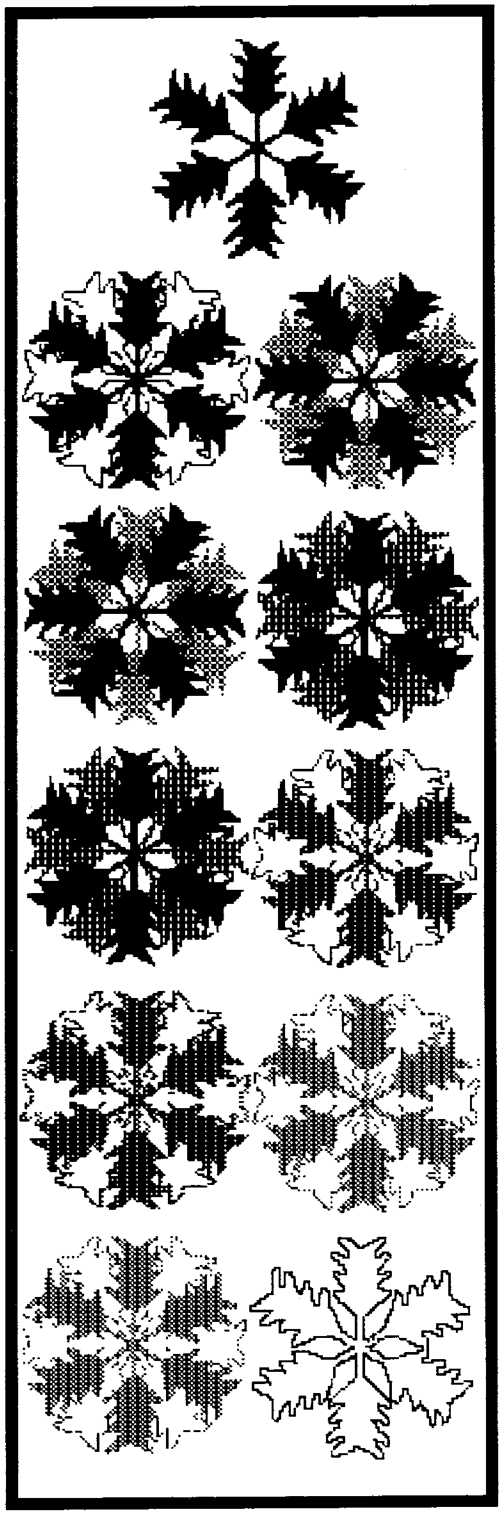


5

Metaclasses

Since all Smalltalk-80 system components are represented by objects and all objects are instances of a class, the classes themselves must be represented by instances of a class. A class whose instances are themselves classes is called a *metaclass*. This chapter describes the special properties of metaclasses. Examples illustrate how metaclasses are used to support instance creation and general class inquiries.

In earlier versions of the Smalltalk system, there was only one metaclass, named Class. It corresponded to the class organization depicted in Figure 5.1. As used in Chapter 4, a box denotes a class and a circle denotes an instance of the class in which it is contained. Where possible, the box is labeled with the name of the class it represents. Note that there is one circle in the box labeled Class for each box in the diagram.

