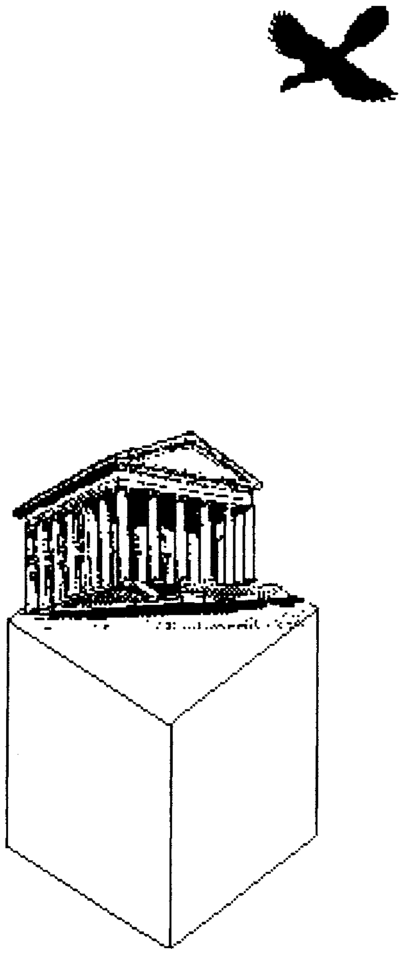

16

Protocol for Classes

- Object
  - Magnitude
    - Character
    - Date
    - Time
    - Number
      - Float
      - Fraction
      - Integer
        - LargeNegativeInteger
        - LargePositiveInteger
        - SmallInteger
    - LookupKey
      - Association
  - Link
    - Process
  - Collection
    - SequenceableCollection
      - LinkedList
        - Semaphore
      - ArrayedCollection
        - Array
        - Bitmap
          - DisplayBitmap
        - RunArray
        - String
          - Symbol
        - Text
        - ByteArray
      - Interval
      - OrderedCollection
        - SortedCollection
    - Bag
    - MappedCollection
    - Set
      - Dictionary
        - IdentityDictionary
  - Stream
    - PositionableStream
      - ReadStream
      - WriteStream
        - ReadWriteStream
          - ExternalStream
            - FileStream
    - Random
  - File
  - FileDirectory
  - FilePage
  - UndefinedObject
  - Boolean
    - False
    - True
  - ProcessorScheduler
  - Delay
  - SharedQueue
  - **Behavior**
    - **ClassDescription**
      - **Class**
      - **MetaClass**
  - Point
  - Rectangle
  - BitBit
    - CharacterScanner
    - Pen
  - DisplayObject
    - DisplayMedium
      - Form
        - Cursor
        - DisplayScreen
    - InfiniteForm
    - OpaqueForm
    - Path
      - Arc
        - Circle
      - Curve
      - Line
      - LinearFit
      - Spline

We have now introduced the protocol for most of the classes that describe the basic components of the Smalltalk-80 system. One notable exception is the protocol for the classes themselves. Four classes—Behavior, ClassDescription, Metaclass, and Class—interact to provide the facilities needed to describe new classes. Creating a new class involves compiling methods and specifying names for instance variables, class variables, pool variables, and the class itself.

Chapters 3, 4, and 5 introduced the basic concepts represented by these classes. To summarize from that discussion, the Smalltalk-80 programmer specifies a new class by creating a subclass of another class. For example, class Collection is a subclass of Object; class Array is a subclass of ArrayedCollection (whose superclass chain terminates with Object).

1. Every class is ultimately a subclass of class Object, except for Object itself, which has no superclass. In particular, Class is a subclass of ClassDescription, which is a subclass of Behavior which is a subclass of Object.

There are two kinds of objects in the system, ones that can create instances of themselves (classes) and ones that can not.

2. Every object is an instance of a class.

Each class is itself an instance of a class. We call the class of a class, its metaclass.

3. Every class is an instance of a metaclass.

Metaclasses are not referenced by class names as are other classes. Instead, they are referred to by a message expression sending the unary message class to the instance of the metaclass. For example, the metaclass of Collection is referred to as Collection class; the metaclass of Class is referred to as Class class.

In the Smalltalk-80 system, a metaclass is created automatically whenever a new class is created. A metaclass has only one instance. The messages categorized as "class methods" in the class descriptions are found in the metaclass of the class. This follows from the way in which methods are found; when a message is sent to an object, the search for the corresponding method begins in the class of the object. When a message is sent to Dictionary, for example, the search begins in the metaclass of Dictionary. If the method is not found in the metaclass, then the search proceeds to the superclass of the metaclass. In this case, the superclass is Set class, the metaclass for Dictionary's superclass. If necessary, the search follows the superclass chain to Object class.

In the diagrams in this chapter, all arrows with solid lines denote a subclass relationship; arrows with dashed lines an instance relationship. A ---> B means A is an instance of B. Solid gray lines indicate the class hierarchy; solid black lines indicate the metaclass hierarchy.

