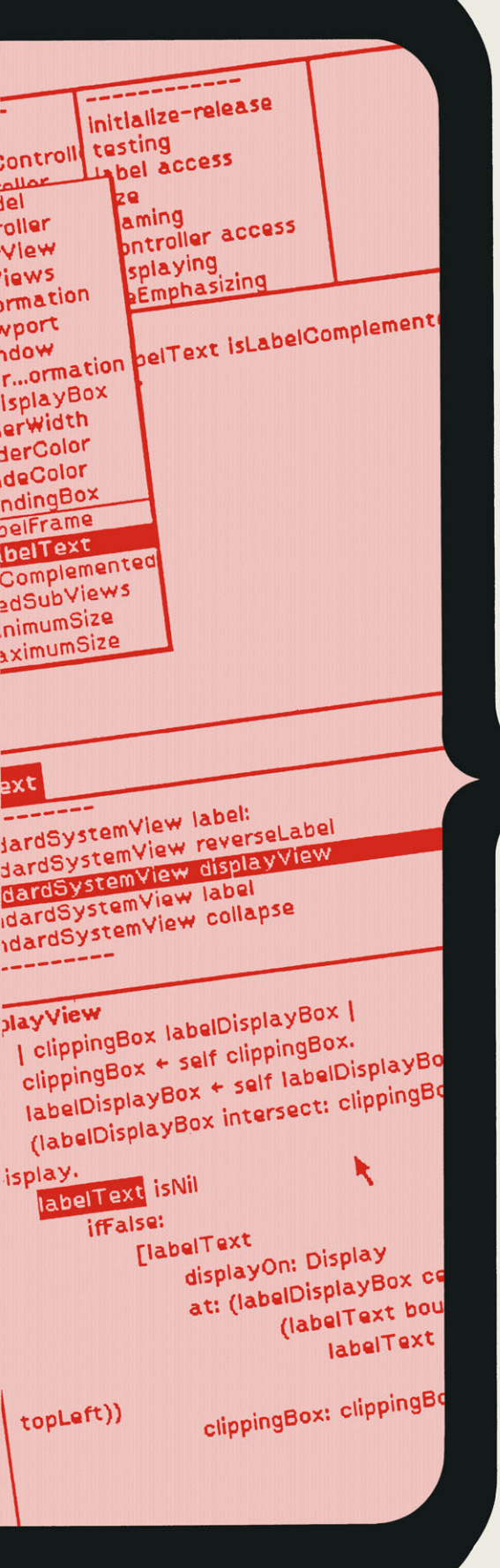


Smalltalk-80

A. Mével and
T. Guéguen

Macmillan Computer Science Series
Consulting Editor
Professor F.H. Sumner, University of Manchester

S.T. Allworth and R.N. Zobel, Introduction to Real-time Software Design, second edition
Ian O. Angell and Gareth Griffith, High-resolution Computer Graphics Using FORTRAN 77
Ian O. Angell and Gareth Griffith, High-resolution Computer Graphics Using Pascal
M.A. Azmoodeh, Abstract Data Types and Algorithms
C. Bamford and P. Curran, Data Structures, Files and Databases
Philip Barker, Author Languages for CAL
A.N. Barrett and A.L. Mackay, Spatial Structure and the Microcomputer
R.E. Berry and B.A.E. Meekings, A Book on C
G.M. Birtwistle, Discrete Event Modelling on Simula
T.B. Boffey, Graph Theory in Operations Research
Richard Bornat, Understanding and Writing Compilers
Linda E.M. Brackenbury, Design of VLSI Systems - A Practical Introduction
J.K. Buckle, Software Configuration Management
W.D. Burnham and A.R. Hall, Prolog Programming and Applications
J.C. Cluley, Interfacing to Microprocessors
J.C. Cluley, Introduction to Low Level Programming for Microprocessors
Robert Cole, Computer Communications, second edition
Derek Coleman, A Structured Programming Approach to Data
Andrew J.T. Colin, Fundamentals of Computer Science
Andrew J.T. Colin, Programming and Problem-solving in Algol 68
S.M. Deen, Fundamentals of Data Base Systems
S.M. Deen, Principles and Practice of Database Systems
Tim Denvir, Introduction to Discrete Mathematics for Software Engineering
P.M. Dew and K.R. James, Introduction to Numerical Computation in Pascal
M.R.M. Dunsmuir and G.J. Davies, Programming the UNIX System
K.C.E. Gee, Introduction to Local Area Computer Networks
J.B. Gosling, Design of Arithmetic Units for Digital Computers
Roger Hutty, Z80 Assembly Language Programming for Students
Roland N. Ibbett, The Architecture of High Performance Computers
Patrick Jaulent, The 68000 - Hardware and Software
J.M. King and J.P. Pardoe, Program Design Using JSP - A Practical Introduction

H. Kopetz, Software Reliability
E.V. Krishnamurthy, Introductory Theory of Computer Science
V.P. Lane, Security of Computer Based Information Systems
Graham Lee, From Hardware to Software - an introduction to computers
A.M. Lister, Fundamentals of Operating Systems, third edition
G.P. McKeown and V.J. Rayward-Smith, Mathematics for Computing
Brian Meek, Fortran, PL/1 and the Algols
A. Mével and T. Guéguen, Smalltalk-80
Barry Morrell and Peter Whittle, CP/M 80 Programmer's Guide
Derrick Morris, System Programming Based on the PDP11
Y. Nishinuma and R. Espesser, UNIX - First contact
Pim Oets, MS-DOS and PC-DOS - A Practical Guide
Christian Queinnec, LISP
Gordon Reece, Microcomputer Modelling by Finite Differences
W.P. Salman, O. Tisserand and B. Toulout, FORTH
L.E. Scales, Introduction to Non-linear Optimization
Peter S. Sell, Expert Systems - A Practical Introduction
Colin J. Theaker and Graham R. Brookes, A Practical Course on Operating Systems
J-M. Trio, 8086-8088 Architecture and Programming
M.J. Usher, Information Theory for Information Technologists
Colin Walls, Programming Dedicated Microprocessors
B.S. Walker, Understanding Microprocessors
Peter J.L. Wallis, Portable Programming
I.R. Wilson and A.M. Addyman, A Practical Introduction to Pascal - with BS6192, second edition

Non-series
Roy Anderson, Management, Information Systems and Computers
J.E. Bingham and G.W.P. Davies, A Handbook of Systems Analysis, second edition
J.E. Bingham and G.W.P. Davies, Planning for Data Communications

Smalltalk-80

A. Mével and T. Guéguen

Edited by
Mario Wolczko,
Department of Computer Science,
University of Manchester

MACMILLAN
EDUCATION

Authorised English Language edition of Smalltalk-80, by A. Mével and T. Guéguen, first published 1987 by Editions Eyrolles, 61 boulevard Saint-Germain, 75005 Paris

Translated by M.J. Stewart

First published 1987

Published by
MACMILLAN EDUCATION LTD
Houndmills, Basingstoke, Hampshire RG21 2XS
and London
Companies and representatives throughout the world

British Library Cataloguing in Publication Data
Mével, A.
Smalltalk-80. --- (Macmillan computer science series).
1. Smalltalk -80 (Computer system)
I. Title II. Guéguen, T.
005.13'3 QA76.8.S635

ISBN 978-0-333-44514-3 ISBN 978-1-349-09653-4 (eBook)
DOI 10.1007/978-1-349-09653-4

Contents

Preface

With the advent of affordable personal computing power, the Smalltalk-80 system is rapidly gaining in popularity. The Smalltalk-80 system can provide the personal computer user with a glimpse into the future of the programming process over the next decade.

This book is an introduction to programming in the Smalltalk-80 system. It describes the major elements of the system, namely the programming language, class structure and user interface, and serves as a concise reference for the most important classes in the system. Also included are many examples of Smalltalk-80 usage, and the source to a complete application. The description is based on the version 2 Virtual Image from Xerox, probably the most popular version of Smalltalk in use today.

The Smalltalk-80 system is large, and one cannot hope to describe it all in a book of this size. For a more comprehensive source, the reader is referred to Smalltalk-80: The language and its Implementation, by A. Goldberg and D. Robson, Addison-Wesley, 1983, and Smalltalk-80: The Interactive Programming Environment, by A. Goldberg, Addison-Wesley, 1984.

1 Introduction

1.1 Presentation

Smalltalk is one of several things: a programming language, an operating system, a programming environment, and a design and programming methodology. Many people consider Smalltalk to be a tool for knowledge representation, thus making it a serious competitor to Lisp and Prolog. It has moved on from its position as internal support system at the Xerox Palo Alto Research Center, which it held for ten years, to being used for the industrial production of software. The application areas in which it excels are application prototyping, interactive graphics systems and simulation. Although there are still some problems with speed of execution (mainly stemming from the interpretive nature of the system), spectacular results have been achieved in cost savings with certain types of software, especially in applications containing a high graphics or interactive component. The principal reason for this is that the Smalltalk environment takes the maximum advantage of the reuse of software components.

1.2 Smalltalk and its growth

Smalltalk is available on an increasing number of modern machines. It was first developed at Palo Alto in the beginning of the 1970s at the instigation of Alan Kay, and has evolved through three important stages:

- Smalltalk-72 was inspired by the Flex machine and the languages Lisp and Plasma
- Smalltalk-76 formalised the class, object and inheritance concepts inspired by Simula-67
- Smalltalk-80 is the latest version of the language, which was released from the Palo Alto laboratories.

Highlighted below are some of the stages in the design and development of Smalltalk at the PARC laboratories leading up to its general publication.

October 1972 building of first Smalltalk interpreter
December 1972 definition of Smalltalk-72 according to the following basic principles

- control with automatic de-allocation
- interaction by messages
- behaviour of objects defined by classes

1974 Smalltalk-74
- introduction of message dictionaries
- graphic control by BitBlt

1976 Smalltalk-76
- unification of classes and contexts as objects
- inheritance by class hierarchy

1978 Smalltalk-78
- portability achieved with NoteTaker
- improvement of performance
- experimental implementation on an 8-bit microcomputer (Z80 or 6502) TinyTalk

Summer 1979 first publications on Smalltalk produced. Invitations to several companies to carry out experimental implementations and participate in the evaluation of the product: Apple, DEC, Tektronix, Hewlett Packard

August 1981 publication of an issue of 'Byte' devoted to Smalltalk

September 1983 first Smalltalk licences granted by Xerox

Summer 1984 appearance of first implementations of Smalltalk on different machines (Sun, SM90, etc)

Early 1985 Tektronix 4404 workstation made available

Spring 1985 'Methods' (a Smalltalk clone) available for the IBM PC (DigiTalk Inc) and Smalltalk for the Apple Lisa (Macintosh XL)

Autumn 1985 Smalltalk available on the Apple Macintosh 512K (Apple) and on the IBM PC/AT (SoftSmarts Inc). More powerful Tektronix 4405 and 4406 models announced.

The essential characteristic of Smalltalk is its economy of design. In fact, this is based on the four concepts of object, message, class and inheritance.

Other mechanisms, such as the block or the dictionary are seen by the programmer as constructs developed from these basic concepts. This effort towards unification can be pushed even further, but it currently has no equivalent in other languages and systems. What it does do is to facilitate rapid mastery of the programming tools available.

From the outset of his research work, Alan Kay established one of the principles that was later to be followed in the different versions of Smalltalk. That was the principle of reaction. At any time, an object is active and must be capable of presenting itself to the user in a tangible form. This property greatly facilitates the creation and development of models.

One important difference between the architecture of a traditional product and that of a product based on Smalltalk is the absence of any frontier between the application and the system, which is thus completely open (this system is called the virtual image in Xerox terminology). To take an example, in a traditional

approach, it is possible to find an application that implements a table control by hash coding while the compiler also uses the same algorithm. With Smalltalk, it would be a design error not to reuse something that is already present in the system.

Programming with Smalltalk is incremental by nature and therefore lends itself very well to the progressive development of models (by stepwise refinement). This has two important consequences:

- There is an intentional fusion of the roles of programmer and user, because the same metaphors are used. This means that the user of a program will be able to move progressively from simple utilisation to modification and extension of the program with the same type of interaction.
- Of course, it will become possible to mix intimately the different activities of programming and debugging.

It would be possible to expand on other characteristics of Smalltalk that improve programming efficiency. For example, one can cite the abandonment of the concept of mode (in the context of insertion mode, query mode, etc). One can also think of the principle of strong modularity which requires that no part of the system depends upon details of the internal functioning of other parts, this property being induced by the architecture of communicating objects. However, it is above all the importance of the graphics interface and its good integration with the system and the language that one associates with Smalltalk.

The graphics interface consists of a bit-mapped screen and a three-button mouse that permits zones of pixels to be designated on the screen. These devices allow

- working in an environment consisting of several windows
- mixing text and graphics
- maintaining a dialogue very easily, thanks to pop-up menus which disappear as soon as the button that calls them is released
- selecting points, areas and text, all with great ease.

It should be appreciated that the graphics screen and the mouse pointing device have nothing to do with ´add-ons´ to an existing system. They are thoroughly integrated into the programming environment which exploits them to the full.

Many research projects are currently evaluating the value of Smalltalk-80 in industrial software production. It is still too soon to pass overall judgement on this product, but some characteristics can be underlined here and now.

The fashion for object oriented languages goes beyond the phenomenon of Smalltalk. A large number of tools that make explicit reference to the same concepts are available or currently appearing. Without striving to be exhaustive, the following can be mentioned: Formes (P. Cointe), Kool (P.Albert), Ulysses (Y. Autret), Mering (J. Ferber), Ceyx (J.M. Hullot), X-Lisp (D.Betz), R-Lisp (J.L. Roos), Ross (D. McArthur), Loops (Bobrow), Glisp (Novak), Actl and Act2 (Liebermann and Theriault), Flavors (Cannon and Moon), LRO (Ch. Roche), C++ (B. Stroustrup), Objective-C (Cox and Love), Clascal (Apple), Methods (Digitalk Inc), ObjectivePascal (N. Wirth and Apple), Oblogis (P.Y. Gloess), Neon (Kryia System Inc), Eiffel (B. Meyer), etc. All of these, mentioned at random, have quite diverse fields of application, but they all, to varying degrees, share certain aspects of Smalltalk. We can therefore say that the language conceived of by Alan Kay has had an important influence on the evolution of software production tools. The question is to establish whether this influence will be direct (effectively using Smalltalk), or indirect (borrowing ideas). Simula-67, which Smalltalk draws on considerably, has had a great influence on current programming languages, but its influence has been mainly indirect.

- The key to the question that has just been put on the future of the industrial use of Smalltalk depends very much on the choice that educational institutions make on the use of this language for their courses. The current availability of Smalltalk would indicate important growth potential from 1986 onwards. The present barrier to the wider industrial spread of Smalltalk is not the cost of the necessary workstations, but the lack of staff trained to use it.

- The extreme flexibility of Smalltalk allows the programmer to redefine the majority of his system functions with no restrictions. This can pose problems in a shared programming environment. Solutions exist, but are not yet integrated. However, for use in prototyping, this is not a serious handicap and the economies achieved promise increasing use of this tool.

There are currently numerous development projects based on Smalltalk. The most serious test of a widespread use of this product will be the existence of an exchange supply of software components written in Smalltalk.

The aim of this book is not to expound the theory of object languages but rather to offer a practical guide to programming in Smalltalk. There can be no question of developing here all the aspects of Smalltalk, because in its most recent versions it contains more than 230

classes and 4500 methods. Nevertheless, we have tried not to neglect any of the important points of Smalltalk; in particular, we have devoted several chapters to the user interface and more precisely to the MVC (Model, View, Controller) interface system that forms the basis of every Smalltalk interface.

2 Principles of the Language

2.1 Objects and messages

All the entities used in Smalltalk-80 are objects. For example, an object can represent a number, a character, a drawing, a list, a program, an editor, etc.

An object has two main characteristics:

- it allows data belonging to it to be stored and accessed
- it can reply to a certain number of messages.

For example, the object 3.14 will respond to the message **sin** by returning the value 0, which is written:

3.14 sin

The only means of manipulating an object is to send it a message. Messages therefore constitute the interface between objects and the external world. The way in which an object responds to a message is private to it and does not have to be known externally.

Writing an application in Smalltalk consists of determining the objects necessary for the description of the problem and in defining the possible operations for each object.

For example, let us take a computer-aided drawing application. This must allow us to draw easily all simple geometric shapes. We will restrict ourselves to points, straight lines and circles. We can see already that just stating the problem indicates some of the objects that will have to be created; this produces a first breakdown of the problem into objects:

- drawing
- point
- line
- circle

Drawings will be defined by a list of points, a list of straight lines and a list of circles.

Points will be defined by two coordinates x and y.

Lines will be defined by two points.

Circles will be defined by a point and a radius.

The following operations are defined on drawings:

- display (displays the drawing on the screen)
- points (returns the list of points in the drawing)
- lines (returns the list of lines in the drawing)

- circles (returns the list of circles in the drawing)
- addPoint: aPoint (adds a point to the drawing)
- addLine: aLine (adds a line to the drawing)
- addCircle: aCircle (adds a circle to the drawing)

The following operations are defined on points:

- display (displays the point on the drawing)
- xy (returns the coordinates of the point)
- x: aNumber (initialises the x-coordinate of the point)
- y: aNumber (initialises the y-coordinate of the point)

The following operations are defined on lines:

- display (displays the line on the drawing)
- origin (returns the startPoint of the line)
- corner (returns the the endPoint of the line)
- origin: aPoint (initialises the startPoint of the line)
- corner: aPoint (initialises the endPoint of the line)

The following operations are defined on circles:

- display (displays the circle on the drawing)
- centre (returns the centre point of the circle)
- radius (returns the radius of the circle)
- centre: aPoint (initialises the centre of the circle)
- radius: aNumber (initialises the radius of the circle).

This example demonstrates the modular aspect of Smalltalk. When the basic objects of an application and the operations on these objects are defined, the programmer can use them without knowing the detail of their construction; he can also reuse them in contexts that are quite different from those for which they were constructed.

2.2 Classes and instances

In the previous example, a drawing is made up of several points, several lines and several circles. All the points must interpret the messages sent to them in the same way.

This leads us to group together objects of the same kind into classes (that is, those objects that represent

the same sort of entity and interpret messages in the same way).

Thus, the class of points whose elements will be particular points is defined.

Objects contained in a class are called instances of this class. The class describes the form in which information about its instances are stored; it also describes the way in which its instances will reply to the messages that they receive. In a class the instances are represented by a certain number of variables called instance variables. For the class of points the instance variables are the coordinates x and y; for the class of lines the instance variables are the origin and the corner.

2.3 Methods

The way in which an object responds to a message is described in a method. All instances of a class use the same method to respond to a message. Each class contains a list of methods that allows its instances to respond to the messages that are sent to them.

3 Classes and Instances

3.1 Declaration of variables

In the Smalltalk environment we encounter several types of variable that differ in their scope. These different types are:

- instance variables
- class variables
- global variables
- pool variables

3.1.1 Instance variables

Instance variables are variables that belong to an object and which allow it to be differentiated from other instances of its class. We can distinguish two types of instance variable, those that are indexed and those that are named.

Named instance variables

These are instance variables that are identified by a name. For the instances of the class **Point** there are two named instance variables, x and y.

When the name of an instance variable appears in an expression, this name refers to the value of the corresponding instance.

When a new instance is created, it contains instance variables specified by its class; the default value of these instance variables is the undefined object called **nil**.

Indexed instance variables

These are instance variables that are not accessed by name but by a numeric index. In contrast to objects that possess only named instance variables, those that possess indexed instance variables may have a varying number of instance variables.

In Smalltalk-80 there is for example a class with indexed instance variables which allows arrays to be represented; for each instance the number of variables corresponds to the size of the table.

3.1.2 Class variables

Class variables are variables that are shared by all the

instances of a class. For example, in the case of a computer-assisted drawing application, the outline of a circle is made up of several line segments; the number of these segments is the same for all circles, and one can therefore define it as a class variable.

3.1.3 Global variables

Global variables are variables that are shared by all objects. They are stored in a dictionary called Smalltalk. For example, all the classess are referenced by global variables whose name is the access key to the dictionary called Smalltalk.

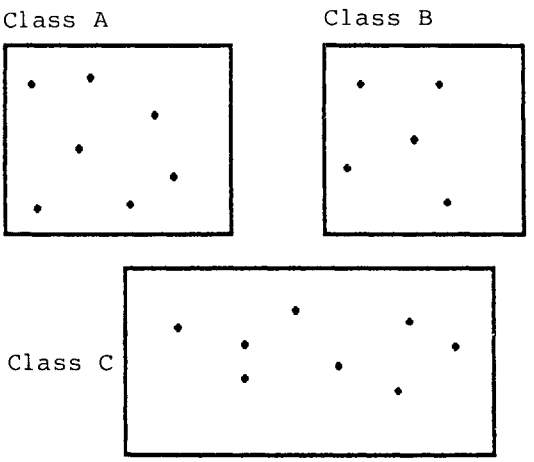
3.1.4 Pool variables

Pool variables are variables that are accessible by the instances of several classes. In order to define a set of variables shared by several classes, it is necessary to define a dictionary that is common to these classes, this dictionary being itself a global variable.

3.2 Classes and subclasses

3.2.1 General remarks

Each object belongs to one and only one class, as can be represented thus:



It can, however, be useful to share some elements of the description between several classes or to describe one class by means of another.(See also section 5.2.2.)

This is what happens in Smalltalk, where each class is described in terms of another class called its superclass. The instances of the new class are identical to those of its superclass, except for the additions

made explicit in the new class. The new class thus defined is a subclass of its superclass.

One subclass is itself a class and can therefore also have one or more subclasses. The set of classes thus takes on a tree structure whose root is the class called **Object.** Object is the only class that does not possess a superclass.

The instances of each subclass inherit from all the instance variables and from all the class variables, as well as from all the methods of the superclass.

3.2.2 Definition of a class

To give a complete definition of a class, we need the following elements:

- its name
- its superclass
- its new class variables
- its new pool variables
- its new instance variables
- the list of its new methods.

The name of the class is obligatory and must be different from the name of any other existing class.

The superclass is obligatory and must correspond to a class already in existence.

Variables are optional, but if they exist their name must be different from any name already defined in the set of its superclasses.

New methods may be added, or methods already defined in the set of its superclasses may be redefined.

3.3 Metaclasses

As we have said earlier, each Smalltalk entity is an object. Consequently, a class is itself an object. We have also seen how each object belongs to one and only one class. A class is therefore itself an instance of another class that we call its metaclass. In earlier versions of Smalltalk all the classes were instances of one single metaclass. For reasons of flexibility (especially when creating new instances of a class), each class is the sole instance of a metaclass.

The metaclasses are classes and therefore contain methods that allow their instances (that is, the classes) to respond to the messages that they receive. As a result, these methods will be called class methods, while the other methods will be called instance methods. When a class is created, a new metaclass is automatically created. Among the messages sent to classes we find for example the message new, which allows a new instance of the class that received this message to be created.

In contrast to other classes, metaclasses have no name; neither are they metaclass instances. They are all instances of the same class called **Metaclass.**

One can access the metaclass of a class by sending it the message class. So, if we call Point the class of points, we can access its metaclass with the expression:

```
Point class
```

The metaclasses are classes and therefore have a superclass called **Class.**

To summarise, we can regard metaclasses in two ways:

- as instances of the class **Metaclass,**
- as a subclass of the the class **Class.**

Metaclass and **Class** are two subclasses of the class **ClassDescription.**

4 Syntax

An expression is a sequence of characters which, when evaluated, always returns an object. In Smalltalk-80 there are four lexical types, as follows:
- literals that represent certain predefined objects like numbers, characters, character strings;
- variable names that represent variables that can be accessed in the context of the expression;
- messages;
- blocks that represent an expression whose evaluation is postponed.

4.1 Literals

4.1.1 Numbers

Numbers are objects that represent numerical values and respond to mathematical messages.

A number may be positive or negative; if it is negative it will be preceded by the minus sign -.

A number may be a decimal or an integer; if it is decimal it includes the decimal point in its sequence of figures, for example:

```
4
4.56
-7
-7.234
432
```

When the base used is not base 10, the number begins with its base expressed in base 10, followed by the character 'r' and ending with the expression of the number in that base. When the base is greater than 10, the first letters of the alphabet are used to represent numbers greater than 9, for example:

Smalltalk expression	Base 10 equivalent
2r10	2
2r1010	10
2r1.1	1.5
8r11	9
8r123	83
16rA1	161
16r-A1	-161

A number may be expressed in scientific notation; in this case the list of figures is followed by the character 'e' and the exponent is expressed in base 10. The number in front of the exponent is multiplied by the base to the power of the exponent, for example:

Scientific notation	Base 10
2.549e3	2459
-2.459e-3	-0.002459
2r10e3	16
2r10e-3	0.25

4.1.2 Characters

Characters represent the elements of the alphabet. They are indicated by the '$' followed by a character, for example:

```
$a
$A
$$
$9
```

4.1.3 Character strings

A string of characters represents a sequence of characters. This is indicated by placing the list of characters between quote marks. If the string itself contains an apostrophe, this must be doubled in order to avoid errors in interpretation, for example:

```
'this is a string'
'it''s a string too'
```

4.1.4 Symbols

Symbols are strings of characters that are used to represent the names of objects in the system. They are written by prefixing them with the hash '#', for example:

```
#name1
#ThisIsASymbol
```

4.1.5 Arrays

An array is a structured object whose elements may be accessed by their numeric index. Each element of the array is a literal. It is indicated by the hash and the list of its elements is enclosed by brackets. If an array or a symbol is included among the elements of the array, there is no need for a hash in front of these elements, for example:

```
#(1)
#(2 7 9)
#(1 'one' one (1 one 'one is a string'))
```

4.2 Variables

A variable allows an object to be referenced; all variables except indexed variables have a name. A variable name may be used in an expression to designate an object. From a given object one may access objects referenced by variables whose name is known by this object.

Because the range of instance variables of an object is limited to the object itself, they will be inaccessible from any other object, and can therefore only be used by methods defined for this object.

A variable name is a sequence of alphanumeric characters and must begin with a letter, for example:

```
v1
nameOfVariable
NameOfVariable
```

By convention, variables whose range is limited to the object (instance variables and temporary variables) begin with a lower case letter, and variables that are accessible by several objects (global variables, pool variables, class variables) begin with an upper case letter.

To assign a value to a variables, the special character '<-' is used, preceded by the variable name followed by an expression whose evaluation will return an object that will then be referenced by the variable, for example:

```
variable1 <- 1
variable2 <- 'variable 2 references a string of
characters'
variable3 <- Point class
variable4 <- variable5 <- 123
```

4.3 Messages

Messages allow interactions between objects, for example:

```
2 sqrt
```

returns the square root of 2,

```
2 + 8
```

returns the sum of 2 and 8,

```
variable1 + 4
```

returns the sum of 4 and the number referenced by variable 1,

```
variable1 > variable4
```

compares the objects referenced by the two variables,

```
10 raisedTo: 2
```

returns 10 raised to the power of 2.

4.3.1 Composition of a message

An expression requires a receiver to appear, followed by

a selector and possibly arguments. The receiver and the arguments are described by expressions, while the selector is described by a literal, for example:

```
2 sqrt
```

the receiver is 2, the selector is sqrt,

```
2 + 8
```

the receiver is 2, the selector is +, the argument is 8,

```
variable1 + 4
```

the receiver is the object referenced by variable1, the selector is +, the argument is 4,

```
variable1 > variable4
```

the receiver is the object referenced by variable1, the selector is >, the argument is the object referenced by variable4,

```
10 raisedTo: 2
```

the receiver is 10, the selector is raisedTo:, the argument is 2.

There are three types of messages, characterised by the number of arguments:

Unary messages
These are messages that have no argument, for example:

```
2 sqrt
```

Keyword messages
This is the most common form of message. The selector is made up of one or more keywords preceding each argument. Each keyword terminates with ':', for example:

```
10 raisedTo: 2
```

the selector is raisedTo:, the unique keyword is raisedTo:, the argument is 2,

```
list at:2 put: 1
```

the selector is at:put:. the keywords are at: and put:, the arguments are 2 and 1.

Binary messages
In the case of a single argument, it is possible not to use a keyword for the selector but to replace it with a selector made up of one or two non-alphanumeric characters; in this case the message is said to be binary, for example:

```
2 + 8
variable1 >= variable4
```

4.3.2 Value returned by a message

In response to the message the receiver returns an object which becomes the value of the expression. In the following example

```
2 + 3 * 4
```

the first message sent to object 2 has as its selector +, and as its argument 3; the object returned in response to this message is object 5. The expression then becomes

```
5 * 4
```

and the result returned by the first expression is the object 20.