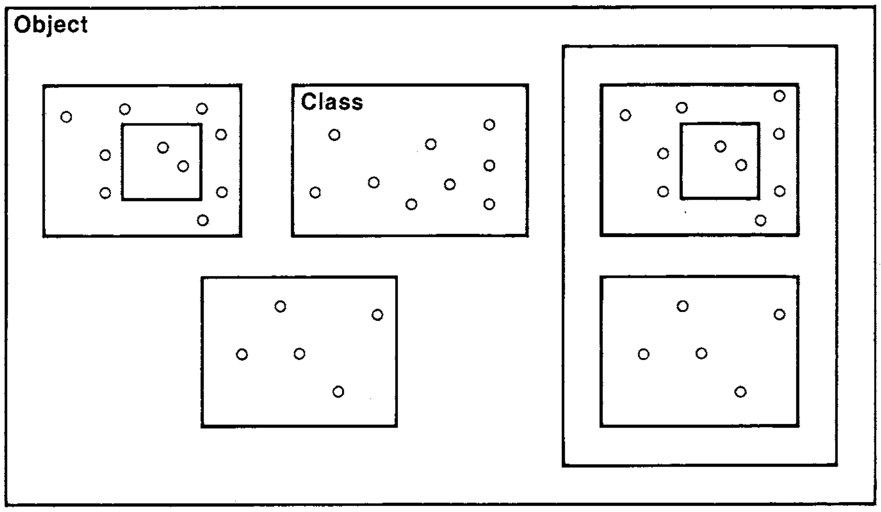


Figure 5.1

This approach had the difficulty that the message protocol of all classes was constrained to be the same since it was specified in one place. In particular, the messages used to create new instances were the same for all classes and could not take any special initialization requirements into account. With a single metaclass, all classes respond to the message new or new: by returning an instance whose instance variables all refer to nil. For most objects, nil is not a reasonable instance variable value, so new instances have to be initialized by sending another message. The programmer must ensure that every time a new or new: is sent, another message is sent to the new object so that it will be properly initialized. Examples of this kind of initialization were shown in Chapter 4 for SmallDictionary and FinancialHistory.

The Smalltalk-80 system removes the restriction that all classes have the same message protocol by making each class an instance of its own metaclass. Whenever a new class is created, a new metaclass is created for it automatically. Metaclasses are similar to other classes because they contain the methods used by their instances. Metaclasses are different from other classes because they are not themselves instances of metaclasses. Instead, they are all instances of a class called Metaclass. Also, metaclasses do not have class names. A metaclass can be accessed by sending its instance the unary message class. For example, Rectangle's metaclass can be referred to with the expression Rectangle class.

The messages of a metaclass typically support creation and initialization of instances, and initialization of class variables.

Initialization of Instances

Each class can respond to messages that request properly initialized new instances. Multiple metaclasses are needed because the initialization messages are different for different classes. For example, we have already seen that Time creates new instances in response to the message now and Date creates new instances in response to the message today.

```
Time now
Date today
```

These messages are meaningless to Point, the class whose instances represent two-dimensional locations. Point creates a new instance in response to a message with selector x:y: and two arguments specifying the coordinates. This message is, in turn, meaningless to Time or Date.

```
Point x: 100 y: 150
```

Class Rectangle understands several messages that create new instances. A message with the selector origin:corner: takes Points representing the upper left and lower right corners as arguments.

```
Rectangle
        origin: (Point x: 50 y: 50)
        corner: (Point x: 250 y: 300)
```

A message with the selector origin:extent: takes as arguments the upper left corner and a Point representing the width and height. The same rectangle could have been created by the following expression.

```
Rectangle
        origin: (Point x: 50 y: 50)
        extent: (Point x: 200 y: 250)
```

In the Smalltalk-80 system, Class is an abstract superclass for all of the metaclasses. Class describes the general nature of classes. Each metaclass adds the behavior specific to its single instance. Metaclasses may add new instance creation messages like those of Date, Time, Point, and Rectangle mentioned above, or they may redefine the fundamental new and new: messages in order to perform some default initialization.

The organization of classes and instances in the system, as described so far, is illustrated in Figure 5.2.

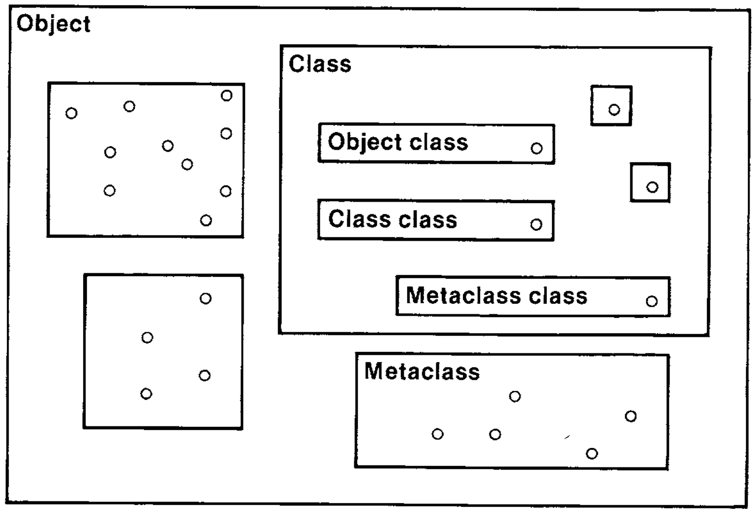


Figure 5.2

In this figure, we indicate classes Object, Metaclass, and Class, and metaclasses for each. Each circle within the box labeled Metaclass denotes a metaclass. Each box within the box labeled Class denotes a subclass of Class. There is one such box for each circle within the box labeled Metaclass. Each of these boxes contains a circle denoting its instance; these instances refer to Object or one of the subclasses of Object, but not to metaclasses.

