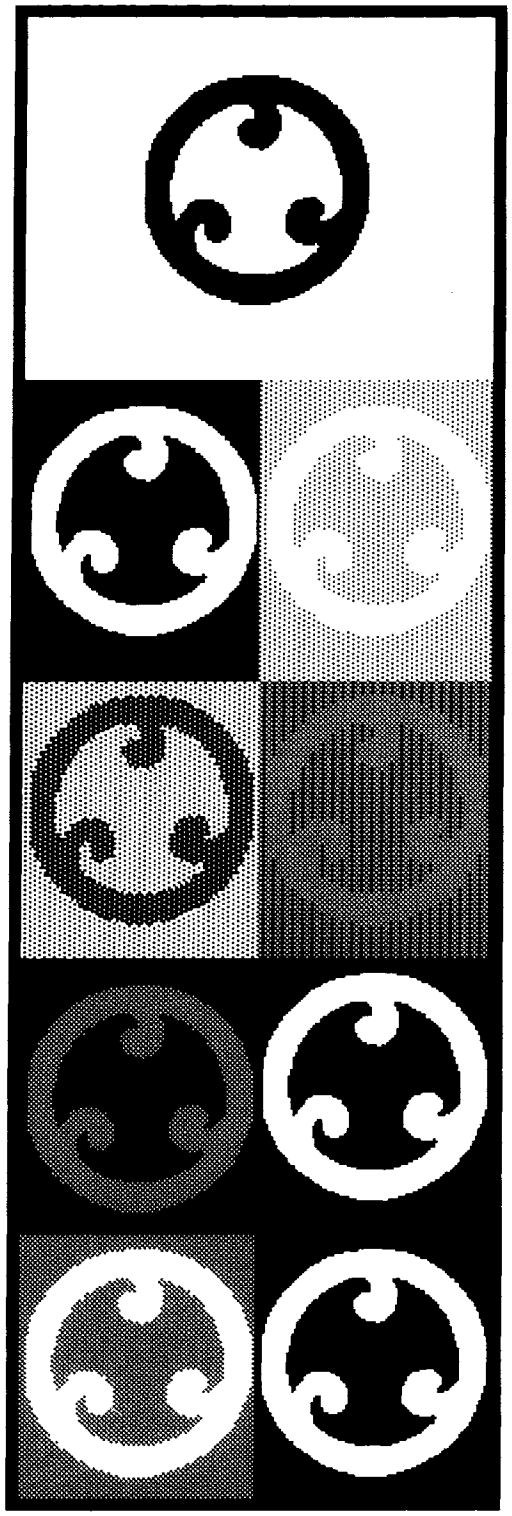


1

Objects and Messages

An *object* represents a component of the Smalltalk-80 software system. For example, objects represent

- numbers
- character strings
- queues
- dictionaries
- rectangles
- file directories
- text editors
- programs
- compilers
- computational processes
- financial histories
- views of information

An object consists of some private memory and a set of operations. The nature of an object's operations depends on the type of component it represents. Objects representing numbers compute arithmetic functions. Objects representing data structures store and retrieve information. Objects representing positions and areas answer inquiries about their relation to other positions and areas.

A *message* is a request for an object to carry out one of its operations. A message specifies which operation is desired, but not how that operation should be carried out. The *receiver*, the object to which the message was sent, determines how to carry out the requested operation. For example, addition is performed by sending a message to an object representing a number. The message specifies that the desired operation is addition and also specifies what number should be added to the receiver. The message does not specify how the addition will be performed. The receiver determines how to accomplish the addition. Computing is viewed as an intrinsic capability of objects that can be uniformly invoked by sending messages.

The set of messages to which an object can respond is called its *interface* with the rest of the system. The only way to interact with an object is through its interface. A crucial property of an object is that its private memory can be manipulated only by its own operations. A crucial property of messages is that they are the only way to invoke an object's operations. These properties insure that the implementation of one ob-

ject cannot depend on the internal details of other objects, only on the messages to which they respond.

Messages insure the modularity of the system because they specify the type of operation desired, but not how that operation should be accomplished. For example, there are several representations of numerical values in the Smalltalk-80 system. Fractions, small integers, large integers, and floating point numbers are represented in different ways. They all understand the same message requesting the computation of their sum with another number, but each representation implies a different way to compute that sum. To interact with a number or any object, one need only know what messages it responds to, not how it is represented.

Other programming environments also use objects and messages to facilitate modular design. For example, Simula uses them for describing simulations and Hydra uses them for describing operating system facilities in a distributed system. In the Smalltalk-80 system, objects and messages are used to implement the entire programming environment. Once objects and messages are understood, the entire system becomes accessible.

An example of a commonly-used data structure in programming is a dictionary, which associates names and values. In the Smalltalk-80 system, a dictionary is represented by an object that can perform two operations: associate a name with a new value, and find the value last associated with a particular name. A programmer using a dictionary must know how to specify these two operations with messages. Dictionary objects understand messages that make requests like "associate the name *Brett* with the value *3*" and "what is the value associated with the name *Dave*?" Since everything is an object, the names, such as *Brett* or *Dave*, and the values, such as *3* or *30*, are also represented by objects. Although a curious programmer may want to know how associations are represented in a dictionary, this internal implementation information is unnecessary for successful use of a dictionary. Knowledge of a dictionary's implementation is of interest only to the programmer who works on the definition of the dictionary object itself.

An important part of designing Smalltalk-80 programs is determining which kinds of objects should be described and which message names provide a useful vocabulary of interaction among these objects. A language is designed whenever the programmer specifies the messages that can be sent to an object. Appropriate choice of objects depends, of course, on the purposes to which the object will be put and the granularity of information to be manipulated. For example, if a simulation of an amusement park is to be created for the purpose of collecting data on queues at the various rides, then it would be useful to describe objects representing the rides, workers who control the rides, the waiting lines, and the people visiting the park. If the purpose of the simula-

tion includes monitoring the consumption of food in the park, then objects representing these consumable resources are required. If the amount of money exchanged in the park is to be monitored, then details about the cost of rides have to be represented.

In designing a Smalltalk-80 application, then, choice of objects is the first key step. There really is nothing definitive to say about the "right way" to choose objects. As in any design process, this is an acquired skill. Different choices provide different bases for extending an application or for using the objects for other purposes. The skilled Smalltalk-80 programmer is mindful that the objects created for an application might prove more useful for other applications if a semantically complete set of functions for an object is specified. For example, a dictionary whose associations can be removed as well as added is generally more useful than an add-only version.

Classes and Instances

A *class* describes the implementation of a set of objects that all represent the same kind of system component. The individual objects described by a class are called its *instances.* A class describes the form of its instances' private memories and it describes how they carry out their operations. For example, there is a system class that describes the implementation of objects representing rectangular areas. This class describes how the individual instances remember the locations of their areas and also how the instances carry out the operations that rectangular areas perform. Every object in the Smalltalk-80 system is an instance of a class. Even an object that represents a unique system component is implemented as the single instance of a class. Programming in the Smalltalk-80 system consists of creating classes, creating instances of classes, and specifying sequences of message exchanges among these objects.

The instances of a class are similar in both their public and private properties. An object's public properties are the messages that make up its interface. All instances of a class have the same message interface since they represent the same kind of component. An object's private properties are a set of *instance variables* that make up its private memory and a set of *methods* that describe how to carry out its operations. The instance variables and methods are not directly available to other objects. The instances of a class all use the same set of methods to describe their operations. For example, the instances that represent rectangles all respond to the same set of messages and they all use the same methods to determine how to respond. Each instance has its own set of instance variables, but they generally all have the same number

of instance variables. For example, the instances that represent rectangles all have two instance variables.

Each class has a name that describes the type of component its instances represent. Class names will appear in a special font because they are part of the programming language. The same font will be used for all text that represents Smalltalk-80 expressions. The class whose instances represent character sequences is named String. The class whose instances represent spatial locations is named Point. The class whose instances represent rectangular areas is named Rectangle. The class whose instances represent computational processes is named Process.

Each instance variable in an object's private memory refers to one object, called its value. The values of a Rectangle's two instance variables are instances of Point that represent opposing corners of its rectangular area. The fact that Rectangles have two instance variables, or that those instance variables refer to Points is strictly internal information, unavailable outside the individual Rectangle.

Each method in a class tells how to perform the operation requested by a particular type of message. When that type of message is sent to any instance of the class, the method is executed. The methods used by all Rectangles describe how to perform their operations in terms of the two Points representing opposing corners. For example, one message asks a Rectangle for the location of its center. The corresponding method tells how to calculate the center by finding the point halfway between the opposing corners.

A class includes a method for each type of operation its instances can perform. A method may specify some changes to the object's private memory and/or some other messages to be sent. A method also specifies an object that should be returned as the value of the invoking message. An object's methods can access the object's own instance variables, but not those of any other objects. For example, the method a Rectangle uses to compute its center has access to the two Points referred to by its instance variables; however, the method cannot access the instance variables of those Points. The method specifies messages to be sent to the Points asking them to perform the required calculations.

A small subset of the methods in the Smalltalk-80 system are not expressed in the Smalltalk-80 programming language. These are called *primitive* methods. The primitive methods are built into the virtual machine and cannot be changed by the Smalltalk-80 programmer. They are invoked with messages in exactly the same way that other methods are. Primitive methods allow the underlying hardware and virtual machine structures to be accessed. For example, instances of Integer use a primitive method to respond to the message +. Other primitive methods perform disk and terminal interactions.

An Example Application

Examples are an important part of the description of a programming language and environment. Many of the examples used in this book are taken from the classes found in the standard Smalltalk-80 system. Other examples are taken from classes that might be added to the system to enhance its functionality. The first part of the book draws examples from an application that might be added to the system to maintain simple financial histories for individuals, households, clubs, or small businesses. The full application allows information about financial transactions to be entered and views of that information to be displayed. Figure 1.1 shows a view of a financial history as it might appear on a Smalltalk-80 display screen. The top two parts of the view show two views of the amount of money spent for various reasons. The next view down shows how the cash-on-hand fluctuated over time as transactions were made.

At the bottom of the picture are two areas in which the user can type in order to add new expenditures and incomes to the history. When new information is added, the three views are automatically updated. In Figure 1.2, a new expenditure for food has been added.

This application requires the addition of several classes to the system. These new classes describe the different kinds of view as well as the underlying financial history information. The class that actually records the financial information is named FinancialHistory and will be used as an example in the next four chapters. This example application will make use of several classes already in the system; it will use numbers to represent amounts of money and strings to represent the reasons for expenditures and the sources of income.

FinancialHistory is used to introduce the basic concepts of the Smalltalk-80 programming language because its functionality and implementation are easy to describe. The functionality of a class can be specified by listing the operations available through its message interface. FinancialHistory provides six operations:

1. Create a new financial history object with a certain initial amount of money available.
2. Remember that a certain amount was spent for a particular reason.
3. Remember that a certain amount was received from a particular source.
4. Find out how much money is available.
5. Find out how much has been spent for a particular reason.
6. Find out how much has been received from a particular source.

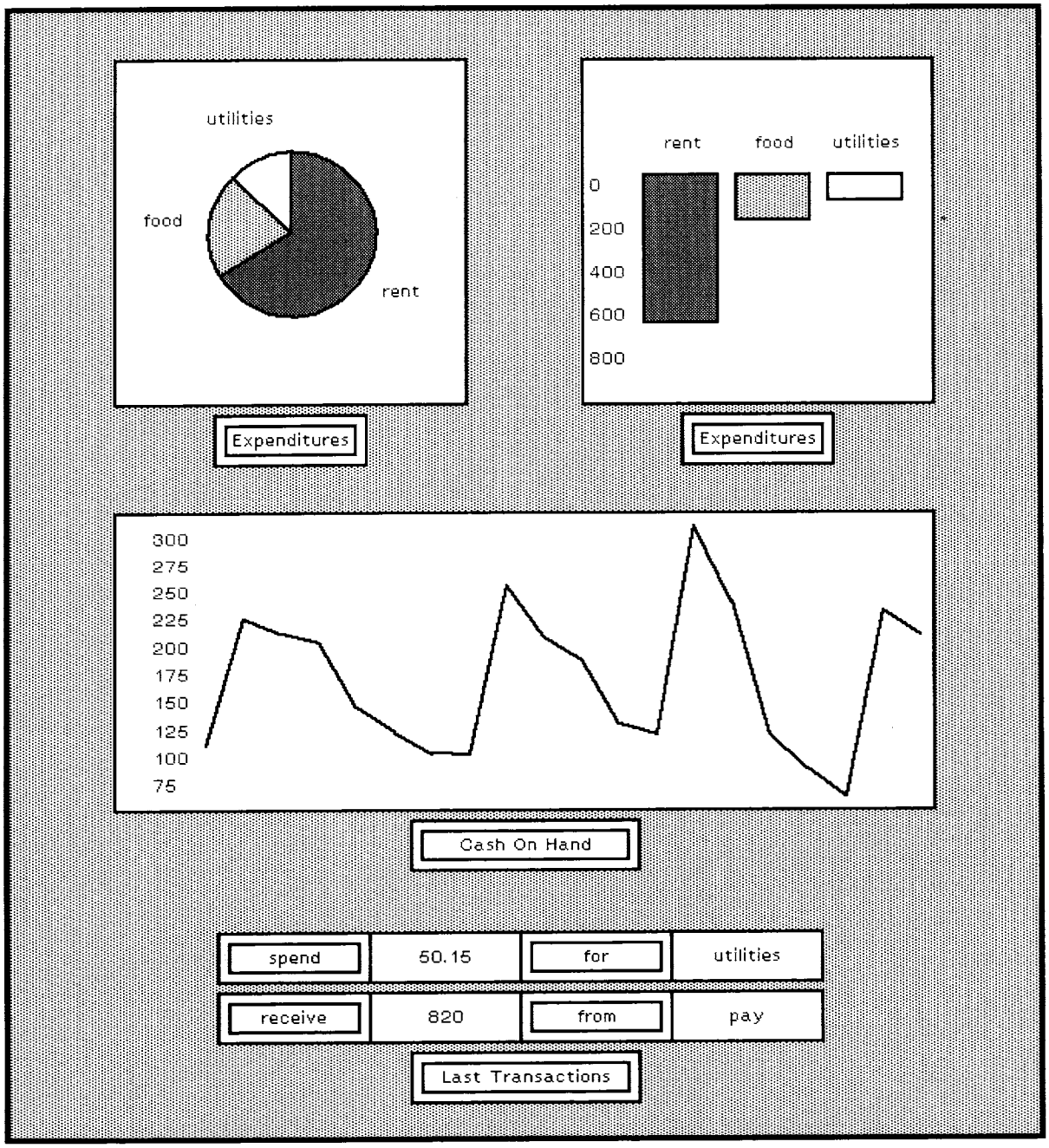

utilities
food
rent
Expenditures
rent
food
utilities
0
200
400
600
800
Expenditures
300
275
250
225
200
175
150
125
100
75
Cash On Hand
spend
50.15
for
utilities
receive
820
from
pay
Last Transactions