In the case where the expression includes an assignment, the object returned is the object referenced by the assignment variable. For example, the expression

```
variable1 <- 2 + 3 * 4
```

returns the object 20.

4.3.3 Message priority

Messages are taken in order from the left of the expression, with the following priorities (in decreasing order of precedence):

- unary message
- binary message
- keywords message

In the example

```
10 raisedTo: 2 + 3 sqrt
```

the highest priority message is the unary message sqrt; the value returned will be 1.73205. The second message will be the binary message +, the returned value being 3.73205. The last message will be the keyword message raisedTo:, with the value returned being 10 raised to the power of 3.73205, that is 5395.74.

In order to alter the order in which messages are sent, parentheses may be used. The highest priority message is the one enclosed between parentheses. For example, in the expression

```
2 + (3 * 4)
```

the first message sent will be *, which will go to the object 3 with argument 4; the result will be 12, which will become the argument for the selector + of the second message. The final object returned will thus be 14.

When several keywords appear in a message, they are grouped under a single selector. For example, in the expression

```
10 raisedTo: 2 raisedTo: 3
```

the selector will be raisedTo: raisedTo:, which is not a correct selector. Parentheses must therefore be used, as follows:

```
(10 raisedTo: 2) raisedTo: 3
```

or

```
10 raisedTo: (2 raisedTo: 3)
```

4.3.4 Cascaded messages

It is possible to send several messages to the same object by separating these messages with semi-colons. In the following example

```
list at: 1 put: 'one'
list at: 2 put: 'two'
list at: 3 put: 'three'
```

could be written

```
list at: 1 put: 'one' ; at: 2 put: 'two' ; at: 3 put: 'three'
```

The use of cascaded messages is not obligatory but it does make the writing of expressions more concise.

4.4 Blocks

4.4.1 Description of blocks

Blocks are Smalltalk objects used in control structures. A block is a sequence of expressions separated by full stops; it begins with the character '[' and ends with the character ']'. Evaluation of a block must be requested explicitly; this allows it to to be used in control structures. An example of a block is:

```
[variable1 <- 1. variable2 <- 2]
```

If there is a point between the last expression and the final bracket, it will be ignored.

A block is an object that can execute expressions placed within brackets if it receives the message value. For example

```
[variable1 <- 1. variable2 <- 2] value
```

and

```
variable1 <- 1. variable2 <- 2
```

will produce the same result.

Evaluation of a block returns the value returned by the last expression in the block. When a block contains no expression the value returned is nil.

4.4.2 Blocks with arguments

It is possible to pass one or more arguments to a block before its execution. In a block the arguments are specified at the beginning of the block by identifiers beginning with the colon :. Arguments are separated from expressions by the bar character |.

For example, the block

```
[:i | variable1 <- i]
```

is a block with an argument that is i. The block

```
[:i :j | variable1 <- i. variable2 <- j]
```

is a block with two arguments, i and j.

The arguments of a block are variables local to that block.

To evaluate a block with an argument, you must send it the message value: with an argument that will be passed to the block. In the event of two arguments, the message to be used is value:value:.

Use value: value: value: and value:value:value:value: for three or four arguments; use valueWithArgs: anArray for more.

4.4.3 Use of blocks in control structures

When we refer to classic programming languages like Pascal or C, we find that they contain three types of control structure: the loop (for, do), the test (if then else) and the conditional loop (while, until). These structures are also found in Smalltalk-80 and make use of blocks. Smalltalk-80 does not have an equivalent to the multi-branch (case) statement.

Loops

Starting with a list of elements it is possible to execute a block for each element in the list, with each element becoming the argument of the block. This is done by sending the message do: to a list of elements. For example

```
listIntegers do: [:eachInteger | sum <- sum +
eachInteger]
```

will have the effect of calculating the sum of all the integers contained in the list.

Tests

Boolean objects are returned from messages like >, <, >=, <= etc. These Boolean objects, true and false, respond to the messages ifTrue:, ifFalse:, ifTrue:ifFalse: and ifFalse:ifTrue:. The arguments of these messages are blocks that will be evaluated or not, depending on whether the receiver is true or false. In the example

```
anInteger < 0 ifTrue: [anInteger <- 0 - anInteger]
```

the sign of anInteger is changed if it is negative.

In the example

```
anInteger < 0 ifTrue: [anInteger <- 0] ifFalse:
[anInteger <- anInteger sqrt]
```

anInteger is set to 0 if it is negative, otherwise its square root is assigned to it. To clarify the expression, it can be formatted differently and may for example be written as follows:

```
anInteger < 0 ifTrue: [anInteger <- 0]
              ifFalse: [anInteger <- anInteger sqrt]
```

Conditional loops

A conditional loop operates for as long as a condition is true or holds. This condition is evaluated at each

iteration of the loop. That can for example be achieved by sending the message whileTrue: or the message whileFalse: to a block with another block as argument. The example below allows the factorial of a number to be calculated iteratively

```
fact <- 1.
[aNumber > 1] whileTrue:
[fact <- fact * aNumber.aNumber <- aNumber - 1]
```

So long as aNumber is greater than 1, the value returned by the evaluation of the first block is true and therefore the second block is evaluated. As soon as aNumber is equal to 1, the evaluation of the first block returns false and the iteration stops. The result will be referenced by the variable fact. This example can be handled with the message whileFalse:

```
fact <- 1.
[aNumber <= 1] whileFalse:
[fact <- fact * aNumber.aNumber <- aNumber - 1]
```

Note that these control structures are constructed from the basic language features, namely, blocks methods and messages, and require no extra features in the language.

For example, the ifTrue: ifFalse: control structure is built from two methods, one in class **True** (whose sole instance is true),

```
ifTrue: aBlock ifFalse aSecondBlock
    ↑aBlock value
```

and another in class **False** (whose sole instance is false),

```
ifTrue: aBlock ifFalse: aSecondBlock
    ↑aSecondBlock value
```

5 Methods

5.1 Messages and methods

A method describes how an object is going to reply to a message. The name of a method is the same as the name of the message to which the method must allow a response to be made. A method consists of its name followed by a list of expressions separated by full stops. For example, the method of the class Point which returns the x-coordinate of that point will be written

```
x                       message name
  ↑x                    return of instance variable x
```

The method of class Point which alters the x-coordinate of the point will be written

```
x: aValue               message name
  x <- aValue           assigns to x the value aValue
```

The method of class Point which displays the point on the drawing will be written

```
display                 message name
 |square|               definition of a variable
                        local to method
 square <- Form new.    square becomes a new
                        instance of Form
 square extent: 2@2     size of square is fixed as 2
                        x 2 pixels
 square black.          square is black
 square displayAt: x@y  display square at position
                        (x,y)
```

Form is a predefined Smalltalk class.

new is a method of class which creates a new instance.

extent:, black and displayAt: are three instance methods of the class Form.

The arguments found in the method name are variables local to the method and cannot be altered. For example

```
x:aValue
  aValue <- aValue + 1.
  x <- aValue
```

produces a syntax error.

5.1.1 Methods search

When an object receives a message, it looks in its class

for a method having the same selector as the message received. If it does not find it, it continues its search in the superclass of its class; this search is continued through the sequence of its superclasses until the correct method is found.

If on reaching the root class, Object, the method has not been found, there is an error. The receiver is then sent the message doesNotUnderstand:, whose argument is the message whose selector has not been found.

The method which replies to the message doesNotUnderstand: is set up in the class Object and is therefore known to all objects. It indicates to the programmer that there is an error.

If the search is successful, each expression of the method found is evaluated. When all expressions have been evaluated, the receiver returns to the sender an object which is the response to the message received.

5.1.2 Object returned by a method

When a method terminates the receiver returns an object to the sender. The default object is the receiver itself. When it is required to return another object, this is indicated in the method by prefixing the object to be returned with the character '↑'.

As soon as this character is found, the object produced by the expression that follows is returned to the sender and the method terminates. For example, the following method of class Point

```
xPositive
  x > 0 ifTrue: [↑true]
        ifFalse: [↑false]
```

will return true if the x-coordinate of the point is positive, and false otherwise. This method may be written more simply as

```
xPositive
  ↑x > 0
```

5.1.3 Temporary variables

As we saw in the example dealing with the display method in class Point, a variable, square, was introduced. Temporary variables like this may be declared in a method, but they are accessible only during execution of the method.

To declare temporary variables their name must be placed between two '|' characters.

Declaration of these variables must occur between the method name and the list of expressions. Every time that a method is executed these variables are set to nil. For

example, the method

invertXandY

```
    |temporaryVariable|
    temporaryVariable <- x.
    x <- y
    y <- temporaryVariable
```

allows the coordinates of a point to be inverted using the temporary variable called temporaryVariable.

5.2 Pseudo-variables

A pseudo-variable is a variable whose value cannot be altered by assignment.

Some pseudo-variables are constants; that is, they always refer to the same object. So far we have met three, namely

- nil the default value of a variable
- true the object returned by a true condition
- false the object returned by an false condition.

Other pseudo-variables depend on the context in which they are used. Such is the case with the pseudo-variables, self and super.

5.2.1 Pseudo-variable self

The pseudo-variable **self** always refers to the receiver of the message itself. Every method has access to the self pseudo-variable. For example, the method

abs

```
  self < 0 ifTrue: [↑0 - self]
```

returns the absolute value of the receiver.

self also allows recursion to be handled, as in the following example:

factorial

```
   self = 0 ifTrue: [↑1].
   ↑self*(self - 1) factorial
```

When the expression 3 factorial is evaluated, self will represent 3 at the first pass. The method will request the response to the message (self - 1) factorial, namely 2 factorial, and so on until 0 factorial. This may be shown diagrammatically as follows:

```
|3 factorial                self = 3
|    |2 factorial           self = 2
|    |    |1 factorial      self = 1
|    |    |   |0 factorial  self = 0
|    |    |   |↑ 1
|    |    |↑ 1
|    |↑ 2
|↑ 6
```

When a message is sent to self, the search for the method corresponding to the message will begin in the

class of the object referenced by self. This class may well be a subclass of the class containing the first method. This is a consequence of the strategy used to look for a method.

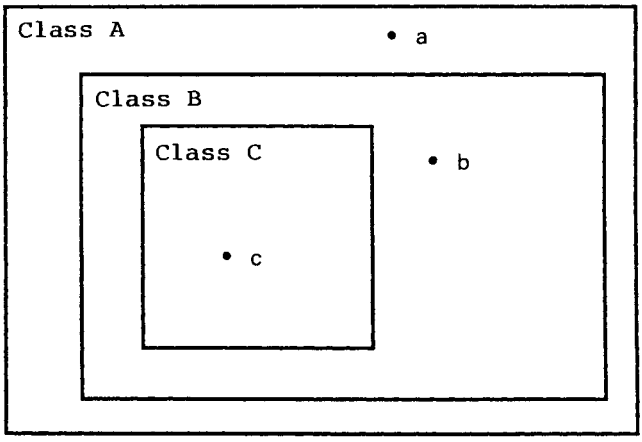
5.2.2 Super pseudo-variable

There is a second pseudo-variable, called super, that is also accessible by all methods.

This pseudo-variable again refers to the receiver, but its strategy for searching for a method is different.

When a message is sent to super, the method search begins not in the receiver's class, but in the superclass of the class that the message was sent from.

For example, say we have a hierarchy of three classes, as shown in the following figure:



in which a b c are three respective instances of classes A B C. Say that in class A the method is

```
name
    ↑'A'
```

in class B the method is

```
name
    ↑'B'
```

and in class C the method is

```
name
    ↑'C'
```

Say that in class B the methods are

```
myName
    ↑self name
```

and

```
superName
    ↑super name
```

The different results of the messages myName and superName sent to objects b and c will be

Expression	Result
c myName	'C'
c superName	'A'
b myName	'B'
b superName	'A'

One of the most frequent cases of the use of the super pseudo-variable is when redefining a method in a subclass. For example:

```
initialise
    super initialise.
    self init
```

This method calls the method with the same name that is located in a superclass of the class in which it is embedded.

5.3 Primitive methods

We have just seen that in order to respond to a message an object executes a method; execution of this method results in the evaluation of one or more expressions and therefore also to the sending of messages to other objects. To avoid an infinite regress of message sending, there must be methods that do not send any messages, but perform basic operations. These methods are not embedded in the virtual image but in the virtual machine; they are written in machine code and are referred to as primitive methods. For example, the mathematical operations + * / - are primitive methods.

There are about a hundred primitive methods; the number varies according to the implementation.

When a method is a primitive method, it begins with an expression of the form

```
<primitive #n>
```

where n is the number of the primitive.

If the primitive cannot be executed correctly, the expressions that follow it in the method are executed; this allows errors to be controlled.

6 Some Elements of the User Interface

Part of this book is devoted to the user interface in Smalltalk. However, at this point we want to give the reader some elements so that he or she may test the examples that have already been presented and those that are to follow in the book. It is worth repeating once again that learning Smalltalk is best done in a practical context; this means that the support of a machine, while not absolutely necessary, is highly desirable for tackling the remainder of this book.

6.1 General description

When compared with a traditional machine, the Smalltalk interface has three special features:

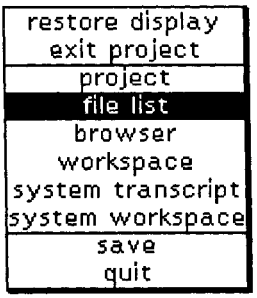
- a bitmap screen
- a three-button mouse
- multiple windows and pop-up menus.

A window is an area of the screen corresponding to the user interface of a particular application.

A pop-up menu is a menu that appears on the screen when the user presses one of the mouse buttons and which disappears when the button is released to execute the selected function.

6.2 Main menu

Depending on the window types, the pop-up menus vary. When the cursor is outside any window, the user can make the menu shown below appear by pressing the middle button of the mouse.



We shall briefly describe the main functions of this menu.

restore display
redisplays the complete screen,

file list
opens a window accessing the file system,

browser
opens a window viewing the classes and methods,

workspace
opens a window in which expressions can be evaluated,

save
saves the virtual image in a file,

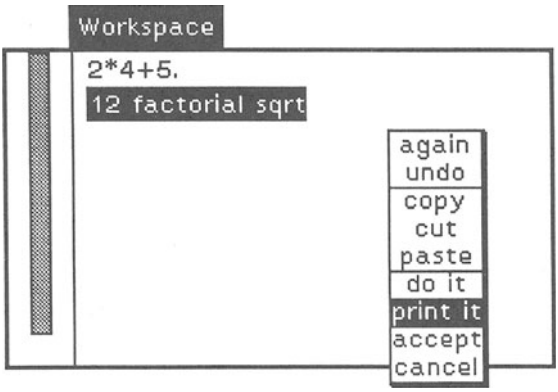
quit
exits the Smalltalk system.

6.3 Workspaces

A workspace is a window in which it is possible to edit text and evaluate expressions.

In this type of window the left-hand button of the mouse is used to designate a position in the text or select a part of this text, showing it in inverse video.

The middle button on the mouse is used to edit the text or evaluate expressions.



The main functions of the middle button menu are:

again
which repeats the last text replacement,

undo
cancels the last editing function,

copy
copies the selected text into a buffer memory,

cut
removes the selected portion of text, placing a copy in the buffer memory

paste
inserts the text held in the buffer at the position marked by the cursor or replaces the selected section of text with the contents of the buffer,

do it
evaluates the selected portion of text,

print it
evaluates the selected portion of text and displays the result of the evaluation.

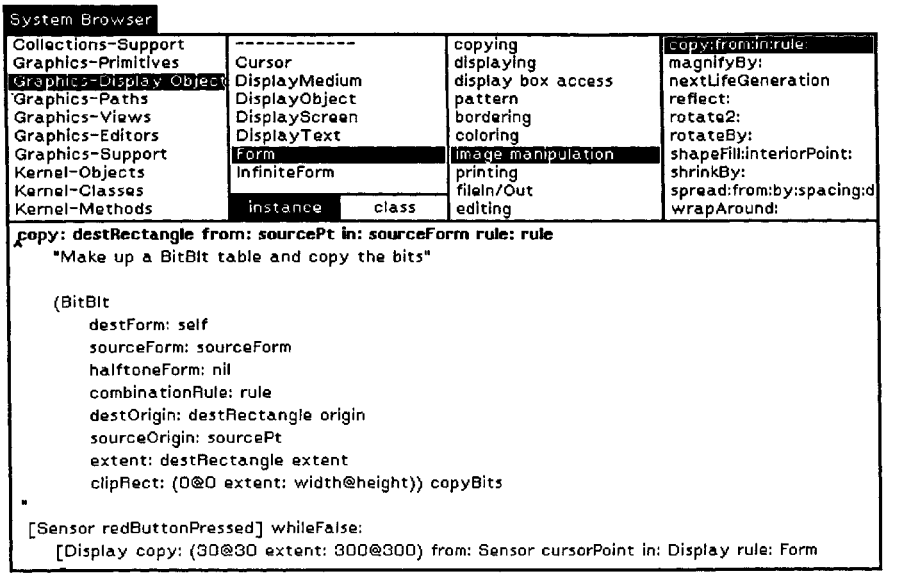
6.4 Browsers

Browsers are windows used to browse through the classes and methods in the system.

For greater clarity of presentation, classes and methods are grouped into categories. These categories are arbitrary and have no effect on the system architecture.

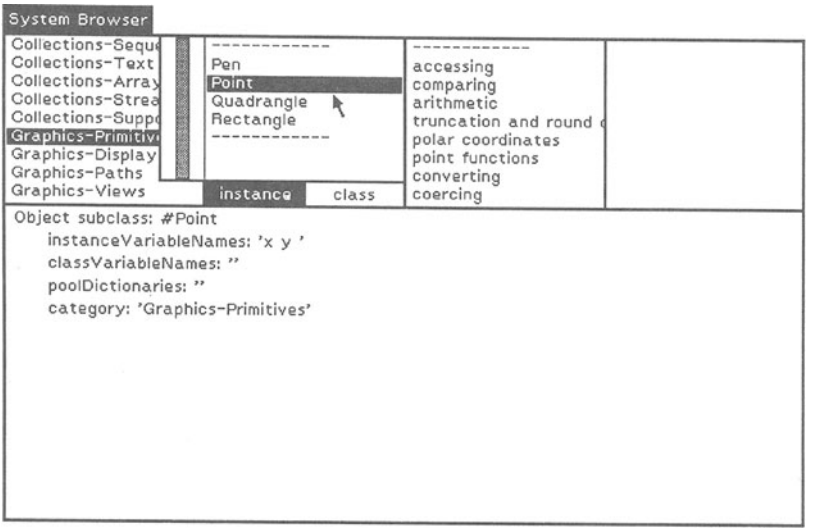
A browser type window contains five sub-windows. The four at the top are windows that can access lists; the one at the bottom is a text editing window. The four top windows contain respectively:

- the class categories - classes belonging to the same category are classes which deal with similar problems
- classes
- the methods (or protocols) categories, methods that



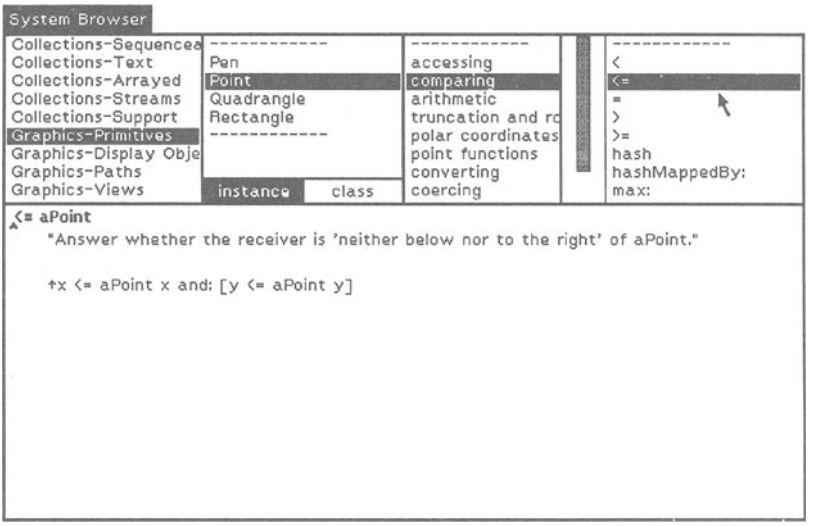
have a close functionality are grouped into the same category
- methods.

The bottom window serves mainly for editing classes and methods. When a class is selected and no method category is chosen, the bottom window contains the definition of the class. The figure below shows a browser containing the definition of the class Point.



Having edited the definition of an existing class, it is possible to alter the characteristics of this class or create a new class. In order to save these characteristics, you need to use the **accept** function of the middle button mouse menu.

When a protocol is chosen, the user can create a new method or alter an existing method in the bottom window. To link the method into the system the accept function must again be used; this has the effect of compiling the text of the method. The figure below shows a browser viewing the text of the method <= in class Point.



The two types of windows that we have just described are sufficient for testing most of the examples in this book. The other principal types of windows are:

- Debuggers - for examining the state of computation during the evaluation of an expression
- FileLists - which allow access to the external files
- Transcripts - there is generally one single window of this type in which the system displays information relating to the progress of the Smalltalk session.

7 The Class Object

The class **Object** is the superclass of all other classes and consequently the methods that are defined in it are accessible by all Smalltalk-80 objects. Of course, these methods can be redefined in any class.

7.1 Identification of an object

The characteristics of an object depend on the class to which it belongs. Certain messages allow access to these characteristics. We have already met one of these:

```
class
```

which when sent to an object returns the class of that object.

For example,

```
2 class
```

returns **SmallInteger**,

```
$A class
```

returns **Character**.

There are others:

```
isKindOf: aClass
```

allows one to find out if the receiver is an instance of the argument aClass or of one of its subclasses.

For example,

```
2 isKindOf: SmallInteger
```

returns true,

```
2 isKindOf: Number
```

returns true, because SmallInteger is a subclass of Number.

```
isMemberOf: aClass
```

allows one to find out if the receiver is a direct instance of aClass.

For example,

```
2 isMemberOf: SmallInteger
```

returns true,

```
2 isMemberOf: Number
```

returns false.

```
respondsTo: aSelector
```

allows one to find out if the receiver understands the message. For example,

```
2 respondsTo: #+
```

returns true,

```
2 respondsTo: #at:
```

returns false.

The selectors of messages are instances of class **Symbols** and are therefore prefixed with the hash.

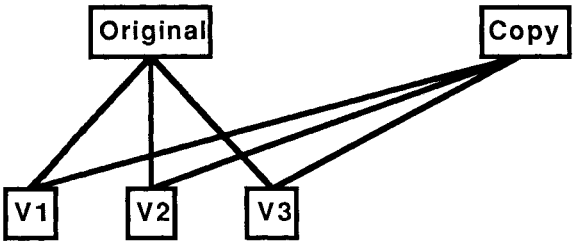
7.2 Copies of objects

It is possible to copy an object, that is, create a new object having the same characteristics as the original.

There are two ways of making a copy, depending on whether the instance variables of the object are duplicated or not.

7.2.1 Non-duplicated instance variables

Instance variables are shared between the original and its copy, as in the following diagram.



If the original or its copy alters one of its instance variables, this alteration is also made for the other object. The message to make this copy is

```
shallowCopy
```

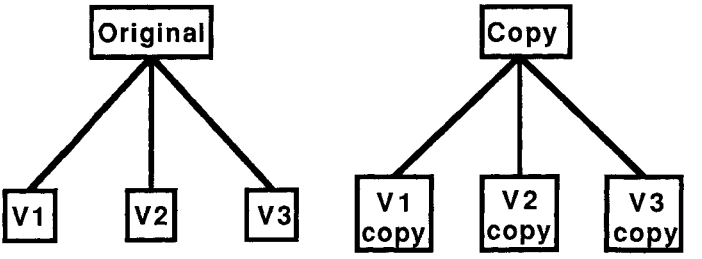
For example,

```
a <- 'ABC'
b <- a shallowCopy
```

assigns a copy of a to b.

7.2.2 Duplicated instance variables

Instance variables belong to the copy; there is no longer any dependence between the original and its copy. We have the following arrangement:



The copy is made in a recursive way at the level of the instance variables; that is, the objects referenced by the instance variables can themselves have their own instance variables. These instance variables are also duplicated up to the point of having objects without instance variables.

The message to make this copy is

deepCopy

For example, if f1 is an instance of class **Form** (that is, a bitmap - see 15.2.3), then after evaluating the following expressions (the reverse message changes black to white and vice versa):

```
f2 <- f1 deepCopy.
f3 <- f1 shallowCopy.
f1 reverse
```

f2 will contain a copy of the original bitmap, whereas f1 and f3 will both refer to the original bitmap, which has been reversed.

7.2.3 Copy message

The class **Object** contains the definition of the method that will respond to the message

copy

This method calls shallowCopy directly. In the sub-classes of Object it is possible to redefine this method according to the nature of the objects to be copied.

Some variables can thus be duplicated while others cannot.

7.3 Comparisons of objects

Two types of comparisons may be made. The equivalence and the equality of two objects can be tested.

7.3.1 Equivalence of two objects

Two expressions are equivalent if they refer to the same object. The message for testing equivalence is

==anObject

For example:

```
variable1 <- anObject.
variable2 <- anObject.
variable1==variable2
```

returns true, because variable 1 and variable 2 refer to the same object. We can also test if two objects are not equivalent with the message

~~anObject

7.3.2 Equality of two objects

Whereas equivalence is defined in class Object, and should not be redefined, equality should be defined appropriately for each class.

For example, two instances of class Point are considered to be equal if their x- and y-values are equal, but are not equivalent unless they are the same object:

```
variable1 <- 3@4
variable2 <- 3@4
variable1 = variable2 returns true
```

but variable1==variable2 returns false.

7.3.3 Comparison with nil

There are two messages to compare an object with the nil object. The message

```
isNil
```

returns true if the receiver is nil. The message

```
notNil
```

returns true if the receiver is not nil.

7.4 Indexed objects

We have seen how an object can have named instance variables and/or indexed instance variables. For objects with indexed instance variables, there are several messages that allow access to the objects referenced by these variables. These are

```
at: anIndex
```

which returns the object referenced by the instance variable that has the index anIndex.

```
at: anIndex put: anObject
```

allows the object anObject to be assigned to the instance variable of index anIndex.

```
basicAt: AnIndex
```

is equivalent to at:, but must not be redefined in the subclasses of Object.

```
basicAt: anIndex put: anObject
```

is equivalent to at:put:, but must not be redefined in the subclasses of Object.

```
size
```

returns the number of indexed instance variables in the receiver.

```
basicSize
```

is equivalent to size, but must not be redefined in the subclasses of Object.

For example,

```
8 size                          returns 0
'ABC' size                      returns 3
#(1 2 3 (2 4)) size             returns 4
(#(1 2 3 (2 4)) at: 4) size     returns 2
```

7.5 Representation of objects

Smalltalk-80 represents objects internally in a binary form which is not readable by users. Therefore, it also provides methods to generate representations of objects in a human-readable form. There are two possible forms of external representation:

- a first external representation in summary form,
- a second representation which is an expression that, if evaluated, will recreate the object.

There are two methods in class Object that create such representations. These are

```
printString
```

which returns a character string giving a summary representation of the receiver, and

storeString

which returns a character string giving a syntactic representation of the receiver.

For example,

#(2 3) printString

will return the character string '(2 3)',

#(2 3) storeString

will return the character string '#(2 3)'.

(Set with: 1 with: 2) printString

will return the character string 'Set(1 2)',

(Set with: 1 with: 2) storeString

will return the character string '((Set new) add: 1; add: 2; yourself)' which allows an identical copy of the receiver to be reconstructed.

7.6 Control of errors

The search for a method begins at the level of the class of the receiver; it continues through the hierarchy of superclasses. If the method is not found, there is an error; this is conveyed by the message doesNotUnderstand: with as its argument the message that caused the error. This method signals the error to the programmer; it is defined within the class Object and is therefore accessible to all objects. This method can be redefined for a particular class, for example, to make an automatic correction.

There are a number of other methods defined within class Object that are intended to control errors. These methods are:

error:aString

indicates to the user that there is an error and displays the string aString,

primitiveFailed

indicates to the user that a primitive method has failed,

shouldNotImplement

indicates to the user that the method that should respond exists but should not be used,

subclassResponsibility

indicates to the user that the method must be redefined in the class of the receiver.

7.7 Control of messages

It is sometimes useful to choose which message to send to an object by computing the selector within an expression.

One cannot simply write:

receiver variableSelector

because the receiver will try to execute the method variableSelector.