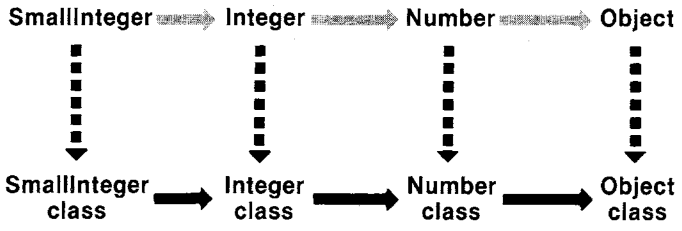

Figure 16.1

Since the superclass chain of all objects ends at Object as shown in Figure 16.1, and Object has no superclass, the superclass of Object's metaclass is not determined by the rule of maintaining a parallel hierarchy. It is at this point that Class is found. The superclass of Object class is Class.

4. All metaclasses are (ultimately) subclasses of Class (Figure 16.2).

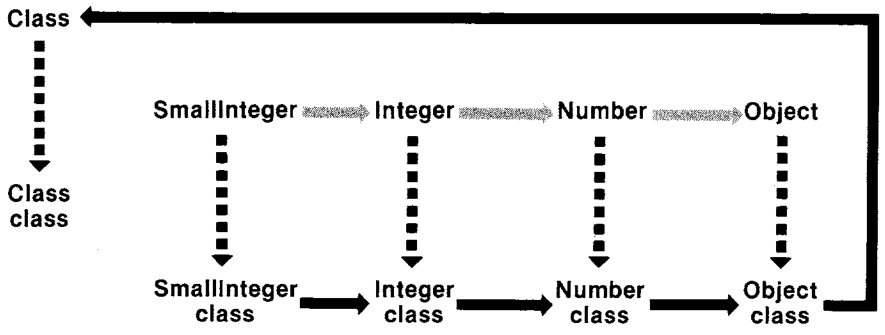

Figure 16.2

Since metaclasses are objects, they too must be instances of a class. Every metaclass is an instance of Metaclass. Metaclass itself is an instance of a metaclass. This is a point of circularity in the system—the metaclass of Metaclass must be an instance of Metaclass.

5. Every metaclass is an instance of Metaclass (Figure 16.3).

Figure 16.4 shows the relationships among Class, ClassDescription, Behavior, and Object, and their respective metaclasses. The class hierarchy follows a chain to Object, and the metaclass hierarchy follows a chain through Object class to Class and on to Object. While the methods of

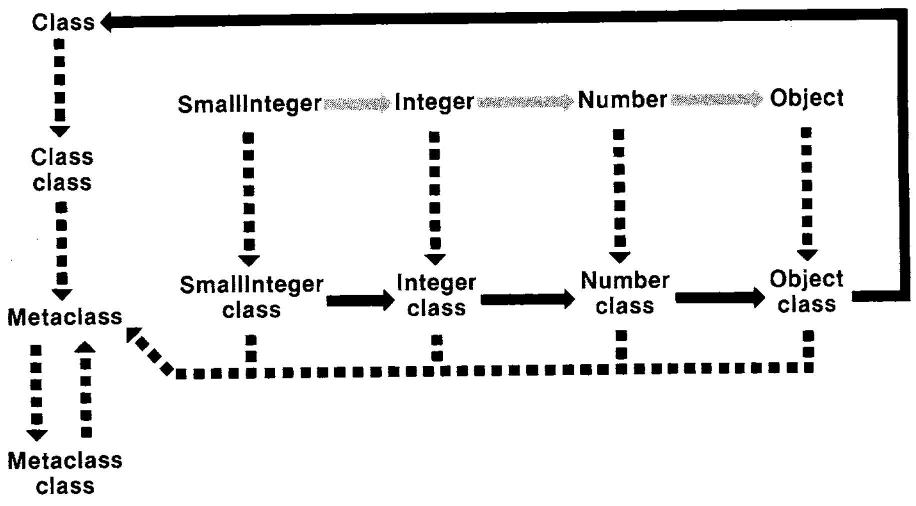

Figure 16.3

Object support the behavior common to all objects, the methods of Class and Metaclass support the behavior common to all classes.

6. The methods of Class and its superclasses support the behavior common to those objects that are classes.
7. The methods of instances of Metaclass add the behavior specific to particular classes.

The correspondence between the class and metaclass hierarchies is shown in Figure 16.5, in which the part of the number hierarchy and the behavior hierarchy of the last two figures are combined.