Figure 1.1

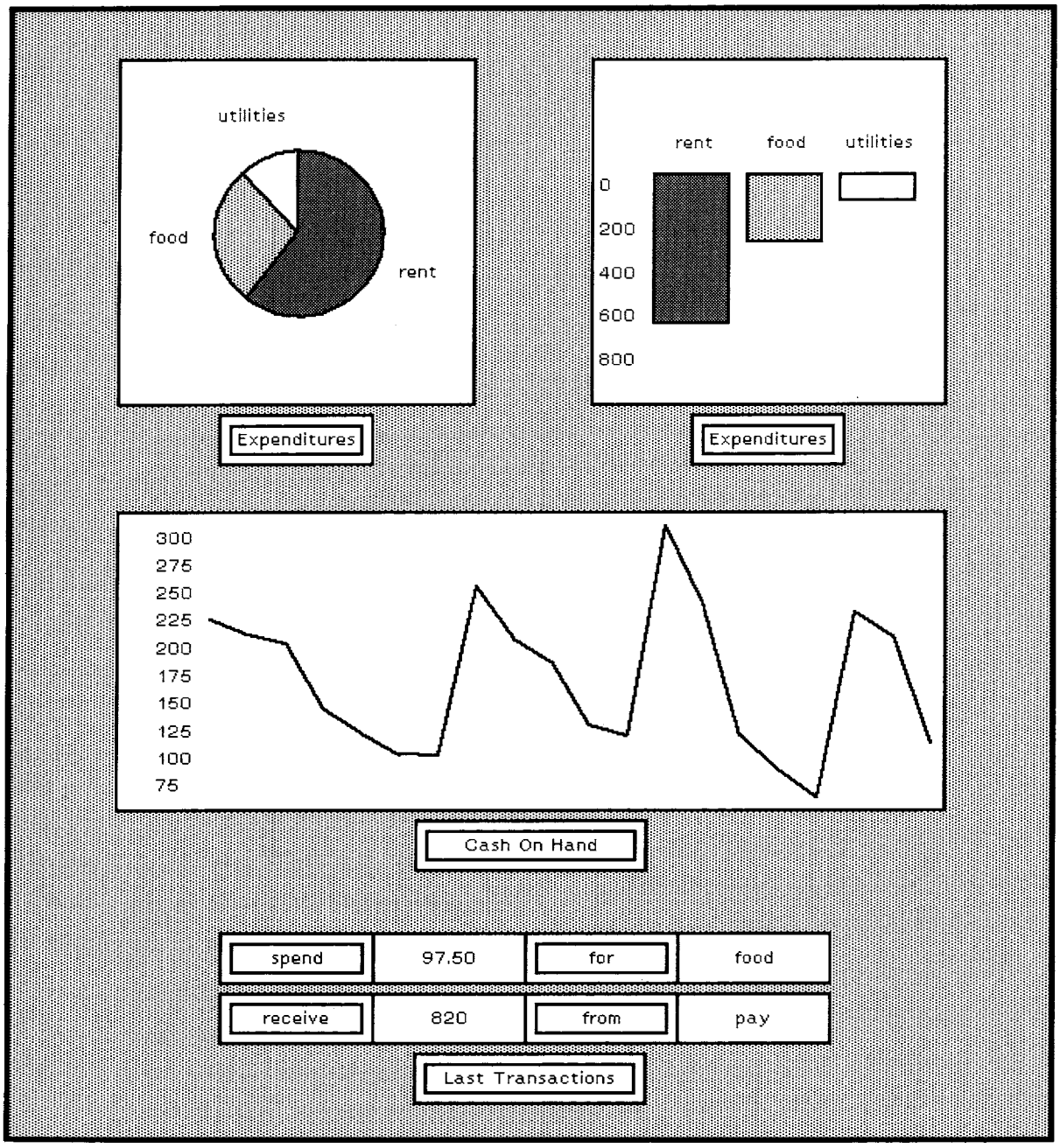

utilities
food
rent
Expenditures
rent
food
utilities
0
200
400
600
800
Expenditures
300
275
250
225
200
175
150
125
100
75
Cash On Hand
spend
97.50
for
food
receive
820
from
pay
Last Transactions

Figure 1.2

An implementation of these operations is specified in the class description shown inside the front cover of this book. The form of class descriptions will be described in Chapters 3, 4, and 5.

System Classes

The Smalltalk-80 system includes a set of classes that provides the standard functionality of a programming language and environment: arithmetic, data structures, control structures, and input/output facilities. The functionality of these classes will be specified in detail in Part Two of this book. Figure 1.3 is a diagram of the system classes presented in Part Two. Lines are drawn around groups of related classes; the groups are labeled to indicate the chapters in which the specifications of the classes can be found.

Arithmetic

The Smalltalk-80 system includes objects representing both real and rational numbers. Real numbers can be represented with an accuracy of about six digits. Integers with absolute value less than 2^{524288} can be represented exactly. Rational numbers can be represented using these integers. There are also classes for representing linear magnitudes (like dates and times) and random number generators.

Data Structures

Most of the objects in the Smalltalk-80 system function as data structures of some kind. However, while most objects also have other functionality, there is a set of classes representing more or less pure data structures. These classes represent different types of collections. The elements of some collections are unordered while the elements of others are ordered. Of the collections with unordered elements, there are bags that allow duplicate elements and sets that don't allow duplication. There are also dictionaries that associate pairs of objects. Of the collections with ordered elements, some have the order specified externally when elements are added and others determine the order based on properties of the elements themselves. The common data structures of arrays and strings are provided by classes that have associative behavior (associating indices and elements) and external ordering (corresponding to the inherent ordering of the indices).

Control Structures

The Smalltalk-80 system includes objects and messages that implement the standard control structures found in most programming languages. They provide conditional selection similar to the if-then-else statements of Algol and conditional repetition similar to its while and until state-

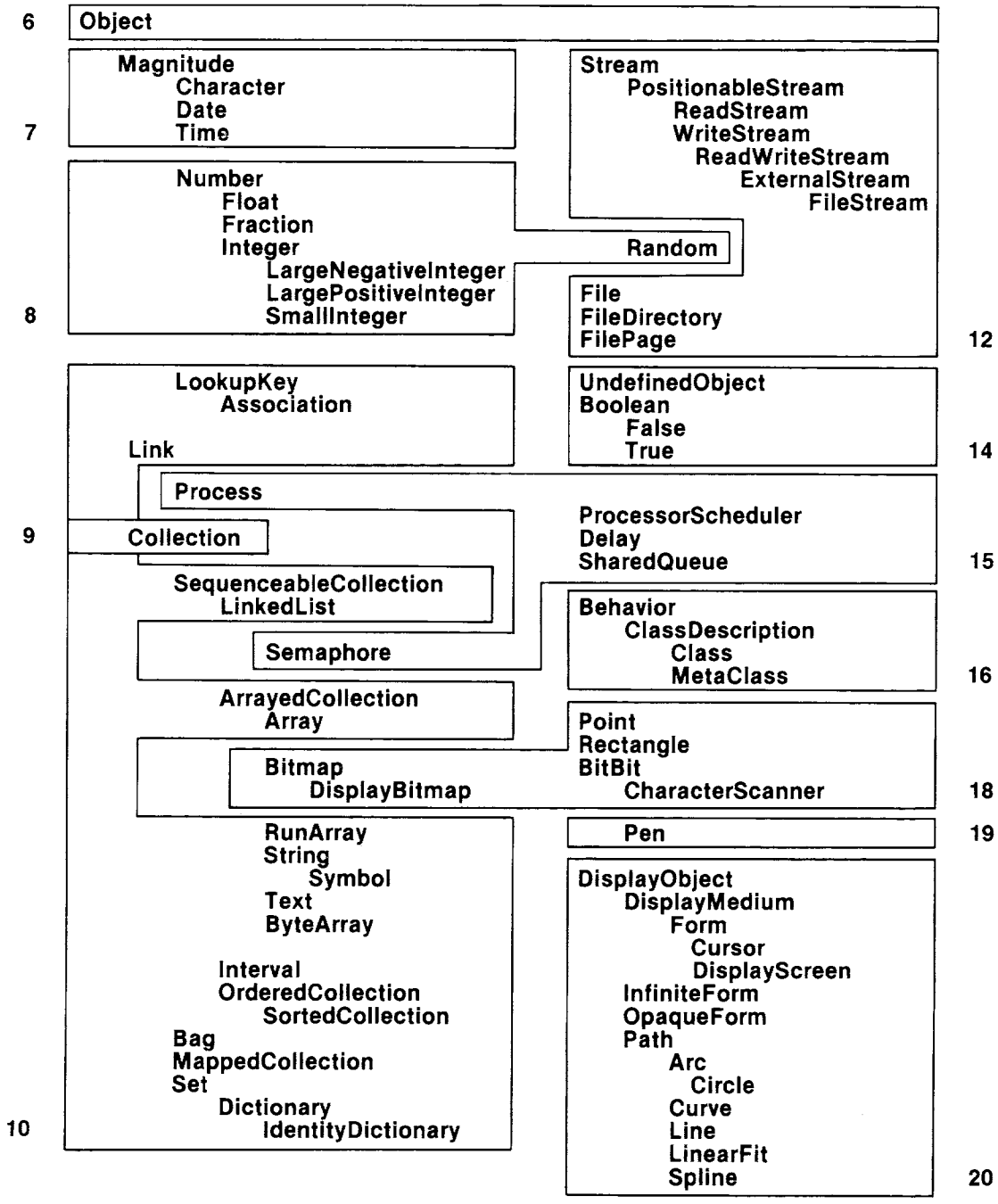

6
Object
Magnitude
Character
Date
7
Time
Number
Float
Fraction
Integer
LargeNegativeInteger
LargePositiveInteger
8
SmallInteger
Stream
PositionableStream
ReadStream
WriteStream
ReadWriteStream
ExternalStream
FileStream
Random
File
FileDirectory
FilePage
12
LookupKey
Association
Link
UndefinedObject
Boolean
False
True
14
Process
9
Collection
ProcessorScheduler
Delay
SharedQueue
15
SequenceableCollection
LinkedList
Semaphore
Behavior
ClassDescription
Class
MetaClass
16
ArrayedCollection
Array
Point
Rectangle
BitBit
Bitmap
DisplayBitmap
CharacterScanner
18
RunArray
String
Symbol
Text
ByteArray
Pen
19
Interval
OrderedCollection
SortedCollection
Bag
MappedCollection
Set
Dictionary
10
IdentityDictionary
DisplayObject
DisplayMedium
Form
Cursor
DisplayScreen
InfiniteForm
OpaqueForm
Path
Arc
Circle
Curve
Line
LinearFit
Spline
20

Figure 1.3

ments. Objects representing independent processes and mechanisms for scheduling and synchronous interaction are also provided. Two classes are provided to support these control structures. Booleans represent the two truth values and blocks represent sequences of actions. Booleans and blocks can also be used to create new kinds of control structures.

Programming Environment

There are several classes in the Smalltalk-80 system that assist in the programming process. There are separate classes representing the source (human-readable) form and the compiled (machine-executable) form of methods. Objects representing parsers, compilers, and decompilers translate between the two forms of method. Objects representing classes connect methods with the objects that use them (the instances of the classes).

Objects representing organizational structures for classes and methods help the programmer keep track of the system, and objects representing histories of software modification help interface with the efforts of other programmers. Even the execution state of a method is represented by an object. These objects are called *contexts* and are analogous to stack frames or activation records of other systems.

Viewing and Interacting

The Smalltalk-80 system includes classes of objects that can be used to view and edit information. Classes helpful for presenting graphical views represent points, lines, rectangles, and arcs. Since the Smalltalk-80 system is oriented toward a bitmap display, there are classes for representing and manipulating bitmap images. There are also classes for representing and manipulating the more specific use of bitmap images for character fonts, text, and cursors.