There are, however, five methods that are available to deal with this problem:

perform:aSymbol
sends to the receiver the unary message aSymbol,

perform:aSymbol with:anObject
sends to the receiver the keyword or binary message aSymbol with argument anObject,

perform:aSymbol with:anObject with:anOtherObject
sends to the receiver the keyword message aSymbol with arguments anObject and anOtherObject,

perform:aSymbol with: anObject1 with: anObject2 with: anObject3
sends to the receiver the keyword message aSymbol with arguments anObject1, anObject2 and anObject3,

perform:aSelector withArguments: anArray
sends to the receiver the message aSelector with the arguments contained in anArray. There will be an error if the number of elements in the array does not correspond to the number of arguments expected by the selector.

For all the messages, there will be an error if the number of arguments does not match the number expected by the selector. For example,

```
sort: aSelector
   |test|
   self size
      to: 1
      by: -1
      do: [:j |1 to: j - 1 do:
              [:k|
                test <- (self at:k)
                   perform: aSelector
                   with: (self at: k + 1).
                test ifFalse:
                   [self swap: k with: k + 1]]]
```

is a method for sorting an array, whose argument aSelector is the sort criterion. This method must be defined in the class **SequenceableCollection.**

```
   #(3 7 0 1) sort: # <
```
will return the array #(0 1 3 7)

```
   #(3 7 0 1) sort: # >
```
will return the array #(7 3 1 0).

7.8 Dependencies between objects

In the Smalltalk-80 system it is possible to define dependencies between objects in such a way that, when an object is dependent on a second one, any alteration of the second is signalled to the first.

The class Object has defined within it methods that allow dependencies between objects to be created explicitly. These methods are:

addDependent:anObject
anObject is added to the list of objects depending on the receiver,

removeDependent:anObject
anObject is removed from the list of objects depending on the receiver,

dependents
returns an instance of **OrderedCollection** containing the objects depending on the receiver,

release
removes all the dependencies on the receiver, leaving it with no dependency on any other object,

changed
the receiver signals to all its dependent objects that it has changed by sending them the message update,

changed:aParameter
the receiver signals to all its dependent objects that it has changed by sending them the message update:aParameter, whose argument aParameter indicates the nature of the change,

update:aParameter
is the method that must respond to the message sent by changed or changed:. This method is used to update the receiver,

broadcast:aMessage
sends the unary message aMessage to all the objects depending on the receiver,

broadcast:aMessage with:anObject
sends the keyword message aMessage with the argument anObject to all the objects depending on the receiver.

We will take as an example the channel-selecting keys of a TV set. Only one of these keys can be selected at once. One key is characterised by its state (selected or not). The class Key of the keys can thus be defined with the following methods

```
class name                          Key
superclass                          Object
name of instance variables          selected
class methods

new
  ↑super new initialise
instance methods
initialise
  selected <- false
select
  selected ifFalse: [selected <- true.self changed]
```

```
update: aKey
  aKey == self ifFalse: [selected <- false]
```

The class Keyboard is also defined

```
class name                          Keyboard
superclass                          Object
instance variables name             keys
class methods

new: numberOfKeys
  ↑super new createKeys: numberOfKeys

instance methods
select: aNumberOfKey
  (keys at: aNumberOfKey) select
createKeys: numberOfKeys
  keys <- Array new: numberOfKeys.
  1 to: numberOfKeys do:
    [:index | keys at: index put: Key new].
  keys do:
    [:aKey |
      keys do:
        [:aDependentKey |
          aKey == aDependentKey ifFalse:
          [aKey addDependent: aDependentKey]]]
```

The method createKeys: allows a keyboard to be initialised with a certain number of keys. The created keys are neither selected nor deselected at the start. A key is dependent on all the other keys. Note the pseudo-variable super in the method new: which avoids recursive calls of this method.

The example can be made to run with:

```
keyboard <- Keyboard new: 5.
keyboard select: 1.
keyboard select: 4.
```

Selection of key 1 of the keyboard will cause the message update: to be sent to all the other keys, which will deselect them whatever their status.

7.9 Primitive messages

There are a certain number of primitive methods described in the class Object. These methods are:

```
become:anObject
```

allows the pointers of the receiver and of the argument anObject to be exchanged. All the variables that refer to the receiver will refer to the argument and vice versa. There will be an error if one of the two objects is an instance of the class SmallInteger,

```
instVarAt: anIndex
```

returns the named instance variable of rank anIndex. The numbering of named instance variables corresponds to their order of definition,

```
instVarAt: anIndex put: anObject
```

assigns the object anObject to the named instance variable of rank anIndex,

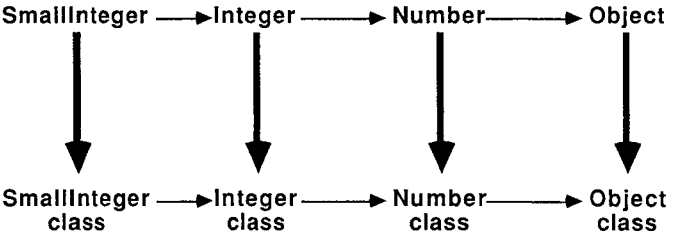
```
nextInstance
```

returns the instance following the receiver in the list of all instances of the class of the receiver. It returns nil if all the instances have been returned.

8 Description of Architecture of Classes

We know that each object is an instance of a class. As a class is itself an object, it is the instance of a class that we call its metaclass. To access the metaclass of a class we send it the message class.

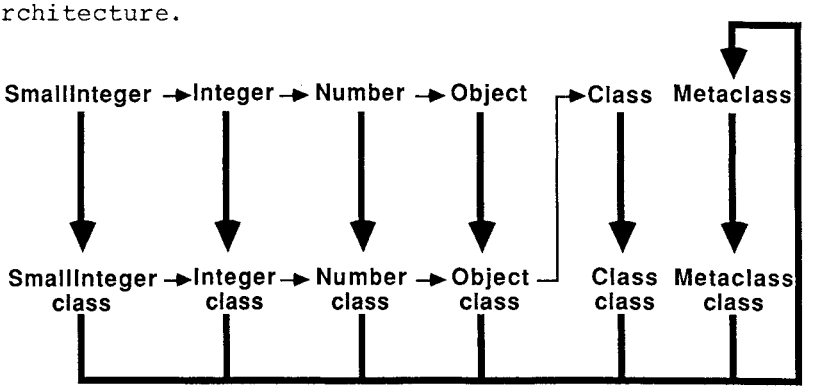
The hierarchy that appears at the level of the metaclasses is the same as that which appears at the level of the classes; that is, if a class A is a subclass of a class B, the metaclass of A will be a subclass of the metaclass of B, as in the diagram below.



Each metaclass has only one instance and thus there are exactly as many metaclasses as classes. The metaclasses are also objects and thus belong to a class that is called **Metaclass**.

Metaclass is a class and thus has a metaclass which is an instance of Metaclass. Metaclass and its metaclass are both instances of each other.

Object is the only class that does not have a superclass; on the other hand its metaclass has a superclass called **Class**. The diagram below illustrates this architecture.



8.1 Class Behavior

The class **Behavior** defines the elementary methods necessary to objects that have instances. The instances of Behavior are objects representing the classes. As instance variables they have a dictionary of methods, a pointer to a superclass and a list of subclasses.

8.1.1 Dictionary of methods

The methods described in a class are held in a dictionary called the dictionary of class methods. The keys to the dictionary are the selectors, and the objects assigned to these keys are methods in compiled form.

Creation

The methods that allow the method dictionary to be manipulated are:

methodDictionary: aDictionary

assigns a dictionary to the method dictionary of the receiver;

addSelector: aSelector withMethod: aMethod

adds a method to the method dictionary of the receiver, with selector aSelector and for compiled method aMethod;

removeSelector: aSelector

removes from the method dictionary of the receiver that method with selector aSelector.

Compilation

The methods that allow one or more methods to be compiled are:

compile: code

compiles the text of the expressions contained in the argument code. The argument code is a string of characters. An error is produced if the argument code cannot be compiled,

compile: code notifying: requestor

compiles the text of the expressions contained in the argument code. If the argument code cannot be compiled, an error is signalled to the requestor object by a message,

recompile: aSelector

compiles the method in the method dictionary whose key is aSelector;

decompile: aSelector

decompiles the method in the method dictionary whose key is aSelector. It returns a string of characters representing the decompiled form of the compiled method;

compileAll

compiles all the methods in the method dictionary;

compileAllSubclasses

compiles all the methods contained in the method dictionaries of the subclasses of the receiver.

Access to selectors or methods

The methods that allow access to the selectors or methods in the method dictionary are:

selectors

returns an instance of Set containing the selectors in the method dictionary of the receiver,

allSelectors

returns an instance of Set containing all the selectors in the method dictionary of the receiver and of its superclasses, that is, all the selectors assigned to the messages that are understandable by an instance of the receiver,

compiledMethodAt: aSelector

returns the compiled method assigned to the selector aSelector contained in the method dictionary of the receiver,

sourceCodeAt: aSelector

returns a string of characters corresponding to the method assigned to the selector aSelector in the method dictionary of the receiver,

sourceMethodAt: aSelector

returns an instance of **Text** corresponding to the method assigned to the selector aSelector in the method dictionary of the receiver. In this book the name of the method is shown in bold type (as are class names).

Some examples of the use of these methods might be:

Number selectors

returns Set (to:do: even roundTo: storeOn: .... log: tan rounded)

Number allSelectors

returns Set (raisedTo: asPoint between:and: negated .... min:max: truncated)

```
Number sourceCodeAt: #abs
returns:
'abs
    "Answer a Number that is the absolute value
(positive magnitude) of the receiver"
    self <0 ifTrue: [↑self negated]
            ifFalse: [↑self]'
```

Testing

The contents of the methods dictionary can be tested in order to find out the methods that are accessible by the instances of the receiver or the methods that possess certain characteristics. The methods that allow these tests to be carried out are:

hasMethods

returns true if the methods dictionary of the receiver is not empty,

includesSelector: aSelector

returns true if the instances of the receiver can res-

pond to the message with selector aSelector, that is, if the selector is a key in the method dictionary of the receiver or of its superclasses,

whichClassIncludesSelector: aSelector

returns the first class contained in the sequence of superclasses of the receiver whose method dictionary contains the key aSelector; it returns nil if no class is found,

whichSelectorsAccess: instanceVariableName

returns an instance of Set containing the selectors of the methods in the method dictionary of the receiver that access the instance variable called instanceVariableName,

whichSelectorsReferTo: anObject

returns an instance of Set containing the selectors of methods in the method dictionary of the receiver that access the object anObject.

scopeHas: aName ifTrue: aBlock

executes the block aBlock if the variable called aName is accessible by the receiver.

Here are some examples of the use of these methods:

Number hasMethods

returns true because the method dictionary of the class Number is not empty;

Number includesSelector: # ==

returns false because this method is not in the method dictionary of the class Number but in the method dictionary of the class Object;

Number canUnderstand: # ==

returns true because this method is contained in the method dictionary of a superclass of the class Number;

Number whichClassIncludesSelector: # ==

returns class Object.

8.1.2 Instances

The instances of a class are created by sending the message new or new: to it. These methods are defined in class Behavior, but can be redefined in any subclass. If a class redefines the method new or new:, the method will necessarily contain the message super new in order to avoid a recursive call of itself. It can be useful to redefine these methods if, for example, you want to initialise instance variables with values other than nil.

Creation

The four methods defined in class Behavior that allow instances to be created are:

new

returns a new instance of the receiver. This instance

has no indexed instance variables;
basicNew
is identical to new, but this method must not be redefined;
new: anInteger
returns a new instance of the receiver with a number of indexed instance variables given by anInteger;
basicNew: anInteger
is identical to new:, but this method must not be redefined.

Here are some examples of the use of these methods:

```
matrix1 <- Array new.
matrix2 <- Array new: 3.
```

matrix1 is an instance of the class Array which has no indexed instance variables. The message new is equivalent to the message new: 0 for those classes that have indexed instance variables.

Accessing

There are different methods that allow one or more instances of a class to be accessed. These methods are:

allInstances
returns an instance of OrderedCollection containing all the direct instances of the receiver (without the instances of its subclasses).
someInstance
returns a direct instance of the receiver,
instanceCount
returns the number of direct instances of the receiver,
instVarNames
returns an instance of Array containing the names of the named instance variables,
subclassInstVarNames
returns an instance of Set containing the names of the named instance variables of the subclasses of the receiver,
allInstVarNames
returns an instance of Array containing the names of the named instance variables of the receiver and of its superclasses,
classVarNames
returns an instance of Set containing the names of the class variables of the receiver,
allClassVarNames
returns an instance of Set containing the names of the class variables of the receiver and of all its superclasses,
sharedPools
returns an instance of Set containing the names of the dictionaries that contains the shared variables,
allSharedPools

returns an instance of Set containing the names of the dictionaries that contain the shared variables defined in the receiver and its superclasses.

Enumeration
There are certain methods that allow an operation to be carried out on each instance. These methods are:

allInstancesDo: aBlock

evaluates the block aBlock for each instance of the receiver,

allSubInstancesDo: aBlock

evaluates the block aBlock for all the subclass instances of the receiver.

Testing
Other methods allow the instance variables of the receiver to be tested

isPointers

returns true if the instance variables of the receiver are stored in the form of pointers,

isBits

returns true if the instance variables of the receiver are stored in the form of bits,

isBytes

returns true if the instance variables of the receiver are stored in the form of bytes,

isWords

returns true if the the instance variables of the receiver are stored in the form of words,

isFixed

returns true if the instances of the receiver have no indexed instance variables,

isVariable

returns true if the instances of the receiver have indexed instance variables,

instSize

returns the number of named instance variables of the receiver.

Some examples are:

Number isFixed

returns true because the instances of Number do not have indexed instance variables,

Array instSize

returns 0 because the instances of Array do not have named instance variables.

8.1.3 Manipulation of the class hierarchy

Creation
Methods defined in class Behavior allow alteration of

the class hierarchy by setting a class's superclass and subclasses. These methods are:

superclass: aClass

fixes the superclass of the receiver at aClass,

addSubclass: aClass

adds the subclass aClass to the receiver,

removeSubclass: aClass

removes the subclass aClass from the receiver.

Accessing

There are methods that allow the hierarchy of the receiver to be accessed. These are:

subclasses

returns an instance of Set containing the immediate subclasses of the receiver,

allSubclasses

returns an instance of OrderedCollection containing all the subclasses of the receiver, that is, its immediate subclasses and the subclasses of its subclasses, and so on,

withAllSubclasses

returns an instance of OrderedCollection containing the receiver and all of its subclasses,

superclass

returns the superclass of the receiver,

allSuperclasses

returns an instance of OrderedCollection containing the superclass of the receiver followed by the other superclasses in hierarchical order; the last element is always Object.

For example,

Number subclasses

returns Set (Float Fraction Integer),

Number allSubclasses

returns OrderedCollection (Fraction Integer Float SmallInteger LargePositiveInteger LargeNegativeInteger),

Integer superclass

returns the object representing the class Number,

Integer allSuperclasses

returns OrderedCollection (Number Magnitude Object).

Testing

There are certain methods that allow the hierarchy of the receiver to be tested. These methods are:

inheritsFrom: aClass

returns true if the class aClass belongs to the sequence of superclasses of the receiver,

kindOfSubclass

returns a string describing the type of the receiver. There are four types of class in Smalltalk-80:

- classes which have no indexed instance variables

where the named instance variables represent pointers to other objects; the returned string will be 'subclass:'
- those classes which have indexed instance variables that can represent
- pointers; the returned string will be 'variableSubclass:'
- bytes; the returned string will be 'variableByteSubclass:'
- words; the returned string will be 'variableWordSubclass:'

For example:
Integer inheritsFrom: Number
returns true because Number is the superclass of Integer,
Integer kindOfSubclass
returns 'subclass:',
Float kindOfSubclass
returns 'variableWordSubclass:'.

Enumeration
There are certain methods that allow operations to be carried out on a part of the class hierarchy related to the receiver. These methods are:
allSubclassesDo: aBlock
evaluates the block aBlock for all the subclasses of the receiver, that is, for its immediate subclasses and the subclasses of its subclasses, and so on
allSuperclassesDo: aBlock
evaluates the block aBlock for all the superclasses of the receiver,
selectSubclasses: aBlock
evaluates the block aBlock for all the subclasses of the receiver and groups in an instance of Set all the classes for which evaluation returns true. This instance of Set is returned.
selectSuperclasses: aBlock
evaluates the block aBlock for all the superclasses of the receiver and groups in an instance of Set all the classes for which evaluation returns true. This instance of Set is returned.

8.2 The classes ClassDescription and Class

The description of a class is not totally described by the class Behavior. Behavior does not allow the names of instance variables and class variables to be stored; nor does it allow a name to be given to the class or a comment describing the class to be stored.

The class **ClassDescription**, which is a subclass of the class Behavior, and the class Class, which is a subclass of the class ClassDescription, are the classes

that allow a name to be given to a class, a comment to be assigned to it and the names of named instance variables to be stored.

8.2.1 Access to the description of a class

The additional methods that allow access to the description of a class are:

name

returns a string representing the name of the receiver,

comment

returns a string giving the comment assigned to the receiver,

comment: aString

assigns the string aString to the commentary associated with the receiver,

addInstVarName: aString

adds to the list of named instance variables of the receiver the named instance variable whose name is aString,

removeInstVarName: aString

removes from the list of named instance variables of the receiver the named instance variable whose name is aString. It returns an error if aString is not the name of a named instance variable.

8.2.2 Category of classes and messages

To clarify the description of the class hierarchy, the classes are grouped into categories. This grouping is arbitrary and simply allows classes that relate to the same problem to be grouped together; for example, the category Numeric-Numbers groups classes Number, Integer, Float, Fraction, SmallInteger, LargePositiveInteger, LargeNegativeInteger, Random. This grouping has no effect on the hierarchy of the classes.

Just as classes can be grouped into categories, so can the methods of classes.

The methods that allow access to the category of a class or a method are:

category

returns the category of the receiver,

category: aString

assigns to the category of the receiver the string aString,

removeCategory: aString

removes all the methods from the methods dictionary of the receiver whose category is aString,

whichCategoryIncludesSelector: aSelector

returns the category of the method whose selector is aSelector; it returns nil if the method is not found in the method dictionary of the receiver.

8.2.3 Copy of messages

Some methods are provided that copy one or more methods from another class. These methods are:

copy: aSelector from: aClass

copies the method of selector aSelector described in the class aClass and places it in the method dictionary of the receiver,

copy: aSelector from: aClass classified: aCategory

copies the method of selector aSelector described in the class aClass and places it in the method dictionary of the receiver under category aCategory,

copyAll: arrayOfSelectors from: aClass

copies the methods whose selectors are contained in the arrayOfSelectors described in the class aClass and places them in the method dictionary of the receiver,

copyAll: arrayOfSelectors from: aClass classified: aCategory

copies the methods whose selectors are contained in the arrayOfSelectors described in the class aClass and places them in the method dictionary of the receiver under the category aCategory,

copyAllCategoriesFrom: aClass

copies all the methods described in the class aClass and places them in the method dictionary of the receiver,

copyCategory: aCategory from: aClass

copies all the methods described in the class aClass under the category aCategory and places them in the method dictionary of the receiver,

copyCategory: aCategory from: aClass classified: aNewCategory

copies all the methods described in the class aClass under the category aCategory and places them in the method dictionary of the receiver under the category aNewCategory.

8.2.4 Compilation of methods

There are two particular methods that allow source code to be compiled and the result placed under a category. These two methods are:

compile: aSourceCode classified: aCategory

compiles the text of the expressions contained in the argument aSourceCode and places the compiled method obtained in the method dictionary of the receiver under category aCategory. An error is indicated to the programmer if the argument aSourceCode cannot be compiled:

compile: aSourceCode classified: aCategory requestor: aRequestor

compiles the text of the expressions contained in the argument aSourceCode and places the compiled method obtained in the methods dictionary of the receiver under category aCategory. An error is sent to the argument

aRequestor if the argument aSourceCode cannot be compiled.

8.2.5 Access to variable names

Three methods allow variable names to be returned in the form of strings. These three methods are:

classVariablesString

returns a string of characters containing the names of all the class variables of the receiver,

instanceVariablesString

returns a string of characters containing the names of all the instance variables of the receiver,

sharedPoolsString

returns a string of characters containing the names of all the dictionaries that contain the shared variables of the receiver.

8.2.6 Saving a class on a file

Three methods allow the description of a class to be saved on a file. These methods are:

fileOutOn: aFile

saves the description of the receiver on the file aFile which is an instance of the class **FileStream**,

fileOutCategory: aCategoryName

creates a file whose name is the name of the receiver concatenated to the string 'st' and places in it all the methods whose category is aCategoryName,

fileOutChangedMessages: aListOfChanges on: aFile

the argument aListOfChanges is made up of class/message couplets; it is an instance of the class Set. The description of the couplets class/message is saved on the file aFile, which is an instance of the class FileStream.

The class ClassDescription possesses another subclass called Metaclass.

8.3 The class Metaclass

The metaclasses are instances of the class Metaclass. We have seen that the role of a metaclass is to allow the initialisation of class variables and the creation of instances of the class that is assigned to it. Each metaclass contains an instance variable called thisClass which allows its unique instance to be referenced.

8.4 Multiple inheritance

We have seen that a class inherits variables and methods from its superclass. In the general case a class may inherit from a single other class (this is so for all the classes of the Smalltalk system), but it is also possible for a class to inherit from several other classes; this is called multiple inheritance.

Multiple inheritance allows the properties of two or more existing classes in the system to be fused into one class. Multiple inheritance imposes certain additional constraints that arise from the conflicts that may appear between the different superclasses:

- A class cannot inherit from two classes that have a same instance variable (that is, a variable having the same name).
- When creating a class with multiple superclasses, there may be a conflict between different inherited methods with the same name. When this occurs, a new method with the same name is automatically created in the new class which signals an error if activated. If the user wishes to resolve the conflict, he should edit the new method to call one of the inherited methods, designating the method by its name, prefixed with the name of its class and a full stop. For example:

```
Point.+
```

designates the method of addition of two points.

To implement multiple inheritance, the Smalltalk system possesses a subclass of Metaclass called **MetaclassForMultipleInheritance**, whose instances have an instance variable called otherSuperclasses which contains the list of additional superclasses. The metaclasses of multiple inheritance classes are instances of MetaclassForMultipleInheritance.

To create a class that inherits from several other classes the following message must be sent to the class Class

```
named: aSymbol
superclass: namesOfSuperclasses
instVariableNames: namesOfInstanceVariables
classVariableNames: namesOfClassVariables
category: nameOfCategory
```

For example, consider the case of simulating a transport network in which the following two classes are defined:

```
MotorVehicle
```

which has two instance variables, namely:

```
tankCapacity and operatingHours
WheeledVehicles
```

which has one instance variable:

```
mileage
```

The class Automobile embraces the two aspects of classes MotorVehicle and WheeledVehicles; thus, to define it we need to construct it as the subclass of these two classes:

```
Class named: #Automobile
    superclasses: 'MotorVehicle WheeledVehicles'
    instVariableNames: 'numberOfSeats'
    classVariablesNames: ''
```

```
category: 'simulation-network'
```

Those methods that conflict will have to redefined; for example, the method printOn: which generates a textual representation of an object.

For multiple inheritance to be attractive to use, the number of conflicting methods needs to be small; that is, the two superclasses should have few aspects in common.

9 The Magnitude Classes

The magnitude classes contain the objects on which an ordering relation can be established. They are objects for which the concept of size has a meaning.

9.1 The class Magnitude

The class **Magnitude** is a subclass of the class Object. All the ordered classes are subclasses of Magnitude; this is why Magnitude describes comparison methods that may be used by all the instances of its subclasses.

These methods are:

< aSize

returns true if the receiver is less than the argument aSize,

<= aSize

returns true if the receiver is less than or equal to the argument aSize,

> aSize

returns true if the receiver is greater than the argument aSize,

>= aSize

returns true if the receiver is greater than or equal to the argument aSize,

between: min and: max

returns true if the receiver is greater than or equal to the argument min and less than or equal to the argument max,

min: aSize

returns the minimum of the receiver and the argument aSize,

max: aSize

returns the maximum of the receiver and the argument aSize,

= aSize

this message must be redefined in subclasses of Magnitude. In class Magnitude the method = is described by the unique expression self subclassResponsibility.

For example:

```
2 > 1                      returns true
1 > 2                      returns false
2 between: 1 and: 3        returns true
2 min: 5                   returns 2
2 max: 5                   returns 5
```

9.2 The class Date

The class **Date** is a subclass of Magnitude. Its instances are dates that represent a day, a month and a year of the calendar. It knows the days of the week and the months of the year by storing index/name pairs.

There are a number of methods that allow these dates to be manipulated.

9.2.1 Creation of instances

When a new date is created, it is initialised with special values. The different creation methods are:

today

returns a new instance of the class Date representing today's date,

fromDay: numberOfDays

returns a new instance of Date representing the date obtained by adding numberOfDays to the first of January 1901,

newDay: aDay month: aMonth year: aYear

returns a new instance of Date whose day is aDay, the month aMonth and the year aYear. The arguments aDay and aYear are integers, whereas the argument aMonth is a symbol.

newDay: numberOfDays year: aYear

returns a new instance of Date representing the date obtained by adding numberOfDays to the first of January of the year aYear.

For example:

```
Date today                          returns 22 October 1986
Date fromDays: 3                    returns  4 January 1901
Date newDay: 8 month: #Oct year: 86
                                    returns  8 October 1986
Date newDay: 3 year: 86             returns  4 January 1986
```

9.2.2 Information about the calendar

It is possible to obtain general information about the calendar through the following methods:

dayOfWeek: nameOfDay

returns the number of the day nameOfDay in the week (1 for #Monday, etc),

nameOfDay: aNumber

returns a symbol representing the day of the week with number aNumber,

indexOfMonth: nameOfMonth

returns the number of the month nameOfMonth in the year (1 for #January or #Jan, etc). The name of a month can be abbreviated,

nameOfMonth: aNumber

returns a symbol representing the month of the year of number aNumber,

```
   daysInMonth: nameOfMonth forYear: aYear
returns  the number of days of the month nameOfMonth for
the year aYear,
   daysInYear: aYear
returns the number of days of the year aYear,
   leapYear: aYear
returns 1 if the year is a leap year, otherwise 0,
   dateAndTimeNow
returns an instance of Array whose first element is  the
actual date and second element the actual time.
   For example:
   Date dayOfWeek: #Tuesday        returns 2
   Date nameOfDay: 3               returns Wednesday
   Date indexOfMonth: #Mar         returns 3
   Date nameOfMonth: 4             returns April
   Date daysInMonth: #February forYear: 1984
                                   returns 29
   Date daysInYear: 1984           returns 366
   Date leapYear: 1984             returns 1
   Date dateAndTimeNow             returns (a Date a Time)
```

9.2.3 Arithmetic

There are three methods for carrying out arithmetic operations on dates. These methods are:

```
   addDays: numberOfDays
returns  a  date  obtained by adding numberOfDays to the
receiver,
   subtractDays: numberOfDays
returns a date obtained by subtracting numberOfDays from
the receiver,
   subtractDate: aDate
returns  the number of days separating the receiver from
the argument aDate.
   For example:
   Date today addDays: 10          returns 1 November 1986
   Date today subtractDays: 10     returns 12 October 1986
   Date today subtractDate:
   (Date newDay: 5 month: #Oct year: 1986)
                                   returns 17
```

9.2.4 Conversion

In order to be able to carry out calculations on dates and hours both are converted into seconds. The method that allows a date to be converted into seconds is:

```
   asSeconds
```

which returns the number of seconds that have elapsed since 1 January 1901 up to the beginning of the current day.

9.3 The class Time

The class **Time** is a subclass of the class Magnitude. The instances of Time represent an instant of a day.

9.3.1 Creation of instances

Three methods allow an instance of Time to be created. These three methods are:

now

returns an instance of Time representing the current time,

fromSeconds: numberOfSeconds

returns an instance of Time representing the time calculated by adding numberOfSeconds to midnight:

dateAndTimeNow

returns an instance of Array whose first element is the actual date and whose second element is the actual time.

9.3.2 Information about the time

The class Time can respond to a certain number of general messages about the time:

millisecondsClockValue

returns the number of milliseconds since the clock passed zero,

millisecondsToRun: aBlock

returns the number of milliseconds it takes to execute the block aBlock,

timeWords

returns the number of seconds that have elapsed since 1 January 1901 at the Greenwich meridian.

totalSeconds

returns the number of seconds that have elapsed since 1 January 1901, allowing for corrections for time zones and summertime.

9.3.3 Conversions

In order to be able to carry out calculations on dates and times, it is useful to be able to convert these into seconds. The method that allows this to be done is:

asSeconds

returns the number of seconds that have elapsed since midnight.

