write a code fragment to do this.
2. Write a code fragment to find all unreachable methods in a class. (A method is unreachable if it is either never sent or if it is sent by a method that is itself unreachable.)
3. Write a code fragment that lists the names of instance protocols of a selected class in the Transcript. Repeat for class protocols.
4. Modify the default comment that appears in the browser for an uncommented class. The new template should automatically display a list of all instance and class variables following the current default comment.
5. Modify the default comment further to list methods that are subclass responsibility.
6. Write a short summary of the logic of the separation of class and metaclass behaviors into Behavior and ClassDescription.
7. Examples presented in this section show the importance of class ClassOrganizer. Write a short summary of its most important features.
8. The file-out operation is implemented in ClassDescription. Which class is responsible for file-in?

### A 4.6 Class Class

Class has the following comment:

Instances of class Class describe the representation and behavior of objects. Class adds more comprehensive programming support facilities to the basic attributes of Behavior and the descriptive facilities of ClassDescription. An example is accessing shared (pool) variables.

Instance Variables:

| | |
|---|---|
| name | <Symbol> name of class for printing and global reference |
| classPool | <PoolDictionary \| nil> of variables common to all instances (i.e., class variables) |
| sharedPools | <Collection of: Dictionary> access to other shared variables |

Two other examples of behaviors that classes need but metaclasses don't are renaming a class (the name of the metaclass is derived from the name of the class) and subclass creation. Class also defines behaviors that classes share with metaclasses but implement differently. These include, for example, adding and removing instance and class variable names, and file out.

Most programmers will never use Class behaviors except for protocol subclass creation which contains methods for defining new classes. Even these methods are rarely used explicitly because new classes are normally defined from the browser template:

```
Paint subclass: #DevicePaint
        instanceVariableNames: 'device devicePaint paintBasis '
```

```
classVariableNames: ''
poolDictionaries: ''
category: 'Graphics-Support'
```

Accepting the edited template simply sends message subclass: instanceVariableNames: classVariableNames: poolDictionaries: category: inherited from Class to class Paint and creates subclass DevicePaint. Classes created with this message only have instance variables accessed by word-size pointers. Most classes have this format and respond true to isFixed from class Behavior.
The definition of this subclass creation message is worth a look:

```
subclass: t instanceVariableNames: f classVariableNames: d poolDictionaries: s category: cat
"This is the standard initialization message for creating a new class as a subclass of an existing class (the receiver)."
    | approved |
    “Check whether arguments have proper form, such as class variables capitalized.”
    approved := SystemUtils
                    validateClassName: t
                    confirm: [:msg :nm | Dialog confirm: msg]
                    warn: [:msg | Dialog warn: msg].
    approved == nil ifTrue: [^nil].
    “Arguments are OK, ask classBuilder to build the class.”
    ^self classBuilder
        “First calculate all necessary parameters.”
        superclass: self;
        environment: self environment;
        className: approved;
        instVarString: f;
        classVarString: d;
        poolString: (self computeFullPoolString: s);
        category: cat;
        beFixed;
        “Now create new class or revise existing class.”
        reviseSystem
```

The most interesting part is the last line which creates a new class or revises the class if it already exists. Its definition is

```
reviseSystem
"Mutate the system, based on whether the class already exists or not."
    ^self needsNewClass
        ifTrue:    [self createNewSubclass]
        ifFalse:   [self modifyExistingClass]
```