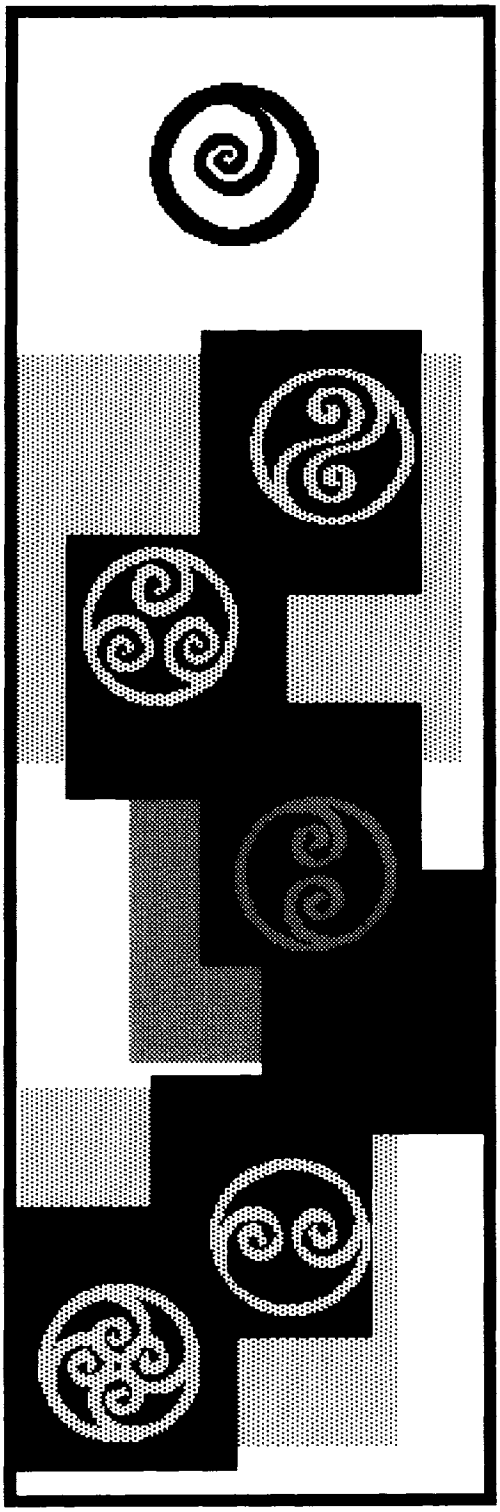
Built from these graphical objects are other objects representing rectangular windows, command menus, and content selections. There are also objects that represent the user's actions on the input devices and how these relate to the information being viewed. Classes representing specific viewing and editing mechanisms constructed from these components provide views for classes, contexts, and documents containing text and graphics. The views of classes provide the fundamental mechanism to interact with the software in the system. Smalltalk-80 views and editors are presented in a separate book.

Communication

The Smalltalk-80 system allows communication with external media. The standard external medium is a disk file system. Objects represent individual files as well as directories. If a connection to a communications network is available, it can be accessed through objects as well.

Summary of Terminology

object	A component of the Smalltalk-80 system represented by some private memory and a set of operations.
message	A request for an object to carry out one of its operations.
receiver	The object to which a message is sent.
interface	The messages to which an object can respond.
class	A description of a group of similar objects.
instance	One of the objects described by a class.
instance variable	A part of an object's private memory.
method	A description of how to perform one of an object's operations.
primitive method	An operation performed directly by the Smalltalk-80 virtual machine.
FinancialHistory	The name of a class used as an example in this book.
system classes	The set of classes that come with the Smalltalk-80 system.



2

Expression Syntax

Chapter 1 introduced the fundamental concepts of the Smalltalk-80 system. System components are represented by *objects.* Objects are *instances* of *classes.* Objects interact by sending *messages.* Messages cause *methods* to be executed. This chapter introduces an expression syntax for describing objects and messages. The next chapter introduces a syntax for describing classes and methods.

An *expression* is a sequence of characters that describes an object called the value of the expression. The syntax presented in this chapter explains which sequences of characters form legal expressions. There are four types of expression in the Smalltalk-80 programming language.

1. *Literals* describe certain constant objects, such as numbers and character strings.
2. *Variable names* describe the accessible variables. The value of a variable name is the current value of the variable with that name.
3. *Message expressions* describe messages to receivers. The value of a message expression is determined by the method the message invokes. That method is found in the class of the receiver.
4. *Block expressions* describe objects representing deferred activities. Blocks are used to implement control structures.

Expressions are found in two places, in methods and in text displayed on the screen. When a message is sent, a method from the receiver's class is selected and its expressions are evaluated. Part of the user interface allows expressions to be selected on the screen and evaluated. The details of selecting and evaluating expressions on the screen fall outside the scope of this book, since they are part of the user interface. Some examples, however, are given in Chapter 17.

Of the four types of expression listed above, only the variable names are context-dependent. An expression's location in the system determines which character sequences are legal variable names. The set of variable names available in a method's expressions depends on the class in which the method is found. For example, methods in class Rectangle and methods in class Point have access to different sets of variable names. The variables available in a class's methods will be fully described in Chapters 3, 4, and 5. The variable names available for use in expressions on the screen depend on where the expressions are displayed on the screen. All other aspects of the expression syntax are independent of the expression's location.

The syntax for expressions is summarized in the diagram that appears inside the back cover of this book. The rest of this chapter describes the four types of expression.

Literals

Five kinds of objects can be referred to by literal expressions. Since the value of a literal expression is always the same object, these expressions are also called *literal constants.* The five types of literal constant are:

1. numbers
2. individual characters
3. strings of characters
4. symbols
5. arrays of other literal constants

Numbers

Numbers are objects that represent numerical values and respond to messages that compute mathematical results. The literal representation of a number is a sequence of digits that may be preceded by a minus sign and/or followed by a decimal point and another sequence of digits. For example,

```
3
30.45
-3
0.005
-14.0
13772
```

Number literals can also be expressed in a nondecimal base by preceding the digits with a radix prefix. The radix prefix includes the value of the digit radix (always expressed in decimal) followed by the letter "r". The following examples specify numbers in octal with their corresponding decimal values.

octal	*decimal*
8r377	255
8r153	107
8r34.1	28.125
8r-37	-31

When the base is greater than ten, the capital letters starting with "A" are used for digits greater than nine. The following examples specify numbers in hexadecimal with their corresponding decimal values.

hexadecimal	*decimal*
16r106	262
16rFF	255
16rAC.DC	172.859
16r−1.C	−1.75

Number literals can also be expressed in scientific notation by following the digits of the value with an exponent suffix. The exponent suffix includes the letter "e" followed by the exponent (expressed in decimal). The number specified before the exponent suffix is multiplied by the radix raised to the power specified by the exponent.

scientific notation	*decimal*
1.586e5	158600.0
1.586e−3	0.001586
8r3e2	192
2r11e6	192

Characters

Characters are objects that represent the individual symbols of an alphabet. A character literal expression consists of a dollar sign followed by any character, for example,

```
$a
$M
$-
$$
$1
```

Strings

Strings are objects that represent sequences of characters. Strings respond to messages that access individual characters, replace substrings, and perform comparisons with other strings. The literal representation of a string is a sequence of characters delimited by single quotes, for example,

```
'hi'
'food'
'the Smalltalk-80 system'
```

Any character may be included in a string literal. If a single quote is to be included in a string, it must be duplicated to avoid confusion with the delimiters. For example, the string literal

```
'can"t'
```

refers to a string of the five characters $c, $a, $n, $', and $t.

Symbols

Symbols are objects that represent strings used for names in the system. The literal representation of a symbol is a sequence of alphanumeric characters preceded by a pound sign, for example,

```
#bill
#M63
```

There will never be two symbols with the same characters; each symbol is unique. This makes it possible to compare symbols efficiently.

Arrays

An array is a simple data structure object whose contents can be referenced by an integer index from one to a number that is the size of the array. Arrays respond to messages requesting access to their contents. The literal representation of an array is a sequence of other literals—numbers, characters, strings, symbols, and arrays—delimited by parentheses and preceded by a pound sign. The other literals are separated by spaces. Embedded symbols and arrays are not preceded by pound signs. An array of three numbers is described by the expression

```
#(1 2 3)
```

An array of seven strings is described by the expression

```
#('food' 'utilities' 'rent' 'household' 'transportation' 'taxes' 'recreation')
```

An array of two arrays and two numbers is described by the expression

```
#(('one' 1) ('not' 'negative') 0 -1)
```

And an array of a number, a string, a character, a symbol, and another array is described by the expression

```
#(9 'nine' $9 nine (0 'zero' $0 ( ) 'e' $f 'g' $h 'i'))
```

Variables

The memory available to an object is made up of variables. Most of these variables have names. Each variable remembers a single object and the variable's name can be used as an expression referring to that object. The objects that can be accessed from a particular place are determined by which variable names are available. For example, the con-

tents of an object's instance variables are unavailable to other objects because the names of those variables can be used only in the methods of the object's class.

A variable name is a simple identifier, a sequence of letters and digits beginning with a letter. Some examples of variable names are:

```
index
initialIndex
textEditor
bin14
bin14Total
HouseholdFinances
Rectangle
IncomeReasons
```

There are two kinds of variables in the system, distinguished by how widely they are accessible. Private variables are accessible only to a single object. Instance variables are private. Shared variables can be accessed by more than one object. Private variable names are required to have lowercase initial letters; shared variable names are required to have uppercase initial letters. The first five example identifiers shown above refer to private variables and the last three refer to shared variables.

Another capitalization convention evident in the examples above is that identifiers formed by concatenating several words capitalize each word following the first one. This convention is not enforced by the system.

Assignments

A literal constant will always refer to the same object, but a variable name may refer to different objects at different times. The object referred to by a variable is changed when an assignment expression is evaluated. Assignments were not listed earlier as a type of expression since any expression can become an assignment by including an assignment prefix.

An assignment prefix is composed of the name of the variable whose value will be changed followed by a left arrow (←). The following example is a literal expression that has an assignment prefix. It indicates that the variable named quantity should now refer to the object representing the number 19.

```
quantity ← 19
```

The following example is a variable-name expression with an assignment prefix. It indicates that the variable named index should refer to the same object as does the variable named initialIndex.

```
index ← initialIndex
```

Other examples of assignment expressions are:

```
chapterName ← 'Expression Syntax'
flavors ← #('vanilla' 'chocolate' 'butter pecan' 'garlic')
```

More than one assignment prefix can be included, indicating that the values of several variables are changed.

```
index ← initialIndex ← 1
```

This expression indicates that both the variables named index and initialIndex should refer to the number 1. Message expressions and block expressions can also have assignment prefixes, as will be seen in the following sections.

Pseudo-variable Names

A pseudo-variable name is an identifier that refers to an object. In this way, it is similar to a variable name. A pseudo-variable name is different from a variable name in that its value cannot be changed with an assignment expression. Some of the pseudo-variables in the system are constants; they always refer to the same objects. Three important pseudo-variable names are nil, true, and false.

nil	refers to an object used as the value of a variable when no other object is appropriate. Variables that have not been otherwise initialized refer to nil.
true	refers to an object that represents logical accuracy. It is used as an affirmative response to a message making a simple yes-no inquiry.
false	refers to an object that represents logical inaccuracy. It is used as a negative response to a message making a simple yes-no inquiry.

The objects named true and false are called *Boolean* objects. They represent the answers to yes-no questions. For example, a number will respond with true or false to a message asking whether or not the number is greater than another number. Boolean objects respond to messages that compute logical functions and perform conditional control structures.

There are other pseudo-variables in the system (for example, self and super) whose values are different depending on where they are used. These will be described in the next three chapters.

Messages

Messages represent the interactions between the components of the Smalltalk-80 system. A message requests an operation on the part of the receiver. Some examples of message expressions and the interactions they represent follow.

Messages to numbers representing arithmetic operations

3 + 4	computes the sum of three and four.
index + 1	adds one to the number named index.
index > limit	inquires whether or not the number named index is greater than the number named limit.
theta sin	computes the sine of the number named theta.
quantity sqrt	computes the positive square root of the number named quantity.

Messages to linear data structures representing the addition or removal of information

list addFirst: newComponent
: adds the object named newComponent as the first element of the linear data structure named list.

list removeLast
: removes and returns the last element in list.

Messages to associative data structures (such as dictionaries) representing the addition or removal of information

ages at: 'Brett Jorgensen' put: 3
: associates the string 'Brett Jorgensen' with the number 3 in the dictionary named ages.

addresses at: 'Peggy Jorgensen'
: looks up the object associated with the string 'Peggy Jorgensen' in the dictionary named addresses.

Messages to rectangles representing graphical inquiries and calculations

frame center
: answers the position of the center of the rectangle named frame.

frame containsPoint: cursorLocation
: answers true if the position named cursorLocation is inside the rectangle named frame, and false otherwise.

frame intersect: clippingBox
computes the rectangle that represents the intersection of the two rectangles named frame and clippingBox.

Messages to financial history records representing transactions and inquiries

HouseholdFinances spend: 32.50 on: 'utilities'
informs the financial history named HouseholdFinances that $32.50 has been spent on utility bills.

HouseholdFinances totalSpentFor: 'food'
asks HouseholdFinances how much money has been spent for food.

Selectors and Arguments

A message expression describes a receiver, *selector*, and possibly some *arguments*. The receiver and arguments are described by other expressions. The selector is specified literally.

A message's selector is a name for the type of interaction the sender desires with the receiver. For example, in the message

```
theta sin
```

the receiver is a number referred to by the variable named theta and the selector is sin. It is up to the receiver to decide how to respond to the message (in this case, how to compute the sine function of its value).

In the two message expressions

```
3 + 4
```

and

```
previousTotal + increment
```

the selectors are +. Both messages ask the receiver to calculate and return a sum. These messages each contain an object in addition to the selector (4 in the first expression and increment in the second). The additional objects in the message are arguments that specify the amount to be added.

The following two message expressions describe the same kind of operation. The receiver is an instance of FinancialHistory and will return the amount of money spent for a particular reason. The argument indicates the reason of interest. The first expression requests the amount spent for utility bills.

```
HouseholdFinances totalSpentOn: 'utilities'
```

The amount spent for food can be found by sending a message with the same selector, but with a different argument.

```
HouseholdFinances totalSpentOn: 'food'
```

The selector of a message determines which of the receiver's operations will be invoked. The arguments are other objects that are involved in the selected operation.