9.3.4 Arithmetic

Two methods allow arithmetic operations to be carried out on times. These are:

addTime: aPeriod

returns an instance of Time representing the time calculated by adding aPeriod to the receiver. The argument aPeriod must be an instance of Date or of Time,

subtractTime: aPeriod

returns an instance of Time representing the time calculated by subtracting aPeriod from the receiver. The argument aPeriod must be an instance of Date or of Time.

9.4 The class Character

The class **Character** is also a subclass of the class Magnitude. It is an ordered class because the instances of this class represent ASCII codes. There are therefore 256 instances in this class; they belong to the Smalltalk system and cannot be altered. It is not possible to make new Characters.

9.4.1 Access to the instances

The methods that allow access to the instances of the class Character are:

```
   value: anInteger
```

returns the Character with ASCII code anInteger,

```
   digitValue: anInteger
```

returns the character representing the digit corresponding to anInteger (when anInteger is greater than 9, it returns the first letters of the alphabet),

```
   asciiValue
```

returns the ASCII code of the receiver,

```
   digitValue
```

returns the number corresponding to the digit represented by the receiver.

For example:

```
Character value: 67                 returns $C
Character digitValue: 1             returns $1
Character digitValue: 10            returns $A
$A asciiValue                       returns 65
$B digitValue                       returns 11
```

9.4.2 Testing the instances

The following methods test for the nature of the instances of the class Character:

```
   isAlphaNumeric
```

returns true if the receiver is a letter or a digit,

```
   isDigit
```

returns true if the receiver is a digit (0-9),

```
   isLetter
```

returns true if the receiver is a letter,

```
   isLowerCase
```

returns true if the receiver is a lower case letter,

```
   isUpperCase
```

returns true if the receiver is an upper case letter,

```
   isSeparator
```

returns true if the receiver is one of the separator characters (cr, tab, line feed, form feed),

```
   isVowel
```

returns true if the receiver is an upper or lower case vowel.

For example:

```
$+ isAlphaNumeric                    returns false
$2 isDigit                           returns true
$t isLetter                          returns true
$G isLowerCase                       returns false
$G isUpperCase                       returns true
$; isSeparator                       returns false
$a isVowel                           returns true
```

The following chapter describes another subclass of Magnitude, namely the class Number.

10 The Numeric Classes

The instances of the numeric classes are the objects that we use to carry out arithmetic operations or mathematical calculations. Each type of numeric value is represented by a class. All the numeric classes are subclasses of class Number which has three direct subclasses.

- the class **Float**
- the class **Fraction**
- the class **Integer.**

The class Integer has three subclasses

- the class **SmallInteger**
- the class **LargePositiveInteger**
- the class **LargeNegativeInteger.**

The classes Number and Integer are abstract subclasses, that is, they have no direct instance.

10.1 The class Number

A certain number of messages are understood by all the numeric objects. These messages allow arithmetic operations and comparisons to be made. The majority of these messages are redefined in the subclasses of class Number. The messages common to all the subclasses of class Number are described below.

10.1.1 Arithmetic

The messages allowing arithmetic operations on numbers to be carried out are:

+ aNumber

returns the sum of the receiver and the argument aNumber,

- aNumber

returns the difference between the receiver and the argument aNumber,

* aNumber

returns the product of the receiver and the argument aNumber,

/ aNumber

returns the quotient between the receiver and the argument aNumber. If the division is not whole the result is an instance of the class Fraction,

// aNumber
returns the whole part of the division of the receiver by the argument aNumber,
\\ aNumber
returns the whole part of the remainder of the division of the receiver by the argument aNumber,
abs
returns the absolute value of the receiver,
negated
returns the opposite of the receiver,
quo: aNumber
returns the whole part of the division of the receiver by the argument aNumber,
rem: aNumber
returns the whole part of the remainder of the division of the receiver by the argument aNumber,
reciprocal
returns the inverse of the receiver, producing an error if the receiver is equal to 0.
For example:

Expressions	**Results**
2 + 3	5
2 - 3	-1
2 * 3	6
4/2	2
4/3	(4/3) instance of the class Fraction with denominator 3 and numerator 4
4//3	1
-4//3	-2
4 \\ 3	1
-4 \\ 3	1
-5 abs	5
5 negated	-5
-4 quo: 3	-1
-4 rem: 3	-1
7 reciprocal	(1/7)

10.1.2 Mathematics

The messages that allow mathematical operations to be carried out on numbers are:
exp
returns the exponent of the receiver,
ln
returns the naperian logarithm of the receiver,
log: aNumber

returns the logarithm to base aNumber of the receiver,
    floorLog: aNumber
returns the whole part of the logarithm to base aNumber of the receiver,
    raisedTo: aNumber
returns the receiver raised to the power aNumber,
    raisedToInteger: anInteger
returns the receiver raised to the power anInteger. The argument anInteger must be an instance of class Integer.
    sqrt
returns the square root of the receiver,
    squared
returns the square of the receiver,
    sin
returns the sine of the receiver expressed in radians,
    cos
returns the cosine of the receiver expressed in radians,
    tan
returns the tangent of the receiver expressed in radians,
    arcSin
returns the arcsine of the receiver in radians,
    arcCos
returns the arccosine of the receiver in radians,
    arcTan
returns the arctangent of the receiver in radians.
    For example:

Expressions	**Results**
1 exp	2.71828
1 ln	0.0
2 log: 2	1.0
3 floorLog: 2	1.0
4 raisedTo: 2.4	27.8576
4 raisedToInteger: 3	64
2 sqrt	1.41421
9 squared	81

10.1.3 Tests

There are the following messages that allow the characteristics of a number to be tested.
    even
returns true if the receiver is an even number,
    odd
returns true if the receiver is an odd number,
    negative
returns true if the receiver is negative,
    positive
returns true if the receiver is positive or zero,
    strictlyPositive

returns true if the receiver is strictly positive,

sign

returns 1 if the receiver is strictly positive; -1 if it is strictly negative; 0 if it is zero.

10.1.4 Truncation and rounding

Several messages allow a number to be truncated or rounded. These messages are:

ceiling

returns the smallest integer greater than or equal to the receiver,

floor

returns the largest integer less than or equal to the receiver (that is, its whole part),

truncated

returns the receiver without its decimal part,

truncateTo: aNumber

returns the largest multiple of the argument aNumber whose absolute value is less than or equal to the absolute value of the receiver,

rounded

returns the integer closest to the receiver,

roundTo: aNumber,

returns the multiple of the argument aNumber closest to the receiver.

For example:

Expressions	Results
2.3 ceiling	3
-2.3 ceiling	-2
2.3 floor	2
-2.3 floor	-3
2.3 truncated	2
-2.3 truncated	-2
2.3 truncateTo: 0.7	2.1
-2.3 truncateTo: 0.7	-2.1
2.3 rounded	2
2.6 rounded	3
-2.3 rounded	-2
-2.6 rounded	-3
2.3 roundTo: 0.7	2.1
2.6 roundTo: 0.7	2.8

10.1.5 Conversions

A number can represent an angle expressed either in degrees or in radians. Two messages allow conversions of angles to be made:

degreesToRadians

the receiver being expressed in degrees, it returns a

number representing the receiver expressed in radians,

radiansToDegrees

the receiver being expressed in radians, it returns a number representing the receiver expressed in degrees.

10.1.6 Nature of the object returned by arithmetic operations

Arithmetic operations can be broken down into two categories.

- Those in which the argument and the receiver are instances of the same class, for example:

  2 + 3

- Those in which the argument and the receiver are not instances of the same class, for example:

  2.2 + 3

When the receiver and the argument are instances of the same class, one is able to determine the class of the result. Depending on the values represented by the argument and the receiver, the result returned may belong to different classes.

For example:

8/4 returns 2, an instance of class Integer

8/3 returns (8/3), an instance of class Fraction.

When the receiver and the argument are not instances of the same class, one of the operands must be transformed to get back to the preceding case. The transformation must be made with the minimum loss of data. The answer is therefore to transform the item that contains the minimum of data.

Thus a hierarchy of numbers can be defined depending on their generality. This is as follows:

Float

Fraction

LargePositiveInteger LargeNegativeInteger

SmallInteger

All numbers can be represented by instances of the class Float (see below), but the precision is not infinite.

There are three messages for making conversions:

coerce: aNumber

returns a number representing the number aNumber that is an instance of the the same class as the receiver. This method must be defined in each subclass of Number.

generality

returns the degree of generality of the receiver,

retry: aSelector coercing: aNumber

the receiver and the argument not being instances of the same class, the least general must be converted, then the selector message aSelector must be re-evaluated. For example, in the expression

3.7*2

the argument 2 and the receiver 3.7 are not instances of the same class. Evaluation of the expression will then look like

3.7 retry #* coercing: 2

which will transform 2 into 2.0 and will return the result 7.4.

10.2 The class Float

The instances of class Float represent real numbers with a finite precision and lying between two limiting values. For example, the precision may be to 6 figures, with the numbers lying between 10^{32} and 10^{-32}.

Some examples of floating number are:

1.2 -1.2 2.1e4 -2.1e4

Whenever one of the operands in an arithmetic expression is a Float, the result is a Float.

10.3 The class Fraction

The instances of the class Fraction represent rational numbers. This representation is exact; a fraction has an integer numerator and denominator. The instances of the class Fraction can be obtained in an arithmetic expression when one of the operands is a fraction and the other is not a floating number (cf 10.1.6 and 10.2). Instances of class Fraction can also be obtained by dividing two integers, when the numerator is not an exact multiple of the denominator.

10.4 The class Integer

The instances of class Integer represent integer numbers. This class has three subclasses which are:

SmallInteger

which represents integers whose absolute value is less than a limit which allows for compact storage (typically in one word less 1 bit),

LargePositiveInteger

which represents arbitrarily large positive integers,

LargeNegativeInteger

which represents arbitrarily large negative integers,

Calculations are much more costly when they involve instances of the classes LargePositiveInteger and LargeNegativeInteger.

If an operation on two instances of the class SmallInteger gives a result outside the limits of the class, the result is transformed into a LargePositiveInteger or a LargeNegativeInteger.

10.4.1 Enumeration

There is a method for evaluating a block several times. This is

timesRepeat: aBlock

which evaluates the block aBlock a number of times equal to the receiver. For example:

```
a <- 1.
5 timesRepeat: [a <- a/20 + a reciprocal]
```

assigns to a the value 1 and then evaluates the block 5 times. The final value of a is 1.41421 (close to the square root of 2).

10.4.2 Some additional arithmetic functions

The class Integer contains some arithmetic functions that belong to integers. The messages that relate to these functions are:

factorial
returns the factorial of the receiver, which must not be negative,

gcd: anInteger
returns the greatest common divisor of the receiver and the argument anInteger,

lcm: anInteger
returns the largest common multiple of the receiver and the argument anInteger.

10.4.3 Bit manipulation

An integer can be interpreted as a sequence of bits (the representation of the integer expressed to the base 2). Some messages allow access to these bits. These are:

allMask: anInteger
returns true if all the bits set in the argument anInteger are also set in the receiver,

anyMask: anInteger
returns true if at least one of the bits set in the argument anInteger is also set in the receiver,

noMask: anInteger
returns true if none of the bits set in the argument anInteger is set in the receiver,

bitAnd: anInteger
returns an integer whose bits are obtained by making a logical "and" between the corresponding bits of the argument anInteger and the receiver,

bitOr: anInteger
returns an integer whose bits are obtained by a logical "or" between the corresponding bits of the argument anInteger and the receiver,

bitXor: anInteger
returns an integer whose bits are obtained by making an "exclusive or" between the corresponding bits of the argument anInteger and the receiver,

bitAt: anIndex
returns the receiver bit positioned at anIndex,

bitInvert
returns an integer whose bits are complements of the

receiver bits,
   highBit
returns the position of the most significant bit of the receiver or the number of bits necessary to represent the receiver to base 2,
   bitShift: anInteger
returns an integer (in two's complement) obtained by shifting the receiver bits (in two's complement) of anInteger to the left.
   For example:

Expressions	**Results**
2r1010	10
2r1000	8
10 allMask: 8	true
8 allMask: 10	false
8 anyMask: 10	true
10 noMask: 8	false
10 bitAnd: 8	8
10 bitOr: 8	10
10 bitXor: 8	2
10 bitAt: 3	0
10 bitInvert	-11
10 highBit	4
10 bitShift: 2	40

10.4.4 Changing base

We have seen how a number could be expressed to another base. There is a message to do this:
   radix: anInteger
returns a representation to base anInteger of the receiver. For example:

```
2 radix: 2          returns 2r10
9 radix: 3          returns 3r100
4r10 radix: 2       returns 2r100
```

11 The Class Collection

A collection represents a set of objects that are not necessarily of the same kind.

Most of the subclasses of **Collection** use indexed instance variables. Indexed instance variables are accessed by an index.

The class Collection is an abstract class that describes the behaviour of all collections. It supports four categories of message described below.

11.1 Creation of instances

Some collections can be expressed in literal form (see chapter 5). For example

```
#(1 5 3 4)
```

represents an instance of Array with four elements,

```
'this is a string'
```

represents an instance of String with 16 elements.

The usual messages for creating an instance of a Smalltalk-80 class are new and new:. These messages are also usable for instances of the class Collection. However, there are additional messages allowing the creation of new collections in which the elements are initialised on creation. These messages are:

```
with: anObject
```

returns a new instance of the receiver containing the single element anObject,

```
with: aFirstObject with: aSecondObject
```

returns a new instance of the receiver containing the two elements aFirstObject and aSecondObject,

```
with: aFirstObject with: aSecondObject with: aThird-
Object
```

returns a new instance of the receiver containing the three elements aFirstObject, aSecondObject and aThird-Object,

```
with: aFirstObject with: aSecondObject with: aThird-
Object with: aFourthObject
```

returns a new instance of the receiver containing the four elements aFirstObject, aSecondObject, aThirdObject and aFourthObject.

For example:

```
Array with: 1 with: 5 with: 3 with: 4
```

is equivalent to #(1 5 3 4).

```
String with: $a with: $b with: $c
```

is equivalent to 'abc'.

11.2 Manipulating the elements of a collection

11.2.1 Adding elements to a collection

There are two messages that allow elements to be added to a collection. They are:

```
add: anObject
```

adds to the receiver the element anObject,

```
addAll: aCollection
```

adds to the receiver all the elements of the collection aCollection.

For example:

Expressions	Contents of a
a <- Bag new.	Bag ()
a add: $a.	Bag ($a)
a addAll: #(2 3 $b)	Bag ($a 2 3 $b)

11.2.2 Removing elements from a collection

Just as you can add elements to a collection, so you can remove them. There are three messages that allow elements to be removed from a collection. They are:

```
remove: anObject
```

removes one of the occurrences of the element anObject from the receiver and returns the object anObject. If no occurrence is found, an error is produced.

```
remove: anObject ifAbsent: aBlock
```

removes one of the occurrences of the element anObject from the receiver and returns the object anObject. If no occurrence is found, the block aBlock is evaluated.

```
removeAll: aCollection
```

removes an occurrence of each element of the collection aCollection. If, for an element of the collection aCollection, there is no occurrence in the receiver, an error is generated. If not, the collection aCollection is returned.

For example:

Expressions	Contents of a
a	Bag ($a 2 3 $b)
a remove: $a	Bag (2 3 $b)
a remove: $a ifAbsent: [a add: $b]	Bag ($b 2 3 $b)
a removeAll: #(2 $b)	Bag (3 $b)

11.3 Tests on a collection

Three messages allow collections to be tested. These are:

includes: anObject

returns true if the object anObject is equal to one of the elements of the receiver,

isEmpty

returns true if the receiver is an empty collection,

occurrencesOf: anObject

returns the number of elements of the receiver equal to the argument anObject.

For example:

Expressions	Result
a	Bag ($a 2 3 $b $b)
a includes: 2	true
a isEmpty	false
a occurrencesOf: $b	2

11.4 Enumeration of a collection

A collection generally includes several elements. There are several messages described in the class Collection that allow each element of a collection to be accessed and allow an operation to be performed on this element. These messages are:

do: aBlockWithOneArgument

evaluates the block aBlock for each element of the receiver, the element being the argument of the block,

select: aBlockWithOneArgument

evaluates the block aBlock for each element of the receiver and returns a collection of the same class as the receiver, containing those elements for which evaluation of the block has returned true,

reject:aBlockWithOneArgument

evaluates the block aBlock for each element of the receiver and returns a collection of the same class as the receiver, containing those elements for which evaluation of the block has returned false,

detect: aBlockWithOneArgument

evaluates the block aBlock for each element of the receiver and returns the first element of the receiver that evaluation of the block has returned as true. If no element is found, an error is produced.

detect: aBlock ifNone: anOtherBlock

evaluates the block aBlock for each element of the receiver and returns the first element of the receiver that evaluation of the block aBlock has returned as true. If no element is found, the block anOtherBlock is evaluated.

inject: anInitialValue into: aBlockWithTwoArguments evaluates the block aBlockWithTwoArguments for each element of the receiver. The second argument of the block is the element, the first argument of the block is the result of its preceding evaluation. At first evaluation, the first argument is anInitialValue.

For example:

```
sum <- 0
a <- # (1 2 3 4 5).
a do: [:anElement | sum <- sum +anElement].
```

places in the variable sum the sum of all the elements in the array a;

```
a <- #(1 2 3 4 5).
a select: [:anElement | anElement even].
```

returns Array (2 4);

```
a reject: [:anElement | anElement even]
```

returns Array (1 3 5);

```
a detect: [:anElement | anElement > 2.5]
```

returns 3;

```
a detect: [:anElement | anElement > 5] ifNone:
[´Element not found´]
```

returns ´Element not found´;

```
a inject: 10 into: [:initialValue: anElement |
initialValue + anElement]
```

returns 25. On the first evaluation of the block, initialValue is 10, anElement is 1; the result of the evaluation of the block, which is also the result of the evaluation of its last expression, is 11. On the second evaluation, initialValue is 11, anElement is 2, and so on.

The message inject: into: allows a local variable to be initialised in the message that will be passed between the successive evaluations of a block. To write the previous example without inject: into:, one could write, for example:

```
counter <- 10.
a do: [:anElement | counter <- counter + anElement]
```

11.5 Conversion of a collection

Five messages understood by the instances of class Collection transform instances of a subclass of class Collection to instances of another subclass of class Collection.

asBag

returns an instance of class Bag whose elements are those of the receiver,

asSet

returns an instance of class Set whose elements are those of the receiver (an instance of class Set does not possess two equal elements),

asOrderedCollection

returns an instance of class OrderedCollection whose elements are those of the receiver,

asSortedCollection

returns an instance of class SortedCollection whose elements are those of the receiver (the elements are stored in ascending order),

asSortedCollection: aBlock

returns an instance of class SortedCollection whose elements are those of the receiver (the criterion for the order of the elements being given by the evaluation of the block aBlock).

11.6 The subclasses of the class Collection

Class Collection has no instances; that is, it is an abstract class. It possesses several subclasses that allow the different structures of useful data to be represented. These subclasses are arranged in the following hierarchy.

11.6.1 The class Bag

Class **Bag** is a subclass of class Collection. The elements of an instance of Bag represent an arbitrary set whose elements are not accessible by keys. Because of this, the messages at: and at: put: are not understood by the instances of class Bag. In fact, an instance of the class Bag just represents a group of elements. There is only one additional message that is understandable by the instances of this class. It is as follows:

add: anObject withOccurrences: anInteger

adds to the receiver a number of occurrences of anObject equal to anInteger.

11.6.2 The class Set

Class **Set** is also a subclass of class Collection. In contrast to the instances of class Bag, those of class Set cannot contain two equal elements.

11.6.3 The class Dictionary

Class **Dictionary** is a subclass of class Set. The elements of an instance of class Dictionary are instances of class **Association.** An instance of class Association is represented by a pair of objects; the first element is called the key, the second element is called the value assigned to the key.

The messages at: and at: put: are still defined for the instances of this class; however, their methods have been redefined. In contrast to the foregoing, the argument of the keyword at: is no longer a numeric index but any object that represents a key.

The additional messages that can be understood by the instances of class Dictionary are as follows.

Access

at: aKey ifAbsent: aBlock
returns the value assigned to the key aKey. If the key is not found, the block aBlock is evaluated.

associationAt: aKey
returns the association whose key is aKey. If the key is not found, an error is produced.

associationAt: aKey ifAbsent: aBlock
returns the association whose key is aKey. If the key is not found, the block aBlock is evaluated.

keyAtValue: aValue
returns a key that refers to the value aValue, and if several keys reference the same value, returns the first one found. If no key is found, nil is returned.

keys
returns an instance of class Set containing all the keys of the receiver,

values
returns an instance of class Bag containing all the values of the receiver.

For example:

```
dictionary <- Dictionary new.
dictionary at: #un put: #one.
dictionary at: #deux put: #two.
dictionary at: #trois put: #three.
dictionary add: (Association key: #quatre value: #four).
```

allows a dictionary to be created. We have used two messages to add an element to the dictionary, the message at:put: and the message add: whose argument must be an Association.

Expressions	Result
dictionary at: #deux	two
dictionary associationAt: #deux	deux -> two
dictionary keyAtValue: #three	trois
dictionary keys	Set (un deux trois quatre)
dictionary values	Bag (one two three four)

Test

Being a subclass of Collection, the class Dictionary inherits the test messages of this class, in particular, includes:, although the method assigned to this message is redefined in the class Dictionary. This message allows for testing the inclusion or otherwise of a value of an association. The same goes for the message occurrencesOf:. The new messages are:

includesAssociation: anAssociation

returns true if the receiver contains the element an-Association,

includesKey: aKey

returns true if one of the elements of the receiver has aKey as its key.

For example:

Expressions	Result
dictionary includes: #two	true
dictionary includes: #deux	false
dictionary includesAssociation: (Association key: #deux value: #two)	true
dictionary includesKey: #two	false
dictionary includesKey: #deux	true

Deletion

Three new messages allow an element to be deleted from a dictionary. The message remove: defined in Collection is unusable for a dictionary. So as to be able not to use it, the method assigned to it is redefined in Dictionary; it is written as follows:

```
remove: anObject
    self shouldNotImplement
```

The new messages are:

removeAssociation: anAssociation

which deletes the element anAssociation from the receiver. If the element anAssociation is not found in the receiver, an error is produced.

removeKey: aKey

deletes the element of the receiver whose key is aKey. If the key is not found, an error is produced; if successful, the value assigned to the key is returned.

For example:

Expressions	Dictionary
dictionary removeAssociation: (Association key: #un value: #one)	Dictionary (deux -> two quatre -> four trois -> three)
dictionary removeKey: #deux	Dictionary (trois -> three quatre -> four)

Enumeration

Being a subclass of class Collection, class Dictionary inherits the messages of this class, in particular, do:, even though the method assigned to this class is redefined in class Dictionary. This message allows evaluation of a block for all values of the receiver and not the associations. The new messages are:

associationsDo: aBlock

evaluates the block aBlock for all associations of the receiver.

keysDo: aBlock

evaluates the block aBlock for all keys of the receiver. For example:

```
dictionary associationDo: [:anAssociation | anAssociation
    value:anAssociation value asString]
```

means that the values of the dictionary are no longer symbols but strings. The result for the dictionary is: Dictionary (one -> 'un' two -> 'deux' three -> 'trois' four -> 'quatre').

11.6.4 The class IdentityDictionary

The class **IdentityDictionary** is a subclass of class Dictionary. The difference between these two classes is that for class Dictionary the search for a key uses the equality (=), whereas for class IdentityDictionary the search for a key uses the equivalence (==).

11.6.5 The class SequenceableCollection

The class **SequenceableCollection** is a subclass of class Collection. The instances of SequenceableCollection have ordered elements that are accessible via a numeric index. An instance of SequenceableCollection possesses a first element and a last element.

Access

Being a subclass of Collection, the class SequenceableCollection inherits from the access messages of this class, especially at: and at:put:. The other messages are:

atAll: aCollection put: anObject

assigns the argument anObject to each index of the receiver that belongs to the argument aCollection,

atAllPut: anObject

assigns the argument anObject to all indices of the receiver,

first

returns the first element of the receiver, producing an error if the receiver is empty,

last

returns the last element of the receiver, producing an error if the receiver is empty,

indexOf: anElement

returns the first index of the element anElement, returning 0 if the receiver does not contain the element anElement,

indexOf: anElement ifAbsent: aBlock

returns the first index of the element anElement, and

evaluates the block aBlock if the receiver does not contain the element anElement,

```
indexOfSubCollection: aSubCollection
startingAt:anIndex
```

if the argument aSubCollection is a subcollection of the receiver, returns the index in the receiver of the first element of the subcollection, the search starting at the index anIndex, and returns 0 if the subcollection is not found,

```
indexOfSubCollection: aSubCollection startingAt:
anIndex ifAbsent: aBlock
```

if the argument aSubCollection is a subcollection of the receiver, returns the index in the receiver of the first element of the subcollection, the search starting at the index anIndex, and evaluates the block aBlock if the subcollection is not found,

```
replaceFrom: beginning to: end with: aCollection
```

replaces the elements whose index lies between beginning and end with the elements of the argument aCollection; if the size of the argument aCollection does not match with beginning and end, an error is produced,

```
replaceFrom: beginning to: end with: aCollection
startingAt: indexBeginning
```

replaces the elements whose index lies between beginning and end with the elements of the argument aCollection, starting at the index indexBeginning; the size of aCollection must not be less than (end - beginning + indexBeginning). For example:

Expressions	Result
`#($a $b $c $d) size`	`4`
`#($a $b $c $d) at: 3`	`$c`
`#($a $b $c $d) at: 3 put: $e`	`#($a $b $e $d)`
`#($a $b $c $d) atAll: #(1 2) put: $f`	`#($f $f $c $d)`
`#($a $b $c $d) atAllPut: $a`	`#($a $a $a $a)`
`#($a $b $c $d) first`	`$a`
`#($a $b $c $d) last`	`$d`
`#($a $b $c $d) indexOf: $b`	`2`
`#($a $b $c $d) indexOf: $t ifAbsent: [5]`	`5`
`'abcdefgh' indexOfSubCollection: 'cd' startingAt: 1`	`3`
`'abcabc' indexOfSubCollection: 'bc' startingAt: 3`	`5`
`'abcdefgh' replaceFrom: 5 to: 7 with 'FGH'`	`'abcdFGHL'`
`'abcdefgh' replaceFrom: 5 to: 8 with: 'ABCDEFGHIJK' startingAt: 3`	`'abcdCDEF'`

Copy

There are messages that allow all or part of an instance of SequenceableCollection to be copied. They are:

```
, aCollection
```

returns a copy of the receiver concatenated with the argument aCollection which must be an instance of SequenceableCollection,

copyFrom: beginning to: end

returns a copy of the subcollection of the receiver starting at the index beginning and finishing at the index end,

copyReplaceAll: aSubCollection with: anOtherSub-Collection

returns a copy of the receiver in which all the occurrences of the subcollection aSubCollection are replaced with the collection anOtherSubCollection,

copyReplaceFrom: beginning to: end with: aCollection

if end is greater than beginning, replaces the subcollection starting at the index beginning and finishing at the index end with the collection aCollection; if end is less than beginning, inserts the collection aCollection in front of the index beginning and returns the copy of the new collection,

copyWith: anObject

returns a copy of the receiver concatenated with anObject,

copyWithout: anObject

returns a copy of the receiver from which all occurrences of the object anObject have been removed.

For example:

Expressions	**Result**
'abc', 'def'	'abcdef'
'abcdef' copyFrom: 2 to: 5	'bcde'
'abcdef' copyReplaceAll: 'cd' with: 'CDE'	'abCDef'
'abcdef' copyReplaceFrom: 2 to: 3 with: 'AAA'	'aAAAdef'
'abcdef' copyReplaceFrom: 3 to: 2 with: 'AAA'	'abAAAcdef'
'abcdef' copyWith: $g	'abcdefg'
'abcdef' copyWithout: $f	'abcde'

Enumeration

Because the elements of an instance of SequenceableCollection are ordered, enumeration of these elements is also made in an ordered manner. The new messages accessible by the instances of SequenceableCollection are:

findFirst: aBlock

which evaluates the block aBlock for each element of the receiver, and returns the index of the first element for which evaluation of the block returns true,

findLast: aBlock

evaluates the block aBlock for each element of the receiver, and returns the index of te last element for

which evaluation of the block returns true,

reverseDo: aBlock

evaluates the block aBlock for each element of the receiver starting with the last and going back to the first,

with: aCollection do: aBlock

evaluates the two argument block aBlock, taking as first argument an element of the receiver and as second argument an element of the SequenceableCollection aCollection. The SequenceableCollection aCollection must be the same size as the receiver.