If the class is new, message createNewSubclass is executed. Its definition in ClassBuilder is as follows:

```
createNewSubclass
"The class does not exist--create a new class-metaclass pair."
    | newMeta |
    self runValidationChecksForNewClass.        “Check that class name is not already in use, etc.”
    newMeta := self metaclassClass new.         “Create new instance of Metaclass and initialize it.”
    newMeta assignSuperclass: (self classOf: superclass).
    newMeta methodDictionary: MethodDictionary new.
    newMeta setInstanceFormat: (self classOf: superclass) format.
    class := newMeta new.                       “Ask the metaclass to create the class.”
    class assignSuperclass: superclass.“Initialize instance variables of the class.”
    class methodDictionary: MethodDictionary new.
    class setInstanceFormat: self computeFormat.
    self setStructureOf: class.
    class setName: className.
```

```
self register: class inPlaceOf: nil.
self changeMicroState.      "Aspects such as category and pools."
^self logNew: class         "Record new class so that changes will be saved on exit."
```

To define a class that does not fall into the usual pattern of fixed-size class with named instance variables, edit the Browser template into one of the following forms

*variableByteSubclass*: instanceVariableNames: classVariableNames: poolDictionaries: category:

*variableSubclass* instanceVariableNames: classVariableNames: poolDictionaries: category:

edit it in the usual way, and *accept*.

If the class category given as the argument does not exist, the message creates it.

Example: Finding which symbols in Smalltalk do not represent classes

*Problem:* The system dictionary Smalltalk includes all global objects. They are mostly classes but also objects of other kinds. This example explores what these other objects are.
*Solution:* Since all class objects are subclasses of Class, we can use the following expression to extract the non-Class objects from the Smalltalk dictionary:

```
Smalltalk reject: [:value| value isKindOf: Class]
```

This returns the following ten or so elements out of the 1200 or so associations in Smalltalk

```
SystemDictionary keys (#SymbolicPaintConstants #Processor #X11InputManagerDictionary #TextConstants
#Transcript #IOConstants #OpcodePool #Smalltalk #InputManagerDictionary #Undeclared
#ScheduledControllers #NullInputManagerDictionary )
```

and includes some familiar global variables such as Transcript, Processor, and ScheduledControllers, and some pool dictionaries such as TextConstants and Undeclared. Undeclared contains all variables that the user did not declare when compiling a program and specified that they should be left undeclared when responding to the compiler query. We leave it to you to find what all these objects are.

Main lessons learned:

- Class inherits the behaviors of Behavior and ClassDescription and adds knowledge of class name and access to class variables and shared pools. It also provides facilities for creating new classes and controlling their format (fixed size, variable size, and others).

Exercises

1. Why is the name of a class defined in Class rather than Behavior or ClassDescription?
2. Trace the creation of a new class and write a short summary.
3. What happens when you redefine and accept an existing class?
4. What is the use of the global pool dictionary Undeclared?

### A 4.7 Class Metaclass

Class Metaclass is the equivalent of Class for metaclasses. It has the following comment:

Metaclasses add instance-specific behavior to various classes in the system. This typically includes messages for initializing class variables and instance creation messages particular to that class. Metaclass has only one instance. A metaclass shares the class variables of its instance.
Instance Variables:
thisClass <Class> the chief instance of the receiver, which the receiver describes

Metaclass mainly redefines behaviors which are different from Class behaviors. One example is definitionMessage which is responsible for providing the text displayed in the Browser for the definition of a class in the instance view, or its metaclass in the class view. Another example is the new message which is automatically invoked to create a metaclass during the creation of a new class. Yet another example is the enumeration message allSubclassesDo: which must enumerate over metaclasses, unlike its class version defined in Behavior.

Main lessons learned:

- Just as Class defines and inherits or redefines behavior needed by classes, Metaclass defines, inherits, and redefines all behavior common to metaclasses.

Exercises

1. As we know, class objects are stored in the Smalltalk SystemDictionary. Where are metaclasses stored?
2. We have now covered the principles of the whole class hierarchy except for its essential class - Object. What is the superclass of Object and what implications does it have?