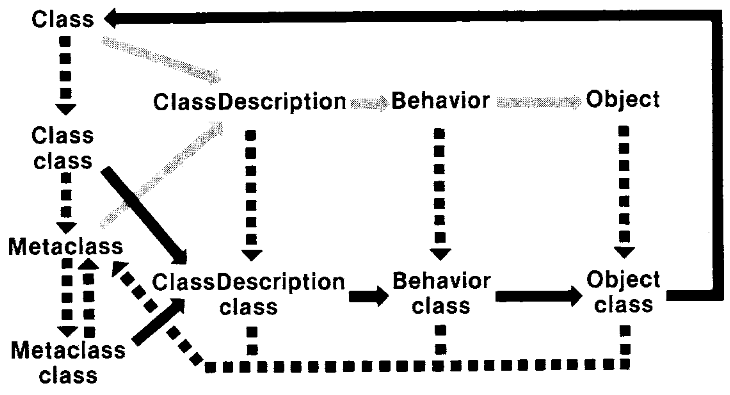

Figure 16.4

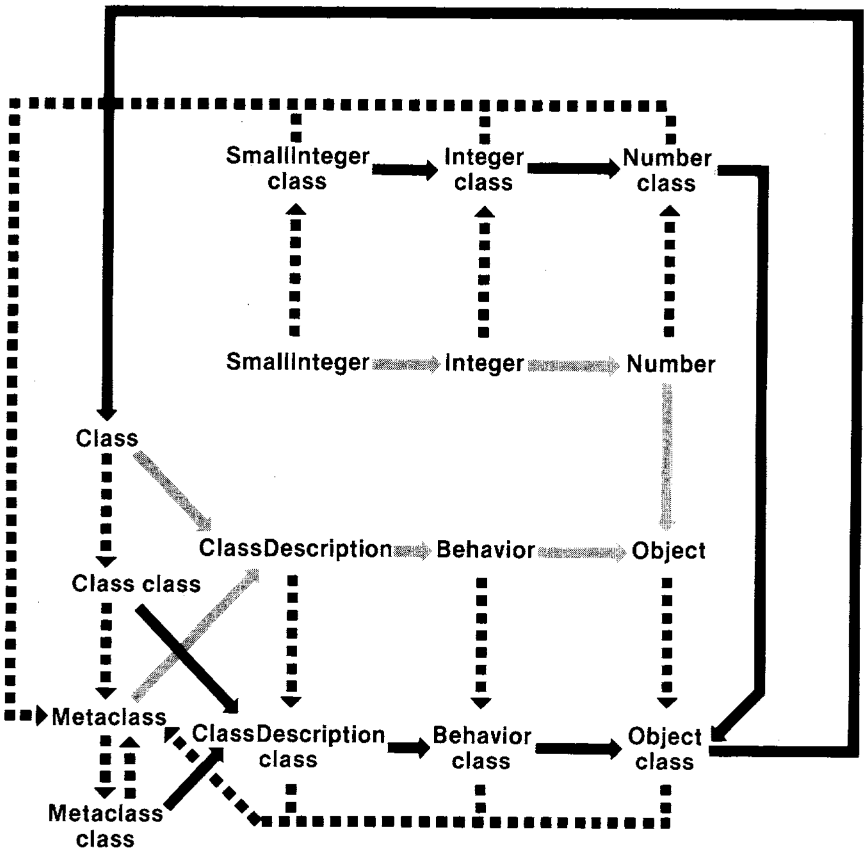

Figure 16.5

Class Behavior

Class Behavior defines the minimum state necessary for objects that have instances. In particular, Behavior defines the state used by the Smalltalk-80 interpreter. It provides the basic interface to the compiler. The state described by Behavior includes a class hierarchy link, a method dictionary, and a description of instances in terms of the number and the representation of their variables.

The message protocol for class Behavior will be described in four categories—*creating, accessing, testing,* and *enumerating.* These categories and their subcategories, as outlined below, provide a model for thinking about the functionality of classes in the Smalltalk-80 system.

Outline of Protocol for All Classes

creating
- creating a method dictionary
- creating instances
- creating a class hierarchy

accessing
- accessing the contents of the method dictionary
- accessing instances and variables: instance, class, and pool
- accessing the class hierarchy

testing
- testing the contents of the method dictionary
- testing the form of the instances
- testing the class hierarchy

enumerating
- enumerating subclasses and instances

□ Behavior's *Creating Protocol* The methods in a class description are stored in a dictionary we refer to as the *method dictionary.* It is also sometimes called a message dictionary. The keys in this dictionary are message selectors; the values are the compiled form of methods (instances of CompiledMethod). The protocol for creating the method dictionary supports compiling methods as well as adding the association between a selector and a compiled method. It also supports accessing both the compiled and noncompiled (source) versions of the method.

Behavior instance protocol

creating method dictionary	
methodDictionary: aDictionary	Store the argument, aDictionary, as the method dictionary of the receiver.
addSelector: selector withMethod: compiledMethod	Add the message selector, selector, with the corresponding compiled method, compiledMethod, to the receiver's method dictionary.
removeSelector: selector	Remove the argument, selector (which is a Symbol representing a message selector), from the receiver's method dictionary. If the selector is not in the method dictionary, report an error.
compile: code	The argument, code, is either a String or an object that converts to a String or it is a PositionableStream accessing an object that is or converts to a String. Compile code as the source code in the context of the receiver's variables. Report an error if the code can not be compiled.

compile: code notifying: requestor	Compile the argument, code, and enter the result in the receiver's method dictionary. If an error occurs, send an appropriate message to the argument, requestor.
recompile: selector	Compile the method associated with the message selector, selector.
decompile: selector	Find the compiled code associated with the argument, selector, and decompile it. Answer the resulting source code as a String. If the selector is not in the method dictionary, report an error.
compileAll	Compile all the methods in the receiver's method dictionary.
compileAllSubclasses	Compile all the methods in the receiver's subclasses' method dictionaries.