An Example Metaclass

Since there is a one-to-one correspondence between a class and its metaclass, their descriptions are presented together. An implementation description includes a part entitled "class methods" that shows the methods added by the metaclass. The protocol for the metaclass is always found by looking at the class methods part of the implementation description of its single instance. In this way, messages sent to the class (class methods) and messages sent to instances of the class (instance methods) are listed together as part of the complete implementation description.

The following new version of the implementation description for FinancialHistory includes class methods.

class name	FinancialHistory
superclass	Object
instance variable names	cashOnHand incomes expenditures

class methods

```
instance creation

    initialBalance: amount
        ↑super new setInitialBalance: amount
    new
        ↑super new setInitialBalance: 0
```

instance methods

```
transaction recording

    receive: amount from: source
        incomes at: source
                put: (self totalReceivedFrom: source) + amount.
        cashOnHand ← cashOnHand + amount
    spend: amount for: reason
        expenditures at: reason
                put: (self totalSpentFor: reason) + amount.
        cashOnHand ← cashOnHand − amount

inquiries

    cashOnHand
        ↑cashOnHand
    totalReceivedFrom: source
        (incomes includesKey: source)
            ifTrue: [↑incomes at: source]
            ifFalse: [↑0]
    totalSpentFor: reason
        (expenditures includesKey: reason)
            ifTrue: [↑expenditures at: reason]
            ifFalse: [↑0]
```

```
private

  setInitialBalance: amount
    cashOnHand ← amount.
    incomes ← Dictionary new.
    expenditures ← Dictionary new
```

Three changes have been made to the implementation description.

1. One category of class methods named instance creation has been added. The category contains methods for initialBalance: and new. By convention, the category instance creation is used for class methods that return new instances.
2. The category of instance methods named initialization has been deleted. It had included a method for initialBalance:.
3. A category of instance methods named private has been added. The category contains one method for setInitialBalance:; this method contains the same expressions that were in the deleted method for initialBalance:.

This example illustrates how metaclasses create initialized instances. The instance creation methods for initialBalance: and new do not have direct access to the instance variables of the new instance (cashOnHand, incomes, and expenses). This is because the methods are not a part of the class of the new instance, but rather of the class's class. Therefore, the instance creation methods first create uninitialized instances and then send an initialization message, setInitialBalance:, to the new instance. The method for this message is found in the instance methods part of FinancialHistory's implementation description; it can assign appropriate values to the instance variables. The initialization message is not considered part of the external protocol of FinancialHistory so it is categorized as private. It is typically only sent once and only by a class method.

The old initialization message initialBalance: was deleted because the proper way to create a FinancialHistory is to use an expression such as

```
FinancialHistory initialBalance: 350
```

not

```
FinancialHistory new initialBalance: 350
```

Indeed, this second expression would now create an error since instances of FinancialHistory are no longer described as responding to initialBalance:. We could have maintained the instance method initialBalance: and implemented the class method for initialBalance: to call on it, but we try not to use the same selectors for both instance and class methods in order to improve the readability of the implementation description. However, there would be no ambiguity if the same selector were used.

Metaclass Inheritance

Like other classes, a metaclass inherits from a superclass. The simplest way to structure the inheritance of metaclasses would be to make each one a subclass of Class. This organization was shown in Figure 5.2. Class describes the general nature of classes. Each metaclass adds behavior specific to its instance. Metaclasses may add new instance creation messages or they may redefine the fundamental new and new: messages to perform some default initialization.

When metaclasses were added to the Smalltalk-80 system, one further step in class organization was taken. The metaclass subclass hierarchy was constrained to be parallel to the subclass hierarchy of the classes that are their instances. Therefore, if DeductibleHistory is a subclass of FinancialHistory, then DeductibleHistory's metaclass must be a subclass of FinancialHistory's metaclass. A metaclass typically has only one instance.

An abstract class named ClassDescription was provided to describe classes and their instances. Class and Metaclass are subclasses of ClassDescription. Since the superclass chain of all objects ends at Object and Object has no superclass, the superclass of Object's metaclass is Class. From Class, the metaclasses inherit messages that provide protocol for the creation of instances (Figure 5.3).

The superclass chain from Class leads eventually to class Object. Notice that the hierarchy of boxes with the box labeled Object class is like that of the hierarchy of boxes within the box labeled Object; this similarity illustrates the parallel hierarchies. A full description of this part of the system, including the relationship between Metaclass and its metaclass, is provided in Chapter 16.