### A 4.7 Metaprogramming - Is this magic useful?

After the first section of this chapter, you may have been wondering whether the subtle architecture of classes provides anything beyond a proof that an environment consisting only of objects can indeed be constructed. By now, it should be obvious that metaclasses and the whole structure of the class hierarchy have important practical roles. First, classes such as Behavior and ClassDescription are heavily involved in the creation of new classes and modification of existing ones. Second, these classes provide access to very valuable information about classes and their instances, and environment tools such as the Browser very much depend on them. Third, programming based on kernel classes and other classes such as compiler classes can be used to extend the programming environment (for example, by adding new browsers), the environment of a running application (for example, by adding new methods or classes at run time), or even the syntax or semantics of the Smalltalk language. This form of programming is called *metaprogramming*[1] and we will now demonstrate it on several examples.

Example 1: A creation method to create a class *and its accessing methods*[2]

The Browser's class definition template makes it possible to create a new class with variables. Most instance variables have accessing methods and these methods must be created manually. Since Smalltalk provides access to its compiler and all class information, it is possible to create tools to create accessing methods automatically.
*Problem:* Define a new class creation message that creates a class and all accessing methods for its instance variables. The usage of the method should be as in

[1] Languages that allow metaprogramming are called *reflective*. Most languages are not reflective.
[2] This example is based on the classic 'blue book' by Goldberg and Robson, a Smalltalk bible that should be read by anybody seriously interested in Smalltalk.

```
Object subclassWithAccessors: NewClass
        instanceVariableNames: 'x y z'
        classVariableNames: 'X Y Z'
        poolDictionaries: 'Pool1 Pool2'
        category: 'Experimental'
```

in other words, the same as the standard Browser template

```
Object subclass: NewClass
        instanceVariableNames: 'x y z'
        classVariableNames: 'X Y Z'
        poolDictionaries: 'Pool1 Pool2'
        category: 'Experimental'
```

except that the first keyword is different.
*Solution:* The algorithm is simple:

1. Create the class using the existing creation message.
2. Generate the source code of accessing methods and ask the class to format it.
3. Compile the methods. This will automatically insert them into the method dictionary.

The implementation of this algorithm (a method to be added to class Class) is relatively simple:

```
subclassWithAccessors: t instanceVariableNames: f classVariableNames: d poolDictionaries: s
category: cat
"Create new class and its accessing methods."
        | newClass |
        "Compile class with its variables using existing creation message."
        newClass := self  subclass: t
                          instanceVariableNames: f
                          classVariableNames: d
                          poolDictionaries: s
                          category: cat.
        "Formulate, format, and compile accessing methods for all instance variables."
        newClass instVarNames
                do:       [:aName | "Enumeration over all instance variables."
                          | formattedText |
                          "Construct the text, format it, and compile it into the accessing protocol."
                          "First the 'get' method."
                          formattedText := Compiler new
                                                format: aName , '^ ' , aName
                                                in: newClass
                                                notifying: nil.
                          "Now the 'set' method."
                          newClass compile: formattedText classified: #accessing.
                          formattedText := Compiler new
                                                format: aName , ':  argument ' , aName , ' :=  argument'
                                                in: newClass
                                                notifying: nil.
                          newClass compile: formattedText classified: #accessing]
```

Main lessons learned:

- A programming environment is called reflective if it provides information about its implementation and allows the programmer modify it.
- Programming based on reflectivity is called metaprogramming.

Exercises

1. Test that the class creation method developed in this section works.
2. Add a new command called *accessing* to the <operate> menu of the text view of the Browser. Its activation should open a window containing two parallel multiple selection lists containing all instance variables of the currently selected class and a *Compile* button. The left list is for creating 'get' methods, the right list is for creating 'set' methods. Selecting variable names in the two lists and clicking *Compile* creates accessing methods for all selections.

### A 4.8 Enhanced Workspace - Another example of metaprogramming

The functionality of the Workspace would be greatly enhanced if it could hold on to objects created by program fragments, in other words, if results of evaluating selected messages could persist for the duration of the existence of the workspace. Such objects could then be reused and we would not have to recreate them every time. We will show how this idea could be implemented, leaving a more complete implementation as a project.