☐ *Unary Messages* Messages without arguments are called *unary messages.* For example, the money currently available according to HouseholdFinances is the value of the unary message expression

```
HouseholdFinances cashOnHand
```

These messages are called unary because only one object, the receiver, is involved. A unary message selector can be any simple identifier. Other examples of unary message expressions are

```
theta sin
quantity sqrt
nameString size
```

☐ *Keyword Messages* The general type of message with one or more arguments is the *keyword message.* The selector of a keyword message is composed of one or more keywords, one preceding each argument. A keyword is a simple identifier with a trailing colon. Examples of expressions describing single keyword messages are

```
HouseholdFinances totalSpentOn: 'utilities'
index max: limit
```

A message with two arguments will have a selector with two keywords. Examples of expressions describing double keyword messages are

```
HouseholdFinances spend: 30.45 on: 'food'
ages at: 'Brett Jorgensen' put: 3
```

When the selector of a multiple keyword message is referred to independently, the keywords are concatenated. The selectors of the last two message expressions are spend:on: and at:put:. There can be any num-

ber of keywords in a message, but most messages in the system have fewer than three.

☐ *Binary Messages* There is one other type of message expression that takes a single argument, the *binary message.* A binary message selector is composed of one or two nonalphanumeric characters. The only restriction is that the second character cannot be a minus sign. Binary selectors tend to be used for arithmetic messages. Examples of binary message expressions are

```
3 + 4
total − 1
total <= max
```

Returning Values

Smalltalk-80 messages provide two-way communication. The selector and arguments transmit information to the receiver about what type of response to make. The receiver transmits information back by returning an object that becomes the value of the message expression. If a message expression includes an assignment prefix, the object returned by the receiver will become the new object referred to by the variable. For example, the expression

```
sum ← 3 + 4
```

makes 7 be the new value of the variable named sum. The expression

```
x ← theta sin
```

makes the sine of theta be the new value of the variable named x. If the value of theta is 1, the new value of x becomes 0.841471. If the value of theta is 1.5, the new value of x becomes 0.997495.

The number referred to by index can be incremented by the expression

```
index ← index + 1
```

Even if no information needs to be communicated back to the sender, a receiver *always* returns a value for the message expression. Returning a value indicates that the response to the message is complete. Some messages are meant only to inform the receiver of something. Examples are the messages to record financial transactions described by the following expressions.

```
HouseholdFinances spend: 32.50 on: 'utilities'
HouseholdFinances receive: 1000 from: 'pay'
```

The receiver of these messages informs the sender only that it is finished recording the transaction. The default value returned in such cases is usually the receiver itself. So, the expression

```
var ← HouseholdFinances spend: 32.50 on: 'utilities'
```

results in var referring to the same financial history as HouseholdFinances.

Parsing

All of the message expressions shown thus far have described the receiver and arguments with literals or variable names. When the receiver or argument of a message expression is described by another message expression, the issue of how the expression is parsed arises. An example of a unary message describing the receiver of another unary message is

```
1.5 tan rounded
```

Unary messages are parsed left to right. The first message in the example is the unary selector tan sent to 1.5. The value of that message expression (a number around 14.1014) receives the unary message rounded and returns the nearest integer, 14. The number 14 is the value of the whole expression.

Binary messages are also parsed left to right. An example of a binary message describing the receiver of another binary message is

```
index + offset * 2
```

The value returned by index from the message + offset is the receiver for the binary message * 2.

All binary selectors have the same precedence; only the order in which they are written matters. Note that this makes mathematical expressions in the Smalltalk-80 language different from those in many other languages in which multiplication and division take precedence over addition and subtraction.

Parentheses can be used to change the order of evaluation. A message within parentheses is sent before any messages outside the parentheses. If the previous example were written as

```
index + (offset * 2)
```

the multiplication would be performed before the addition.

Unary messages take precedence over binary messages. If unary messages and binary messages appear together, the unary messages will all be sent first. In the example

```
frame width + border width * 2
```

the value of frame width is the receiver of the binary message whose selector is + and whose argument is the value of border width. The value of the + message is, in turn, the receiver of the binary message * 2. The expression parses as if it had been parenthesized as follows:

```
((frame width) + (border width)) * 2
```

Parentheses can be used to send binary messages before unary messages. The expression

```
2 * theta sin
```

calculates twice the sine of theta, while the expression

```
(2 * theta) sin
```

calculates the sine of twice theta.

Whenever keywords appear in an unparenthesized message, they compose a single selector. Because of this concatenation, there is no left-to-right parsing rule for keyword messages. If a keyword message is to be used as a receiver or argument of another keyword message, it must be parenthesized. The expression

```
frame scale: factor max: 5
```

describes a single two-argument keyword message whose selector is scale:max:. The expression

```
frame scale: (factor max: 5)
```

describes two single keyword messages whose selectors are scale: and max:. The value of the expression factor max: 5 is the argument for the scale: message to frame.

Binary messages take precedence over keyword messages. When unary, binary, and keyword messages appear in the same expression without parentheses, the unaries are sent first, the binaries next, and the keyword last. The example

```
bigFrame width: smallFrame width * 2
```

is evaluated as if it had been parenthesized as follows:

```
bigFrame width: ((smallFrame width) * 2)
```

In the following example, a unary message describes the receiver of a keyword message and a binary message describes the argument.

```
OrderedCollection new add: value * rate
```

To summarize the parsing rules:

1. Unary expressions parse left to right.
2. Binary expressions parse left to right.
3. Binary expressions take precedence over keyword expressions.
4. Unary expressions take precedence over binary expressions.
5. Parenthesized expressions take precedence over unary expressions.

Formatting Conventions

A programmer is free to format expressions in various ways using spaces, tabs, and carriage returns. For example, multiple keyword messages are often written with each keyword-argument pair on a different line, as in

```
ages at: 'Brett Jorgensen'
      put: 3
```

or

```
HouseholdFinances
        spend: 30.45
        on: 'food'
```

The only time that a space, tab, or carriage return affects the meaning of an expression is when its absence would cause two letters or two numbers to fall next to each other.

Cascading

There is one special syntactic form called *cascading* that specifies multiple messages to the same object. Any sequence of messages can be expressed without cascading. However, cascading often reduces the need for using variables. A cascaded message expression consists of one description of the receiver followed by several messages separated by semicolons. For example,

```
OrderedCollection new add: 1; add: 2; add: 3
```

Three add: messages are sent to the result of OrderedCollection new. Without cascading, this would have required four expressions and a

variable. For example, the following four expressions, separated by periods, have the same result as the cascaded expression above.

```
temp ← OrderedCollection new.
temp add: 1.
temp add: 2.
temp add: 3
```

Blocks

Blocks are objects used in many of the control structures in the Smalltalk-80 system. A block represents a deferred sequence of actions. A block expression consists of a sequence of expressions separated by periods and delimited by square brackets. For example,

```
[index ← index + 1]
```

or

```
[index ← index + 1.
 array at: index put: 0]
```

If a period follows the last expression, it is ignored, as in

```
[expenditures at: reason.]
```

When a block expression is encountered, the statements enclosed in the brackets are not executed immediately. The value of a block expression is an object that can execute these enclosed expressions at a later time, when requested to do so. For example, the expression

```
actions at: 'monthly payments'
        put: [HouseholdFinances spend: 650 on: 'rent'.
              HouseholdFinances spend: 7.25 on: 'newspaper'.
              HouseholdFinances spend: 225.50 on: 'car payment']
```

does not actually send any spend:on: messages to HouseholdFinances. It simply associates a block with the string 'monthly payments'.

The sequence of actions a block describes will take place when the block receives the unary message value. For example, the following two expressions have identical effects.

```
index ← index + 1
```

and

```
[index ← index + 1] value
```

The object referred to by the expression

```
actions at: 'monthly payments'
```

is the block containing three spend:on: messages. The execution of the expression

```
(actions at: 'monthly payments') value
```

results in those three spend:on: messages being sent to HouseholdFinances.

A block can also be assigned to a variable. So if the expression

```
incrementBlock ← [index ← index + 1]
```

is executed, then the expression

```
incrementBlock value
```

increments index.

The object returned after a value message is sent to a block is the value of the last expression in the sequence. So if the expression

```
addBlock ← [index + 1]
```

is executed, then another way to increment index is to evaluate

```
index ← addBlock value
```

A block that contains no expressions returns nil when sent the message value. The expression

```
[] value
```

has the value nil.

Control Structures

A control structure determines the order of some activities. The fundamental control structure in the Smalltalk-80 language provides that a sequence of expressions will be evaluated sequentially. Many

nonsequential control structures can be implemented with blocks. These control structures are invoked either by sending a message to a block or by sending a message with one or more blocks as arguments. The response to one of these control structure messages determines the order of activities with the pattern of value messages it sends to the block(s).

Examining the evaluation of the following sequence of expressions gives an example of the way blocks work.

```
incrementBlock ← [index ← index + 1].
sumBlock ← [sum + (index * index)].
sum ← 0.
index ← 1.
sum ← sumBlock value.
incrementBlock value.
sum ← sumBlock value
```

The 15 actions taken as a result of evaluating this sequence of expressions are

1. *Assign* a block to incrementBlock.
2. *Assign* a block to sumBlock.
3. *Assign* the number 0 to sum.
4. *Assign* the number 1 to index.
5. *Send* the message value to the block sumBlock.
6. *Send* the message * 1 to the number 1.
7. *Send* the message + 1 to the number 0.
8. *Assign* the number 1 to sum.
9. *Send* the message value to the block IncrementBlock.
10. *Send* the message + 1 to the number 1.
11. *Assign* the number 2 to index.
12. *Send* the message value to the block sumBlock.
13. *Send* the message * 2 to the number 2.
14. *Send* the message + 4 to the number 1.
15. *Assign* the number 5 to sum.

An example of a control structure implemented with blocks is simple repetition, represented by a message to an integer with timesRepeat: as the selector and a block as the argument. The integer will respond by sending the block as many value messages as its own value indicates. For example, the following expression doubles the value of the variable named amount four times.

```
4 timesRepeat: [amount ← amount + amount]
```

Conditionals

Two common control structures implemented with blocks are *conditional selection* and *conditional repetition.* Conditional selection is similar to the if-then-else statements in Algol-like languages and conditional repetition is similar to the while and until statements in those languages. These conditional control structures use the two Boolean objects named true and false described in the section on pseudo-variables. Booleans are returned from messages that ask simple yes-no questions (for example, the magnitude comparison messages: <, =, <=, >, >=, ~=).

☐ *Conditional Selection* The conditional selection of an activity is provided by a message to a Boolean with the selector ifTrue:ifFalse: and two blocks as arguments. The only objects that understand ifTrue:ifFalse: messages are true and false. They have opposite responses: true sends value to the first argument block and ignores the second; false sends value to the second argument block and ignores the first. For example, the following expression assigns 0 or 1 to the variable parity depending on whether or not the value of the variable number is divisible by 2. The binary message \\ computes the modulus or remainder function.

```
(number \\ 2) = 0
        ifTrue: [parity ← 0]
        ifFalse: [parity ← 1]
```

The value returned from ifTrue:ifFalse: is the value of the block that was executed. The previous example could also be written

```
parity ← (number \\ 2) = 0 ifTrue: [0] ifFalse: [1]
```

In addition to ifTrue:ifFalse:, there are two single-keyword messages that specify only one conditional consequent. The selectors of these messages are ifTrue: and ifFalse:. These messages have the same effect as the ifTrue:ifFalse: message when one argument is an empty block. For example, these two expressions have the same effect.

```
index <= limit
        ifTrue: [total ← total + (list at: index)]
```

and

```
index <= limit
        ifTrue: [total ← total + (list at: index)]
        ifFalse: []
```

Since the value of an empty block is nil, the following expression would set lastElement to nil if index is greater than limit.

```
lastElement ← index > limit ifFalse: [list at: index]
```

☐ *Conditional Repetition* The conditional repetition of an activity is provided by a message to a block with the selector whileTrue: and another block as an argument. The receiver block sends itself the message value and, if the response is true, it sends the other block value and then starts over, sending itself value again. When the receiver's response to value becomes false, it stops the repetition and returns from the whileTrue: message. For example, conditional repetition could be used to initialize all of the elements of an array named list.

```
index ← 1.
[index <= list size]
        whileTrue: [list at: index put: 0.
                    index ← index + 1]
```

Blocks also understand a message with selector whileFalse: that repeats the execution of the argument block as long as the value of the receiver is false. So, the following expressions are equivalent to the one above.

```
index ← 1.
[index > list size]
        whileFalse: [list at: index put: 0.
                     index ← index + 1]
```

The programmer is free to choose whichever message makes the intent of the repetition clearest. The value returned by both whileTrue: and whileFalse: is always nil.

Block Arguments

In order to make some nonsequential control structures easy to express, blocks may take one or more arguments. Block arguments are specified by including identifiers preceded by colons at the beginning of a block. The block arguments are separated from the expressions that make up the block by a vertical bar. The following two examples describe blocks with one argument.

```
[ :array |  total ← total + array size]
```

and

```
[ :newElement |
  index ← index + 1.
  list at: index put: newElement]
```

A common use of blocks with arguments is to implement functions to be applied to all elements of a data structure. For example, many ob-

jects representing different kinds of data structures respond to the message do:, which takes a single-argument block as its argument. The object that receives a do: message evaluates the block once for each of the elements contained in the data structure. Each element is made the value of the block argument for one evaluation of the block. The following example calculates the sum of the squares of the first five primes. The result is the value of sum.

```
sum ← 0.
#(2 3 5 7 11) do: [ :prime |  sum ← sum + (prime * prime)]
```

The message collect: creates a collection of the values produced by the block when supplied with the elements of the receiver. The value of the following expression is an array of the squares of the first five primes.

```
#(2 3 5 7 11) collect: [ :prime |  prime * prime]
```

The objects that implement these control structures supply the values of the block arguments by sending the block the message value:. A block with one block argument responds to value: by setting the block argument to the argument of value: and then executing the expressions in the block. For example, evaluating the following expressions results in the variable total having the value 7.

```
sizeAdder ← [ :array |  total ← total + array size].
total ← 0.
sizeAdder value: #(a b c).
sizeAdder value: #(1 2).
sizeAdder value: #(e f)
```

Blocks can take more than one argument. For example

```
[ :x :y |  (x * x) + (y * y)]
```

or

```
[ :frame :clippingBox |  frame intersect: clippingBox]
```

A block must have the same number of block arguments as the number of value: keywords in the message to evaluate it. The two blocks above would be evaluated by means of a two-keyword message with selector value:value:. The two arguments of the message specify the values of the two block arguments, in order. If a block receives an evaluation message with a different number of arguments from the number of block arguments it takes, an error will be reported.