As an example of the metaclass inheritance hierarchy, consider the implementation of initialBalance: in FinancialHistory class.

```
initialBalance: amount
    ↑super new setInitialBalance: amount
```

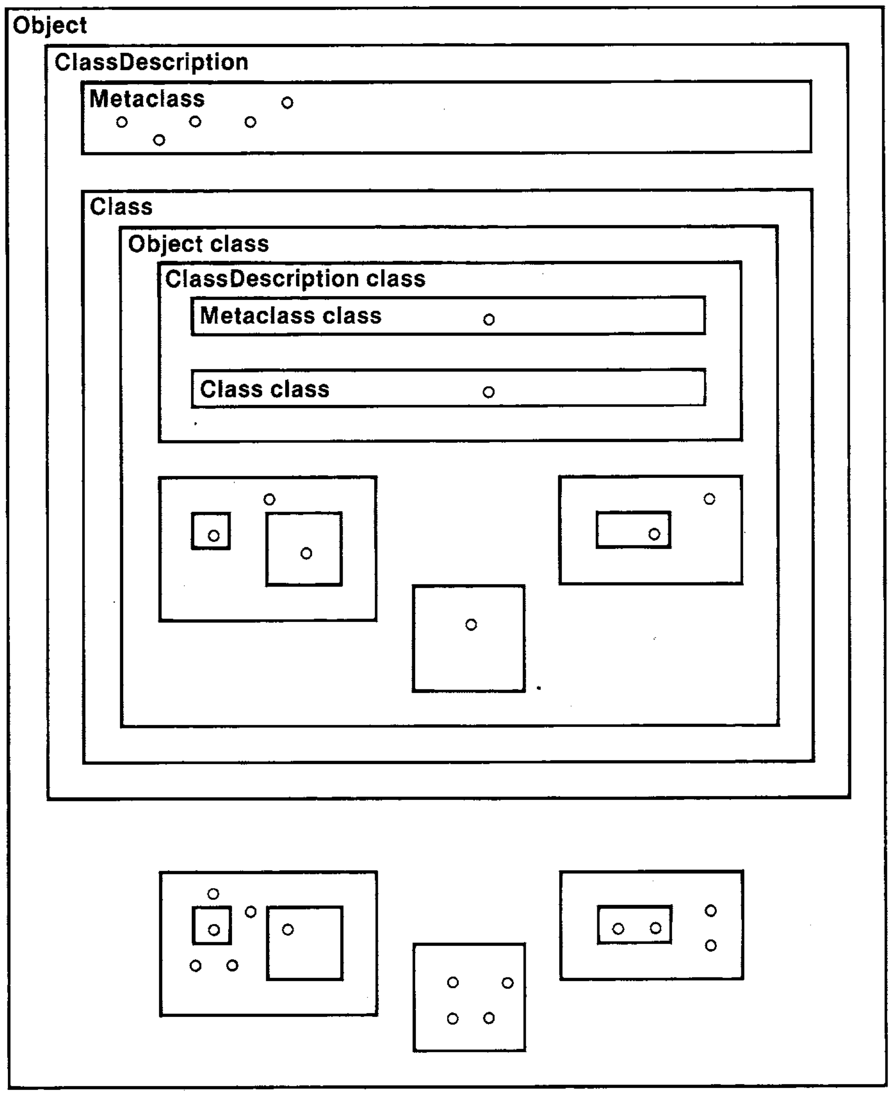


Figure 5.3

This method creates a new instance by evaluating the expression super new; it uses the method for new found in the class methods of the superclass, not the class methods found in this class. It then sends the new instance the message setInitialBalance: with the initial amount of the balance as the argument. Similarly, new is reimplemented as creating an instance using super new followed by setInitialBalance:.

```
new
    ↑super new setInitialBalance: 0
```