*Problem:* Implement an enhanced workspace with an additional <operate> menu command *do and keep* that will execute just like *do it* but save the resulting object as a persistent part of the workspace and bind it to an automatically generated variable. An additional pop up command called *variables* will open a multiple choice menu on all variables created in this way, and allow the user to select and inspect one. Within the scope of the workspace, these variables will be treated as global.

*Solution:* We will create the Extended Workspace as a new an application, an instance of a new class called ExtendedWorkspace, a subclass of ApplicationModel. In its design, we will consider two basic scenarios: executing a code fragment to create a workspace-persistent object, and executing a code fragment containing a reference to such an object via the variable name associated with it.

Scenario 1: Creating a workspace-persistent object

Since there is no limit on the number of objects that the user may want to add to the extended workspace, the obvious approach is to store them in a dictionary whose key is the name of the variable and whose value is the object. However, this implementation would force us to refer to the object as a value in a dictionary as in

```
worskpaceVars at: varname
```

rather than as a variable as in

```
x := y factorial
```

To be able to access the persistent objects as named variables, we can use one of at least the following two approaches: Create the new variable as a global variable (adding it to Smalltalk), or add it as a named instance variable to the ExtendedWorkspace at run time.

Using the first approach, we could automatically generate numbered variables, such as Var0001, Var0002, and so on, and delete them when the user closes the workspace. This approach is relatively simple but it does not restrict the variable to one workspace; we leave it as an exercise.

The other approach is to add a new named instance variable to the ExtendedWorkspace class every time when the user requests a new persistent object; this is the strategy that we will use here. This approach requires that we modify the definition of the extended workspace class at run time every time when the user adds a new workspace-persistent object. Since there may be any number of workspaces, each of them with a different number of variables, we will do this by treating ExtendedWorkspace as an abstract class, creating a new numbered subclass called, for example, ConcreteExtendedWorkspace7 when we open a new workspace, and adding a new instance variable to this class whenever we add a new persistent object to this workspace. This way, different workspaces will be instances of different classes and their 'persistency' variables will be mutually unknown. The new class will be automatically deleted when the user closes the workspace.

This principle can be implemented as follows: When the user creates a new workspace-persistent object, 'mutate' the corresponding ConcreteExtendedWorkspace class by adding a new instance variable to it, and bind the object to this variable. This procedure is relatively simple as it only requires one message from Class to perform the mutation, one message to add an accessing method for the new variable, and one message to access the variable and assign the new value to it.

To test the idea of mutation, assume the existence of a test class called MutationTest, a subclass of Object, with one instance variable called var1 and its accessing message var1:. Our program will assign a value to var1 using this accessing message, add new variable var2 and its accessing method, assign a value to this variable, and inspect the result:

```
| x |
"Create an instance of the test class and assign a value to var1."
x := MutationTest new.
x var1: 10.
"Mutate the class by adding a new variable."
MutationTest addInstVarName: 'var2'.
"Add its accessing method."
MutationTest      compile: 'var2: anObject ^var2 := anObject'
                  classified: 'accessing'
                  notifying: nil.
"Assign value to the new instance variable."
x perform: #var2: with: 20.
"x var2: 20 would not work, var: does not exist as this code fragment is being compiled."
x inspect
```

Everything works as we hoped: The x object has both var1 and var2 variables and their values are as expected. You can also open the browser on MutationTest to see that the new definition of MutationTest captures the changes made by this program.

The principle thus works and we can apply it to our problem. In our implementation of ExtendedWorkspace, the value assigned to the new variable will be the object that we want to make persistent and the mutation will be performed when the user executes, for example,

```
| x y |
x := 130 * 60 cos.
y := 130 * 60 sin.
^x @ y
```

with *do and keep*. This will create a new workspace-defined persistent variable and store the Point in it.

We now have the solution to the problem of creating workspace-persistent objects and their associated variables. Our next problem is how will these variables be accessed when we execute a program in the workspace.