Summary of Terminology

The syntax of expressions is summarized inside the back cover of this book.

expression — A sequence of characters that describes an object.

literal — An expression describing a constant, such as a number or a string.

symbol — A string whose sequence of characters is guaranteed to be different from that of any other symbol.

array — A data structure whose elements are associated with integer indices.

variable name — An expression describing the current value of a variable.

assignment — An expression describing a change of a variable's value.

pseudo-variable name — An expression similar to a variable name. However, unlike a variable name, the value of a pseudo-variable name cannot be changed by an assignment.

receiver — The object to which a message is sent.

message selector — The name of the type of operation a message requests of its receiver.

message argument — An object that specifies additional information for an operation.

unary message — A message without arguments.

keyword — An identifier with a trailing colon.

keyword message — A message with one or more arguments whose selector is made up of one or more keywords.

binary message — A message with one argument whose selector is made up of one or two special characters.

cascading — A description of several messages to one object in a single expression.

block — A description of a deferred sequence of actions.

block argument — A parameter that must be supplied when certain blocks are evaluated.

value — A message to a block asking it to carry out the set of actions it represents.

value: — A keyword used in a message to a block that has block arguments; the corresponding message asks the block to carry out its set of actions.

ifTrue:ifFalse: — Message to a Boolean requesting conditional selection.

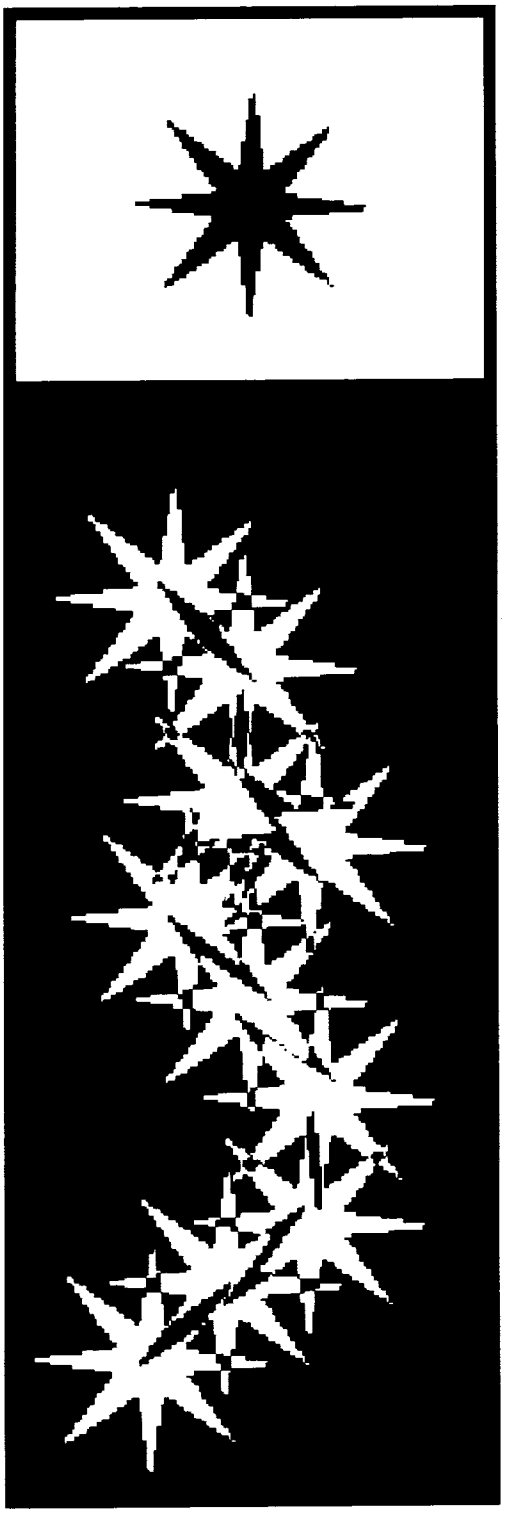
ifFalse:ifTrue: — Message to a Boolean requesting conditional selection.

ifTrue: — Message to a Boolean requesting conditional selection.

ifFalse: — Message to a Boolean requesting conditional selection.

whileTrue: — Message to a block requesting conditional repetition.

whileFalse:	Message to a block requesting conditional repetition.
do:	A message to a collection requesting enumeration of its elements.
collect:	A message to a collection requesting transformation of its elements.



3

Classes and Instances

Objects represent the components of the Smalltalk-80 system—the numbers, data structures, processes, disk files, process schedulers, text editors, compilers, and applications. Messages represent interactions between the components of the Smalltalk-80 system—the arithmetic, data accesses, control structures, file creations, text manipulations, compilations, and application uses. Messages make an object's functionality available to other objects, while keeping the object's implementation hidden. The previous chapter introduced an expression syntax for describing objects and messages, concentrating on how messages are used to access an object's functionality. This chapter introduces the syntax for describing methods and classes in order to show how the functionality of objects is implemented.

Every Smalltalk-80 object is an *instance* of a *class.* The instances of a class all have the same message interface; the class describes how to carry out each of the operations available through that interface. Each operation is described by a *method.* The selector of a message determines what type of operation the receiver should perform, so a class has one method for each selector in its interface. When a message is sent to an object, the method associated with that type of message in the receiver's class is executed. A class also describes what type of private memory its instances will have.

Each class has a name that describes the type of component its instances represent. A class name serves two fundamental purposes; it is a simple way for instances to identify themselves, and it provides a way to refer to the class in expressions. Since classes are components of the Smalltalk-80 system, they are represented by objects. A class's name automatically becomes the name of a globally shared variable. The value of that variable is the object representing the class. Since class names are the names of shared variables, they must be capitalized.

New objects are created by sending messages to classes. Most classes respond to the unary message new by creating a new instance of themselves. For example,

```
OrderedCollection new
```

returns a new collection that is an instance of the system class OrderedCollection. The new OrderedCollection is empty. Some classes create instances in response to other messages. For example, the class whose instances represent times in a day is Time; Time responds to the message now with an instance representing the current time. The class whose instances represent days in a year is Date; Date responds to the message today with an instance representing the current day. When a new instance is created, it automatically shares the methods of the class that received the instance creation message.

This chapter introduces two ways to present a class, one describing the functionality of the instances and the other describing the implementation of that functionality.

1. A *protocol description* lists the messages in the instances' message interface. Each message is accompanied by a comment describing the operation an instance will perform when it receives that type of message.
2. An *implementation description* shows how the functionality described in the protocol description is implemented. An implementation description gives the form of the instances' private memory and the set of methods that describe how instances perform their operations.

A third way to present classes is an interactive view called a *system browser.* The browser is part of the programming interface and is used in a running Smalltalk-80 system. Protocol descriptions and implementation descriptions are designed for noninteractive documentation like this book. The browser will be described briefly in Chapter 17.

Protocol Descriptions

A protocol description lists the messages understood by instances of a particular class. Each message is listed with a comment about its functionality. The comment describes the operation that will be performed when the message is received and what value will be returned. The comment describes *what* will happen, not *how* the operation will be performed. If the comment gives no indication of the value to be returned, then the value is assumed to be the receiver of the message.

For example, a protocol description entry for the message to a FinancialHistory with the selector spend:for: is

spend: amount for: reason	Remember that an amount of money, amount, has been spent for reason.

Messages in a protocol description are described in the form of *message patterns.* A message pattern contains a message selector and a set of argument names, one name for each argument that a message with that selector would have. For example, the message pattern

spend: amount for: reason

matches the messages described by each of the following three expressions.

```
HouseholdFinances spend: 32.50 for: 'utilities'
HouseholdFinances spend: cost+ tax for: 'food'
HouseholdFinances spend: 100 for: usualReason
```

The argument names are used in the comment to refer to the arguments. The comment in the example above indicates that the first argument represents the amount of money spent and the second argument represents what the money was spent for.

Message Categories

Messages that invoke similar operations are grouped together in *categories.* The categories have names that indicate the common functionality of the messages in the group. For example, the messages to FinancialHistory are grouped into three categories named transaction recording, inquiries, and initialization. This categorization is intended to make the protocol more readable to the user; it does not affect the operation of the class.

The complete protocol description for FinancialHistory is shown next.

FinancialHistory protocol

transaction recording	
receive: amount from: source	Remember that an amount of money, amount, has been received from source.
spend: amount for: reason	Remember that an amount of money, amount, has been spent for reason.
inquiries	
cashOnHand	Answer the total amount of money currently on hand.
totalReceivedFrom: source	Answer the total amount received from source, so far.
totalSpentFor: reason	Answer the total amount spent for reason, so far.
initialization	
initialBalance: amount	Begin a financial history with amount as the amount of money on hand.

A protocol description provides sufficient information for a programmer to know how to use instances of the class. From the above protocol description, we know that any instance of FinancialHistory should respond to the messages whose selectors are receive:from:, spend:for:, cashOnHand, totalReceivedFrom:, totalSpentFor:, and initialBalance:. We can guess that when we first create an instance of a FinancialHistory, the message initialBalance: should be sent to the instance in order to set values for its variables.

Implementation Descriptions

An implementation description has three parts.

1. a class name
2. a declaration of the variables available to the instances
3. the methods used by instances to respond to messages

An example of a complete implementation description for FinancialHistory is given next. The methods in an implementation description are divided into the same categories used in the protocol description. In the interactive system browser, categories are used to provide a hierarchical query path for accessing the parts of a class description. There are no special character delimiters separating the various parts of implementation descriptions. Changes in character font and emphasis indicate the different parts. In the interactive system browser, the parts are stored independently and the system browser provides a structured editor for accessing them.

```
class name                          FinancialHistory
instance variable names             cashOnHand
                                    incomes
                                    expenditures
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]
```

initialization

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

This implementation description is different from the one presented for FinancialHistory on the inside front cover of this book. The one on the inside front cover has an additional part labeled "class methods" that will be explained in Chapter 5; also, it omits the initialization method shown here.

Variable Declarations

The methods in a class have access to five different kinds of variables. These kinds of variables differ in terms of how widely they are available (their scope) and how long they persist.

There are two kinds of private variables available only to a single object.

1. *Instance variables* exist for the entire lifetime of the object.
2. *Temporary variables* are created for a specific activity and are available only for the duration of the activity.

Instance variables represent the current state of an object. Temporary variables represent the transitory state necessary to carry out some activity. Temporary variables are typically associated with a single execution of a method: they are created when a message causes the method to be executed and are discarded when the method completes by returning a value.

The three other kinds of variables can be accessed by more than one object. They are distinguished by how widely they are shared.

3. *Class variables* are shared by all the instances of a single class.
4. *Global variables* are shared by all the instances of all classes (that is, by all objects).
5. *Pool variables* are shared by the instances of a subset of the classes in the system.

The majority of shared variables in the system are either class variables or global variables. The majority of global variables refer to the classes in the system. An instance of FinancialHistory named

HouseholdFinances was used in several of the examples in the previous chapters. We used HouseholdFinances as if it were defined as a global variable name. Global variables are used to refer to objects that are not parts of other objects.

Recall that the names of shared variables (3-5) are capitalized, while the names of private variables (1-2) are not. The value of a shared variable will be independent of which instance is using the method in which its name appears. The value of instance variables and temporaries will depend on the instance using the method, that is, the instance that received a message.

Instance Variables

There are two types of instance variables, named and indexed. They differ in terms of how they are declared and how they are accessed. A class may have only named instance variables, only indexed variables, or some of each.

☐ *Named Instance Variables* An implementation description includes a set of names for the instance variables that make up the individual instances. Each instance has one variable corresponding to each instance variable name. The variable declaration in the implementation description of FinancialHistory specified three instance variable names.

```
instance variable names        cashOnHand
                               incomes
                               expenditures
```

An instance of FinancialHistory uses two dictionaries to store the total amounts spent and received for various reasons, and uses another variable to keep track of the cash on hand.

- expenditures refers to a dictionary that associates spending reasons with amounts spent.
- incomes refers to a dictionary that associates income sources with amounts received.
- cashOnHand refers to a number representing the amount of money available.

When expressions in the methods of the class use one of the variable names incomes, expenditures, or cashOnHand, these expressions refer to the value of the corresponding instance variable in the instance that received the message.

When a new instance is created by sending a message to a class, it has a new set of instance variables. The instance variables are initialized as specified in the method associated with the instance creation message. The default initialization method gives each instance variable a value of nil.

For example, in order for the previous example messages to HouseholdFinances to work, an expression such as the following must have been evaluated.

```
HouseholdFinances ← FinancialHistory new initialBalance: 350
```

FinancialHistory new creates a new object whose three instance variables all refer to nil. The initialBalance: message to that new instance gives the three instance variables more appropriate initial values.

☐ *Indexed Instance Variables* Instances of some classes can have instance variables that are not accessed by names. These are called *indexed instance variables.* Instead of being referred to by name, indexed instance variables are referred to by messages that include integers, called indices, as arguments. Since indexing is a form of association, the two fundamental indexing messages have the same selectors as the association messages to dictionaries—at: and at:put:. For example, instances of Array have indexed variables. If names is an instance of Array, the expression

```
names at: 1
```

returns the value of its first indexed instance variable. The expression

```
names at: 4 put: 'Adele'
```

stores the string 'Adele' as the value of the fourth indexed instance variable of names. The legal indices run from one to the number of indexed variables in the instance.

If the instances of a class have indexed instance variables, its variable declaration will include the line indexed instance variables. For example, part of the implementation description for the system class Array is

```
class name                          Array
indexed instance variables
```

Each instance of a class that allows indexed instance variables may have a different number of them. All instances of FinancialHistory have three instance variables, but instances of Array may have any number of instance variables.