For example:

Expressions	Result
'abcDEFG' findFirst: [:i | i isUpperCase]	4
'abcDEFG' findLast: [:i | i isUpperCase]	7
mult <- Bag new. #(1 2 3 4) with: #(5 6 7 8) do: [:i :j | mult add: i*j]. mult	Bag(5 12 21 32)

SequenceableCollection possesses several subclasses; these are OrderedCollection, LinkedList, Interval and ArrayedCollection.

11.6.5.1 The class OrderedCollection

The class **OrderedCollection** is a subclass of SequenceableCollection. The order of the elements of an instance of OrderedCollection is fixed by the order in which they are added. The new messages accessible by the instances of OrderedCollection are:

addLast: anObject

which adds the object anObject to the end of the receiver,

addFirst: anObject

adds the object anObject to the beginning of the receiver,

add: anObject after: anOtherObject

adds the object anObject to the receiver after the first occurrence of anOtherObject. If the object anOtherObject is not an element of the receiver, an error message is produced,

add: anObject before: anOtherObject

adds the object anObject to the receiver in front of the first occurrence of anOtherObject. If the object anOtherObject is not an element of the receiver, an error message is produced,

addAllFirst: aCollection

adds to the beginning of the receiver all the elements of the OrderedCollection aCollection,

addAllLast: aCollection

adds to the end of the receiver all the elements of the OrderedCollection aCollection,

removeFirst

removes the first element of the receiver and returns this element,

removeLast

removes the last element of the receiver and returns this element. An error is produced if the receiver is empty,

after: anObject

returns the element that appears after the first occurrence of the object anObject. If the receiver contains no object anObject or there is no element after anObject, an error is produced,

before: anObject

returns the element that appears in front of the first occurrence of the object anObject. If the receiver contains no object aObject or there is no element in front of anObject, an error is produced.

For example:

Expressions	List
list <- OrderedCollection new.	OrderedCollection ()
list add: 1	OrderedCollection (1)
list addFirst: 2	OrderedCollection (2 1)
list addLast: 3	OrderedCollection (2 1 3)
list add: 4 after: 1	OrderedCollection (2 1 4 3)
list add: 5 before: 1	OrderedCollection (2 5 1 4 3)
list removeFirst	OrderedCollection (5 1 4 3)
list removeLast	OrderedCollection (5 1 4)
list before: 1	returns 5
list after: 1	returns 4
list addAllFirst: list	OrderedCollection (5 1 4 5 1 4)
list addAllLast: list	OrderedCollection (5 1 4 5 1 4 5 1 4 5 1 4)

11.6.5.2 The class SortedCollection

The class **SortedCollection** is a subclass of OrderedCollection whose elements are ordered. To order these elements the class uses a function represented by a two argument block. To create an instance of SortedCollection, it is sufficient to send the message sortBlock: to the class. The argument of sortedBlock: is a two argu-

ment block, for example:

```
SortedCollection sortBlock: [:arg 1 :arg 2 | arg 1 > arg2]
```

This instance of SortedCollection will contain a collection of objects stored from largest to smallest.

The message new allows creation of a SortedCollection with the following as default sort block

```
[:arg1 :arg2 | arg1 <= arg2]
```

There are two other messages that allow any collection to be converted into a SortedCollection. These are:

```
asSortedCollection
```

which transforms the receiver into a SortedCollection with the block as default sort function,

```
asSortedCollection: aBlock
```

which transforms the receiver into a SortedCollection with the block given as argument being the sort function.

There are two messages sortBlock and sortBlock: that are understood by instances of SortedCollection; the first returns the sort block of the receiver, the second allows the sort block of the receiver to be altered.

For example:

Expressions	**Results**
collection <- SortedCollection new	SortedCollection ()
collection add: 'abcdefg'	'abcdefg'
collection add: 'abdef'	'abdef'
collection add: 'aa'	'aa'
collection	SortedCollection ('aa' 'abcdefg' 'abdef')
collection sortBlock: [:a :b | a size > b size]	SortedCollection ('abcdefg' 'abdef' 'aa')
collection add: 'abc'	'abc'
collection	SortedCollection ('abcdefg' 'abdef' 'abc' 'aa')

The only constraint imposed on the sort block is that it returns true or false. In particular, it is important to ensure that the objects of the SortedCollection can be compared by the sort block.

11.6.5.3 The class LinkedList

The class **LinkedList** is a subclass of SequenceableCollection whose instances contain a list of instances of the class Link ordered explicity at the time of addition or removal. The class Link is a subclass of Object whose instances make reference to another instance of Link.

To create an instance of the class Link, the message

nextLink: aLink

needs to be sent to the class. The argument of the message nextLink: must be an instance of Link.

There are two messages that are understood by instances of Link:

nextLink

which returns the instance of Link referenced by the receiver; and

nextLink: aLink

which allows the reference of the receiver to be altered.

The class Link has no direct instances because they would not be usable (that is, the class Link is an abstract class). To use the class LinkedList, subclasses of Link have to be defined whose instances will have other usable variables.

The instances of LinkedList understand the same messages as the instances of Link; they also understand all the messages of the class SequenceableCollection (addFirst, removeFirst, etc). The following example, which allows a circular list of n objects to be created, shows how the class LinkedList may be used:

```
name of class                                   Bond
superclass                                      Link
name of instance variables                      object
instance methods
object
     ↑object
object: anObject
     object <- anObject
name of class                                   CircularList
superclass                                      LinkedList
class method
new: ASize
   |aList|
   aList <- super new.
   (1 to: aSize) inject: nil into:
        [:i :j| aList addFirst: (Bond nextLink: i)].
   aList last nextLink: aList first
```

11.6.5.4 The class Interval

The class **Interval** is a subclass of SequenceableCollection whose instances represent arithmetic sequences. An arithmetic sequence is characterised by its first element and the increment which steps from one element to the next up to a finite limit. There are two messages that create Interval instances.

from: beginning to: end

returns an instance of the class Interval starting at the beginning number and finishing at the end number, with increments of 1.

from: beginning to: end by: anIncrement

returns an instance of the class Interval starting at the beginning number and finishing at the end number and incrementing by anIncrement.

The class Number also supports two messages that create Interval instances:

to: aNumber

which returns an instance of the class Interval starting at the receiver and finishing at the number aNumber, with an increment of 1.

to: aNumber by: anIncrement

which returns an instance of the class Interval starting at the receiver and finishing at the number aNumber, with an increment of anIncrement.

For example:

1 to: 5 by: 2

returns an instance of Interval whose elements are 1, 3 and 5.

1 to: 5 by: 2.1

returns an instance of Interval whose elements are 1 and 3.1.

11.6.5.5 The class ArrayedCollection

The class **ArrayedCollection** is a subclass of SequenceableCollection whose instances are arrays of objects. Each object is accessible by a numeric key representing its position in the array. The class ArrayedCollection, which is an abstract class, has five subclasses, as described below.

- The class **Array**

An instance of the class Array can store any type of object. The messages understood by instances of Array are the messages supported by the class Collection.

- The class **String**

In contrast to class Array the instances of class String only contain characters. The instances of String are therefore strings of characters. The additional messages supported by the class String are:

Creation

Two messages allow instances of String to be created:

fromString: aString

returns a copy of the string aString;

readFrom: aStreamOfCharacters

returns an instance of String created from the characters contained in the stream of characters aStreamOfCharacters (an instance of **Stream**, see chapter 12).

Comparison

Six messages allow strings of characters to be compared with one another. The distinction between upper and lower case is ignored. These messages are:

<aString

returns true if the receiver precedes the argument aString alphabetically,

<=aString

returns true if the receiver precedes the argument aString alphabetically or if two strings are equal,

>aString

returns true if the receiver follows the argument aString alphabetically,

>=aString

returns true if the receiver follows the argument aString alphabetically or if the two strings are equal,

match: aString

returns true if the argument aString appears in the receiver. The characters # and * have a special meaning in the receiver:

- # can represent any character
- * can represent any string of characters (including the empty string).

sameAs: aString

returns true if the receiver and the argument aString are equal.

Conversion

Three messages allow character strings to be converted:

asLowerCase

returns a string of characters equivalent to the receiver containing only lower case characters,

asUpperCase

returns a string of characters equivalent to the receiver containing only upper case characters,

asSymbol

returns the symbol made up of characters cf the receiver.

For example:

Expressions	Result
'String' = 'String'	true
'String' = 'string'	false
'String' sameAs: 'String'	true
'String' sameAs: 'string'	true
'#tring' match: 'string'	true
'*ng' match: 'string'	true
'string' asUpperCase	'STRING'
'string' asSymbol	string

The class Symbol
The class **Symbol** is a subclass of the class String whose instances are used to name objects in the system. The objects that have to be named are for example classes, messages, variables, etc. In the interests of saving space, symbols are only represented once in the system. Two messages allow new symbols to be created:

```
intern: aString
```

returns the unique symbol made up of the characters of the string aString;

```
internCharacter: aCharacter
```

returns the unique symbol made up of the character aCharacter.

Remember that the syntax for writing a symbol as a literal form is #nameOfSymbol.

The class Text
The class **Text** is a subclass of ArrayedCollection. The class Text allows strings of characters to be stored as String, with the additional possibility of assigning to each character a further characteristic that can for example represent the style of the character when displayed.

The class ByteArray
The class **ByteArray** is a subclass of the class Arrayed-Collection whose instances contain integers coded into a byte (between 0 and 255). This class allows arrays of such integers to be stored efficiently.

The class RunArray
The class **RunArray** is a subclass of the class Arrayed-Collection. This class allows the convenient storage of arrays containing sequences of similar objects. The instances of RunArray have two named instance variables that are runs and values. The variable runs contains a list of integers and the variable values contains a list of objects. Each object of values is assigned to an integer of runs that represents the number of consecutive occurrences of this object.

12 The Class Stream

Collections allow objects to be grouped. Each object of a collection can be accessed individually; also, an operation can be carried out on all the objects of a collection, provided that this is done in a loop. However, it is not easy to mix the two types of access and operate on the enumeration of objects.

The class **Stream** is a subclass of the class Object that allows streams of objects to be controlled. The instances of Stream have a variable representing the current object in the objects stream. The instances of Stream do not directly store objects, but they do control access to these objects.

Several types of access are possible; they will be defined in the subclasses of Stream, which accounts for why Stream is an abstract class.

To summarise, the instances of Stream allow access to the collections to be controlled.

The messages of class Stream can be classified into several categories, as follows.

Reading

Four access messages are understood by the instances of Stream:

next

returns the next object accessed by the receiver,

next: anInteger

returns the next anInteger objects accessed by the receiver, in the collection form of the same class as the collection accessible by the receiver,

nextMatchFor: anObject

accesses the next object and returns true if this object is equal to the argument anObject,

contents

returns all the objects accessed by the receiver.

Writing

Three write messages are understood by the instances of Stream:

nextPut: anObject

stores the argument anObject at the next position accessed by the receiver and returns anObject,

nextPutAll: aCollection

stores all the objects of the argument aCollection at the next positions accessed by the receiver and returns aCollection,

next: anInteger put: anObject

stores the argument anObject at the next anInteger positions accessed by the receiver and returns anObject.

Test

The message atEnd establishes whether the receiver can access any more objects.

Enumeration

The message do: aBlock evaluates the argument aBlock for all the objects accessible by the receiver.

12.1 The class PositionableStream

The class **PositionableStream** is a subclass of Stream. It is again an abstract class that has three subclasses:

- the class **ReadStream**
- the class **WriteStream**
- the class **ReadWriteStream**

In addition to the protocol defined within class Stream, instances of the class PositionableStream allow positioning within the object stream; these streams of objects therefore must themselves be ordered, that is, they must be instances of the class SequenceableCollection.

To create an instance of PositionableStream, two messages are possible:

on: aCollection

returns an instance of PositionableStream allowing control of the stream made up of the objects of the argument aCollection.

on: aCollection from: indexBeginning to: indexEnd

returns an instance of PositionableStream allowing control of the stream made up of the the objects of the argument aCollection that lie between the arguments indexBeginning and indexEnd.

The additional messages understood by the instances of PositionableStream fall into several categories, as follows.

Test

The message isEmpty returns true if the collection controlled by the receiver is empty.

Access

Four messages allow access to the objects of the collection controlled by the receiver:

peek

returns the next object of the collection, but does not

alter the current position in the stream; it returns nil if the receiver is at the end,

peekFor: anObject

returns true and advances one position in the stream if the next object is equal to the argument anObject; if the object is not found, it returns the remainder of the collection,

reverseContents

returns a collection in inverse order of that controlled by the receiver.

Positioning

The messages that alter or give the position in the objects stream are:

position

returns the integer representing the current position in the stream,

position: anInteger

places the receiver at the position given by the argument anInteger; if the position given as argument is not within the limits of the collection controlled by the receiver, an error is produced,

reset

places the receiver at the beginning of the collection that it controls,

setToEnd

places the receiver at the end of the collection that it controls,

skip: anInteger

shifts the current position of the argument anInteger, ensuring that this position never goes beyond the permitted bounds,

skipTo: anObject

shifts the current position to place it immediately after the next occurrence of the argument anObject in the collection controlled by the receiver; it returns true if such an object exists and false otherwise.

For example:

Expression	**Result**
pS <- ReadStream on: # (peter paul john)	
pS reset	
pS next	peter
pS next	paul
pS next	john
pS next	nil
pS position	3
pS position: 2	2
pS peek	paul
pS next	paul
pS contents	(peter paul john)
pS reverseContents	(john paul peter)

12.1.1 The class ReadStream

The class **ReadStream** is a subclass of the class PositionableStream. The instances of this class can read the stream of objects of a collection but cannot alter this stream. None of the messages nextPut:, next: and nextPutAll: is recognised.

12.1.2 The class WriteStream

The class **WriteStream** is a subclass of the class PositionableStream. The instances can alter the objects stream of a collection but cannot read the objects of this stream. In the system this class is used on numerous occasions to implement the methods printString and printOn that allow objects to be displayed in a textual form.

Additional messages are supported by the class WriteStream. They fall into the following categories.

Creation

The messages that allow instances to be created are:

with: aCollection

creates an instance of WriteStream allowing the collection given as argument to be controlled, and places it at the end of the stream (the next objects will be added at the end),

with: aCollection from: indexBeginning to: indexEnd

creates an instance of WriteStream allowing control of the collection of objects lying between the arguments indexBeginning and indexEnd in the collection given as argument, and places it at the end of the stream (the next objects will be added at the end).

Writing characters

The following methods allow a character to be added to an objects stream:

cr

adds a carriage-return after the current position of the receiver,

tab

adds a tabulation after the current position of the receiver,

space

adds a space after the current position of the receiver,

crTab

adds a carriage-return and a tabulation after the current position of the receiver,

crTab: n

adds a carriage-return and tabulations after the current position of the receiver.

An interesting case is where the collection is a string of characters. In such a case the control of the character stream amounts to carrying out editing funct-

ions on a string of characters.

12.1.3 The class ReadWriteStream

The class **ReadWriteStream** is a subclass of WriteStream which allows control of a stream of objects when reading or writing. It supports all the messages of ReadStream and WriteStream.

12.2 The class ExternalStream

The class **ExternalStream** is a subclass of the class ReadWriteStream. In the Smalltalk system, communication with the external files is also carried out by means of data streams. In contrast to the subclasses of Stream that we have already met, the input/output streams do not contain any objects but only bytes. The messages supported by the class ExternalStream fall into two categories.

Access

The access messages to byte streams are:

```
nextNumber: n
```

returns the next n bytes of the collection accessed by the receiver and interprets them as a positive instance of SmallInteger or of LargePositiveInteger, depending on the value read,

```
nextNumber: n put: anInteger
```

stores in n bytes the positive integer anInteger (an instance of SmallInteger or of LargePositiveInteger) in the collection accessed by the receiver,

```
nextString
```

returns in the form of a character string (instance of String) the next bytes of the collection accessed by the receiver,

```
nextStringPut: aString
```

stores the string of characters aString in the form of bytes in the collection accessed by the receiver,

```
nextWord
```

returns the next two bytes interpreted as an integer,

```
nextWordPut: anInteger
```

stores in two bytes the integer anInteger in the collection accessed by the receiver.

Positioning

The messages for positioning on the byte streams are:

```
padTo: aSize
```

places itself at the next multiple position of aSize, and returns the number of bytes skipped,

```
padTo: aSize put: aCharacter
```

places itself at the next multiple position of aSize writing the character aCharacter the required number of

times, and returns the number of bytes skipped,

padToNextWord

places itself on the next word (first even byte position), and returns the character skipped if there is one,

padToNextWordPut: aCharacter

places itself on the next word writing the character aCharacter as many times as necessary,

skipWords: n

skips n words,

wordPosition

returns the current position in words,

wordPosition: aPosition

places itself at the position aPosition in words.

12.2.1 The class FileStream

The class **FileStream** is a subclass of the abstract class ExternalStream. All accesses to files are carried out by means of instances of FileStream. In contrast to its superclasses, the class FileStream does not work on collections present in central memory, but on streams of bytes present in auxiliary memories. In addition to the messages supported by the class ExternalStream, the class FileStream supports messages relating to the status of a file (open, closed, etc). Three class methods allow instances of FileStream to be created:

oldFileNamed: aString

creates an instance of FileStream accessing the file named aString and verifies that it exists,

newFileNamed: aString

creates an instance of FileStream accessing the file named aString, creating it if it does not exist and deleting it if it does,

fileNamed: aString

creates an instance of FileStream accessing the file named aString and creating it if it does not exist.

The two important new messages supported by FileStream are:

open

which allows the file accessed by the receiver to be opened; read and write operations cannot take place until this message has been sent.

close

which allows the file accessed by the receiver to be closed.

12.2.2 The class FileDirectory

The class **FileDirectory** is a subclass of the class FileStream. The class FileStream allows files to be named, while the class FileDirectory allows them to be located in the logical organisation of the secondary memory. The

methods of the class FileDirectory depend on the system support, that is, the file organisation of the operating system on which the Smalltalk interpreter has been written. The instances of the class FileDirectory represent directories or volumes.

13 Processes

A process represents a sequence of instructions that has to execute a particular task. A conventional machine, for example, contains processes for reading the keyboard, writing to disk, printing out text via a printer, etc. When a machine is what is known as a uniprocessor, it can only execute one process at a time; it is therefore necessary to be able to select a process to execute and synchronise the various processes required. Smalltalk-80 contains several classes that allow such concepts to be defined.

13.1 The class Process

A process is a sequence of expressions that are interpreted by Smalltalk. There are already in the system processes that inspect the input interfaces (keyboard and mouse) and a process that is activated when the amount of available memory is running low, notifying the user of a potential problem.

Processes are instances of the class **Process** which is a subclass of Link. To create a new process, four messages are supported by the class **BlockContext:**

fork

creates a new process from the receiver which is a block containing the expressions to be interpreted, and places it on the list of processes to be executed; the priority of the process created is equal to the priority of the process that creates it,

forkAt: aPriority

creates a new process from the receiver which is a block containing the expressions to be interpreted, and places it on the list of processes to be executed with the priority aPriority,

newProcess

creates a new process from the receiver which is a block containing the expressions to be interpreted, but does not place it on the list of processes to be executed,

newProcessWith: anArray

creates a new process from the receiver which is a block containing the expressions to be interpreted, without

putting it on the list of processes to be executed. The arguments of the block are initialised by the objects contained in the argument anArray.

The messages that allow action on processes are:

resume
which authorises execution of the receiver,

suspend
which halts executions of the receiver, but keeps it on the list of processes to be executed; to authorise continuation, the message resume must be sent,

terminate
halts execution of the receiver and removes it from the list of processes tc be executed,

priority
returns the priority of the receiver,

priority: aPriority
alters the priority of the receiver and gives the value of the argument aPriority.

13.2 The class ProcessorScheduler

The Smalltalk virtual machine has only one processor; since as a consequence only one process can be executed at a time, the activation of the various processes must be controlled.

It is the role of the class **ProcessorScheduler**, a subclass of the class Object, to control a list of processes. The class ProcessorScheduler has only one instance which is stored in a global variable called Processor.

The messages supported by the class ProcessorScheduler are:

activePriority
returns the priority of the process currently being executed,

activeProcess
returns the process (instance of Process) currently being executed,

terminateActive
terminates the process currently being executed,

yield
authorises execution of a process of the same priority as the active process,

highIOPriority
returns the priority of the fast input/output processes,

lowIOPriority
returns the priority of the slow input/output processes,

systemBackgroundPriority
returns the priority of the Smalltalk processes operating in the "background",

timingPriority

returns the priority of the processes that consult the real-time clock,

userBackgroundPriority

returns the priority of the user processes operating in the "background",

userInterruptPriority

returns the priority of the user processes considered to be of high priority,

userSchedulingPriority

returns the priority of the non-priority user processes.

The list below gives the rank order of priorities, from the highest to the lowest:

```
timingPriority                -> Priority 8
highIOPriority                -> Priority 7
lowIOPriority                 -> Priority 6
userInterruptPriority         -> Priority 5
userSchedulingPriority        -> Priority 4
userBackgroundPriority        -> Priority 3
systemBackgroundPriority      -> Priority 2
```

13.3 The class Delay

The class **Delay** is a subclass of the class Object. The instances of the class Delay allow the active process to be suspended for a determined period. There are three messages to create instances of Delay, as follow.

forMilliseconds: aNumberOfMilliseconds

returns a new instance of the class Delay that will suspend execution of the active process for the number of milliseconds equal to the argument,

forSeconds: aNumberOfSeconds

returns a new instance of the class Delay that will suspend execution of the active process for the number of seconds equal to the argument,

untilMilliseconds: aNumberOfMilliseconds

returns a new instance of the class Delay that will suspend execution of the active process until the real-time clock reaches the value of the argument.

To suspend execution of the active process, the message wait must be sent to an instance of Delay that is initialised for the required delay.

The message resumptionTime sent to an instance of Delay returns the value in milliseconds of the real-time clock at which the interrupted process can resume.

For example:

```
[[true] whileTrue:
   [Time now printString asDisplayText displayAt:
   Sensor cursorPoint. (Delay forSeconds: 1) wait]] fork
```

will display the current time every second at the cursor position.

13.4 Semaphores

We have seen that processes are independent of one ancther. In some circumstances it is useful to be be able to synchronise processes and communicate data between them.

The instances of the class **Semaphore**, which is a subclass of the class LinkedList, allow processes to be synchronised.

To allow a process to wait for an event from another process, it places itself in a wait state by sending a wait message to a semaphore. When the second process sends a signal to the first, the latter is able to resume.

The order in which these messages are sent has no effect on the execution of the processes, except that it may or may not cause the process placed in signal await state to wait.

The two messages that are involved are:

wait

which suspends the active process until the semaphore receives a signal,

signal

which allows a process to signal an event to the semaphore.

Several processes can wait for a semaphore; they resume execution in the same order that they were suspended as and when the semaphore receives the messages signalling an event.

One of the useful applications of semaphores is for achieving mutual exclusion between two processes. Two processes are in mutual exclusion when one of them cannot interrupt the other when it is in a critical section.

To define a mutual exclusion semaphore, a new semaphore must be created and the signal message sent to it. This is done by:

forMutualExclusion

A critical section is a part of a process that cannot be interrupted by another process. To define it, there must be a mutual exclusion semaphore to which the message wait is sent; that is, from that moment any process that sends the message wait to that same semaphore is blocked. When a critical section is terminated, it is then sufficient to signal to the semaphore to free access to the protected resource. The following message defines a critical section in a method:

critical: aBlock

which evaluates the argument aBlock, while preventing any other process from being activated.

The class SharedQueue

The instances of this class are queues that are used in Smalltalk to transmit data between processes while protecting their integrity. The instances of this class have two instance variables that refer to two semaphores. The first counts the elements in the queue and blocks the process that reads this queue when it is empty. The second is used for read and write operations to protect the queue (two processes cannot alter the queue at the same time, because of the danger of losing data).

The messages understood by a queue are:

isEmpty

returns true if the queue is empty, false if otherwise,

next

returns the next object in the queue and positions itself at the following object; if there is no object in the queue, the process is suspended until an object is added,

peek

returns the next object in the queue without positioning itself at the following object; if there is no object in the queue, the process is suspended until an object is added,

nextPut: anObject

places the argument anObject at the next position in the queue.

14 The Classes Point and Rectangle

Just as we have numbers to represent scalar sizes, we also need objects to represent sizes in two dimensions. These sizes are essential in order to represent screen positions. The class **Point**, a subclass of the class Object, was introduced for this purpose.

14.1 The class Point

The instances of the class Point have two instance variables called x and y, which represent respectively the abscissa and ordinate of a point.

14.1.1 Creating a point

A class method creates new points:

```
x: anAbscissa y: anOrdinate
```

returns a new instance of the class Point whose abscissa is anAbscissa and ordinate anOrdinate.

There is another instance method cf the class Number that allows new points to be created:

```
@ aNumber
```

returns a new instance of the class Point whose abscissa is the receiver and whose ordinate is aNumber.

14.1.2 Messages supported by the class Point

Like numbers (instances of the class Number), points are basic objects and therefore they understand a large number of messages. These can be divided into the following categories.

Access

The messages that allow access to the characteristics of a point are:

```
x
```

returns the abscissa of the receiver,

```
y
```

returns the ordinate of the receiver,

```
x: aNumber
```

replaces the abscissa of the receiver by the argument aNumber,

```
y: aNumber
```

replaces the ordinate of the receiver by the argument aNumber.

Arithmetic
The arithmetic operations are:

+ anOffset

returns a new point whose x-coordinate is the sum of the x-coordinates of the receiver and the argument anOffset; similarly for the y-coordinate. If the offset is not a point but a number, the two coordinates are incremented by the same value,

- anOffset

returns a new point obtained by subtracting from each coordinate of the receiver the corresponding coordinate of the argument anOffset; if the offset is not a point but a number, the two coordinates are decremented by the same value,

* aCoefficient

returns a new point obtained by multiplying each coordinate of the receiver by the argument aCoefficient; if the coefficient is not a point but a number, the two coordinates are multiplied by the same value,

/ aCoefficient

returns a new point obtained by dividing the receiver by the argument aCoefficient; if the coefficient is not a point but a number, the two coordinates are divided by the same value,

// aCoefficient

returns a new point obtained by carrying out the integer division for each coordinate between the receiver and the argument aCoefficient; if the coefficient is not a point but a number, the two coordinates are divided by the same value,

abs

returns a new point obtained by taking the absolute value of each coordinate of the receiver,

rounded

returns a new point whose coordinates are the integers closest to each coordinate of the receiver,

truncateTo: aCoefficient

returns a new point whose coordinates are the largest multiples of the argument aCoefficient smaller than the coordinates of the receiver,

dist: aPoint

returns the Euclidean distance between the receiver and the argument aPoint,

grid: aPoint

returns the multiple of the argument aPoint closest to the receiver,

truncateGrid: aPoint

returns the largest multiple of the argument aPoint smaller than the receiver,

transpose

returns a new point whose x-coordinate is the receiver's y-coordinate, and whose y-coordinate is the receiver's x-coordinate.

Comparison

The messages that allow two points to be compared are:

```
< aPoint
```

returns true if the two coordinates of the receiver are less than the two coordinates of the argument,

```
<= aPoint
```

returns true if the two coordinates of the receiver are less than or equal to the two coordinates of the argument,

```
> aPoint
```

returns true if the two coordinates of the receiver are greater than the two coordinates of the argument,

```
>= aPoint
```

returns true if the two coordinates of the receiver are greater than or equal to the two coordinates of the argument.