Scenario 2: Executing a code fragment containing a reference to a persistent object

Assume that the user created two persistent objects and assigned them to automatically created variables var1 and var2. Assume that the user now executes the following fragment in the extended workspace using, for example, *do it*.

```
| radius |
radius := (var1 squared) + (var2 squared)
...
```

To make this work as we wish, we must first understand how *do it* works. When we examine implementors of doIt, we find the following definition in class ParagraphEditor:

```
doIt
"Evaluate the current text selection as an expression."
        self selectionStartIndex = self selectionStopIndex
```

```
            ifTrue: [^self].       "If the current selection is empty, just return."
        self class compilationErrorSignal
            handle: [:ex | ex returnWith: nil]
            do: [self evaluateSelection]
```

The evaluation of the selection is obviously done by evaluateSelection whose definition

```
evaluateSelection
"Evaluate the current text selection as an expression"
        | result |
        result := Cursor execute showWhile:
            [self doItReceiver class evaluatorClass new
                    evaluate: self selectionAsStream
                    in: self doItContext
                    receiver: self doItReceiver
                    notifying: self
                    ifFail: [self class compilationErrorSignal raise]].
        self doItValue: result.
        SourceFileManager default logChange: self selection string.
        ^result
```

shows that evaluation uses the stream which is the first argument of evaluate:in:receiver:notifying:ifFail:. When you examine its receiver, you will find that evaluatorClass returns SmalltalkCompiler and the receiver thus amounts to SmalltalkCompiler new. Expression self selectionAsStream appears to refer to the highlighted text that is being evaluated. However, this text only includes the variables defined in the code fragment. Can we include our workspace-based variables as the context of evaluation of the code fragment? The definition of evaluate:in:receiver:notifying:ifFail: is as follows:

```
evaluate: textOrStream in: aContext receiver: receiver notifying: aRequestor ifFail: failBlock
"Compiles the sourceStream into a parse tree, then generates code into a method.  If receiver is not nil, then
the text can refer to instance variables of that receiver ... etc.
```

and this shows that if we use our extended workspace as receiver, the evaluated code can make references to our workspace-based variables.

The next question is how to access the doIt method, in other words, how to get a suitable instance of a ParagraphEditor. Using the Browser, we find that ParagraphEditor is the controller (the object that handles user input) of its text editor. To obtain the controller, we must ask the text editor widget as in

```
(aBuilder componentAt: #textEditor) widget controller
```

However, if we just sent doIt to the ParagraphEditor, doIt would use its definition of doItReceiver which returns nil and this is not what we want. This means that we must define a new controller with a new definition of doItReceiver that returns the application model of our workspace which contains the our workspace-based variables.

We will thus define a subclass of ParagraphEditor containing only a new definition of doItReceiver that returns the application model. We will then assign this new controller to our text widget instead of the default ParagraphEditor. This opens two questions: How do we assign a new controller, and how do we get from the controller back to the application model.
Assigning a new controller to a widget is simple - just send controller: to the widget as in

```
(aBuilder componentAt: #textEditor) widget controller
```

To see how to get from a controller back to the application model, we will create a test application whose window contains only a text editor. In its postBuildWith: method, we will open an inspector on the builder:

```
postBuildWith: aBuilder
```

```
aBuilder inspect
```

In the inspector, we can locate the controller of the widget and trace how to get back to the application model. We find that the controller has a reference to its view which is wrapped is several wrappers, the last of which is a part of a window and this window can access the builder. We can thus access the builder and its model, and this is the object that should be returned by doItReceiver. We leave it to you to implement the whole idea.

Exercises

1. Implement ExtendedWorkspace. Don't forget to delete the class when the user closes the workspace.
2. Reimplement ExtendedWorkspace using global variables. Don't forget to delete the global variables when the user closes the workspace.
3. Modify ExtendedWorkspace so that the user can delete selected workspace-persistent variables.
4. Modify ExtendedWorkspace so that the user can rename workspace-persistent variables.
5. Add an inspector view for workspace variables at the bottom of the extended workspace.