A class whose instances have indexed instance variables can also have named instance variables. All instances of such a class will have the same number of named instance variables, but may have different numbers of indexed variables. For example, a system class representing a collection whose elements are ordered, OrderedCollection, has indexed instance variables to hold its contents. An OrderedCollection might have more space for storing elements than is currently being used. The two

named instance variables remember the indices of the first and last element of the contents.

class name	OrderedCollection
instance variable names	firstIndex
	lastIndex
indexed instance variables	

All instances of OrderedCollection will have two named variables, but one may have five indexed instance variables, another 15, another 18, and so on.

The named instance variables of an instance of FinancialHistory are private in the sense that access to the values of the variables is controlled by the instance. A class may or may not include messages giving direct access to the instance variables. Indexed instance variables are not private in this sense, since direct access to the values of the variables is available by sending messages with selectors at: and at:put:. Since these messages are the only way to access indexed instance variables, they must be provided.

Classes with indexed instance variables create new instances with the message new: instead of the usual message new. The argument of new: tells the number of indexed variables to be provided.

```
list ← Array new: 10
```

creates an Array of 10 elements, each of which is initially the special object nil. The number of indexed instance variables of an instance can be found by sending it the message size. The response to the message size

```
list size
```

is, for this example, the integer 10.

Evaluating each of the following expressions, in order,

```
list ← Array new: 3.
list at: 1 put: 'one'.
list at: 2 put: 'two'.
list at: 3 put: 'three'
```

is equivalent to the single expression

```
list ← #('one' 'two' 'three')
```

Shared Variables

Variables that are shared by more than one object come in groups called *pools*. Each class has two or more pools whose variables can be accessed by its instances. One pool is shared by all classes and contains the global variables; this pool is named Smalltalk. Each class also has a

pool which is only available to its instances and contains the class variables.

Besides these two mandatory pools, a class may access some other special purpose pools shared by several classes. For example, there are several classes in the system that represent textual information; these classes need to share the ASCII character codes for characters that are not easily indicated visually, such as a carriage return, tab, or space. These numbers are included as variables in a pool named TextConstants that is shared by the classes implementing text display and text editing.

If FinancialHistory had a class variable named SalesTaxRate and shared a pool dictionary whose name is FinancialConstants, the declaration would be expressed as follows.

instance variable names	cashOnHand
	incomes
	expenditures
class variable names	SalesTaxRate
shared pools	FinancialConstants

SalesTaxRate is the name of a class variable, so it can be used in any methods in the class. FinancialConstants, on the other hand, is the name of a pool; it is the variables *in* the pool that can be used in expressions.

In order to declare a variable to be global (known to all classes and to the user's interactive system), the variable name must be inserted as a key in the dictionary Smalltalk. For example, to make AllHistories global, evaluate the expression

```
Smalltalk at: #AllHistories put: nil
```

Then use an assignment statement to set the value of AllHistories.

Methods

A method describes how an object will perform one of its operations. A method is made up of a message pattern and a sequence of expressions separated by periods. The example method shown below describes the response of a FinancialHistory to messages informing it of expenditures.

```
spend: amount for: reason
    expenditures at: reason
                put: (self totalSpentFor: reason) + amount.
    cashOnHand ← cashOnHand − amount
```

The message pattern, **spend: amount for: reason**, indicates that this method will be used in response to all messages with selector spend:for:.

The first expression in the body of this method adds the new amount to the amount already spent for the reason indicated. The second expression is an assignment that decrements the value of cashOnHand by the new amount.

Argument Names

Message patterns were introduced earlier in this chapter. A message pattern contains a message selector and a set of argument names, one for each argument that a message with that selector would have. A message pattern matches any messages that have the same selector. A class will have only one method with a given selector in its message pattern. When a message is sent, the method with matching message pattern is selected from the class of the receiver. The expressions in the selected method are evaluated one after another. After all the expressions are evaluated, a value is returned to the sender of the message.

The argument names found in a method's message pattern are pseudo-variable names referring to the arguments of the actual message. If the method shown above were invoked by the expression

```
HouseholdFinances spend: 30.45 for: 'food'
```

the pseudo-variable name amount would refer to the number 30.45 and the pseudo-variable name reason would refer to the string 'food' during the evaluation of the expressions in the method. If the same method were invoked by the expression

```
HouseholdFinances spend: cost + tax for: 'food'
```

cost would be sent the message + tax and the value it returned would be referred to as amount in the method. If cost referred to 100 and tax to 6.5, the value of amount would be 106.5.

Since argument names are pseudo-variable names, they can be used to access values like variable names, but their values cannot be changed by assignment. In the method for spend:for:, a statement of the form

```
amount ← amount * taxRate
```

would be syntactically illegal since the value of amount cannot be reassigned.

Returning Values

The method for spend:for: does not specify what the value of the message should be. Therefore, the default value, the receiver itself, will be returned. When another value is to be specified, one or more return expressions are included in the method. Any expression can be turned

into a return expression by preceding it with an uparrow (↑). The value of a variable may be returned as in

```
↑cashOnHand
```

The value of another message can be returned as in

```
↑expenditures at: reason
```

A literal object can be returned as in

```
↑0
```

Even an assignment statement can be turned into a return expression, as in

```
↑initialIndex ← 0
```

The assignment is performed first. The new value of the variable is then returned.

An example of the use of a return expression is the following implementation of totalSpentFor:.

```
totalSpentFor: reason
    (expenditures includesKey: reason)
        ifTrue: [↑expenditures at: reason]
        ifFalse: [↑0]
```

This method consists of a single conditional expression. If the expenditure reason is in expenditures, the associated value is returned; otherwise, zero is returned.

The Pseudo-variable self

Along with the pseudo-variables used to refer to the arguments of a message, all methods have access to a pseudo-variable named self that refers to the message receiver itself. For example, in the method for spend:for:, the message totalSpentFor: is sent to the receiver of the spend:for: message.

```
spend: amount for: reason
    expenditures at: reason
                put: (self totalSpentFor: reason) + amount.
    cashOnHand ← cashOnHand - amount
```

When this method is executed, the first thing that happens is that totalSpentFor: is sent to the same object (self) that received spend:for:. The result of that message is sent the message + amount, and the result of that message is used as the second argument to at:put:.

The pseudo-variable self can be used to implement recursive functions. For example, the message factorial is understood by integers in order to compute the appropriate function. The method associated with factorial is

```
factorial
    self = 0 ifTrue: [↑1].
    self < 0
        ifTrue: [self error: 'factorial invalid']
        ifFalse: [↑self * (self − 1) factorial]
```

The receiver is an Integer. The first expression tests to see if the receiver is 0 and, if it is, returns 1. The second expression tests the sign of the receiver because, if it is less than 0, the programmer should be notified of an error (all objects respond to the message error: with a report that an error has been encountered). If the receiver is greater than 0, then the value to be returned is

```
self * (self − 1) factorial
```

The value returned is the receiver multiplied by the factorial of one less than the receiver.

Temporary Variables

The argument names and self are available only during a single execution of a method. In addition to these pseudo-variable names, a method may obtain some other variables for use during its execution. These are called *temporary variables.* Temporary variables are indicated by including a temporary variable declaration between the message pattern and the expressions of a method. A temporary declaration consists of a set of variable names between vertical bars. The method for spend:for: could be rewritten to use a temporary variable to hold the previous expenditures.

```
spend: amount for: reason
    | previousExpenditures |
    previousExpenditures ← self totalSpentFor: reason.
    expenditures at: reason
                put: previousExpenditures + amount.
    cashOnHand ← cashOnHand − amount
```

The values of temporary variables are accessible only to statements in the method and are forgotten when the method completes execution. All temporary variables initially refer to nil.

In the interactive Smalltalk-80 system, the programmer can test algorithms that make use of temporary variables. The test can be carried out by using the vertical bar notation to declare the variables for the duration of the immediate evaluation only. Suppose the expressions to

be tried out include reference to the variable list. If the variable list is undeclared, an attempt to evaluate the expressions will create a syntax error message. Instead, the programmer can declare list as a temporary variable by prefixing the expressions with the declaration | list |. The expressions are separated by periods, as in the syntax of a method.

```
| list |
list ← Array new: 3.
list at: 1 put: 'one'.
list at: 2 put: 'four'.
list printString
```

The programmer interactively selects all five lines—the declaration and the expressions—and requests evaluation. The variable list is available only during the single execution of the selection.

Primitive Methods

When an object receives a message, it typically just sends other messages, so where does something *really* happen? An object may change the value of its instance variables when it receives a message, which certainly qualifies as "something happening." But this hardly seems enough. In fact, it is not enough. All behavior in the system *is* invoked by messages, however, all messages are *not* responded to by executing Smalltalk-80 methods. There are about one hundred *primitive methods* that the Smalltalk-80 virtual machine knows how to perform. Examples of messages that invoke primitives are the + message to small integers, the at: message to objects with indexed instance variables, and the new and new: messages to classes. When 3 gets the message + 4, it does not execute a Smalltalk-80 method. A primitive method returns 7 as the value of the message. The complete set of primitive methods is included in the fourth part of this book, which describes the virtual machine.

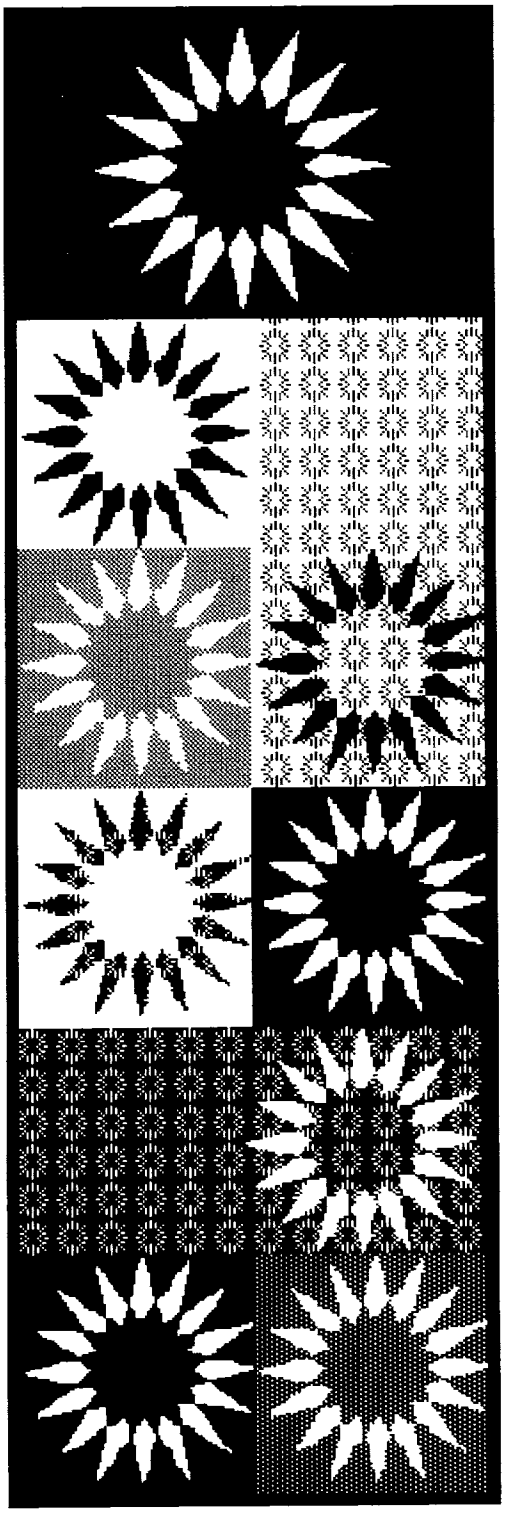
Methods that are implemented as primitive methods begin with an expression of the form

```
<primitive #>
```

where # is an integer indicating which primitive method will be followed. If the primitive fails to perform correctly, execution continues in the Smalltalk-80 method. The expression <primitive #> is followed by Smalltalk-80 expressions that handle failure situations.

Summary of Terminology

class	An object that describes the implementation of a set of similar objects.
instance	One of the objects described by a class; it has memory and responds to messages.
instance variable	A variable available to a single object for the entire lifetime of the object; instance variables can be named or indexed.
protocol description	A description of a class in terms of its instances' public message protocol.
implementation description	A description of a class in terms of its instances' private memory and the set of methods that describe how instances perform their operations.
message pattern	A message selector and a set of argument names, one for each argument that a message with this selector must have.
temporary variable	A variable created for a specific activity and available only for the duration of that activity.
class variable	A variable shared by all the instances of a single class.
global variable	A variable shared by all the instances of all classes.
pool variable	A variable shared by the instances of a set of classes.
Smalltalk	A pool shared by all classes that contains the global variables.
method	A procedure describing how to perform one of an object's operations; it is made up of a message pattern, temporary variable declaration, and a sequence of expressions. A method is executed when a message matching its message pattern is sent to an instance of the class in which the method is found
argument name	Name of a pseudo-variable available to a method only for the duration of that method's execution; the value of the argument names are the arguments of the message that invoked the method.
↑	When used in a method, indicates that the value of the next expression is to be the value of the method.
self	A pseudo-variable referring to the receiver of a message.
message category	A group of methods in a class description.
primitive method	An operation performed directly by the Smalltalk-80 virtual machine; it is not described as a sequence of Smalltalk-80 expressions.



4

Subclasses

Every object in the Smalltalk-80 system is an instance of a class. All instances of a class represent the same kind of system component. For example, each instance of Rectangle represents a rectangular area and each instance of Dictionary represents a set of associations between names and values. The fact that the instances of a class all represent the same kind of component is reflected both in the way the instances respond to messages and in the form of their instance variables.

- All instances of a class respond to the same set of messages and use the same set of methods to do so.
- All instances of a class have the same number of named instance variables and use the same names to refer to them.
- An object can have indexed instance variables only if all instances of its class can have indexed instance variables.

The class structure as described so far does not explicitly provide for any intersection in class membership. Each object is an instance of exactly one class. This structure is illustrated in Figure 4.1. In the figure, the small circles represent instances and the boxes represent classes. If a circle is within a box, then it represents an instance of the class represented by the box.