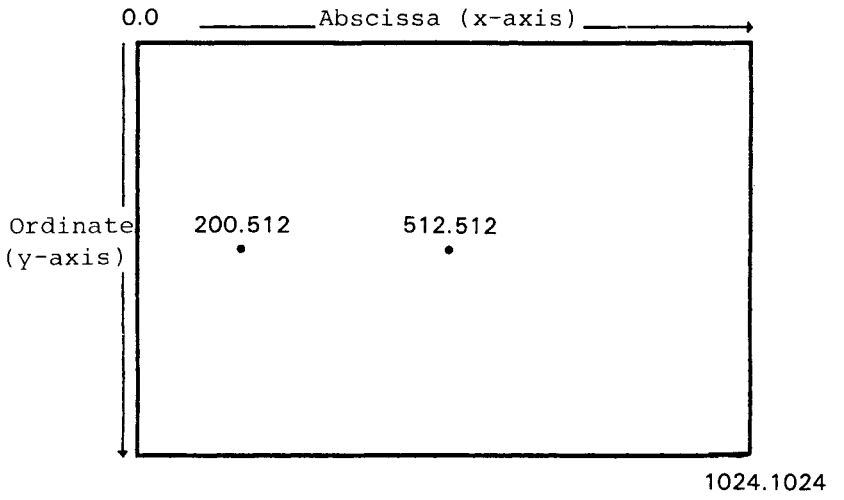
For example:

Expressions	Results
Point x: 10 y: 20	10@20
10@20	10@20
(10@20)x	10
(19@20)y	20
(10@20)x: 30	30@20
(10@20)y: 30	10@30
(10@20)+(5@8)	15@28
(10@20)+5	15@25
(10@20)-(5@8)	5@12
(19@20)-5	14@15
(10@20)*(2@3)	20@60
(10@20)*2	20@40
(10@20)/(2@4)	5@5
(10@20)/3	(10/3)@(20/3)
(10@20)//(3@7)	3@2
(10@20)//3	3@6
(-10@20) abs	10@20
(10.7@20.2) rounded	11@20
(10@20) truncateTo: 3	9@18
(4@4) dist: (1@0)	5
(10@20) grid: (3.3@7)	9.9@21
(10@20) truncatedGrid: (3.3@7)	9.9@14
(10@20) transpose	20@10

14.2 Coordinate system within Smalltalk environment

Each point on the screen can be defined by two coordinates, an abscissa and an ordinate. The diagram

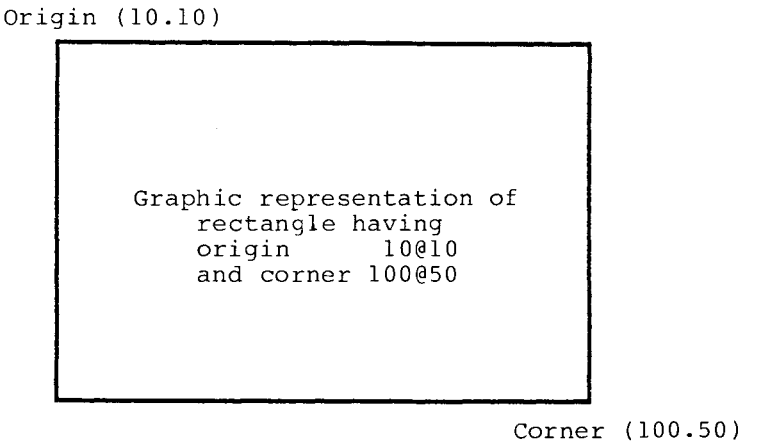
below shows the position and orientation of the two co-ordinate axes in the Smalltalk system.



14.3 The class Rectangle

A rectangle parallel to the axes can be defined by two points. In the Smalltalk system many objects use this concept of a rectangle (for example, windows, menus, etc). The class Rectangle, which is a subclass of the class Object, allows rectangles to be defined by two points, an origin and a corner.

The origin and the corner of a rectangle are defined as shown in the following diagram:



14.3.1 Creating a rectangle

Three messages create new rectangles:

origin: anOriginPoint corner: aCornerPoint

returns a new instance of the class Rectangle whose origin is anOriginPoint and corner aCornerPoint,

origin: anOrigin extent: aVector

returns a new instance of the class Rectangle whose origin is anOriginPoint and whose corner is obtained by adding to the argument anOrigin the argument aVector (which is an instance of Point),

left: limitLeft right: limitRight top: limitTop bottom: limitBottom

returns a new instance of the class Rectangle whose coordinates are delimited by the limits given as arguments.

There are two messages understood by instances of the class Point that also allow new rectangles to be created:

extent: aPoint

returns a new instance of the class Rectangle whose origin is given by the receiver and whose corner is obtained by adding the origin to the argument aPoint,

corner: aPoint

returns a new instance of the class Rectangle whose origin is given by the receiver and whose corner is the argument aPoint.

14.3.2 Instance methods

The instance methods of the class Rectangle may be divided into several categories:

Access

There are nine methods for accessing particular points of a rectangle:

topLeft

returns the point located at the top left of the receiver,

topCenter

returns the point located at the top centre of the receiver,

topRight

returns the point located at the top right of the receiver,

leftCenter

returns the point located at the left centre of the receiver,

center

returns the point located at the centre of the receiver,

rightCenter

returns the point located at the right centre of the receiver,

bottomLeft

returns the point located at the bottom left of the receiver,

bottomCenter

returns the point located at the bottom centre of the receiver,

bottomRight

returns the point located at the bottom right of the receiver,

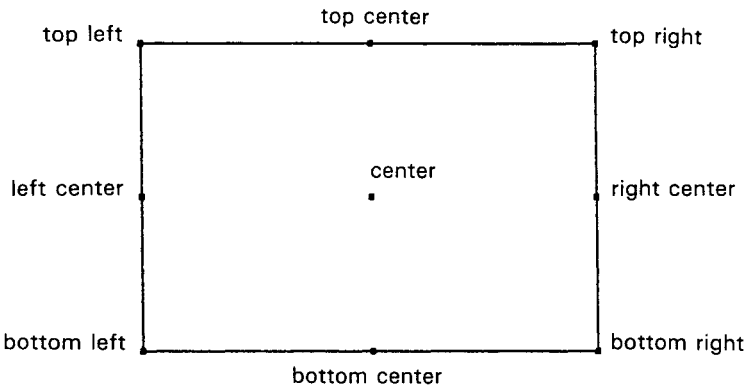
origin

returns the same point as the method topLeft,

corner

returns the same point as the method bottomRight.

These points are located as shown in the following diagram:



Other messages give access to the characteristics of the receiver:

origin: anOrigin corner: aCorner

replaces the origin and the corner of the receiver by the arguments anOrigin and aCorner,

origin: anOrigin extent: aSize

replaces the origin by the argument anOrigin and the corner by the point obtained by adding the origin with the argument aSize,

width

returns the width of the receiver,

height

returns the height of the receiver,

extent

returns aPoint whose abscissa is the width and whose ordinate is the height,

top

returns the ordinate of the origin of the receiver,
   bottom
returns the ordinate of the corner of the receiver,
   left
returns the abscissa of the origin,
   right
returns the abscissa of the corner,
   area
returns the area of the rectangle (the product of its height and its width).

Operations on rectangles

These operations return new objects, in contrast to those that we shall examine later, which transform the receiver. These operations are:

   amountToTranslateWithin: aRectangle
returns a point corresponding to the minimal translation that is required to be made to the receiver for it to be placed within the argument aRectangle,
   areasOutside: aRectangle
returns a collection of rectangles contained in the receiver and not intersecting the argument aRectangle,
   expandBy: anIncrement
returns a new rectangle obtained by increasing the receiver by the value given as argument; the argument can be a number, a point or a rectangle,
   insetBy: aDecrement
returns a new rectangle obtained by decreasing the receiver by the value given as argument; the argument can be a number, a point or a rectangle,
   insetOriginBy: anOriginIncrement
   cornerBy: aCornerDecrement
returns a new rectangle whose origin is the origin of the receiver increased by the argument anOriginIncrement and whose corner is the corner of the receiver decreased by the argument aCornerDecrement; the arguments can be points or numbers,
   intersect: aRectangle
returns the rectangle obtained by taking the intersection of the receiver and the argument,
   merge: aRectangle
returns the smallest rectangle containing the receiver and the argument,
   rounded
returns a new rectangle whose coordinates are the rounded coordinates of the receiver,
   translateBy: aTranslation
returns a new rectangle whose coordinates are the coordinates of the receiver translated from the argument aTranslation; aTranslation can be a number or a point,
   scaleBy: aCoefficient

returns a new rectangle whose coordinates are the coordinates of the receiver multiplied by the argument aCoefficient; aCoefficient can be a number or a point.

Test

Three methods allow the characteristics of a rectangle to be tested:

```
   contains: aRectangle
```

returns true if the argument aRectangle is within the receiver,

```
   containsPoint: aPoint
```

returns true if the argument aPoint is within the receiver,

```
   intersects: aRectangle
```

returns true if the intersection of the receiver and the argument aRectangle is non-empty.

Transformations

The methods that alter rectangles are:

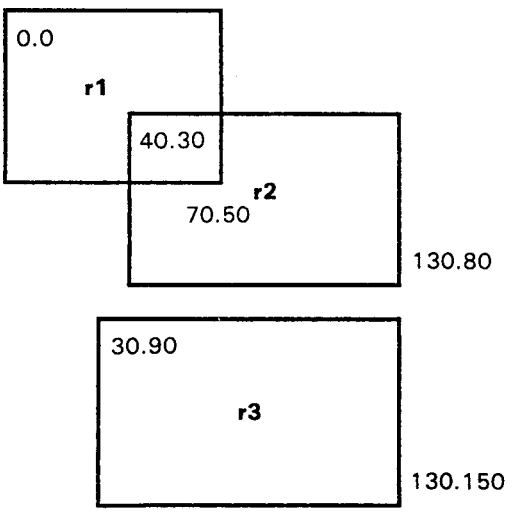
```
   moveBy: aTranslation
```

translates the origin and the corner of the receiver by the argument aTranslation; aTranslation can be a number or a point,

```
   moveTo: aPoint
```

translates the receiver in such a way that the origin of the receiver is located at the argument aPoint.

We shall take as an example three rectangles placed as in the following diagram:



Expression	Result
r1 <- Rectangle origin: (0@0) corner: (70@50)	
r2 <- Rectangle origin: (40@30) extent: (90@50)	
r3 <- 30@90 corner: (130@150)	
r1 centre	35@25
r2 width	90
r3 area	6000
r1 amountToTranslateWithin: r3	30@90
r1 areasOutside: r2	OrderedCollection ((0@0 corner: 70@30) (0@0 corner: 40@50))
r3 expandBy: 20	10@70 corner: 150@170
r3 expandBy: 20@10	10@80 corner: 150@160
r3 expandBy: (20@10 corner: 10@5)	10@80 corner: 140@155
r1 intersect: r2	40@30 corner: 70@50
r1 merge: r3	0@0 corner: 130@150
r1 contains: r2	false
r1 intersects: r2	true
r3 containsPoint: 50@100	true
r1 moveBy: 10@20	10@20 corner: 80@70
r1 moveTo: 100@100	100@100 corner: 170@150
r1 scaleBy: 2	200@200 corner: 340@300
r1 scaleBy: 2@4	200@400 corner: 340@600
r1 translateBy: -50	50@50 corner: 120@100
r1 translateBy: -50@50	50@150 corner: 120@200

15 The Graphics Classes

An object can be represented in different ways. For example, a collection of numbers may be seen as
- a sequence of numbers
- a histogram
- a piechart
- a curve passing through points

a rectangle may be seen as
- two points giving the extremities of the rectangle
- four right-angled segments
- a system of four mathematical equations.

We have seen that all the objects of the Smalltalk system respond to the message printString by returning a string of characters. This string allows the characteristics of the object to be explained, that is, its class and its instance variables. For certain objects, this representation is not helpful; they require graphic representation in addition.

In the Smalltalk system, the graphics interface consists of a bit-mapped screen. A bit-map screen is a screen in which each point is individually accessible, and where the characteristics of each point are stored in memory. A point on the screen is called a pixel, and in the case of Smalltalk each pixel can have one of two possible states, on or off. Thus, a single bit is sufficient to store this state and there are as many bits in memory as there are pixels on the screen.

15.1 The class Bitmap

The class **Bitmap** is a subclass of the abstract class ArrayedCollection. It therefore has indexed instance variables, each of which allows a machine word to be stored. The instances of the class Bitmap are able to store sequences of pixels. Since the size of one of these instance variables is the same as that of the machine word, pixel manipulation is carried out efficiently.

15.2 The class DisplayObject

The class **DisplayObject** is an abstract subclass of the class Object. An instance of the class DisplayObject represents a picture described by its width, its height,

and a point that provides the offset applied when it is displayed. The class DisplayObject contains five subclasses, each of which adopts a different way of storing a graphics object. The methods supported by the class DisplayObject are:

width
returns the width of the minimum rectangle containing the receiver,

height
returns the height of the minimum rectangle containing the receiver,

extent
returns a point whose coordinates are the width and height of the minimum rectangle containing the receiver,

offset
returns the point giving the offset of the minimum bounding rectangle when the receiver is displayed,

offset: anOffset
replaces the offset of the receiver by the argument anOffset,

scaleBy: aPoint
multiplies the offset of the receiver by the argument aPoint, coordinate by coordinate,

translateBy: aPoint
adds the argument aPoint to the offset of the receiver,

align: aPointOfAlignment with: anOtherPoint
translates the offset of the receiver such that the argument aPointOfAlignment is placed at the position of the argument anOtherPoint,

boundingBox
returns the minimum rectangle containing the receiver, taking the offset into account,

display
displays the receiver at the point 0@0,

displayAt: aPoint
displays the receiver at the point aPoint.

15.2.1 The class DisplayText

The class **DisplayText** is a subclass of the abstract class DisplayObject. The objects that we want to represent in the class DisplayText are instances of the class Text. Recall that a text is made up of a string of characters and of an instance of the class RunArray representing the style of the characters. To display a text, additional data is required, like for example the graphical representation of the character fonts or what action is to occur in the case of tabulation. Such information is collected together in the class **TextStyle**, each instance of which represents a style of display.

15.2.1.1 Creating an instance of the class DisplayText

The message asDisplayText, when sent to an instance of the class String or to an instance of the class Text, returns a new instance of the class DisplayText initialised with the character string or the text. For example:

```
Time now printString asDisplayText displayAt: 100@100
```

displays the current time at position 100@100.

Two other messages sent to the class DisplayText allow instances to be created:

```
text: aText
```

returns an instance of the class DisplayText consisting of the text given as argument; the style of the instance is given by a shared variable (pool variable) placed in the dictionary TextConstants and called DefaultTextStyle (recall that the dictionaries containing the pool variables are referenced by global variables).

```
text: aText textStyle: aStyle
```

returns an instance of the class DisplayText consisting of the text given as argument with as style the argument aStyle.

15.2.1.2 Instance methods

The instance methods of the class DisplayText are:

```
string
```

returns the character string that makes up the text of the receiver,

```
text
```

returns the text of the receiver,

```
text: aText
```

replaces the text of the receiver by the argument aText,

```
setText: aText textStyle: aStyle offset: anOffset
```

replaces the text, style and offset of the receiver by the values given as argument,

```
textStyle
```

returns the style of the receiver,

```
textStyle: aStyle
```

replaces the style of the receiver by the argument aStyle.

15.2.2 The class DisplayMedium

The abstract class **DisplayMedium** is another subclass of the class DisplayObject. The graphics object is stored in pixel form. The class DisplayMedium implements additional messages that allow a drawing to be filled with a motif and a border added to the rectangle containing the drawing. A motif is a 16 x 16 pixel drawing that is repeated in every direction. The messages can be divided into categories, as follows.

15.2.2.1 Background colour

The messages that fill a drawing with a motif are:

```
   black
```

fills the receiver with black,

```
   black: aRectangle
```

fills the intersection between the argument aRectangle and the rectangle containing the receiver with black,

```
   darkGrey
```

fills the receiver with dark grey,

```
   darkGrey: aRectangle
```

fills the intersection between the argument aRectangle and the rectangle containing the receiver with dark grey,

```
   lightGrey
```

fills the receiver with light grey,

```
   lightGrey: aRectangle
```

fills the intersection between the argument aRectangle and the rectangle containing the receiver with light grey,

```
   veryLightGrey
```

fills the receiver with very light grey,

```
   veryLightGrey: aRectangle
```

fills the intersection between the argument aRectangle and the rectangle containing the receiver with very light grey,

```
   white
```

fills the receiver with white,

```
   white: aRectangle
```

fills the intersection between the argument aRectangle and the rectangle containing the receiver with white.

15.2.2.2 Drawing borders

The messages that add a border to an instance of the class DisplayMedium are:

```
   border: aRectangle width: aWidth
```

draws a black border inside the rectangle aRectangle of width aWidth,

```
   border: aRectangle width: aWidth mask: aMotif
```

draws a border with the motif aMotif inside the rectangle aRectangle of width aWidth,

```
   border: aRectangle widthRectangle: aRectangleInterior
   mask: aMotif
```

constructs a rectangle from the rectangle aRectangle and from the rectangle aRectangleInterior, in which the new rectangle has as origin the origin of the first, plus the origin of the second, and as corner the corner of the first less the corner of the second. It fills the difference between the rectangle aRectangle and the new rectangle with the motif aMotif.

15.2.2.3 Filling with a motif

The aim of these messages is to compose two pictures, the first having the same size as the receiver, with background colour a motif propagated in all directions; the second being the picture of the receiver.

There are several ways of composing the two pictures. In the Smalltalk system an integer is assigned to each composition rule. There are 16 of them.

Composition rule	Final image
0	all the pixels are white
1	logical AND between the two pictures
2	logical AND between the first picture and the second inverted picture
3	the final picture is the first picture
4	logical AND between the first inverted picture and the second picture
5	the final picture is the second picture
6	exclusive OR between the two pictures
7	logical OR between the two pictures
8	logical AND between the two inverted pictures
9	exclusive OR between the first inverted picture and the second picture
10	the final picture is the second inverted picture
11	logical OR between the first picture and the second inverted picture
12	the final picture is the first inverted picture
13	logical OR between the first inverted picture and the second picture
14	logical OR between the two inverted pictures
15	all the pixels are black

The messages that allow a picture to be composed with a motif are:

fill: aRectangle mask: aMotif

fills the intersection between the argument aRectangle and the rectangle containing the receiving with the motif aMotif, applying composition rule 3,

fill: aRectangle rule: aCompositionRule mask: aMotif

fills the intersection between the argument aRectangle and the rectangle containing the receiver with the motif aMotif, applying composition rule aCompositionRule,

reverse

inverts the picture of the receiver, that is, fills the

picture of the receiver with a black motif, applying composition rule 6 (exclusive OR),

reverse: aRectangle

inverts the intersection of the receiver and the argument aRectangle,

reverse: aRectangle mask: aMotif

fills the intersection of the receiver and the argument aRectangle with the motif given as argument, applying composition rule 6 (exclusive OR).

15.2.3 The class Form

The class **Form** is a subclass of the class DisplayMedium. An instance of the class Form allows a rectangular picture to be stored. This picture is described by its height, its width and the value of the pixels that make it up. The instance variable called bits which stores pixels is an instance of the class Bitmap.

15.2.3.1 Creating a form

Six messages create and initialise a Form:

extent: aSize

returns a new instance of Form whose size is equal to the argument aSize, and whose bitmap is not yet initialised,

extent: aSize fromArray: anArray

returns a new instance of Form whose size is equal to the argument aSize, and whose bitmap is initialised to the values contained in the argument anArray,

extent: aSize fromArray: anArray offset: anOffset

returns a new instance of Form whose size is equal to the argument aSize, and whose bitmap is initialised to the values contained in the argument anArray and offset is initialised by the argument anOffset (the offset is a shift that is added when the form is displayed),

fromDisplay: aRectangle

returns a new instance of Form, created from the screen by taking the bitmap contained in the rectangle given as argument,

fromUser

returns a new instance of Form, created from the screen by taking the bitmap contained in the rectangle designated by the user,

fromUser: aPoint

returns a new instance of Form, created from the screen by taking the bitmap contained in the rectangle designated by the user; the position and size of this rectangle are multiples of the argument aPoint,

readFrom: aFileName

returns a new instance of Form read from the file called aFileName.

15.2.3.2 Other class methods

The Smalltalk system contains a number of predefined forms used to construct motifs. The size of these forms is 16 x 16 pixels. The forms available are:

black	returns a form in which all the pixels are black
darkGrey	returns a form in which almost all the pixels are black
grey	returns a form in which one pixel in two is black
lightGrey	returns a form in which the number of white pixels is greater than the number of black
veryLightGrey	returns a form in which almost all the pixels are white
white	returns a form in which all the pixels are white

Some class methods return an integer corresponding to a composition rule. These methods are:

and
returns 1 which corresponds to the logical AND between the two pictures,

over
returns 3 which corresponds to the replacement of the first picture by the second,

erase
returns 4 which corresponds to the erasure in the first picture of all the pixels that are black in the second,

reverse
returns 6 which corresponds to the exclusive OR between the two pictures,

under
returns 7 which corresponds to the logical OR between the two pictures, giving the impression that the second picture is located under the first.

15.2.3.3 Instance methods of the class Form

Some useful messages supported by the class Form are:

bits
returns the instance of the class Bitmap that stores the pixels,

bits: aBitmap
replaces the bitmap of the receiver by the argument aBitmap

borderWidth: aWidth
inserts a black border of width aWidth around the receiver,

displayOn: aDisplayMedium
at: aPoint
clippingBox: aRectangle
rule: aCompositionRule
mask: aMotif

displays the receiver on the argument a DisplayMedium at the position aPoint, without exceeding the limits of aRectangle and applying the mask aMotif with the composition rule aCompositionRule,

follow: aBlock while: aBlockCondition

so long as evaluation of the block aBlockCondition returns true, the block aBlock is evaluated to give the position at which the receiver must be displayed and the zone hidden by the receiver at the previous evaluation is redisplayed,

valueAt: aPoint

returns 0 or 1 depending on whether the pixel located at the position aPoint in the receiver is white or black,

valueAt: aPoint put: aBit

sets the pixel located at the position aPoint in the receiver to white or black depending on whether the argument aBit is 0 or 1.

15.2.4 The class Cursor

The class Cursor is a subclass of the class Form. Depending on the system status, different cursors appear. These cursors indicate, by their position, the place on the screen pointed to by the mouse, and by their form the status of the system. The cursor forms occupy 16 x 16 pixels.

The message cursorLink: sent to the class Cursor allows movement of the cursor to be linked or not to the movement of the mouse, depending on whether its argument is true or false.

The messages supported by the class cursor are:

show

where the receiver becomes the current cursor,

showGridded: aPoint

the receiver beccmes the current cursor, with its position being a multiple of the argument aPoint,

showWhile: aBlock

the receiver becomes the current cursor while the block given as argument is evaluated, after which the current cursor becomes the previous cursor.

15.2.5 The class DisplayScreen

The class **DisplayScreen** is a subclass of the class DisplayMedium. The instances of the class DisplayScreen serve to represent the complete screen (the screen is a rectangular form comprising a certain number of pixels; it may therefore be likened to an instance of the class

Form). There is in the Smalltalk system a single instance of the class DisplayScreen referenced by the global variable Display. Note the following message:

```
flash: aRectangle
```

which causes to flash that part of the screen corresponding to the rectangle given as argument.

15.2.6 The class Path

The class **Path** is a subclass of the class DisplayObject which is used to draw a form along the length of a "trajectory". A trajectory is an OrderedCollection of points. The instances of the class Path have two instance variables that are a form and a trajectory. The messages of the class Path may be divided into the following categories.

Access

The messages that allow access to the characteristics of the instances are:

```
form
```

returns the form referenced by the receiver,

```
form: aForm
```

replaces the form referenced by the receiver by the form aForm,

```
at: anIndex
```

returns the point of the trajectory referenced by the receiver of index anIndex,

```
at: anIndex put: aPoint
```

replaces the trajectory point of index anIndex by the point aPoint,

```
size
```

returns the number of points of the trajectory referenced by the receiver.

Test

The message isEmpty returns true if the trajectory referenced by the receiver contains no point.

Add

The message add: adds the point given as argument at the end of the trajectory referenced by the receiver.

Remove

The message removeAllSuchThat: removes from the trajectory referenced by the receiver all the points for which evaluation of the block given as argument returns true.

Enumeration

The three messages that allow the points of a trajectory to be handled successively are:

```
do: aBlock
```

evaluates the block given as argument for each point of the trajectory referenced by the receiver,

collect: aBlock

evaluates the block given as argument for each point of the trajectory referenced by the receiver and returns an instance of OrderedCollection containing the results of each evaluation,

select: aBlock

evaluates the block given as argument for each point of the trajectory referenced by the receiver and returns an instance of OrderedCollection containing the points for which the evaluation has returned true.

The subclasses of the class Path correspond to particular trajectories. These are:

- straight lines
- arcs of circles
- circles

15.2.6.1 The class Line

The class **Line** is a subclass of the class Path that allows rectilinear trajectories to be represented by giving the two extremities of the trajectory. The additional messages implemented in this class are:

beginPoint

returns the first point of the receiver,

endPoint

returns the second point of the receiver,

beginPoint: aPoint

replaces the first point of the receiver by the point aPoint,

endPoint: aPoint

replaces the second point of the receiver by the point aPoint.

15.2.6.2 The classes Arc and Circle

The classes **Arc** and **Circle** are subclasses of the class Path which allow sections of circles and circles to be represented.

15.3 The class BitBlt

The class **BitBlt** is a subclass of the class Object. (The name BitBlt stands for Bit Block Transfer, a technique for altering and copying rectangular areas of a picture.) The instances of the class are objects that are capable of transferring pixels of one form to another form, while respecting certain constraints given by their instance variables. The instance variables that control pixel transfer are:

destForm

which is a form in which the transferred pixels will be stored,

sourceForm

which is a form from which the pixels to be transferred are taken,

halftoneForm

which is the motif with which the starting form will be composed. There are four possible cases:

- sourceForm and halftoneForm are equal to nil; the form transferred will be a black form
- sourceForm is equal to nil; the form transferred will be the motif propagated in all directions
- halftoneForm is equal to nil; the form transferred will be the sourceForm
- halftoneForm and sourceForm are not nil; the form transferred will be obtained by making a logical AND between the sourceForm and the motif propagated in all directions.

combinationRule

which is an integer representing the composition rule of the starting form and arrival form (cf composition rules defined in 15.2.2).

sourceX, sourceY, width, height

which are four integers designating the region of the starting form from which the pixels will be copied,

destX, destY

which are two integers designating the region of the destination form where the pixels will be placed,

clipX, clipY, clipWidth, clipHeight

which are four integers that limit the region that can be altered by the copy.

The message that makes the copy, while respecting the constraints imposed by the instance variables, is copyBits. Generally, the call to BitBlt is made by other classes like Form, Line, etc., and the user does not have to invoke it explicitly.

The class BitBlt implements another important message:

drawFrom: aStartPoint to: anEndPoint

which allows a form to be propagated between the two points given as argument.

15.4 The class Pen

The class **Pen** is a subclass of the class BitBlt. The instances of the class Pen simulate a stylus that can be moved on the screen and which leaves a trace, or does not, depending upon whether it is lowered or raised.

Since the class Pen is a subclass of the class BitBlt, drawing operations are carried out more quickly using it rather than instances of the class Line.

The instances of the class Pen have four instance variables:

penDown

which is a boolean indicating whether the stylus is lowered or raised,

location
which is a point indicating the current stylus position,

direction
which is an angle expressed in degrees indicating the direction of the stylus trace,

frame
which is a rectangle indicating the limits within which the stylus can trace.

The messages of the class Pen may be divided into the following categories.

Access

The messages that allow access to the characteristics of the stylus are:

down
lowers the receiver,

up
raises the receiver,

location
returns the current position of the receiver,

direction
returns the direction in degrees of the trace of the receiver,

turn: anAngleInDegrees
adds to the current direction of the trace of the receiver the argument anAngleInDegrees, which is equivalent to turning the angle anAngleInDegrees in relation to its current direction,

north
orients the trace of the receiver towards the top of the screen,

frame
returns the rectangle in which the receiver may trace,

frame: aRectangle
replaces the rectangle in which the receiver may trace by the argument aRectangle,

defaultNib: aWidth
replaces the width of the receiver by the argument aWidth,

colour: aMotif
replaces the motif of the receiver by the motif given as argument.

Movement

The messages that allow the stylus to be moved are:

place: aPoint
replaces the current position of the receiver by the point aPoint,

home

places the receiver at the centre of the rectangle in which it may trace,

```
go: aDistance
```

moves in the current direction by a distance aDistance, tracing if the receiver is lowered,

```
goto: aPoint
```

moves in a straight line to the point given as argument, tracing if the receiver is lowered.

16 Basic Elements of the Interface

Every information system has an interface to allow communication with a user. This interface usually consists of a keyboard and a screen.

There are two basic types of screen, a character type and a graphics type. Smalltalk uses a graphics screen.

Two technologies are used for graphics screens, depending on their function - vector scan screens and raster scan screens. The former allow lines to be drawn very quickly, and they are used for CAD and similar applications. The latter manipulate points; if these points are individually accessible, such screens are said to be bit-mapped. It is this last type of screen that we find in the Smalltalk system.

The number of points on the screen depends on the machine and the Smalltalk system can adapt itself to this number of points.

The Smalltalk system uses a standard keyboard as input device, but another device, commonly known as a "mouse", is also required.

16.1 Mouse

A mouse is a device that allows the user to designate a point on the screen by moving the mouse around on a flat surface. The mouse incorporates buttons that cause different operations to occur when they are pressed. There are usually three buttons in the Smalltalk system.

These buttons are identified by colour:

the left button is the red button (redButton),

the centre button is the yellow button (yellowButton),

the right button is the blue button (blueButton).