### A 4.9 Another example: Wrapping objects to intercept messages

A common debugging problem is that an object is changing in some undesirable way and you don't know where. You can, of course, put a breakpoint in front of the creation message and trace your program message by message until you find where the problem occurs but this may be very time-consuming. A much better solution is to intercept all messages sent to the object in question and to open the debugger or perform some other action when such a message occurs.

If we have complete control over the object, the solution is simple - just insert self halt in all methods that refer to it. Very often, however, the object is a system object or there may be too many references to it. In such a case, we would like to have a tool that allows us to create this interception mechanism automatically.

One way to intercept messages to object x is to insert another object 'in front' of it - a 'proxy' as in Figure A 4.9. All messages to x then go to this proxy which can process them in an arbitrary way, for example opening the Debugger and allowing the user to pass the messages on to x.

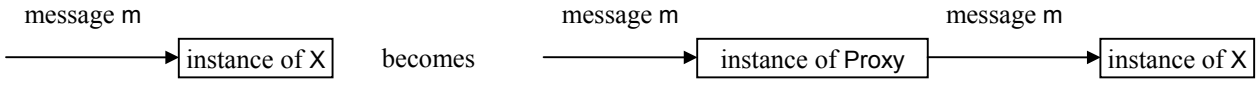


Figure A 4.9. Intercepting messages to x by a proxy.

A possible solution is as follows: Assume that the object in question (call it x) is an instance of class X. Modify creation messages of class X so that they create an instance of X but return an instance p of Proxy that knows about x. All communication with the returned object will now go to p, and p can direct it to x and perform any other operations specified by the programmer.

Having formulated the principle, let's consider how we can implement it. The obvious way is to define Proxy so that it understands all messages understood by x (defined in X and all its superclasses). These messages would have to be modified to allow the programmer to execute the programmer-specified action and then pass the message on to x. Although this approach can be automated, it is very unattractive because it requires defining possibly hundreds of messages in Proxy.

An interesting alternative to making sure that p *understands all messages* of x is to make sure that p *understands as few messages* of x *as possible*. When a messages understood by x is then sent to p, p does not understand it, this sends doesNotUnderstand: aMessage to p, and if we redefine doesNotUnderstand: in Proxy to insert a programmer action and pass the original message to x, the problem is solved (Figure A 4.10). This approach is very simple and we will now implement it.

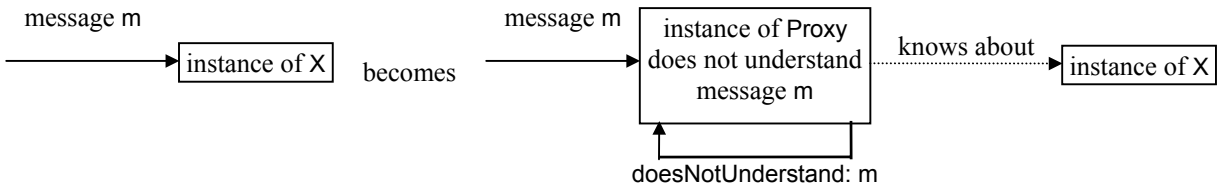


Figure A 4.10. Intercepting messages to x by a proxy via doesNotUnderstand:.

The essential question in defining Proxy is what should be its superclass. Since we want it to share as few method with class X as possible, we must place it as high in the hierarchy as possible so that it inherits as few messages as possible. This seems to suggest that Proxy should be a subclass of Object. But if it is, it will still understand all Object's messages and these quite likely include messages that we would like to intercept. As an example, if X is an Array, we may want to intercept at:put: - but this message is defined in Object. We thus want to eliminate even inheritance from Object. But if we don't want even Object to be the superclass of Proxy, then what should the superclass be? The obvious question is *nothing* – the proxy should not have a superclass – just like Object which also does not have a superclass. In other words, in place of a superclass, use nil. And this exactly what we will do.