Instances of classes are created by sending the message new or new:. These two messages can be overridden in the method dictionary of a metaclass in order to supply special initialization behavior. The purpose of any special initialization is to guarantee that an instance is created with variables that are themselves appropriate instances. We have demonstrated this idea in many previous chapters. Look, for example, at the definition of class Random in Chapter 12; the method dictionary of Random class (the class methods) contains an implementation for new in which a new instance is sent the message setSeed; this initialization guarantees that the random number generation algorithm refers to a variable that is an appropriate kind of number.

Suppose a class overrides the method for new and then one of its subclasses wishes to do the same in order to avoid the behavior created by its superclass's change. The method for the first class might be

```
new
    ↑super new setVariables
```

where the message setVariables is provided in the protocol for instances of the class. By sending the message new to the pseudo-variable super, the method for creating an instance as specified in class Behavior is evaluated; the result, the new instance, is then sent the message setVariables. In the subclass, it is not possible to utilize the message super new because this will invoke the method of the first class—precisely the method to be avoided. In order to obtain the basic method in Behavior for creating an instance, the subclass must use the expression self basicNew. The message basicNew is the primitive instance creation message that should not be reimplemented in any subclass. In Behavior, new and basicNew are identical. A similar pair for creating variable-length objects, new: and basicNew:, are also provided in the protocol of class Behavior. (Note, this technique of dual messages is also used in class Object for accessing messages such as at: and at:put:.)

Behavior instance protocol

instance creation	
new	Answer an instance of the receiver with no indexed variables. Send the receiver the message new: 0 if the receiver is indexable.
basicNew	Same as new, except this method should not be overridden in a subclass.
new: anInteger	Answer an instance of the receiver with anInteger number of indexed variables. Report an error if the receiver is not indexable.
basicNew: anInteger	Same as basicNew, except this method should not be overridden in a subclass.

The protocol for creating classes includes messages for placing the class within the hierarchy of classes in the system. Since this hierarchy is linear, there is only a need to set the superclass and to add or remove subclasses.

Behavior instance protocol

creating a class hierarchy	
superclass: aClass	Set the superclass of the receiver to be the argument, aClass.
addSubclass: aClass	Make the argument, aClass, be a subclass of the receiver.
removeSubclass: aClass	Remove the argument, aClass, from the subclasses of the receiver.

Although the creating protocol for Behavior makes it possible to write expressions for creating a new class description, the usual approach is to take advantage of the graphical environment in which the Smalltalk-80 language is embedded, and to provide an interface in which the user fills out graphically-presented forms to specify information about the various parts of a class.

☐ Behavior's *Accessing Protocol* The messages that access the contents of a method dictionary distinguish among the selectors in the class's locally specified method dictionary, and those in the method dictionaries of the class and each of its superclasses.

Behavior instance protocol

accessing the method dictionary	
selectors	Answer a Set of all the message selectors specified in the receiver's local method dictionary.
allSelectors	Answer a Set of all the message selectors that instances of the receiver can understand. This consists of all message selectors in the receiver's method dictionary and in the dictionaries of each of the receiver's superclasses.

compiledMethodAt: selector	Answer the compiled method associated with the argument, selector, a message selector in the receiver's local method dictionary. Report an error if the selector can not be found.
sourceCodeAt: selector	Answer a String that is the source code associated with the argument, selector, a message selector in the receiver's local method dictionary. Report an error if the selector can not be found.
sourceMethodAt: selector	Answer a Text for the source code associated with the argument, selector, a message selector in the receiver's local method dictionary. This Text provides boldface emphasis for the message pattern part of the method. Report an error if the selector can not be found.

An instance can have named instance variables, indexed instance variables, class variables, and dictionaries of pool variables. Again, the distinction between locally specified variables and variables inherited from superclasses is made in the accessing protocol.

Behavior instance protocol

accessing instances and variables	
allInstances	Answer a Set of all direct instances of the receiver.
someInstance	Answer an existing instance of the receiver.
instanceCount	Answer the number of instances of the receiver that currently exist.
instVarNames	Answer an Array of the instance variable names specified in the receiver.
subclassInstVarNames	Answer a Set of the instance variable names specified in the receiver's subclasses.
allInstVarNames	Answer an Array of the names of the receiver's instance variables, those specified in the receiver and in all of its superclasses. The Array ordering is the order in which the variables are stored and accessed by the Smalltalk-80 interpreter.
classVarNames	Answer a Set of the class variable names specified locally in the receiver.
allClassVarNames	Answer a Set of the names of the receiver's and the receiver's superclasses' class variables.
sharedPools	Answer a Set of the names of the pools (dictionaries) that are specified locally in the receiver.
allSharedPools	Answer a Set of the names of the pools (dictionaries) that are specified in the receiver and each of its superclasses.