In the Smalltalk system each button has a particular function:

the red button is used to select or designate a visible area,

the yellow button is used to call up the menu that allows the inside of the viewed area to be changed,

the blue button is used to call up the menu that allows the viewed area itself to be changed.

16.2 Windows

A window is a rectangular portion of the screen in which

a program specific to this window runs, independently of the content of other windows. For each application there are specialised windows. For example, there are windows in which text can be edited and others in which drawings can be edited.

In the Smalltalk system, activities are not assigned to the screen but to the individual windows that go to make up the screen.

A window is said to be active when the user can interact with the contents of that window. There can be only one window active at a time.

When windows are superimposed, the active window is the one that is visible.

If there is no active window, as soon as the cursor moves inside a window, that window becomes active. To change windows, the cursor must be placed in an inactive window and a button on the mouse pressed. If one of the mouse buttons is pressed with the cursor outside any window, there is no longer any window active.

16.3 Pop-up menus

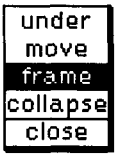
A pop-up menu is a window that appears at the cursor position when the yellow or blue button on the mouse is pressed. This menu remains displayed for as long as the user holds the button down.

These menus offer a list of commands. To select a command the user must move the cursor to the command that he wishes to execute and then release the button. Positioning the cursor on the command has the effect of inverting the display of this command line. If the user releases the button without selecting any command, the pop-up menu disappears and no command is executed.

Pop-up menus are instances of the class **PopUpMenu** which is a subclass of the class Object.

16.4 Standard windows

One of the important window classes is the class **StandardSystemView**. The instances of this class are windows that have a title indicating the functions of the view. These windows can be manipulated; they can be moved, altered in size, etc. These actions are carried out by means of a pop-up menu which appears on the screen when the blue button is pressed while within the window. The diagram below shows a menu example that appears when the blue button is pressed in a standard window:



The actions possible are:

under

calls for a window below the cursor to be activated. If there are several inactive windows under the cursor, one of these will be activated. If there is no inactive window under the cursor, this command has no effect,

move

moves the active window without altering its size. The window follows the movement of the cursor until the red button is pressed,

frame

changes the dimensions of the active window. An origin cursor follows the movement of the mouse until the red button is pressed. The position thus obtained corresponds to the top left corner of the window. Subsequently, a corner cursor follows the movement of the mouse for as long as the red button remains pressed. The new position obtained corresponds to the bottom right corner bottom of the window.

collapse

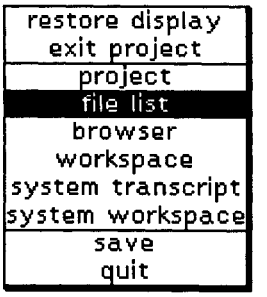
removes all of the window except its title. To make this window reappear, the command frame must be used to change its dimensions,

close

causes the window to disappear completely from the screen.

16.5 The system menu

The system menu is the pop-up menu that appears when the yellow button is pressed outside any window. This is the menu that, for example, is used to create windows or to exit the Smalltalk system. The diagram below shows an example of a system menu.



The commands offered by this menu are:

restore display

redisplays the complete screen, beginning with the background in a grey motif, then displaying all the windows,

exit project

allows you to exit from the current project. A project is a list of windows that relate to a particular application. When you return to a project, the screen is cleared and the windows of this project are displayed. When you have entered a project, the system menu is the same,

project
opens a window that will allow entry to a project,

file list
opens an access window to the external files,

browser
opens a window that allows classes and methods to be edited,

workspace
opens a window that allows text to be edited and expressions evaluated,

system transcript
opens a window used by the Smalltalk system to display its messages,

system workspace
opens a text window whose contents are initialised by the Smalltalk system with a number of expressions that are useful to know,

save
allows the Smalltalk virtual image to be saved to a file,

quit
allows you to quit the Smalltalk system.

17 Text Editing Windows

Numerous types of window use a text editor. Whatever the window type, the editor used is always the same.

To edit text, the user can use the keyboard and the mouse. The keyboard provides the means of entering new characters, while the mouse allows a section of text to be selected with the red button or commands to be executed using the yellow button.

If a section of text is too long to be displayed all at once in the window, there needs to be a means of displaying only part while retaining easy access to the remainder. This is what happens on a conventional word processor page, where at most 24 lines are displayed, but the rest of the text can be accessed by means of arrow keys. In the Smalltalk system, the windows have another means of accessing a particular section of text.

17.1 Scrollbars

A "scrollbar" is a rectangular area that appears on the left of an editing window of active text when the cursor is located inside the window.

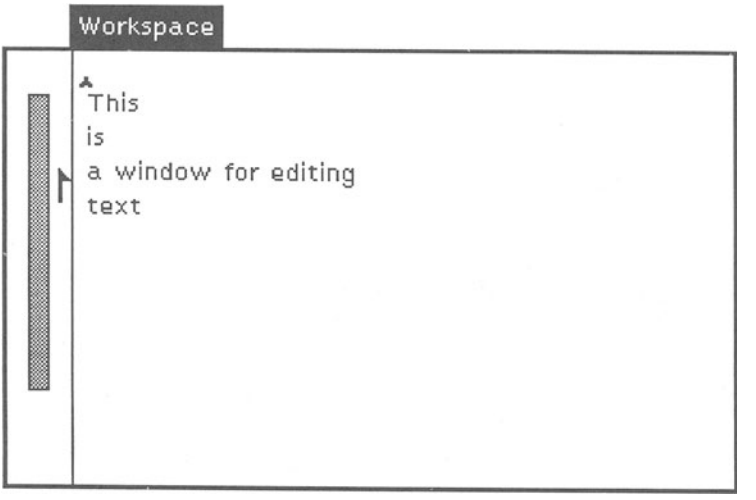
Whenever the cursor moves outside of this window, the scrollbar disappears, only to reappear again when the cursor returns to the window.

This scrollbar represents the totality of the text. It contains a grey rectangle whose position in the scrollbar represents the position of the text shown on the screen relative to the text in its entirety. Its height, in relation to the height of the scrollbar, represents the proportion of text shown on the screen in relation to the text in its entirety.

To access a particular section of text, the rectangle within the scrollbar must be moved. When the mouse points to the interior of the scrollbar, there are three different cursors, depending on the relative position of the cursor and of the interior rectangle.

If the mouse points to the left of the rectangle, the down cursor appears. If the user pressess the red button, the inside rectangle moves up and the text displayed in the text window moves down, as shown in the following diagram.

If the mouse points to the right of the interior rectangle, the up cursor appears. If the user presses the red button, the interior rectangle moves down and the text displayed in the text window moves up, as shown below.

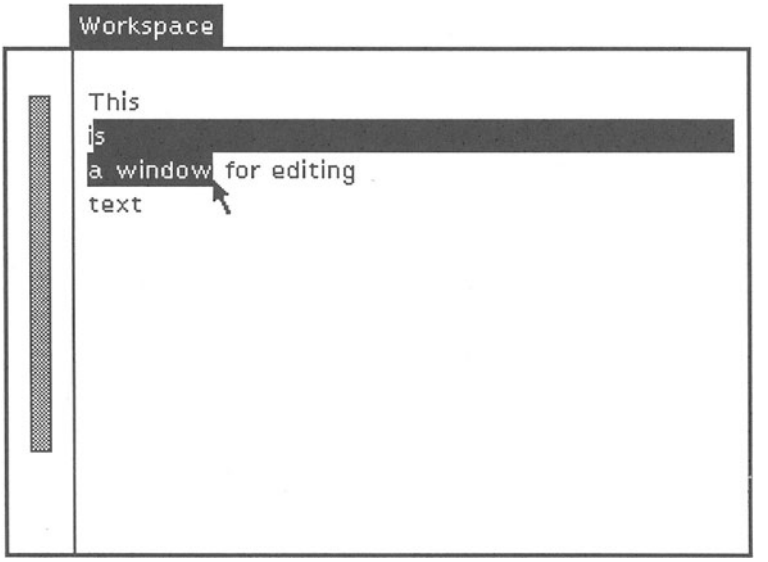


If the mouse points along the same level as the interior rectangle, the marker cursor appears. If the user presses the red button, the interior rectangle centres itself on the cursor and causes the text to move, as shown below.

17.2 How to select text

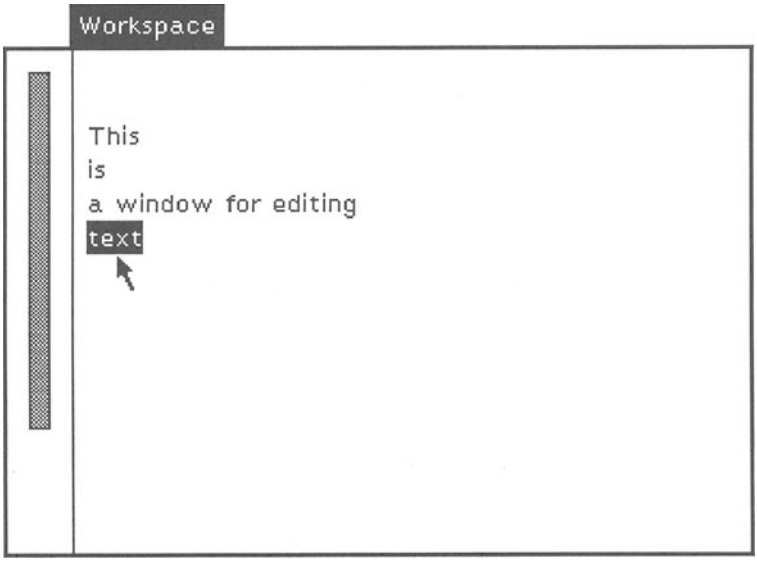
Selecting part of a text means choosing a continuous section of text with a view to manipulating it with an edit command.

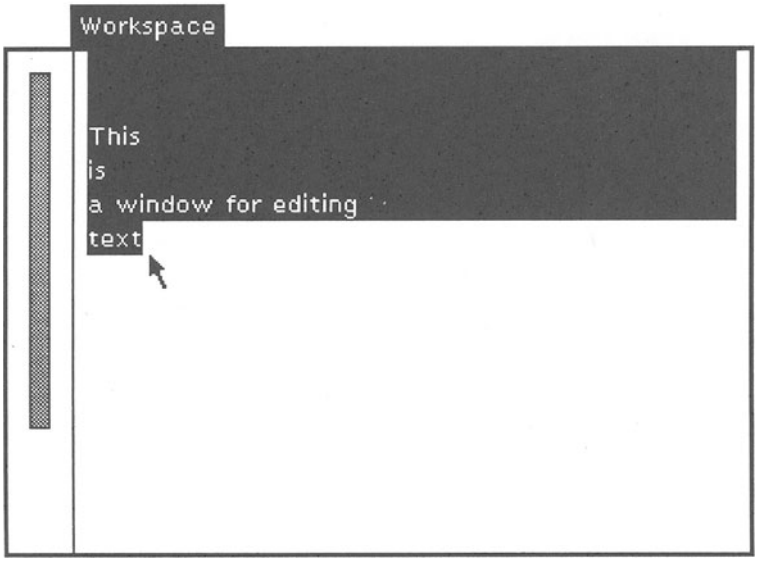
In a text window there is a special cursor that indicates the position of the next text input. This cursor is shaped like an inverted "v", as can be seen on the previous three diagrams. This cursor can be moved around in the window by pressing and releasing the red button on the mouse until the correct position for the mouse cursor is found. There are several ways of selecting a text; these are described in the following paragraphs.



The mouse cursor is placed at a position on the text, the red button is pressed without being released and the mouse cursor is moved to another position on the text, then the button is released. The text located between these two positions will be displayed in reverse video, as shown in the figure on page 124 (bottom).

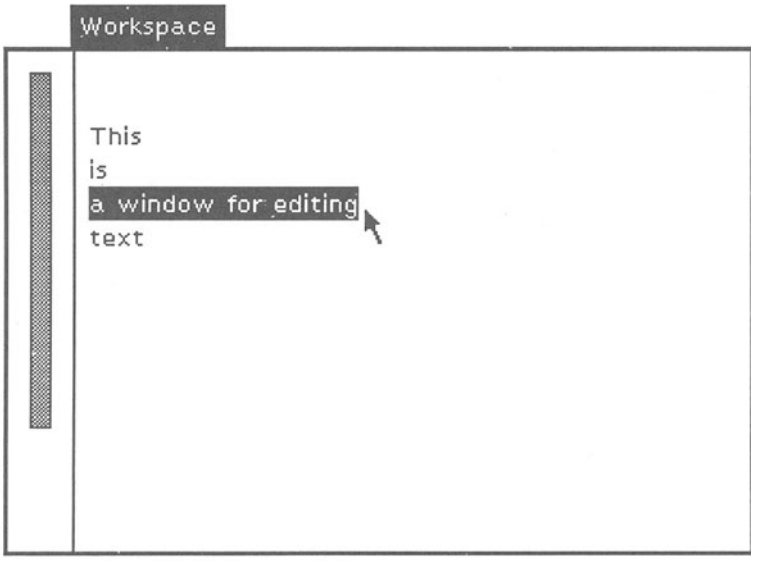
The mouse cursor is placed on a word, the red button is pressed and released twice without moving the mouse. This will select a complete word, as shown below.

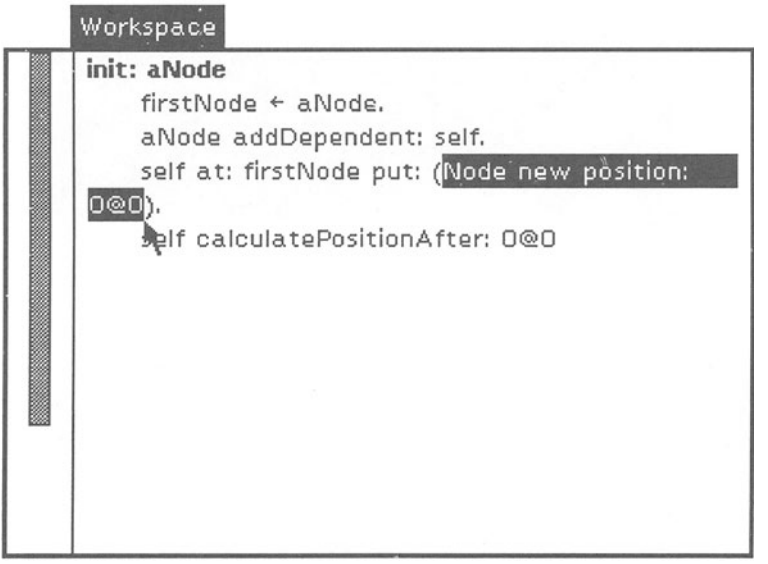




The cursor is placed at the beginning or at the end of the text, the red button is pressed and released twice without moving the mouse. This will select all of the text, as shown on page 125.

The cursor is placed after or in front of a carriage return, the red button is pressed and released twice without moving the mouse. This will select that section of text that extends backwards or forwards to the next carriage return, as shown below.





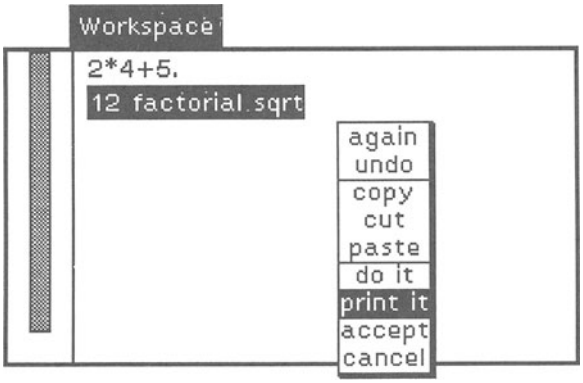
The cursor is placed after or in front of a delimiter, the red button is pressed and released twice without moving the mouse. This will select that portion of text located between delimiters, as shown in the example at the foot of page 126.

If the Escape key on the keyboard is pressed, all text entered since the last use of the mouse buttons will be selected.

If the text contains a selected section and the user enters characters via the keyboard, the entered characters will replace those in the selected section.

17.3 Edit commands

The edit commands can be accessed by a menu that appears when the yellow button is pressed. Each command is executed as soon as the button is released. This menu is illustrated below.



The commands are:

cut
removes the selected section from the text,

copy
places a copy of the selected text in a buffer,

paste
replaces the selected text with the contents of the buffer,

again
re-executes the last command,

undo
cancels the effect of the last command.

17.4 Workspaces

A workspace is a text-editing window which makes it possible to evaluate expressions contained in a selected section of text. Smalltalk expressions can be evaluated

by two commands that can be accessed via the yellow button menu. They are:

doIt

which evaluates the selected text, and

printIt

which evaluates the selected text and displays the text that describes the object returned by the last evaluation. The returned text becomes the current selection.

When a text is evaluated, the text expressions are compiled. If there is a syntax error, an error message is displayed immediately in front of the position corresponding to the detected error. This message becomes the selected section of text, which allows it to be amended immediately or replaced with other text.

In the basic version of Smalltalk-80, when there is an unknown variable in an expression, a menu appears offering a choice of possible declarations for this variable. This menu also offers the possibility of correcting the variable name or of abandoning the evaluation.

More recent versions of Smalltalk-80 consider that variables can be local to a workspace. These variables will be defined when first assigned and will be stored in a dictionary local to the workspace.

During evaluation of expressions, the scrollbar disappears from the window. It reappears when the evaluation is finished.

17.5 System workspace

The System workspace is a workspace whose contents consists of expressions that it may be useful to know in order to change certain operating parameters in Smalltalk. The parameters that can be changed are, for example:

the size of the screen

the "sources" and "changes" files

The System workspace also contains expressions for reconstructing an image after an accidental break.

18 Browsers

The browsers are windows that allow access to the classes and methods. A browser has two possible uses; it allows the user to:

- consult or change the existing classes and methods. Since the Smalltalk environment is itself written in Smalltalk, the user has access to all of its code. From this point of view Smalltalk can be regarded as a self-documented language. The advantage of having access to the existing classes encourages the reuse of code. It is not advisable to change the existing classes or methods, as any such changes could have consequences well beyond any that could be foreseen.
- create new classes or new methods. In the Smalltalk environment, the building of an application amounts to enriching the environment with new classes and new methods.

18.1 Description

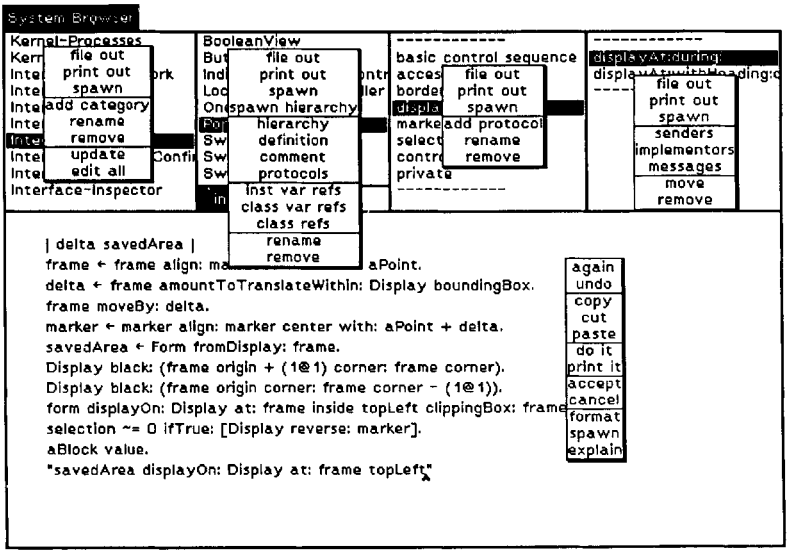
A browser is made up of five main windows. Four of these give access to the following lists:

- list of categories of classes,
- list of classes belonging to the selected category,
- list of categories of methods or protocols of the selected class,
- list of methods of the selected protocol.

The fifth window, located beneath the other four, is a Smalltalk code editing window, the contents of which will be interpreted according to the options selected from the lists.

There are two additional windows located under the list of classes; the one on the left contains the 'instance' text, that on the right the 'class' text. These two windows allow access to a class or to its metaclass, depending on whether the left or right window is selected (indicated by reverse video). If the left window is selected, the methods displayed are the instance methods; if the right window is selected, the class methods are displayed. These two windows cannot be selected simultaneously.

Each window has a particular menu linked to the central button on the mouse. The diagram below shows these different menus.



18.2 The categories of class

The categories of class group together the classes by function. Once again, be reminded that the categories have no influence on the behaviour of the system. The list of these categories appears in the left window. The menu in this window, linked to the central button, contains eight functions, namely:

file out

which, for all classes of the selected category, outputs the text of the class and its methods to a file. This file can then be reread by the message fileIn. The name of the file produced is the name of the category with the suffix '.st',

print out

which causes the same action as the function 'file out', except that in this case the file created cannot be re-read. Special formatting is inserted into the text so that when printed it can be more easily read by humans. In particular, the names of methods appear in bold characters. The name of the file produced is the name of the category with the suffix '.pp'. (Note that the exact nature of this option varies from machine type to type, so your machine may do something different.)

spawn

which opens a browser dedicated to the selected category,

rename

which allows the name of the selected category to be changed,

remove

which removes the selected category, along with all the classes that belong to it,

update

which updates the list of categories. This list may have been changed in another browser or by reading a file,

edit all

which presents the list of categories and their classes in the Smalltalk code editing window of the browser. This list can then be changed using the menu available in that window.

18.3 The classes

The list of classes of the selected category appears in the second window. If no class category is selected, this window is empty. The functions offered by the menu in this window are:

file out

as before, this outputs the definition of the selected class and the text of its methods to an external file. The name of the file produced is the name of the class concatenated with the suffix '.st',

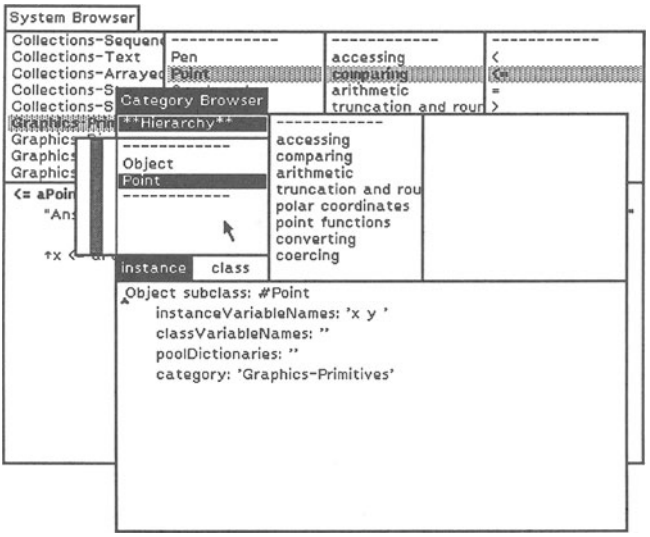
print out

as for the class categories, this outputs the definition of the selected class and the text of its methods to an external file. The name of the file produced is the name of the class with the suffix '.pp',

spawn

opens a browser dedicated to the selected class, as shown in the diagram below for the class Point:

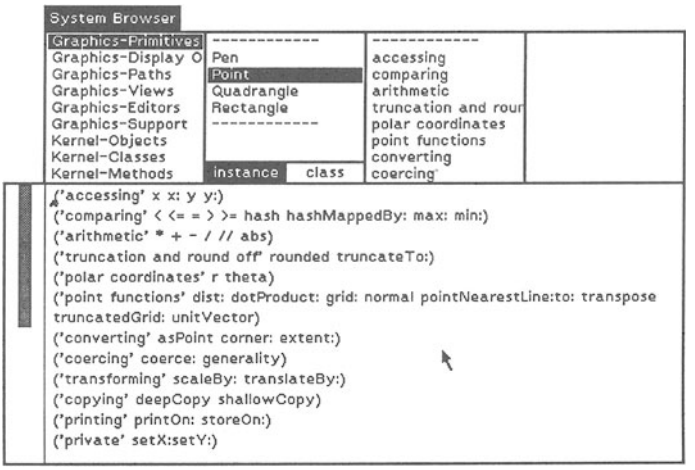
spawn hierarchy
opens a browser on the hierarchy of the selected class (that is, its subclasses and superclasses), as shown in the following diagram:



hierarchy
displays in the code window the hierarchy of the selected class; the depth of the classes in their hierarchy is represented by indentation of the text,

definition
displays in the text window the definition of the selected class,

comment
displays in the code window the commentary relating to



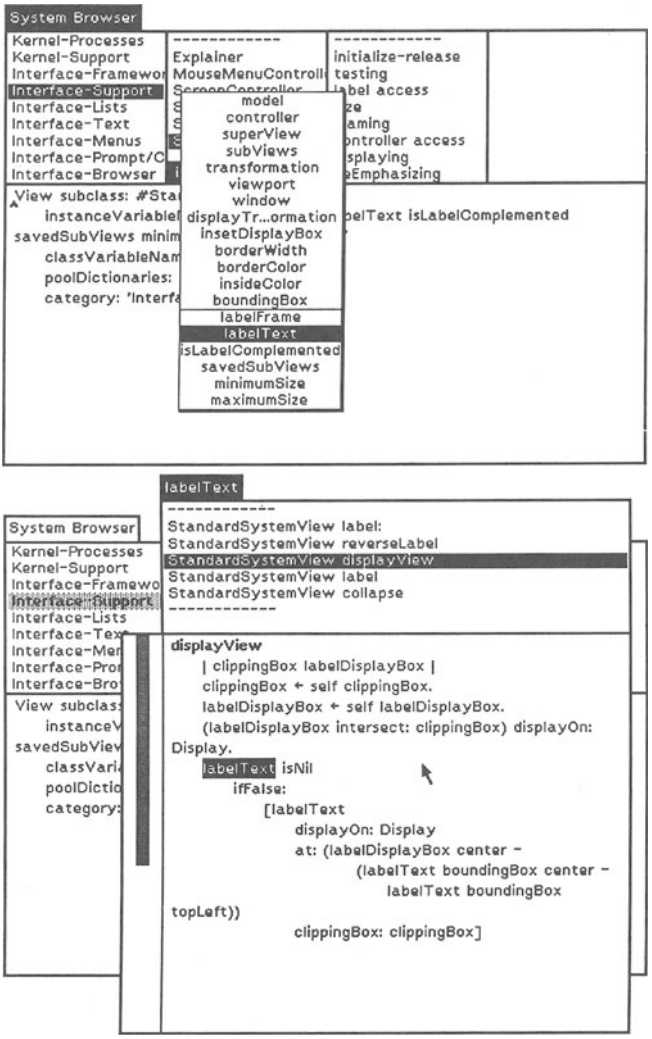
the selected class. This can then be changed,

protocols

displays in the code window the list of protocols and methods belonging to these protocols, as shown on page 132 (foot). The name of a protocol is a string of characters, whereas the name of a method is a symbol.

inst var refs

causes a pop-up menu to appear containing the list of named instance variables of the selected class and of its superclasses. It is then possible to select one of the variables from the offered menu, which has the effect of opening a browser on the list of methods that make reference to this variable. The following two diagrams show the selection of the instance variable labelText from the class StandardSystemView:

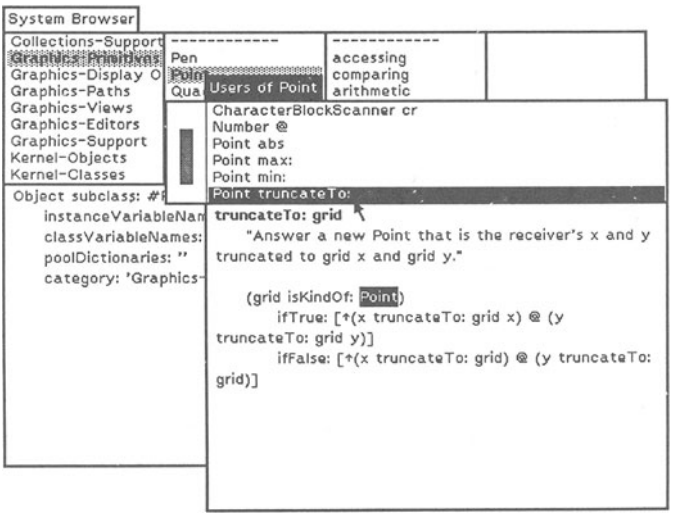


class var refs

causes a pop-up menu to appear containing the list of class variables of the selected class and its superclasses. It is then possible to select one of the class variables from the menu, which has the effect of opening a browser on the list of methods that make reference to this variable.

class refs

opens a browser on the list of methods that make reference to the selected class, as shown in the diagram below:



rename

allows the name of the selected class to be changed. If any methods make reference to this class, a browser is opened on the list of these methods,

remove

removes the selected class and its subclasses if they exist.