Having decided this, the next question is what behaviors should Proxy implement and what attributes it needs. Instances of Proxy must obviously have access to the instance of the original addressee of X messages; we will call its instance variable addressee. The only behavior that appears necessary is a new definition of doesNotUnderstand: that allows the programmer to specify a block to be executed before the intercepted message is passed to addressee. To understand the possible implications, let's examine the existing definition of doesNotUnderstand: in class Object:

```
doesNotUnderstand: aMessage
"The default behavior is to create a Notifier containing the appropriate message and to allow the user to
open a Debugger. Subclasses can override this message in order to modify this behavior."
| 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
```

The argument is an instance of class Message with information about the selector of the intercepted message and its argument. doesNotUnderstand: first attempts to convert the selector to a string and raises an exception (Chapter 11) if it fails. It then opens the usual exception window, and if the user selects *proceed*, it attempts to execute the original message. This normally fails but in our case, we can take advantage of this behavior to pass the message to addressee or execute another action.

We must now decide what should happen when p intercepts a message to x. We may or may not want to open the exception window (we might, for example, only want to collect some statistics about messages to x without ever opening the Debugger). To make this possible, we will create p with a Boolean specifying whether to open the exception window or not.

The next thing we want is to be able to execute an arbitrary block because we don't know what the user might want to do with the intercepted message. What should this block be allowed to do? We cannot know, but in principle it should be able to do anything with the information now available which includes: addressee, selector, and arguments. We will thus assume that the block is a three-argument block with these three objects as its arguments, to provide the maximum possible flexibility.
The design is now complete and we are ready to implement Proxy. Its definition is

```
nil subclass: #Proxy
        instanceVariableNames: 'addressee doesNotUnderstandBlock openNotifier '
```

```
classVariableNames: ''
poolDictionaries: ''
category: 'tests - proxy'
```

Instance variable doesNotUnderstandBlock is the three-argument block to be executed when a message is intercepted, and openNotifier is a Boolean that determines whether the intercepted message should open an exception window or not. *Accepting* this definition in the Browser will open a confirmation window because nil should not normally be the superclass; click *OK* and proceed.

The creation message for Proxy must create a new Proxy object with an addressee, a block, and information about whether to open an exception window or not:

```
newOn: anObject block: aBlock openNotifier: aBoolean
    ^(self new) addressee: anObject; block: aBlock; openNotifier: aBoolean
```

Finally the doesNotUnderstand: method. In VisualWorks, compilation of a subclass of nil automatically adds a few essential methods and these include the standard doesNotUnderstand:. We change it according to our earlier analysis as follows:

```
doesNotUnderstand: aMessage
"Provide the default behavior but add control over the exception window, execution of a programmer supplied block, and finally execution by addressee."
    (Object canUnderstand: aMessage selector)
        ifTrue:
            [self class copy: aMessage selector from: Object.
            ^self perform: aMessage selector withArguments: aMessage arguments].
    openNotifier ifTrue: [Object messageNotUnderstoodSignal raiseRequestWith: aMessage
                errorString: 'Message not understood: ' , aMessage selector].
    doesNotUnderstandBlock
        value: aMessage
        value: aMessage arguments
        value: addressee.
    ^addressee perform: aMessage selector withArguments: aMessage arguments
```

This completes the definition of Proxy and we are now ready to test it. To do this, we will define a new class called TestProxy with a single instance variable called var and a single instance method

```
var: anObject
    var := anObject
```

Assume that we don't want to open an exception window with each message to a TestOfProxy, and that we want it to print which object is being addressed, what is the selector of the intercepted message, and what are its arguments. For the purpose of this test, we will thus write a new TestOfProxy creation method defined as follows:

```
newWithProxy
"Create a proxy that does not open exception window and prints information about myself and the intercepted message."
    | newInstance |
    newInstance := self new.
    ^Proxy  newOn: newInstance
        block: [:message :arguments :addressee | Transcript cr; show: addressee printString ,
                ' gets message ' , message selector , ' with arguments: ' , arguments printString]
        openNotifier: false
```

To test our scheme, execute

```
TestOfProxy newWithProxy var: 13; var: 15
```

and you will get the expected result. When you are finished testing, remove this method so that TestProxy again behaves 'normally'.