Thus, for example,

expression	*result*
OrderedCollection instVarNames	('firstIndex' 'lastIndex')
OrderedCollection subclassInstVarNames	Set ('sortBlock')
SortedCollection allInstVarNames	('firstIndex' 'lastIndex' 'sortBlock')
String classVarNames	Set (StringBlter)
String allClassVarNames	Set (StringBlter DependentsFields ErrorRecursion)
Text sharedPools	a Set containing one element, TextConstants, a Dictionary

The accessing protocol includes messages for obtaining collections of the superclasses and subclasses of a class. These messages distinguish between a class's immediate superclass and subclasses, and all classes in the class's superclass chain.

Behavior instance protocol

accessing class hierarchy	
subclasses	Answer a Set containing the receiver's immediate subclasses.
allSubclasses	Answer a Set of the receiver's subclasses and the receiver's descendent's subclasses.
withAllSubclasses	Answer a Set of the receiver, the receiver's subclasses and the receiver's descendent's subclasses.
superclass	Answer the receiver's immediate superclass.
allSuperclasses	Answer an OrderedCollection of the receiver's superclass and the receiver's ancestor's superclasses. The first element is the receiver's immediate superclass, followed by its superclass, and so on; the last element is always Object.

Thus, for example

expression	*result*
String superclass	ArrayedCollection
ArrayedCollection subclasses	Set (Array ByteArray RunArray Bitmap String Text)

ArrayedCollection allSubclasses	Set (Array ByteArray RunArray Bitmap String Text DisplayBitmap Symbol CompiledMethod)
ArrayedCollection withAllSubclasses	Set (ArrayedCollection Array ByteArray RunArray Bitmap String Text DisplayBitmap Symbol CompiledMethod)
ArrayedCollection allSuperclasses	OrderedCollection (SequenceableCollection Collection Object)
ArrayedCollection class allSuperclasses	OrderedCollection (SequenceableCollection class Collection class Object class Class ClassDescription Behavior Object)

☐ Behavior's *Testing Protocol* Testing protocol provides the messages needed to find out information about the structure of a class and the form of its instances. The structure of a class consists of its relationship to other classes, its ability to respond to messages, the class in which a message is specified, and so on.

The contents of a method dictionary can be tested to find out which class, if any, implements a particular message selector, whether a class can respond to a message, and which methods reference particular variables or literals. These messages are all useful in creating a programming environment in which the programmer can explore the structure and functionality of objects in the system.

Behavior instance protocol

testing the method dictionary	
hasMethods	Answer whether the receiver has any methods in its (local) method dictionary.
includesSelector: selector	Answer whether the message whose selector is the argument, selector, is in the local method dictionary of the receiver's class.
canUnderstand: selector	Answer whether the receiver can respond to the message whose selector is the argument.

	The selector can be in the method dictionary of the receiver's class or any of its superclasses.
whichClassIncludesSelector: selector	Answer the first class on the receiver's superclass chain where the argument, selector, can be found as a message selector. Answer nil if no class includes the selector.
whichSelectorsAccess: instVarName	Answer a Set of selectors from the receiver's local method dictionary whose methods access the argument, instVarName, as a named instance variable.
whichSelectorsReferTo: anObject	Answer a Set of selectors whose methods access the argument, anObject.
scopeHas: name ifTrue: aBlock	Determine whether the variable name, name, is within the scope of the receiver, i.e., it is specified as a variable in the receiver or in one of its superclasses. If so, evalaute the argument, aBlock.

Thus, for example

expression	*result*
OrderedCollection includesSelector: #addFirst:	true
SortedCollection includesSelector: #size	false
SortedCollection canUnderstand: #size	true
SortedCollection whichClassIncludesSelector: #size	OrderedCollection
OrderedCollection whichSelectorsAccess: #firstIndex	Set (makeRoomAtFirst before: size makeRoomAtLast insert:before: remove:ifAbsent: addFirst: first removeFirst find: removeAllSuchThat: at: at:put: reverseDo: do: setIndices:)

The last example expression is useful in determining which methods must be changed if an instance variable is renamed or deleted. In addition to the messages intended for external access, the Set includes all messages implemented in support of the implementation of the external messages.

The testing protocol includes messages to a class that test how its variables are stored, whether the number of variables is fixed-length or variable-length, and the number of named instance variables.

Behavior instance protocol

testing the form of the instances	
isPointers	Answer whether the variables of instances of the receiver are stored as pointers (words).
isBits	Answer whether the variables of instances of the receiver are stored as bits (i.e., not pointers).
isBytes	Answer whether the variables of instances of the receiver are stored as bytes (8-bit integers).
isWords	Answer whether the variables of instances of the receiver are stored as words.
isFixed	Answer true if instances of the receiver do not have indexed instance variables; answer false otherwise.
isVariable	Answer true if instances of the receiver do have indexed instance variables; answer false otherwise.
instSize	Answer the number of named instance variables of the receiver.