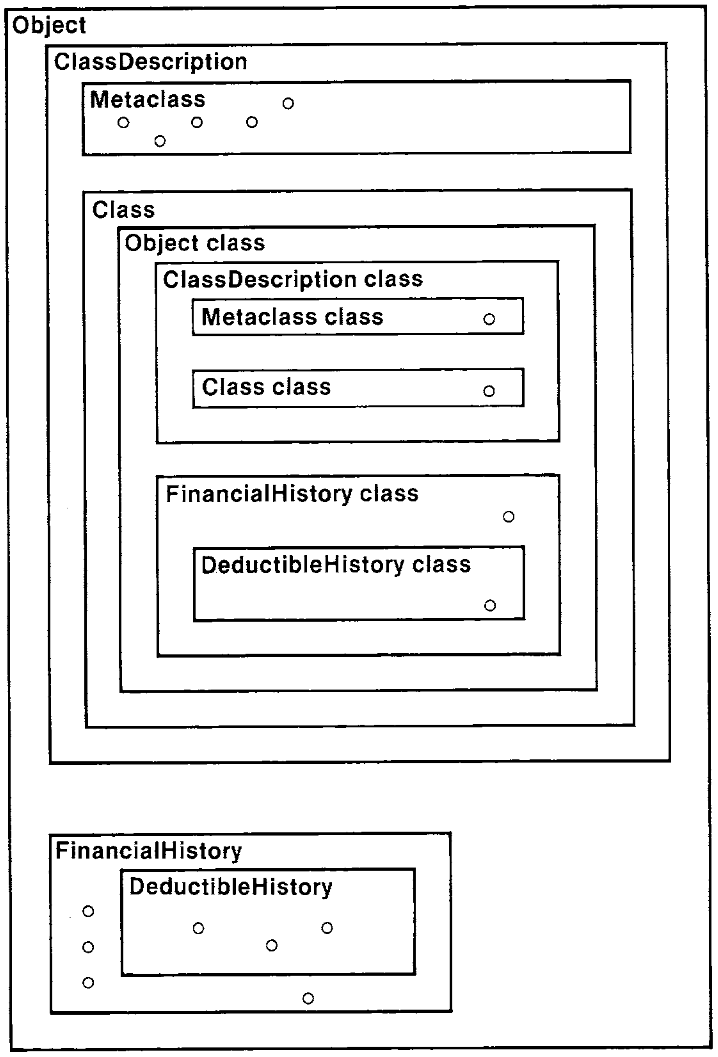


Figure 5.4

Where is the method for the message new sent to super actually found? The subclass hierarchy of the metaclasses parallels the hierarchy of their instances. If one class is a subclass of another, its metaclass will be a subclass of the other's metaclass, as indicated in Figure 5.3. The parallel class and metaclass hierarchies for the FinancialHistory application are shown in Figure 5.4.

If we evaluate the expression

```
FinancialHistory initialBalance: 350
```

the search for the response to initialBalance: begins in FinancialHistory class, i.e., in the class methods for FinancialHistory. A method for that selector is found there. The method consists of two messages:

1. Send super the message new.
2. Send the result of 1 the message setInitialBalance: 0.

The search for new begins in the superclass of FinancialHistory class, that is, in Object class. A method is not found there, so the search continues up the superclass chain to Class. The message selector new is found in Class, and a primitive method is executed. The result is an uninitialized instance of FinancialHistory. This instance is then sent the message setInitialBalance:. The search for the response begins in the class of the instance, i.e., in FinancialHistory (in the instance methods). A method is found there which assigns a value to each instance variable.

The evaluation of

```
FinancialHistory new
```

is carried out in a similar way. The response to new is found in FinancialHistory class (i.e., in the class methods of FinancialHistory). The remaining actions are the same as for initialBalance: with the exception of the value of the argument to setInitialBalance:. The instance creation methods must use super new in order to avoid invoking the same method recursively.

Initialization of Class Variables

The main use of messages to classes other than creation of instances is the initialization of class variables. The implementation description's variable declaration gives the names of the class variables only, not their values. When a class is created, the named class variables are created, but they all have a value of nil. The metaclass typically defines a method that initializes the class variables. By convention, the class-variable initialization method is usually associated with the unary message initialize, categorized as class initialization.

Class variables are accessible to both the class and its metaclass. The assignment of values to class variables can be done in the class methods, rather than indirectly via a private message in the instance methods (as was necessary for instance variables).

The example DeductibleHistory, this time with a class variable that needs to be initialized, is shown next. DeductibleHistory is a subclass of FinancialHistory. It declares one class variable, MinimumDeductions.

class name	DeductibleHistory
superclass	FinancialHistory
instance variable names	deductibleExpenditures
class variable names	MinimumDeductions
class methods	

```
instance creation

   initialBalance: amount
      | newHistory |
      newHistory ← super initialBalance: amount.
      newHistory initializeDeductions.
      ↑newHistory
   new
      | newHistory |
      newHistory ← super initialBalance: 0.
      newHistory initializeDeductions.
      ↑newHistory

class initialization

   initialize
      MinimumDeductions ← 2300
```

instance methods

```
transaction recording

   spendDeductible: amount for: reason
      self spend: amount for: reason.
      deductibleExpenditures ←
         deductibleExpenditures + amount
   spend: amount for: reason deducting: deductibleAmount
      self spend: amount for: reason.
      deductibleExpenditures ←
         deductibleExpenditures + deductibleAmount

inquiries

   isItemizable
      ↑deductibleExpenditures >= MinimumDeductions
   totalDeductions
      ↑deductibleExpenditures

private

   initializeDeductions
      deductibleExpenditures ← 0
```