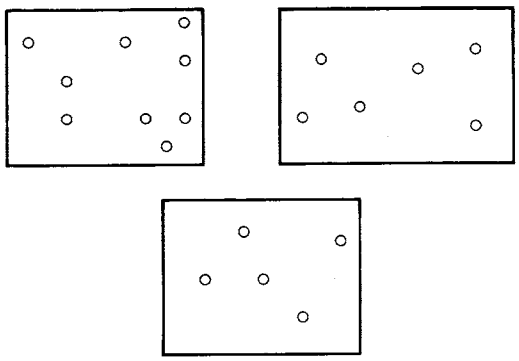


Figure 4.1

Lack of intersection in class membership is a limitation on design in an object-oriented system since it does not allow any sharing between class descriptions. We might want two objects to be substantially similar, but to differ in some particular way. For example, a floating-point number and an integer are similar in their ability to respond to arithmetic messages, but are different in the way they represent numeric values. An ordered collection and a bag are similar in that they are containers to which elements can be added and from which elements can be removed,

but they are different in the precise way in which individual elements are accessed. The difference between otherwise similar objects may be externally visible, such as responding to some different messages, or it may be purely internal, such as responding to the same message by executing different methods. If class memberships are not allowed to overlap, this type of partial similarity between two objects cannot be guaranteed by the system.

The most general way to overcome this limitation is to allow arbitrary intersection of class boundaries (Figure 4.2).

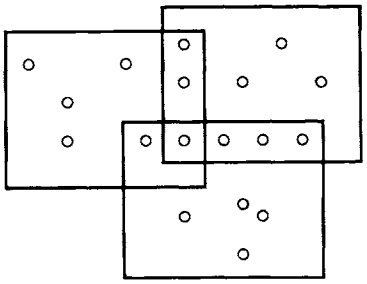


Figure 4.2

We call this approach *multiple inheritance.* Multiple inheritance allows a situation in which some objects are instances of two classes, while other objects are instances of only one class or the other. A less general relaxation of the nonintersection limitation on classes is to allow a class to include all instances of another class, but not to allow more general sharing (Figure 4.3).

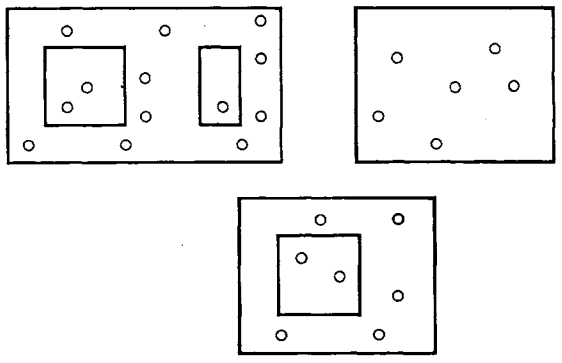


Figure 4.3

We call this approach *subclassing.* This follows the terminology of the programming language Simula, which includes a similar concept. Subclassing is strictly hierarchical; if any instances of a class are also

instances of another class, then all instances of that class must also be instances of the other class.

The Smalltalk-80 system provides the subclassing form of inheritance for its classes. This chapter describes how subclasses modify their superclasses, how this affects the association of messages and methods, and how the subclass mechanism provides a framework for the classes in the system.

Subclass Descriptions

A subclass specifies that its instances will be the same as instances of another class, called its *superclass*, except for the differences that are explicitly stated. The Smalltalk-80 programmer always creates a new class as a subclass of an existing class. A system class named Object describes the similarities of *all* objects in the system, so every class will at least be a subclass of Object. A class description (protocol or implementation) specifies how its instances differ from the instances of its superclass. The instances of a superclass can not be affected by the existence of subclasses.

A subclass is in all respects a class and can therefore have subclasses itself. Each class has one superclass, although many classes may share the same superclass, so the classes form a tree structure. A class has a sequence of classes from which it inherits both variables and methods. This sequence begins with its superclass and continues with its superclass's superclass, and so on. The inheritance chain continues through the superclass relationship until Object is encountered. Object is the single root class; it is the only class without a superclass.

Recall that an implementation description has three basic parts:

1. A class name
2. A variable declaration
3. A set of methods

A subclass must provide a new class name for itself, but it inherits both the variable declaration and methods of its superclass. New variables may be declared and new methods may be added by the subclass. If instance variable names are added in the subclass variable declaration, instances of the subclass will have more instance variables than instances of the superclass. If shared variables are added, they will be accessible to the instances of the subclass, but not to instances of the superclass. All variable names added must be different from any declared in the superclass.

If a class does not have indexed instance variables, a subclass can declare that its instances will have indexed variables; these indexed variables will be in addition to any inherited named instance variables. If a class has indexed instance variables, its subclasses *must* also have indexed instance variables; a subclass can also declare new named instance variables.

If a subclass adds a method whose message pattern has the same selector as a method in the superclass, its instances will respond to messages with that selector by executing the new method. This is called *overriding* a method. If a subclass adds a method with a selector not found in the methods of the superclass, the instances of the subclass will respond to messages not understood by instances of the superclass.

To summarize, each part of an implementation description can be modified by a subclass in a different way:

1. The class name *must* be overridden.
2. Variables *may* be added.
3. Methods *may* be added or overridden.

An Example Subclass

An implementation description includes an entry, not shown in the previous chapter, that specifies its superclass. The following example is a class created as a subclass of the FinancialHistory class introduced in Chapter 3. Instances of the subclass share the function of FinancialHistory for storing information about monetary expenditures and receipts. They have the additional function of keeping track of the expenditures that are tax deductible. The subclass provides the mandatory new class name (DeductibleHistory), and adds one instance variable and four methods. One of these methods (initialBalance:) overrides a method in the superclass.

The class description for DeductibleHistory follows.

class name	DeductibleHistory
superclass	FinancialHistory
instance variable names	deductibleExpenditures
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

totalDeductions
    ↑deductibleExpenditures

initialization

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

In order to know all the messages understood by an instance of DeductibleHistory, it is necessary to examine the protocols of DeductibleHistory, FinancialHistory, and Object. Instances of DeductibleHistory have four variables—three inherited from the superclass FinancialHistory, and one specified in the class DeductibleHistory. Class Object declares no instance variables.

Figure 4.4 indicates that DeductibleHistory is a subclass of FinancialHistory. Each box in this diagram is labeled in the upper left corner with the name of class it represents.

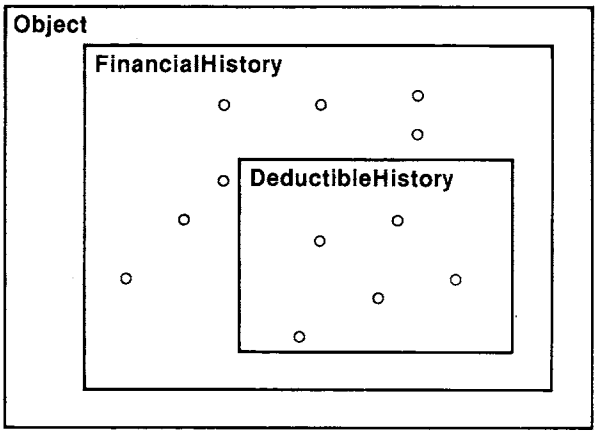


Figure 4.4

Instances of DeductibleHistory can be used to record the history of entities that pay taxes (people, households, businesses). Instances of FinancialHistory can be used to record the history of entities that do not

pay taxes (charitable organizations, religious organizations). Actually, an instance of DeductibleHistory could be used in place of an instance of FinancialHistory without detection since it responds to the same messages in the same way. In addition to the messages and methods inherited from FinancialHistory, an instance of DeductibleHistory can respond to messages indicating that all or part of an expenditure is deductible. The new messages available are spendDeductible:for:, which is used if the total amount is deductible; and spend:for:deducting:, which is used if only part of the expenditure is deductible. The total tax deduction can be found by sending a DeductibleHistory the message totalDeductions.

Method Determination

When a message is sent, the methods in the receiver's class are searched for one with a matching selector. If none is found, the methods in that class's superclass are searched next. The search continues up the superclass chain until a matching method is found. Suppose we send an instance of DeductibleHistory a message with selector cashOnHand. The search for the appropriate method to execute begins in the class of the receiver, DeductibleHistory. When it is not found, the search continues by looking at DeductibleHistory's superclass, FinancialHistory. When a method with the selector cashOnHand is found there, that method is executed as the response to the message. The response to this message is to return the value of the instance variable cashOnHand. This value is found in the receiver of the message, that is, in the instance of DeductibleHistory.

The search for a matching method follows the superclass chain, terminating at class Object. If no matching method is found in any class in the superclass chain, the receiver is sent the message doesNotUnderstand:; the argument is the offending message. There is a method for the selector doesNotUnderstand: in Object that reports the error to the programmer.

Suppose we send an instance of DeductibleHistory a message with selector spend:for:. This method is found in the superclass FinancialHistory. The method, as given in Chapter 3, is

```
spend: amount for: reason
    expenditures at: reason
                put: (self totalSpentFor: reason) + amount.
    cashOnHand ← cashOnHand − amount
```

The values of the instance variables (expenditures and cashOnHand) are found in the receiver of the message, the instance of DeductibleHistory.

The pseudo-variable self is also referenced in this method; self represents the DeductibleHistory instance that was the receiver of the message.

Messages to self

When a method contains a message whose receiver is self, the search for the method for that message begins in the instance's class, regardless of which class contains the method containing self. Thus, when the expression self totalSpentFor: reason is evaluated in the method for spend:for: found in FinancialHistory, the search for the method associated with the message selector totalSpentFor: begins in the class of self, i.e., in DeductibleHistory.

Messages to self will be explained using two example classes named One and Two. Two is a subclass of One and One is a subclass of Object. Both classes include a method for the message test. Class One also includes a method for the message result1 that returns the result of the expression self test.

```
class name                      One
superclass                      Object
instance methods

    test
        ↑1
    result1
        ↑self test

class name                      Two
superclass                      One
instance methods

    test
        ↑2
```

An instance of each class will be used to demonstrate the method determination for messages to self. example1 is an instance of class One and example2 is an instance of class Two.

```
example1 ← One new.
example2 ← Two new
```

The relationship between One and Two is shown in Figure 4.5. In addition to labeling the boxes in order to indicate class names, several of the circles are also labeled in order to indicate a name referring to the corresponding instance.

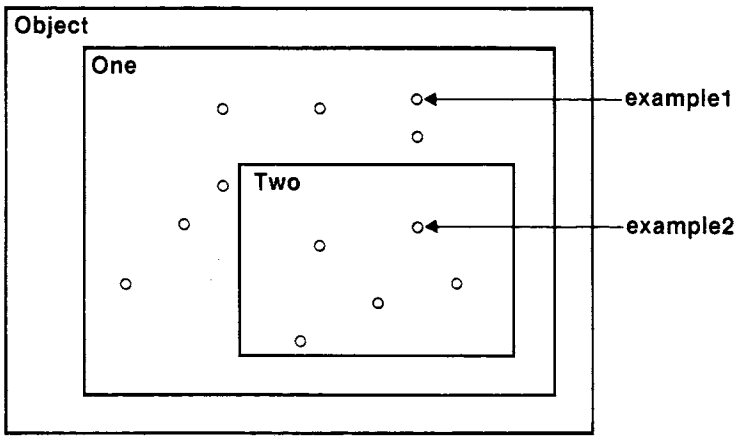


Figure 4.5

The following table shows the results of evaluating various expressions.

expression	*result*
example1 test	1
example1 result1	1
example2 test	2
example2 result1	2

The two result1 messages both invoke the same method, which is found in class One. They produce different results because of the message to self contained in that method. When result1 is sent to example2, the search for a matching method begins in Two. A method is not found in Two, so the search continues by looking in the superclass, One. A method for result1 is found in One, which consists of one expression, ↑self test. The pseudo-variable self refers to the receiver, example2. The search for the response to test, therefore, begins in class Two. A method for test is found in Two, which returns 2.

Messages to super

An additional pseudo-variable named super is available for use in a method's expressions. The pseudo-variable super refers to the receiver of the message, just as self does. However, when a message is sent to super, the search for a method does not begin in the receiver's class. Instead, the search begins in the superclass of the class containing the method. The use of super allows a method to access methods defined in

a superclass even if the methods have been overridden in subclasses. The use of super as other than a receiver (for example, as an argument), has no different effect from using self; the use of super only affects the initial class in which messages are looked up.

Messages to super will be explained using two more example classes named Three and Four. Four is a subclass of Three, Three is a subclass of the previous example Two. Four overrides the method for the message test. Three contains methods for two new messages—result2 returns the result of the expression self result1, and result3 returns the result of the expression super test.

```
class name                    Three
superclass                    Two
instance methods

    result2
        ↑self result1
    result3
        ↑super test

class name                    Four
superclass                    Three
instance methods

    test
        ↑4
```

Instances of One, Two, Three, and Four can all respond to the messages test and result1. The response of instances of Three and Four to messages illustrates the effect of super (Figure 4.6).

```
example3 ← Three new.
example4 ← Four new
```

An attempt to send the messages result2 or result3 to example1 or example2 is an error since instances of One or Two do not understand the messages result2 or result3.