So we have

expression	*result*
LinkedList isFixed	true
String isBytes	true
Integer isBits	false
Float isWords	true
OrderedCollection isFixed	false
OrderedCollection instSize	2
oc ← OrderedCollection with: $a with: $b with: $c	OrderedCollection ($a $b $c)
oc size	3

The last four example lines show that instances of OrderedCollection are variable-length; the instance oc has three elements. In addition, instances of OrderedCollection have two named instance variables.

There are four kinds of classes in the system. Classes that have indexed instance variables are called *variable-length* and classes that do not are called *fixed-length.* The variables of all fixed-length classes are stored as pointers (word-sized references). The variables of variable-length classes can contain pointers, bytes, or words. Since a pointer is a word-sized reference, an object that contains pointers will answer true when asked whether it contains words, but the inverse is not always the case. Initialization messages specified in Class and itemized in a later section support creation of each kind of class.

Behavior instance protocol

testing the class hierarchy	
inheritsFrom: aClass	Answer whether the argument, aClass, is on the receiver's superclass chain.
kindOfSubclass	Answer a String that is the keyword that describes the receiver as a class: either a regular (fixed length) subclass, a variableSubclass, a variableByteSubclass, or a variableWordSubclass.

Thus

expression	*result*
String inheritsFrom: Collection	true
String kindOfSubclass	' variableByteSubclass: '
Array kindOfSubclass	' variableSubclass: '
Float kindOfSubclass	' variableWordSubclass: '
Integer kindOfSubclass	' subclass: '

☐ Behavior's *Enumerating Protocol* Messages specified in class Behavior also support listing out particular sets of objects associated with a class and applying each as the argument of a block. This enumeration of objects is similar to that provided in the collection classes, and consists of enumerating over all subclasses, superclasses, instances, and instances of subclasses. In addition, two messages support selecting those subclasses or superclasses for which a block evaluates to true.

Behavior instance protocol

enumerating	
allSubclassesDo: aBlock	Evaluate the argument, aBlock, for each of the receiver's subclasses.
allSuperclassesDo: aBlock	Evaluate the argument, aBlock, for each of the receiver's superclasses.
allInstancesDo: aBlock	Evaluate the argument, aBlock, for each of the current instances of the receiver.

allSubinstancesDo: aBlock	Evaluate the argument, aBlock, for each of the current instances of the receiver's subclasses.
selectSubclasses: aBlock	Evaluate the argument, aBlock, for each of the receiver's subclasses. Collect into a Set only those subclasses for which aBlock evaluates to true. Answer the resulting Set.
selectSuperclasses: aBlock	Evaluate the argument, aBlock, with each of the receiver's superclasses. Collect into a Set only those superclasses for which aBlock evaluates to true. Answer the resulting Set.

As an example, in order to understand the behavior of an instance of the collection classes, it might be useful to know which subclasses of Collection implement the adding message addFirst:. With this information, the programmer can track down which method is actually evaluated when the message addFirst: is sent to a collection. The following expression collects each such class into a Set named subs.

```
subs ← Set new.
Collection allSubclassesDo:
        [ :class |
          (class includesSelector: #addFirst:)
                 ifTrue: [subs add: class]]
```

The same information is accessible from

```
Collection selectSubclasses:
               [ :class | class includesSelector: #addFirst:]
```

Both create a Set of the three subclasses LinkedList, OrderedCollection, and RunArray.

The following expression returns a collection of the superclasses of SmallInteger that implement the message =.

```
SmallInteger selectSuperclasses:
               [ :class | class includesSelector: #=]
```

The response is

```
Set (Integer Magnitude Object )
```

Several subclasses of Collection implement the message first. Suppose we wish to see a list of the code for each implementation. The following expressions print the code on the file whose name is 'classMethods.first'.

```
| aStream |
aStream ← Disk file: 'classMethods.first'.
Collection allSubclassesDo:
        [ :class |
         (class includesSelector: #first)
          ifTrue:
             [class name printOn: aStream.
              aStream cr.
              (class sourceCodeAt: #first) printOn: aStream.
              aStream cr; cr]].
aStream close
```

The resulting contents of the file is

```
SequenceableCollection
'first
   self emptyCheck.
   ↑self at: 1'
OrderedCollection
'first
   self emptyCheck.
   ↑self basicAt: firstIndex'
Interval
'first
   ↑start'
LinkedList
'first
   self emptyCheck.
   ↑firstLink'
```

The protocol described in the next sections is not generally used by programmers, but may be of interest to system developers. The messages described are typically accessed in the programming environment by selecting items from a menu presented in a graphically-oriented interface.

Although most of the facilities of a class are specified in the protocol of Behavior, a number of the messages can not be implemented because Behavior does not provide a complete representation for a class. In particular, Behavior does not provide a representation for instance variable names and class variable names, nor for a class name and a comment about the class.