This version of DeductibleHistory adds five instance methods, one of which is isItemizable. The response to this message is true or false depending on whether enough deductions have been accumulated in order to itemize deductions on a tax report. The tax law specifies that a minimum deduction of 2300 can be taken, so if the accumulation is less, the standard deduction should be used. The constant, 2300, is referred to by the class variable MinimumDeductions. In order to successfully send an instance of DeductibleHistory the message isItemizable, the class variable MinimumDeductions must be assigned its numeric value. This is done by sending the class the message initialize before any instances are created.

```
DeductibleHistory initialize
```

This message only has to be sent once, after the class initialization message is first defined. The variable is shared by each new instance of the class.

According to the above class description, a new instance of DeductibleHistory can be created by sending the class the messages initialBalance: or new, just as for the superclass FinancialHistory. Suppose we evaluate the expression

```
DeductibleHistory initialBalance: 100
```

The determination of which methods are actually followed in order to evaluate the expression depends on the class/superclass chain for DeductibleHistory. The method for initialBalance: is found in the class methods of DeductibleHistory.

```
initialBalance: amount
    | newHistory |
    newHistory ← super initialBalance: amount.
    newHistory initializeDeductions.
    ↑newHistory
```

This method declares newHistory as a temporary variable. The first expression of the method is an assignment to the temporary variable.

```
newHistory ← super initialBalance: amount
```

The pseudo-variable super refers to the receiver. The receiver is the class DeductibleHistory; its class is its metaclass. The superclass of the metaclass is the metaclass for FinancialHistory. Thus we can find the method that will be followed by looking in the class methods of FinancialHistory. The method is

```
initialBalance: amount
    ↑super new setInitialBalance: amount
```

We have already followed evaluation of this method. The response to new is found in Class. A new instance of the original receiver, DeductibleHistory, is created and sent the message setInitialBalance:. The search for setInitialBalance: begins in the class of the new instance, i.e., in DeductibleHistory. It is not found. The search proceeds to the superclass FinancialHistory. It is found and evaluated. Instance variables declared in FinancialHistory are assigned values. The value of the first expression of the class method for initialBalance: in DeductibleHistory, then, is a partially initialized new instance. This new instance is assigned to the temporary variable newHistory.

newHistory is then sent the message initializeDeductions. The search begins in the class of the receiver, newHistory; the class is DeductibleHistory. The method is found. It assigns the value of the fourth instance variable to be 0.

The third expression of the instance creation message returns the new instance.

An alternative way to implement the class DeductibleHistory is presented next. In this alternative class description, the instance-creation class methods of FinancialHistory are not reimplemented. Rather, the private instance-method message setInitialBalance: is overridden in order to account for the additional instance variable.

```
class name                    DeductibleHistory
superclass                    FinancialHistory
instance variable names       deductibleExpenditures
class variable names          MinimumDeductions
class methods

class initialization

   initialize
      MinimumDeductions ← 2300

instance methods

transaction recording

   spendDeductible: amount for: reason
      self spend: amount for: reason.
      deductibleExpenditures ←
         deductibleExpenditures + amount
   spend: amount for: reason deducting: deductibleAmount
      self spend: amount for: reason.
      deductibleExpenditures ←
         deductibleExpenditures + deductibleAmount

inquiries

   isItemizable
      ↑deductibleExpenditures > = MinimumDeductions
```

```
        totalDeductions
            ↑deductibleExpenditures

    private

        setInitialBalance: amount
            super setInitialBalance: amount.
            deductibleExpenditures ← 0
```

Using this alternative class description for DeductibleHistory, the evaluation of the response to initialBalance: in

```
    DeductibleHistory initialBalance: 350
```

is to search in DeductibleHistory class for initialBalance:. It is not found. Continue the search in the superclass, FinancialHistory class. It is found. The method evaluated consists of the expression

```
    super new setInitialBalance: amount
```

The method for new is found in Class. Search for setInitialBalance: beginning in the class of the new instance, a DeductibleHistory. The method for setInitialBalance: is found in DeductibleHistory. The response of setInitialBalance: in DeductibleHistory is to send the same message to super so that the search for the method begins in FinancialHistory. It is found and three instance variables are assigned values. The second expression of setInitialBalance: in DeductibleHistory sets the fourth variable to 0. The result of the original message is a fully initialized instance of DeductibleHistory.