Main lessons learned:

- It is possible to define classes outside the Object hierarchy. In fact, one can create another completely separate hierarchy by using nil as the 'superclass' of the root of this hierarchy.

Exercises

1. Write a program to run two series of tests of Array methods using Proxy. In the first series, print a log of message sends by your program in the Transcript. In the second series, let each intercepted message open an exception window.
2. The idea of parallel class hierarchies rooted in nil is used in the complete VisualWorks library. Find all classes in the library whose superclass is nil.
3. Use Proxy to display a confirmer with the name of the intercepted message (Figure A 4.11). An exception window opens only if the user clicks *yes*.
4. Proxy will be even more powerful if the programmer can specify an additional object to be used in the block. This object could, for example, collect some statistics. Extend the definition of Proxy accordingly.
5. Automate the proxy creation mechanism as much as possible.
6. Write a method that will insert a breakpoint into every method in a given class that contains a reference to a given instance variable. Write also a method that will remove all these breakpoints.


Figure A 4.11. Confirmer for Exercise 3.

**Conclusion**

Every class in Smalltalk is the single instance of a matching metaclass which is automatically created when the class is first compiled. Metaclasses use the same names as their corresponding classes and cannot be accessed through the System Browser in the same way as 'regular' classes. Instead, the class message must be addressed to the corresponding class.

Metaclasses form a hierarchy that parallels the hierarchy of classes, with the metaclass of Object at the top. Metaclass Object, however, has itself several superclasses that define behavior shared by all classes and all metaclasses, and their top superclass is class Object. Class Object thus remains a superclass of all classes, including metaclasses.

Like all classes in Smalltalk, each metaclasses is itself an instance of a class. Each metaclass is an instance of class Metaclass.

Whereas class Metaclass defines behaviors particular to metaclasses, behaviors particular to classes are defined in class Class. Much of the behavior of Metaclass and Class is shared and defined in class ClassDescription which is a subclass of Behavior, itself a subclass of Object. As the names suggest, Behavior defines functionality related to basic class behaviors such as methods for creating new classes. ClassDescription adds information about details of class descriptions such as comments and variables. Class, Metaclass, Behavior, and ClassDescription are grouped together in category Kernel Classes.

Although the basic class hierarchy has Object as its root, it is possible to create additional parallel hierarchies rooted in new classes whose superclass is undefined - nil.

On the theoretical and esthetic side, the hierarchy of metaclasses and kernel classes gives a constructive proof that a system can be designed consisting of objects and nothing else. On the practical side, these classes provide many essential behaviors that make possible Smalltalk programming tools such as the Browser, allow creation of classes and access to their descriptions, and provide a very powerful mechanism that makes it possible to reshape the Smalltalk environment from within, even at run-time. This property is called reflectivity and programming based on reflectivity is called metaprogramming. Reflectivity and metaprogramming make Smalltalk one of the most powerful programming environments.

**Important classes introduced in this chapter**

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

*Behavior*, *Class*, *ClassDescription*, *ClassOrganizer*, Metaclass.

**Terms introduced in this chapter**

*metaclass* - the class of a class; accessed by sending class to class name
*kernel class* - a class providing access to shared class behavior and system organization
*metaprogramming* - programming based on kernel classes
*reflectivity* - features of a programming language allowing the programmer to modify the language and its programming environment programmatically