Representations for a class name, class comment, and instance variable names are provided in ClassDescription, a subclass of Behavior. ClassDescription has two subclasses, Class and Metaclass. Class describes the representation for class variable names and pool variables. A metaclass shares the class and pool variables of its sole instance. Class adds additional protocol for adding and removing class variables and pool variables, and for creating the various kinds of subclasses. Metaclass adds an initialization message for creating a subclass of itself, that is, a message for creating a metaclass for a new class.

Class ClassDescription

ClassDescription represents class naming, class commenting, and naming instance variables. This is reflected in additional protocol for accessing the name and comment, and for adding and removing instance variables.

ClassDescription instance protocol

accessing class description	
name	Answer a String that is the name of the receiver.
comment	Answer a String that is the comment for the receiver.
comment: aString	Set the receiver's comment to be the argument, aString.
addInstVarName: aString	Add the argument, aString, as one of the receiver's instance variables.
removeInstVarName: aString	Remove the argument, aString, as one of the receiver's instance variables. Report an error if aString is not found.

ClassDescription was provided as a common superclass for Class and Metaclass in order to provide further structuring to the description of a class. This helps support a general program development environment. Specifically, ClassDescription adds structure for organizing the selector/method pairs of the method dictionary. This organization is a simple categorization scheme by which the subsets of the dictionary are grouped and named, precisely the way we have been grouping and naming messages throughout the chapters of this book. ClassDescription also provides the mechanisms for storing a full class description on an external stream (a file), and the mechanisms by which any changes to the class description are logged.

The classes themselves are also grouped into system category classifications. The organization of the chapters of this part of the book parallels that of the system class categories, for example, magnitudes, numbers, collections, kernel objects, kernel classes, and kernel support. Protocol for message and class categorization includes the following messages.

ClassDescription instance protocol

organization of messages and classes

category	Answer the system organization category for the receiver.
category: aString	Categorize the receiver under the system category, aString, removing the receiver from any previous category.
removeCategory: aString	Remove each of the messages categorized under the name aString and then remove the category itself.
whichCategoryIncludesSelector: selector	Answer the category of the argument, selector, in the organization of the receiver's method dictionary, or answer nil if the selector can not be found.

Given a categorization of the messages, ClassDescription is able to support a set of messages for copying messages from one method dictionary to another, retaining or changing the category name. Messages to support copying consists of

```
copy: selector from: aClass
copy: selector from: aClass classified: categoryName
copyAll: arrayOfSelectors from: class
copyAll: arrayOfSelectors from: class classified: categoryName
copyAllCategoriesFrom: aClass
copyCategory: categoryName from: aClass
copyCategory: categoryName
        from: aClass
        classified: newCategoryName
```

The categorization scheme has an impact on protocol for compiling since a compiled method must be placed in a particular category. Two messages are provided: compile: code classified: categoryName and compile: code classified: categoryName notifying: requestor.

We also note, for the next example, that Behavior supports special printing protocol so that arguments to the compiling messages can be computed. These are

Behavior instance protocol

printing

classVariableString	Answer a String that contains the names of each class variable in the receiver's variable declaration.
instanceVariableString	Answer a String that contains the names of each instance variable in the receiver's variable declaration.

sharedVariableString	Answer a String that contains the names of each pool dictionary in the receiver's variable declaration.

Take as an example the creation of a class named AuditTrail. This class should be just like LinkedList, except that removing elements should not be supported. Therefore, the class can be created by copying the accessing, testing, adding, and enumerating protocol of LinkedList. We assume that the elements of an AuditTrail are instances of a subclass of Link that supports storing the audit information. First, let's create the class. We assume that we do not know internal information about LinkedList so that the superclass name and variables must be accessed by sending messages to LinkedList.

```
LinkedList superclass
        subclass: #AuditTrail
        instanceVariableNames: LinkedList instanceVariableString
        classVariableNames: LinkedList classVariableString
        poolDictionaries: LinkedList sharedPoolString
        category: 'Record Keeping'.
```

AuditTrail is created as a subclass of whichever class is the superclass for LinkedList (LinkedList superclass). Now we copy the categories we are interested in from class LinkedList.

```
AuditTrail copyCategory: #accessing from: LinkedList.
AuditTrail copyCategory: #testing from: LinkedList.
AuditTrail copyCategory: #adding from: LinkedList.
AuditTrail copyCategory: #enumerating from: LinkedList.
AuditTrail copyCategory: #private from: LinkedList.
```

AuditTrail declared two instance variable names, firstLink and lastLink, and copied messages first, last, size, isEmpty, add:, addFirst:, and addLast:. We also copied all the messages in the category private on the assumption that at least one of them is needed in the implementation of the external messages.