Summary of Method Determination

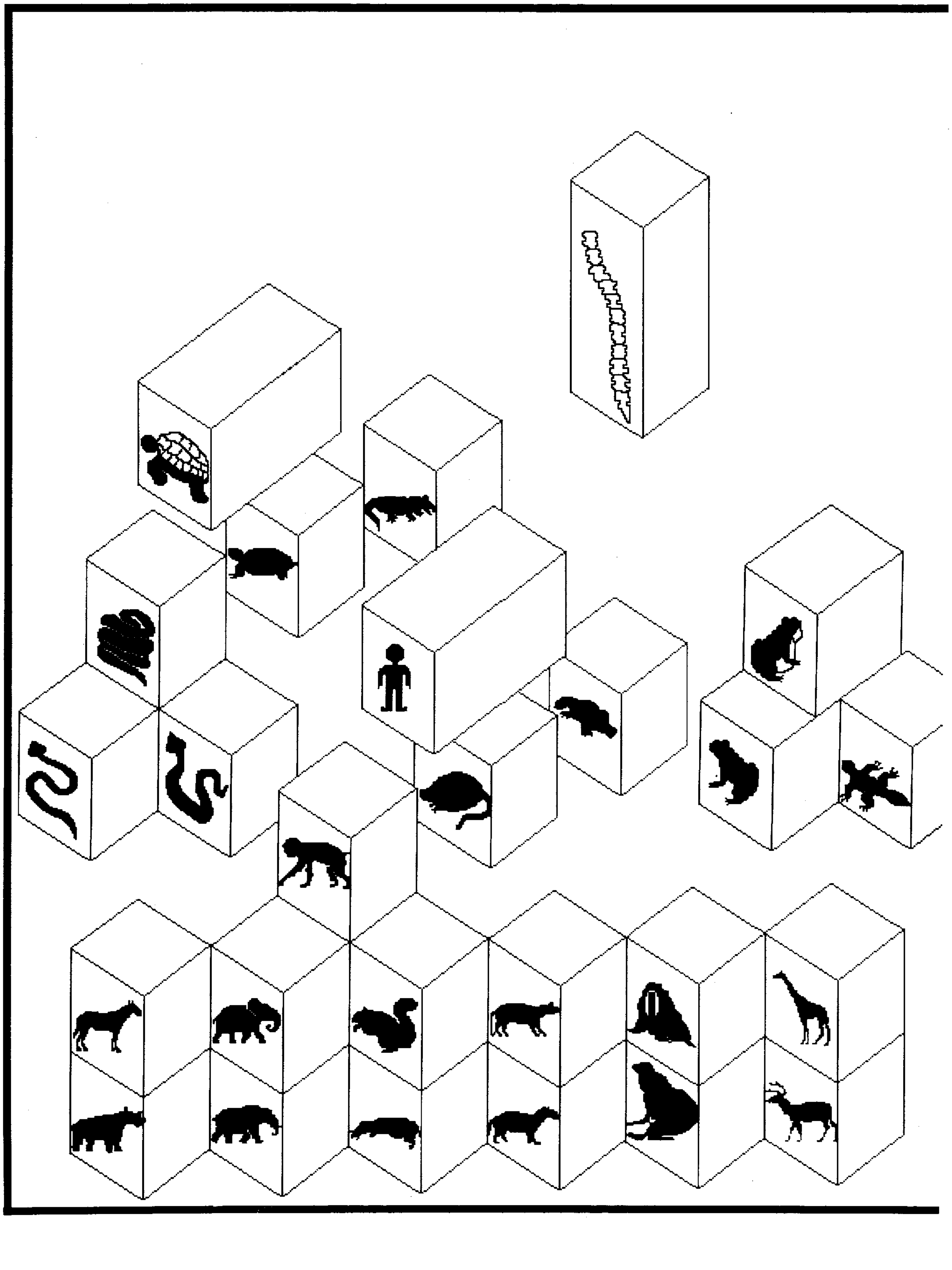
Determining the actual actions taken when a message is sent involves searching the methods in the class hierarchy of the receiver. The search begins with the class of the receiver and follows the superclass chain. If not found after searching the last superclass, Object, an error is reported. If the receiver is a class, its class is a metaclass. The messages to which a class can respond are listed in the implementation description in the part entitled "class methods." If the receiver is not a class, then the messages to which it can respond are listed in its implementation description in the part entitled "instance methods."

The pseudo-variable self refers to the receiver of the message that invoked the executing method. The search for a method corresponding to a message to self begins in the class of self. The pseudo-variable super also refers to the receiver of the message. The search for a method corresponding to a message to super begins in the superclass of the class in which the executing method was found.