The following table shows the results of sending various messages.

expression	*result*
example3 test	2
example4 result1	4
example3 result2	2
example4 result2	4
example3 result3	2
example4 result3	2

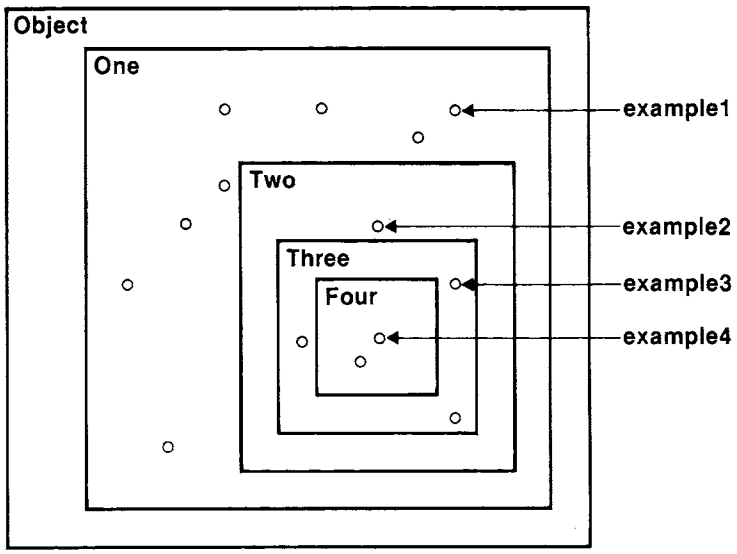


Figure 4.6

When test is sent to example3, the method in Two is used, since Three doesn't override the method. example4 responds to result1 with a 4 for the same reason that example2 responded with a 2. When result2 is sent to example3, the search for a matching method begins in Three. The method found there returns the result of the expression self result1. The search for the response to result1 also begins in class Three. A matching method is not found in Three or its superclass, Two. The method for result1 is found in One and returns the result of self test. The search for the response to test once more begins in class Three. This time, the matching method is found in Three's superclass Two.

The effect of sending messages to super will be illustrated by the responses of example3 and example4 to the message result3. When result3 is sent to example3, the search for a matching method begins in Three. The method found there returns the result of the expression super test. Since test is sent to super, the search for a matching method begins not in class Three, but in its superclass, Two. The method for test in Two returns a 2. When result3 is sent to example4, the result is still 2, even though Four overrides the message for test.

This example highlights a potential confusion: super does not mean start the search in the superclass of the receiver, which, in the last example, would have been class Three. It means start the search in the superclass of the class containing the method in which super was used, which, in the last example, was class Two. Even if Three had overridden the method for test by returning 3, the result of example4 result3 would still be 2. Sometimes, of course, the superclass of the class in which the

method containing super is found is the same as the superclass of the receiver.

Another example of the use of super is in the method for initialBalance: in DeductibleHistory.

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

This method overrides a method in the superclass FinancialHistory. The method in DeductibleHistory consists of two expressions. The first expression passes control to the superclass in order to process the initialization of the balance.

```
super initialBalance: amount
```

The pseudo-variable super refers to the receiver of the message, but indicates that the search for the method should skip DeductibleHistory and begin in FinancialHistory. In this way, the expressions from FinancialHistory do not have to be duplicated in DeductibleHistory. The second expression in the method does the subclass-specific initialization.

```
deductibleExpenditures ← 0
```

If self were substituted for super in the initialBalance: method, it would result in an infinite recursion, since every time initialBalance: is sent, it will be sent again.

Abstract Superclasses

Abstract superclasses are created when two classes share a part of their descriptions and yet neither one is properly a subclass of the other. A mutual superclass is created for the two classes which contains their shared aspects. This type of superclass is called *abstract* because it was not created in order to have instances. In terms of the figures shown earlier, an abstract superclass represents the situation illustrated in Figure 4.7. Notice that the abstract class does not directly contain instances.

As an example of the use of an abstract superclass, consider two classes whose instances represent dictionaries. One class, named SmallDictionary, minimizes the space needed to store its contents; the other, named FastDictionary, stores names and values sparsely and uses a hashing technique to locate names. Both classes use two parallel lists

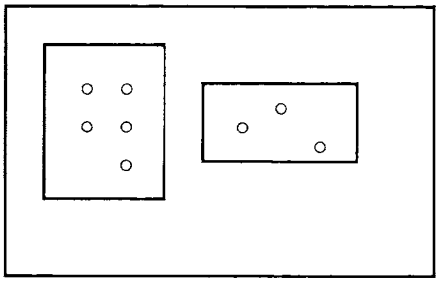


Figure 4.7

that contain names and associated values. SmallDictionary stores the names and values contiguously and uses a simple linear search to locate a name. FastDictionary stores names and values sparsely and uses a hashing technique to locate a name. Other than the difference in how names are located, these two classes are very similar: they share identical protocol and they both use parallel lists to store their contents. These similarities are represented in an abstract superclass named DualListDictionary. The relationships among these three classes is shown in Figure 4.8.

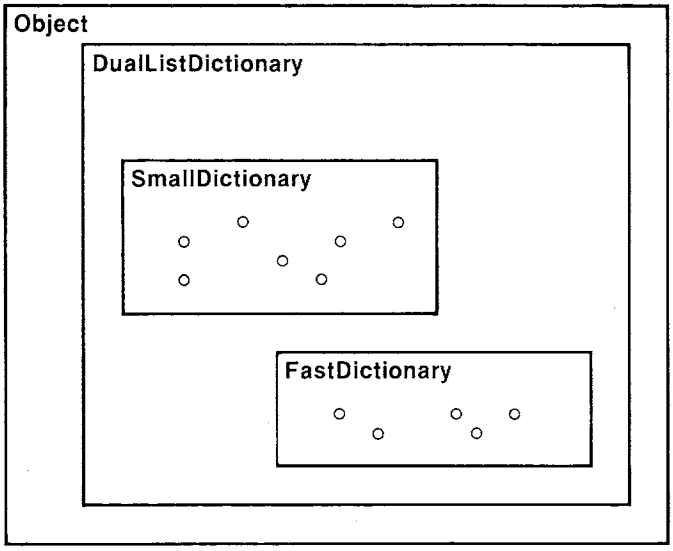


Figure 4.8

The implementation description for the abstract class, DualListDictionary is shown next.

```
class name                      DualListDictionary
superclass                      Object
instance variable names         names
                                values
instance methods

accessing

    at: name
        | index |
        index ← self indexOf: name.
        index = 0
            ifTrue: [self error: 'Name not found']
            ifFalse: [↑values at: index]
    at: name put: value
        | index |
        index ← self indexOf: name.
        index = 0
            ifTrue: [index ← self newIndexOf: name].
        ↑values at: index put: value

testing

    includes: name
        ↑(self indexOf: name) ~= 0
    isEmpty
        ↑self size = 0

initialization

    initialize
        names ← Array new: 0.
        values ← Array new: 0
```

This description of DualListDictionary uses only messages defined in DualListDictionary itself or ones already described in this or in the previous chapters. The external protocol for a DualListDictionary consists of messages at:, at:put:, includes:, isEmpty, and initialize. A new DualListDictionary (actually an instance of a subclass of DualListDictionary) is created by sending it the message new. It is then sent the message initialize so that assignments can be made to the two instance variables. The two variables are initially empty arrays (Array new: 0).

Three messages to self used in its methods are not implemented in DualListDictionary—size, indexOf:, and newIndexOf:. This is the reason that DualListDictionary is called abstract. If an instance were created, it would not be able to respond successfully to all of the necessary messages. The two subclasses, SmallDictionary and FastDictionary, must implement the three missing messages. The fact that the search always

starts at the class of the instance referred to by self means that a method in a superclass can be specified in which messages are sent to self, but the corresponding methods are found in the subclass. In this way, a superclass can provide a framework for a method that is refined or actually implemented by the subclass.

SmallDictionary is a subclass of DualListDictionary that uses a minimal amount of space to represent the associations, but may take a long time to find an association. It provides methods for the three messages that were not implemented in DualListDictionary—size, indexOf:, and newIndexOf:. It does not add variables.

class name SmallDictionary
superclass DualListDictionary
instance methods

accessing

size
```
↑names size
```

private

indexOf: name
```
1 to: names size do:
   [ :index | (names at: index) = name ifTrue: [↑index]].
↑0
```
newIndexOf: name
```
self grow.
names at: names size put: name.
↑names size
```
grow
```
| oldNames oldValues |
oldNames ← names.
oldValues ← values.
names ← Array new: names size + 1.
values ← Array new: values size + 1.
names replaceFrom: 1 to: oldNames size with: oldNames.
values replaceFrom: 1 to: oldValues size with: oldValues
```

Since names are stored contiguously, the size of a SmallDictionary is the size of its array of names, names. The index of a particular name is determined by a linear search of the array names. If no match is found, the index is 0, signalling failure in the search. Whenever a new association is to be added to the dictionary, the method for newIndexOf: is used to find the appropriate index. It assumes that the sizes of names and values are exactly the sizes needed to store their current elements. This means no space is available for adding a new element. The message grow creates two new Arrays that are copies of the previous ones, with

one more element at the end. In the method for newIndexOf:, first the sizes of names and values are increased and then the new name is stored in the new empty position (the last one). The method that called on newIndexOf: has the responsibility for storing the value.

We could evaluate the following example expressions.

expression	*result*
ages ← SmallDictionary new	a new, uninitialized instance
ages initialize	instance variables initialized
ages isEmpty	true
ages at: 'Brett' put: 3	3
ages at: 'Dave' put: 30	30
ages includes: 'Sam'	false
ages includes: 'Brett'	true
ages size	2
ages at: 'Dave'	30

For each of the above example expressions, we indicate in which class the message is found and in which class any messages sent to self are found.

message selector	*message to* self	*class of method*
initialize		DualListDictionary
at:put:		DualListDictionary
	indexOf:	SmallDictionary
	newIndexOf:	SmallDictionary
includes:		DualListDictionary
	indexOf:	SmallDictionary
size		SmallDictionary
at:		DualListDictionary
	indexOf:	SmallDictionary
	error:	Object

FastDictionary is another subclass of DualListDictionary. It uses a hashing technique to locate names. Hashing requires more space, but takes less time than a linear search. All objects respond to the hash message by returning a number. Numbers respond to the \\ message by returning their value in the modulus of the argument.

class name FastDictionary
superclass DualListDictionary
instance methods

accessing

size

```
| size |
size ← 0.
names do: [ :name | name notNil ifTrue: [size ← size + 1]].
↑size
```

initialization

initialize

```
names ← Array new: 4.
values ← Array new: 4
```

private

indexOf: name

```
| index |
index ← name hash \\ names size + 1.
[(names at: index) = name]
   whileFalse: [(names at: index) isNil
                     ifTrue: [↑0]
                     ifFalse: [index ← index \\ names size + 1]].
↑index
```

newIndexOf: name

```
| index |
names size - self size <= (names size / 4)
   ifTrue: [self grow].
index ← name hash \\ names size + 1.
[(names at: index) isNil]
   whileFalse: [index ← index \\ names size + 1].
names at: index put: name.
↑index
```

grow

```
| oldNames oldValues |
oldNames ← names.
oldValues ← values.
names ← Array new: names size * 2.
values ← Array new: values size * 2.
1 to: oldNames size do:
   [ :index |
     (oldNames at: index) isNil
          ifFalse: [self at: (oldNames at: index)
                     put: (oldValues at: index)]]
```

FastDictionary overrides DualListDictionary's implementation of initialize in order to create Arrays that already have some space allocated (Array new: 4). The size of a FastDictionary is not simply the size of one of its variables since the Arrays always have empty entries. So the size is determined by examining each element in the Array and counting the number that are not nil.

The implementation of newIndexOf: follows basically the same idea as that used for SmallDictionary except that when the size of an Array is changed (doubled in this case in the method for grow), each element is explicitly copied from the old Arrays into the new ones so that elements are rehashed. The size does not always have to be changed as is necessary in SmallDictionary. The size of a FastDictionary is changed only when the number of empty locations in names falls below a minimum. The minimum is equal to 25% of the elements.

```
names size – self size <= (names size / 4)
```

Subclass Framework Messages

As a matter of programming style, a method should not include messages to self if the messages are neither implemented by the class nor inherited from a superclass. In the description of DualListDictionary, three such messages exist—size, indexOf:, and newIndexOf:. As we shall see in subsequent chapters, the ability to respond to size is inherited from Object; the response is the number of indexed instance variables. A subclass of DualListDictionary is supposed to override this method in order to return the number of names in the dictionary.

A special message, subclassResponsibility, is specified in Object. It is to be used in the implementation of messages that cannot be properly implemented in an abstract class. That is, the implementation of size and indexOf: and newIndexOf:, by Smalltalk-80 convention, should be

```
self subclassResponsibility
```

The response to this message is to invoke the following method defined in class Object.

```
subclassResponsibility
    self error: 'My subclass should have overridden one of my messages.'
```

In this way, if a method should have been implemented in a subclass of an abstract class, the error reported is an indication to the programmer of how to fix the problem. Moreover, using this message, the programmer creates abstract classes in which all messages sent to self are

implemented, and in which the implementation is an indication to the programmer of which methods must be overridden in the subclass.

By convention, if the programmer decides that a message inherited from an abstract superclass should actually *not* be implemented, the appropriate way to override the inherited method is

```
self shouldNotImplement
```

The response to this message is to invoke the following method defined in class Object.

```
shouldNotImplement
    self error: 'This message is not appropriate for this object.'
```

There are several major subclass hierarchies in the Smalltalk-80 system that make use of the idea of creating a framework of messages whose implementations must be completed in subclasses. There are classes describing various kinds of collections (see Chapters 9 and 10). The collection classes are arranged hierarchically in order to share as much as possible among classes describing similar kinds of collections. They make use of the messages subclassResponsibility and shouldNotImplement. Another example of the use of subclasses is the hierarchy of linear measures and number classes (see Chapters 7 and 8).

Summary of Terminology

subclass	A class that inherits variables and methods from an existing class.
superclass	The class from which variables and methods are inherited.
Object	The class that is the root of the tree-structured class hierarchy.
overriding a method	Specifying a method in a subclass for the same message as a method in a superclass.
super	A pseudo-variable that refers to the receiver of a message; differs from self in where to start the search for methods.
abstract class	A class that specifies protocol, but is not able to fully implement it; by convention, instances are not created of this kind of class.
subclassResponsibility	A message to report the error that a subclass should have implemented one of the superclass's messages.
shouldNotImplement	A message to report the error that this is a message inherited from a superclass but explicitly not available to instances of the subclass.