Some messages in ClassDescription that support storing the class description on an external stream are

ClassDescription instance protocol

filing	
fileOutOn: aFileStream	Store a description of the receiver on the file accessed by the argument, aFileStream.
fileOutCategory: categoryName	Create a file whose name is the name of the receiver concatenated by an extension, '.st'. Store on it a description of the messages categorized as categoryName.

fileOutChangedMessages: setOfChanges on: aFileStream	The argument, setOfChanges, is a collection of class/message pairs that were changed. Store a description of each of these pairs on the file accessed by the argument, aFileStream.

We can write a description of class AuditTrail on the file 'AuditTrail.st' by evaluating the expression

```
AuditTrail fileOutOn: (Disk file: 'AuditTrail.st')
```

Class Metaclass

The primary role of a metaclass in the Smalltalk-80 system is to provide protocol for initializing class variables and for creating initialized instances of the metaclass's sole instance. Thus the key messages added by Metaclass are themselves initialization messages—one is sent to Metaclass itself in order to create a subclass of it, and one is sent to an instance of Metaclass in order to create its sole instance.

Metaclass class protocol

instance creation

subclassOf: superMeta	Answer an instance of Metaclass that is a subclass of the metaclass, superMeta.
name: newName environment: aSystemDictionary subclassOf: superClass instanceVariableNames: stringOfInstVarNames variable: variableBoolean words: wordBoolean pointers: pointerBoolean classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames category: categoryName comment: commentString changed: changed	Each of these arguments, of course, is needed in order to create a fully initialized class.

The Smalltalk-80 programming environment provides a simplified way, using graphical interface techniques, in which the user specifies the information to create new classes.

Class Class

Instances of Class describe the representation and behavior of objects. Class adds more comprehensive programming support facilities to the basic ones provided in Behavior and more descriptive facilities to the ones provided in ClassDescription. In particular, Class adds the representation for class variable names and shared (pool) variables.

Class instance protocol

accessing instances and variables	
addClassVarName: aString	Add the argument, aString, as a class variable of the receiver. The first character of aString must be capitalized; aString can not already be a class variable name.
removeClassVarName: aString	Remove the receiver's class variable whose name is the argument, aString. Report an error if it is not a class variable or if it is still being used in a method of the class.
addSharedPool: aDictionary	Add the argument, aDictionary, as a pool of shared variables. Report an error if the dictionary is already a shared pool in the receiver.
removeSharedPool: aDictionary	Remove the argument, aDictionary, as one of the receiver's pool dictionaries. Report an error if the dictionary is not one of the receiver's pools.
classPool	Answer the dictionary of class variables of the receiver.
initialize	Initialize class variables.

Additional accessing messages store a description of the class on a file, where the file has the same name as that of the class (fileOut), and remove the class from the system (removeFromSystem).

A variety of messages for creating one of the four kinds of subclasses in the system are specified in the method dictionary of Class. In addition, Class provides a message for renaming a class (rename: aString); this message is provided in Class rather than in ClassDescription because it is not an appropriate message to send to a metaclass.

Class instance protocol

instance creation	
subclass: classNameString instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames category: categoryNameString	Create a new class that is a fixed-length (regular) subclass of the receiver. Each of the arguments provides the information needed to initialize the new class and categorize it.

Three other messages, like the one above except that the first keyword is variableSubclass:, variableByteSubclass:, or variableWordSubclass, support the creation of the other kinds of classes. Note also that the system requires that a subclass of a variable-length class be a variable-length class. When possible, the system makes the appropriate conversion; otherwise, an error is reported to the programmer.

Suppose that every time we created a new subclass, we wanted to install messages for storing and retrieving the instance variables of that class. For example, if we create a class Record with instance variable names name and address, we wish to provide messages name and address, to respond with the values of these variables, and name: argument and address: argument, to set the values of these variables to the value of the message argument. One way to accomplish this is to add the following method to the instance creation protocol of class Class.

```
accessingSubclass: className
    instanceVariableNames: instVarString
    classVariableNames: classVarString
    poolDictionaries: stringOfPoolNames
    category: categoryName
  | newClass |
  newClass ← self subclass: className
                   instanceVariableNames: instVarString
                   classVariableNames: classVarString
                   poolDictionaries: stringOfPoolNames
                   category: categoryName.
  newClass instVarNames do:
    [ :aName |
      newClass compile: (aName , '
    ↑ ', aName) classified: #accessing.
      newClass compile: (aName , ': argument
    ', aName, ' ← argument.
    ↑argument') classified: #accessing].
  ↑newClass
```

The method creates the class as usual, then, for each instance variable name, compiles two methods. The first is of the form

```
name
  ↑name
```

and the second is of the form

```
name: argument
  name ← argument.
  ↑argument
```

So, if we create the class Record, we can do so by sending Object the following message.

```
Object accessingSubclass: #Record
        instanceVariableNames: 'name address'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Example'.
```

The message is found in the method dictionary of Class, and creates the following four messages in the category accessing of class Record.

```
accessing

   name
      ↑name
   name: argument
      name ← argument.
      ↑argument
   address
      ↑address
   address: argument
      address ← argument.
      ↑argument
```