This ends the description of the Smalltalk-80 programming language. To use the system, the programmer must have general knowledge of the system classes. Part Two gives detailed accounts of the protocol descriptions for each of the system classes and provides examples, often by presenting the implementation descriptions of system classes. Part Three introduces a moderate-size application. Before delving into the details of the actual system classes, the reader might want to skip to Part Three to get a sense of what it is like to define a larger application.

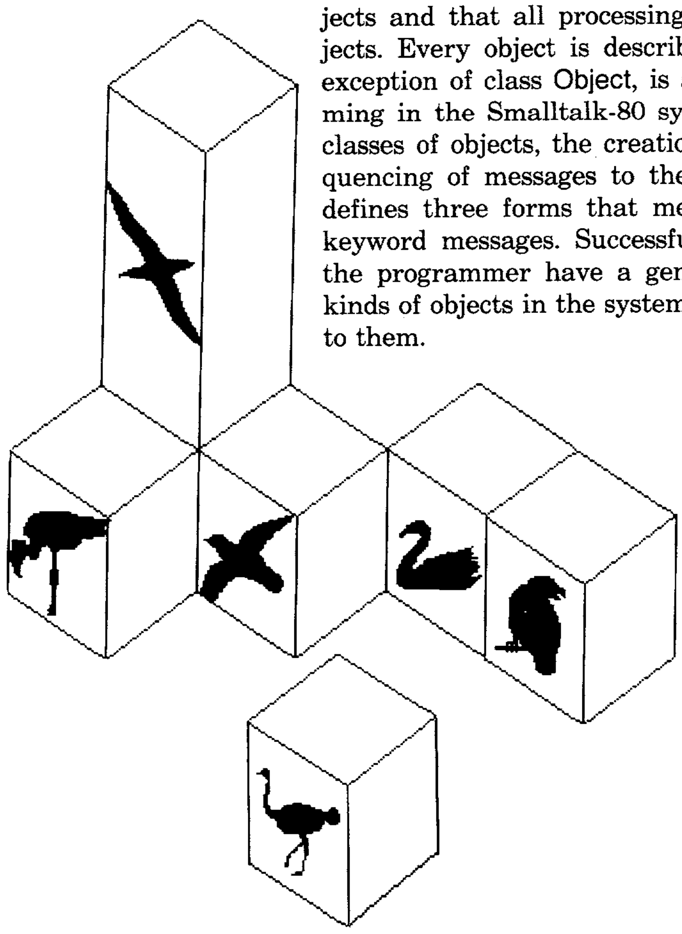
Summary of Terminology

metaclass	The class of a class.
Class	An abstract superclass of all classes other than metaclasses.
Metaclass	A class whose instances are classes of classes.



PART TWO

Part One provided an overview of the Smalltalk-80 language both from the semantic view of objects and message sending and from the syntactic view of the form that expressions take. The Smalltalk-80 programmer must first understand the semantics of the language: that all information is represented in the form of objects and that all processing is done by sending messages to objects. Every object is described by a class; every class, with the exception of class Object, is a subclass of another class. Programming in the Smalltalk-80 system involves the description of new classes of objects, the creation of instances of classes, and the sequencing of messages to the instances. The Smalltalk-80 syntax defines three forms that messages can take: unary, binary, and keyword messages. Successful use of the language requires that the programmer have a general knowledge of each of the basic kinds of objects in the system and of the messages that can be sent to them.



The semantics and syntax of the language are relatively simple. Yet the system is large and powerful due to the numbers of and kinds of available objects. There are eight significant categories of classes in the Smalltalk-80 system: kernel and kernel support, linear measures, numbers, collections, streams, classes, independent processes, and graphics. The protocol of these kinds of objects is reviewed in 12 chapters of Part Two. In each of these chapters, the diagram of the class hierarchy given in Chapter 1 is re-presented in order to highlight the portion of the hierarchy discussed in that chapter. Three additional chapters in Part Two provide examples of Smalltalk-80 expressions and class descriptions.

The classes in the Smalltalk-80 system are defined in a linear hierarchy. The chapters in Part Two take an encyclopedic approach to reviewing class protocol: categories of messages are defined, each message is annotated, and examples are given. In presenting the protocol of a class, however, only those messages added by the class are described. The complete message protocol is determined by examining the protocol specified in the class and in each of its superclasses. Thus it is useful to present the classes starting with a description of class Object and to proceed in a mostly depth-first manner so that inherited protocol can be understood in conjunction with the new protocol.