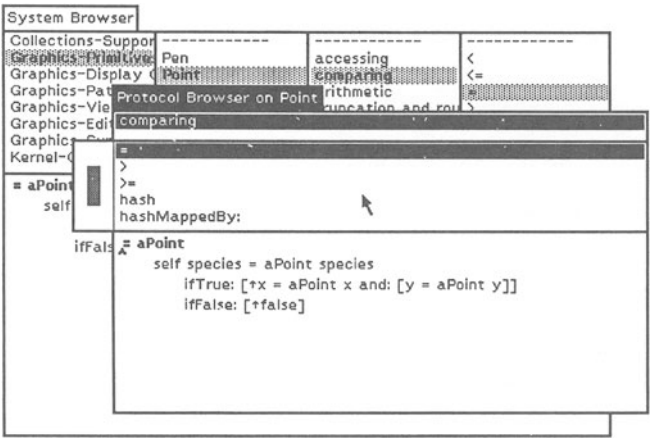
18.4 Protocols

The third window contains the list of protocols of the selected class. If no class is selected, this window is empty. The reader is reminded that the sole function of protocols in the system is to clarify the presentation of the methods in browsers. The functions offered by this menu are:

file out

which outputs the text of all the methods belonging to the selected protocol. The name of the file produced is the name of the class concatenated with the name of the protocol, concatenated with the string '.st',

print out
causes the same action as file out, but produces a formatted text,
spawn
opens a browser on the list of methods of the selected protocol. The figure below shows an opened browser on the protocol comparing of the class Point:



add protocol
allows a protocol to be added to the list of existing protocols. The name of this protocol is requested in an edit window,
rename
allows the name of the selected protocol to be changed,
remove
removes the selected protocol, together with the list of all its methods.

18.5 Methods

The list of methods belonging to the selected protocol is given in the right window. If no protocol is selected, this window is empty. The functions offered by the menu are:
file out
which allows the text of the selected method to be output. The name of the file produced is the name of the class, concatenated with the name of the method, concatenated with with the string ´.st´,
print out
causes the same action as file out, but produces a formatted text,
spawn
opens a browser on the selected method,
senders
opens a browser on the list of methods that send the message with the same name as the selected method,

implementors
opens a browser on the list of methods with the same name as the selected method,

messages
causes a menu to appear giving the list of all the messages sent in the selected method. Using this menu, it is possible to access all the implementations of one of the messages given in the menu,

move
allows the selected method to be moved. If it is moved to another class, it is copied; if its protocol is changed, it is simply moved,

remove
removes the selected method.

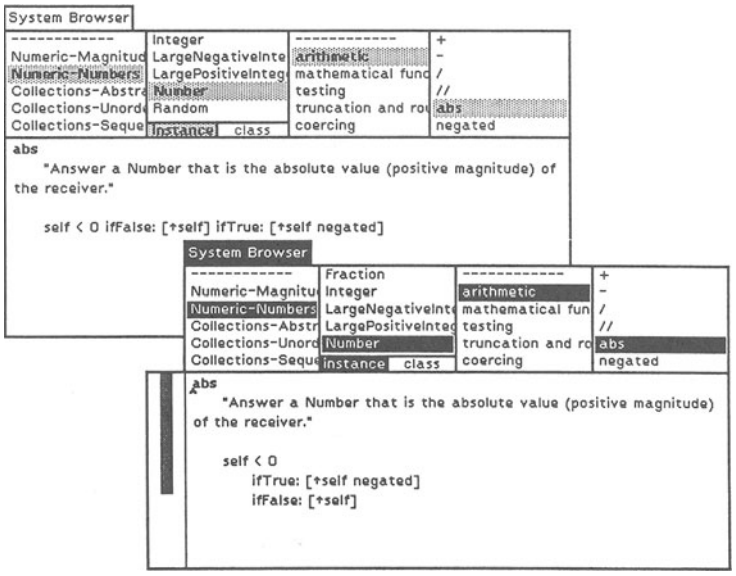
18.6 The code window

The Smalltalk code window, which appears in the browser, is like the text windows described in the previous chapter. In addition to its editing functions, this window offers three more functions in its menu:

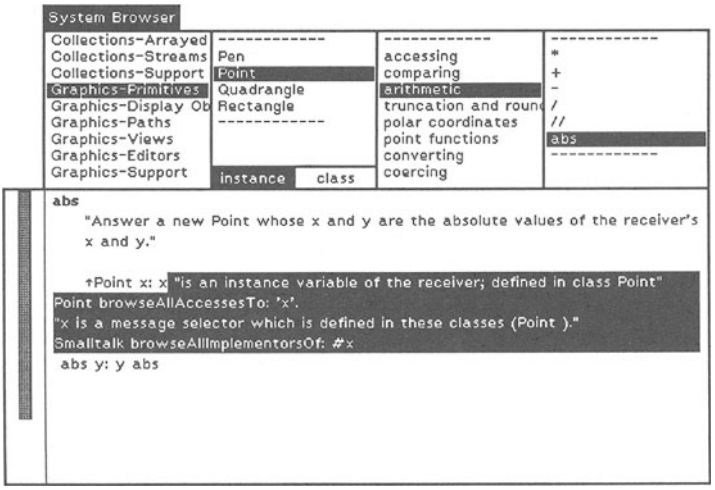
format
this function relates exclusively to methods and will thus only be valid when a method is displayed. Its purpose is to present the code of the method in the clearest possible form. To achieve this, the name of the method is shown in bold and some conventions in its presentation are observed; the most important of these are:

- there is only one statement per line
- loops are shown with additional indentation.

The figure below shows the effect of the formatting on a method.



spawn
which opens a new browser on the selected method,
explain
this function is very useful; it allows the nature of the selected word to be explained. Only one word can be selected at a time. The following example shows the explanation provided by the system for the word ´x´. This word can be explained in two ways: x is an instance variable of the class Point, and x is a message defined in the class Point.



Depending upon the options chosen from the list at the top, the code window has two main uses, described in the following sections.

18.6.1 Editing a class

When a category is selected, without a class being selected, a text giving the general form of a class is displayed; this can be changed. To capture this newly defined class, the accept function accessible via the middle button must be used.

When a class is selected, without a protocol being selected, the definition of this class appears in the text window. It is then possible to change and accept it, as before.

18.6.2 Editing a method

When a protocol is selected, without a method being selected, the general form of a method is displayed in the text window. The user may change this general form in order to produce a new method. If the name of the method that is displayed is the same as that of a method already in existence in this class, the newly accepted

method will replace the old. To compile the method, the accept option (via the centre button) must be used.

When a method is selected, the text of this method is displayed in the text window. This text may then be changed. If the method name is changed, a new method will be created.

19 Other Windows

19.1 Inspectors

Just as a browser allows classes to be accessed and changed, so an inspector allows access to any object in the system. It provides access to all the instance variables of the object inspected and all the Smalltalk objects can be inspected. Inspectors are used chiefly during the development phase of an application.

To open an inspector on an object, the message inspect has to be sent to it. For example:

```
#(1 2 3 4) inspect
```

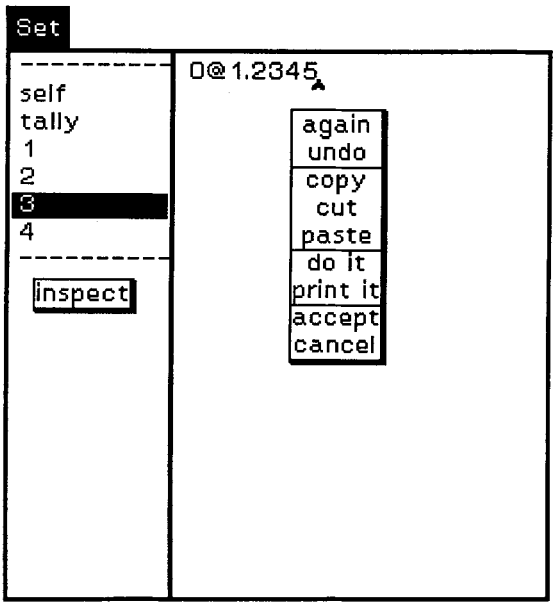
opens an inspector on the receiver of the message,

```
Set someInstance inspect
```

opens an inspector on an instance of the class Set.

19.1.1 Description

An inspector type window consists of two subwindows, as shown in the figure below.



The left window contains a list made up of the symbol self, representing the object inspected, and the list of names of its instance variables. In the case where the inspected object has indexed instance variables, the latter are represented by their index.

In the left window, the menu linked to the middle button offers the choice of inspecting the object that is selected. If the selected object is self, a second inspector will be opened on the same object.

The right window is a text window which contains the textual description of the object that is selected from the list on the left. The reader is reminded that this textual description is obtained by sending the message printString to the selected object.

The menu in this window, linked to the middle button, offers the usual functions suggested in a text window, namely:

- three text-editing functions: cut, copy and paste
- two evaluation functions: do it and print it
- two functions for manipulating the window contents: accept and cancel. The contents are the text that appears when it is open.

In contrast to the text windows that we have met before (for example, workspaces), evaluations in an inspector are carried out in the context of the inspected object, that is, at the time of evaluation references can be made to an object contained in the list on the left. In particular, an expression can be built by sending a message to self or to one of its instance variables, which is not the case in a workspace.

The function accept assigns the result of the evaluation of the text present in the text window to the variable selected from the list on the left.

19.1.2 Other inspectors

We have seen how in order to inspect an object the message inspect has to be sent to it. The method for responding to this message is defined in the class Object; thus all objects can be inspected. In some classes, this message is redefined, which allows the opening of windows adapted to certain types of objects.

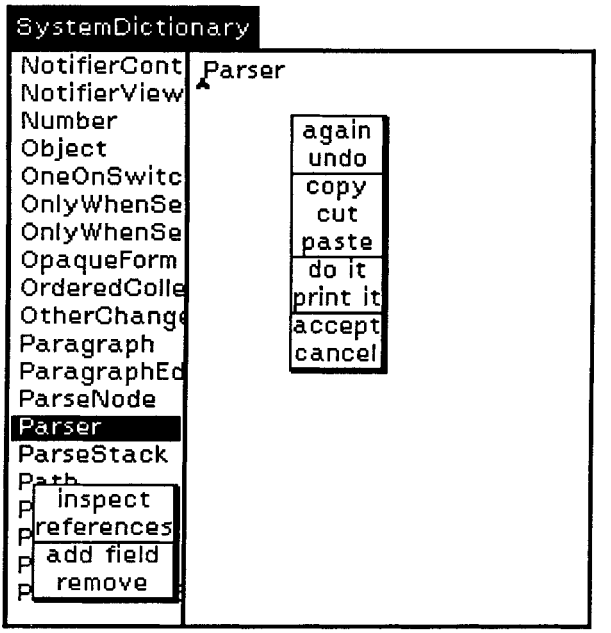
Dictionaries have special inspectors that allow keys to be manipulated. The following figure shows an inspector on a well known dictionary, namely the dictionary called Smalltalk, which contains all the global variables in the system, it will be recalled. In the left window the keys of the dictionary appear, while in the right window the value assigned to the selected key appears.

The left window offers the following options:

inspect

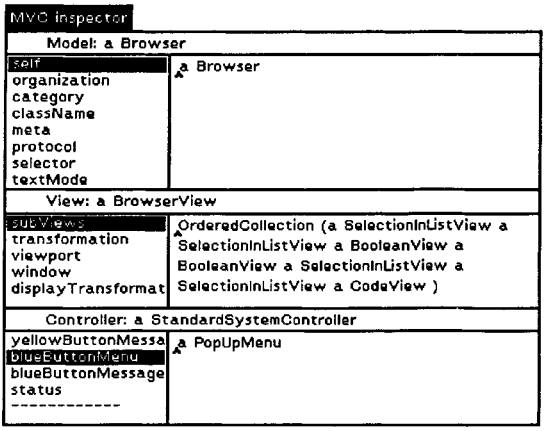
which opens an inspector on the selected variable,

references
which opens a browser on all the methods that refer to the selected variable,
add field
which allows a key to be added to the dictionary,
remove
which removes the selected key from the dictionary.



Views (instances of the class View) also have special inspectors which allow access to the model and to the controller assigned to the view to form a MVC (model-view-controller) system. The MVC system is described in detail in chapter 21.

The figure below shows an inspector on an instance of



BrowserView, which is the view assigned to a browser. This inspector is in fact made up of three inspectors on the three objects of the MVC system.

19.2 Debuggers

Development tools are very important tools in programming environments and in Smalltalk they have been very carefully designed.

Debuggers are windows that serve to visualise the methods and variables implicated in the evaluation of an expression during the actual evaluation. A debugger can be opened from a window called a Notifier, in which the menu, controlled by the middle button, offers two actions:

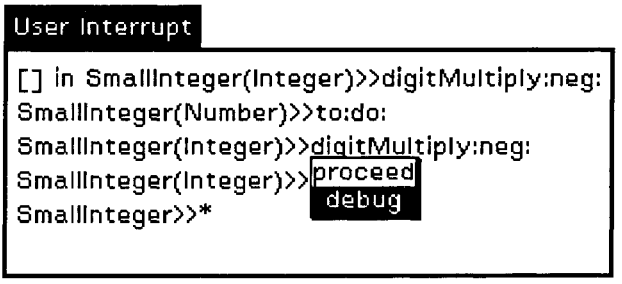
proceed

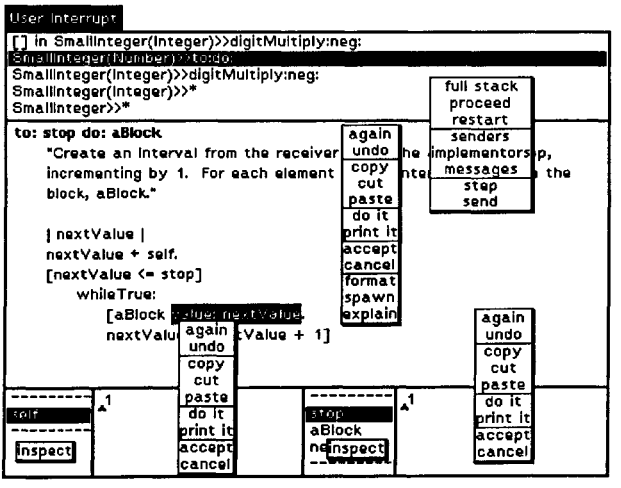
which resumes the interrupted activity,

debug

which opens a debugger on the interrupted context.

The Notifier contains the list of the last methods that were used before the interruption, as shown in the following figure.





Several circumstances can cause the opening of a Notifier. These are usually:

- an object does not understand a message that it receives,
- the user causes interruption of an activity by pressing control C,
- an object receives a halt message.

A debugger type window is made up of several subwindows, as shown in the bottom figure on page 142. The six windows involved are described below.

19.2.1 List of methods

This list contains the name of the last methods that were active before the interruption. When the user selects one of the methods contained in the list, the text of this method appears in the code window located below, with the expression that was being evaluated when the interruption occurred highlighted. This subwindow contains a menu controlled by the middle button which offers the following options:

full stack
which completes the list of methods by adding to it all the methods that were active within the process,

proceed
which resumes execution of the interrupted process,

restart
which restarts evaluation of the selected method from the beginning,

senders
which opens a browser on the list of methods that send a message with the same name as the selected method,

implementors
which opens a browser on the list of methods with the same name as the selected method,

messages
which cause a pop-up menu to appear giving the list of all messages sent from the selected method. From this menu it is possible to access all the implementations of a message present in the menu,

step
which causes the next expression to be evaluated. The selected part in the code window is updated,

send
which causes the next message in the selected expression to be sent.

19.2.2 The code window

This window is located under the window that provides the list of methods that have been used. When a method is selected from the list of methods, its text appears in the code window, as in a browser. The expression that

was being evaluated at the time of the interruption is shown in inverse video. The functions offered in the menu, which is controlled by the centre button, are the same as those offered in the code window of a browser.

19.2.3 Inspector of the receiver

This inspector is an inspector which describes the receiver of the message selected from the list at the top. In this window it is possible to access all the instance variables of the receiver and change them.

19.2.4 Inspector of local variables

The inspector enables access to the arguments of the method selected from the list of methods, as well as access to all its local variables. It is possible to consult or change these variables.

19.3 Views of external files

These windows, called 'fileLists', allow access to the file system underlying the Smalltalk virtual machine. The windows allow access to both directories and files. They are used in particular for integrating new classes into the system; rather than transfer complete images, it is more economical to transfer definitions of classes and methods that have been obtained by means of a fileOut.

The fileList type windows are made up of three sub-windows, as shown in the following figure.

The top window is a text window in which the user enters the name of the file that is to be accessed. The character '*' can be used to replace any character string when designating the file; for example:

```
*.st
```

designates all the files suffixed by '.st', that is, all the files obtained by a fileOut in a browser.

Once the name of the file(s) has been entered, the user can obtain a list of all the filenames matching the name entered by using the accept function controlled by the middle button.

The central window contains the list of the files whose name is compatible with the filename specification of the top window.

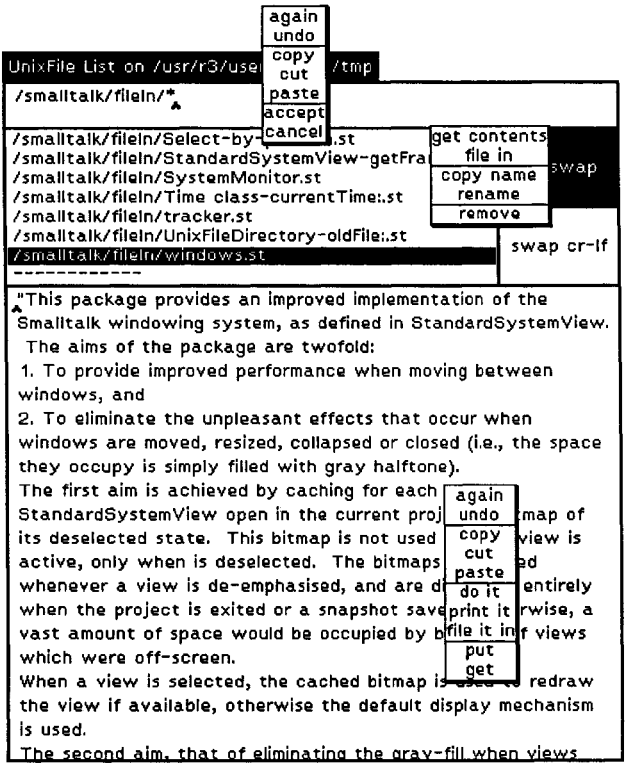
If there are files whose name corresponds to the name in the top window, the menu controlled by the middle button offers the following options:

```
get contents
```

allows the text from the selected file to be loaded into the bottom window,

```
fileIn
```

this operation carries out the opposite function to the browser fileOut; it allows the text of the selected file



to be compiled. The latter must contain the definitions of classes and methods; that is, it must have been obtained by a fileOut,

copy name

which allows the name of the selected file to be copied into a text buffer (this is the equivalent of the copy function in a text window),

rename

which allows the name of the selected file to be changed. The name of a new file must not be the name of a file that already exists,

remove

which allows the selected file to be destroyed.

When the name of the top window does not correspond to any existing file, the central window contains a list, whose only element is the name entered in the top window, together with a menu whose options are:

copy name

which allows, as before, the name of the selected file to be copied into the text buffer,

rename

which allows the selected file to be renamed,

new file

which allows a new file to be created whose name will be

the selected name

new directory

which allows a new directory to be created whose name will be the selected name.

The bottom window is a text window; it contains the text of the selected file in the central window if the option 'get contents' has been used in this window. The menu offered is the one generally offered in this type of window, but it contains three additional options, namely:

file it in

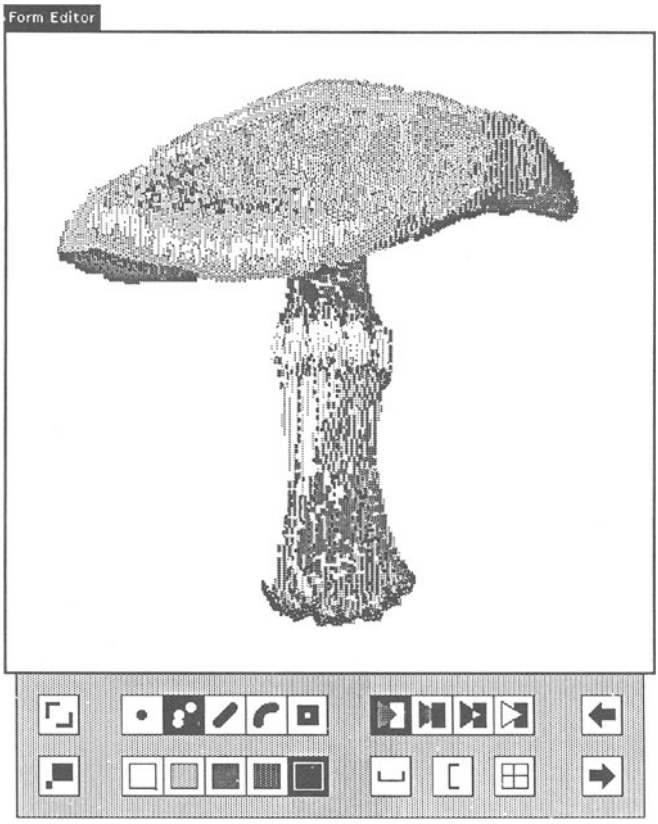
which allows the selected text to be compiled by interpreting it as text defining classes and methods, that is, this text must be in the format of the external files,

put

which writes the text in the bottom window to the selected external file,

get

which replaces the text in the bottom window with the text from the selected file.



19.4 Graphics editors

The Smalltalk system has two graphics editors that allow a form to be drawn. Such forms are instances of the class Form (see chapter 15) or of one of its subclasses.

All the entities displayed on the screen are forms (drawing, cursor, character font) and can therefore be edited graphically.

19.4.1 The Form Editor

This editor consists of two windows, one containing the form to be edited and the other a graphics menu from which the user selects the functions required to draw on the form. To edit a form with this graphics editor necessitates sending it the message edit. The figure on page 146 illustrates the graphics editor.

To select an option from the graphics menu the user places the cursor on the option and presses the red button on the mouse. The options in this menu are as follows:

allows selection of the pattern that will serve as the paintbrush. This pattern is obtained by designating a rectangle on the screen.

indicates that the paintbrush is to be copied at the position indicated by the mouse as soon as the red button is pressed.

indicates that the paintbrush is to be copied at the position indicated by the mouse as long as the red button is pressed.

indicates that the paintbrush is to be copied along the line defined by two presses on the red button.

allows a curve to be drawn with the paintbrush.

allows a rectangle to be filled with the selected pattern.

indicates that paintbrush overwrites the background when it is applied.

indicates that the paintbrush is ORed on to the background.

indicates that paintbrush is XORed on to the background.

indicates that paintbrush is ANDed on to the background.

allows part of the drawing to be edited with the bit editor (see next section).

initialises the fill colour with a white pattern.

initialises the fill colour with a light grey pattern.

initialises the fill colour with a grey pattern.

initialises the fill colour with a dark grey pattern.

initialises the fill colour with a black pattern.

allows the horizontal grid to be used.

allows the vertical grid to be used.

allows the minimum number of points between two lines and two columns of the grid to be set.

allows the edited form to be output to an external file.

allows a form to be read in from an external file.

The menu linked to the yellow button has two options, namely:

accept

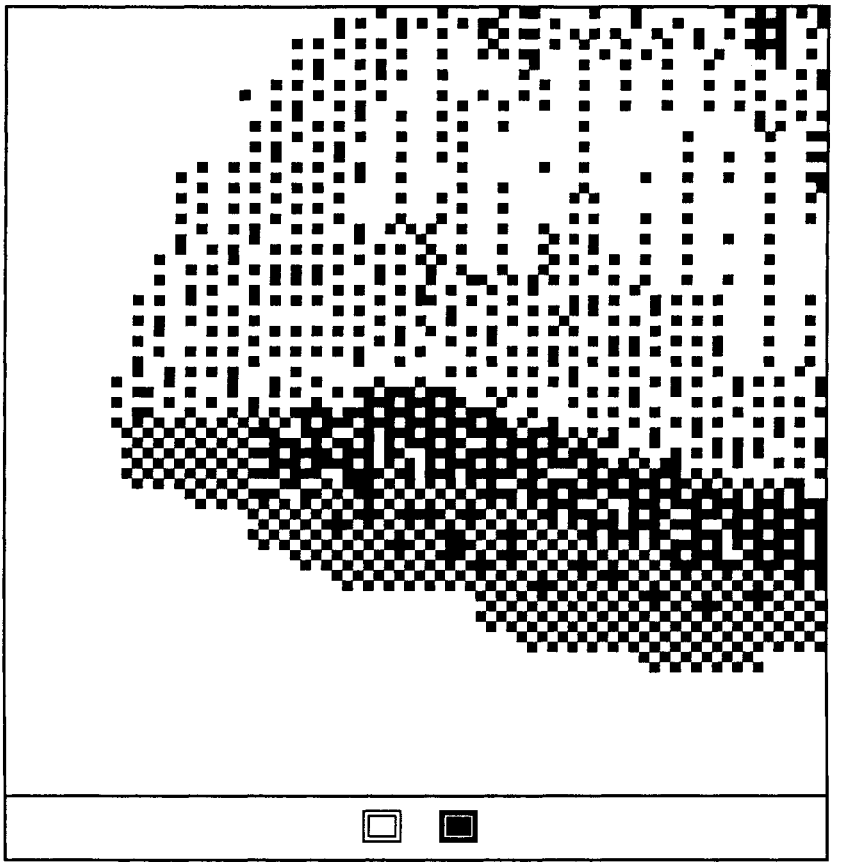
which allows the existing drawing form to be replaced by the one currently shown on the screen,

cancel

which replaces the drawing on the screen with the original edited drawing.

19.4.2 The Bit Editor

There is a second graphics editor called by the message bitEdit which allows access to each pixel of the drawing. In this case, the drawing is enlarged and each pixel is represented by a square of 8 x 8 points. The figure below shows an example of the use of this editor.



This view is made up of two subviews, one giving access to the enlarged drawing and the other which lets you choose to draw in either black or white. The menu, controlled by the yellow button, is the same as that offered by the editor edit. This editor is generally used for small size forms; for example, it can be used to create a new cursor:

```
Cursor new bitEdit
```

20 Management of User Events

In order to interact with the Smalltalk system, the user can use two different devices - the keyboard and the mouse. Any alteration to the state of the keyboard or the mouse is called a user event. The user events that are possible are:

- a movement of the mouse,
- the pressing of one of the mouse buttons,
- the striking of a key on the keyboard.

Within the Smalltalk system there are several classes that define methods for detecting and managing user events. Because only one user may interact with a Smalltalk system at once, these classes have only one instance.

20.1 The class InputState

The class **InputState**, which is a subclass of the class Object, interfaces Smalltalk with the hardware that detects user events. There is in the system one instance of this class which at any time represents the status of the keyboard and the mouse. This class has a class variable called **InputProcess**, which references the process reading the status of the keyboard and the mouse; this process is an instance of Process (cf chapter 13).

20.2 The class InputSensor

The class **InputSensor**, which is a subclass of the class Object, interfaces between the user and the instance of InputState that represents the status of the keyboard and the mouse. It has a class variable called **CurrentInputState**, which references the instance of the class **InputState**. This class supports the methods that allow the keyboard and the mouse to be tested.

Because there is only one keyboard and one mouse, this class only has one instance that is referenced by the global variable called **Sensor**. To test the status of the keyboard or the mouse, messages have to be sent to the global variable Sensor. The messages understood by Sensor are:

anyButtonPressed

returns true if one of the buttons on the mouse is pressed,

```
   noButtonPressed
returns true if none of the buttons on the mouse is
pressed,
   redButtonPressed
returns true if the red button on the mouse is pressed,
   yellowButtonPressed
returns true if the yellow button on the mouse is
pressed,
   blueButtonPressed
returns true if the blue button on the mouse is pressed,
   waitButton
waits for one of the buttons on the mouse to be pressed
and returns the cursor position,
   waitNoButton
waits until none of the buttons on the mouse is pressed
and returns the cursor position,
   keyboardPressed
returns true if one of the keys on the keyboard has been
pressed,
   keyboard
waits until a character is struck on the keyboard and
returns this character,
   keyboardPeek
reads a character from the keyboard queue without
removing it from the queue,
   keyboardNext
reads a character from the keyboard queue and withdraws
it from this queue,
   cursorPoint
returns the current cursor position,
   cursorPoint: aPoint
places the cursor at the position given as argument,
   currentCursor
returns the current cursor,
   currentCursor: aCursor
replaces the current cursor by the argument aCursor.
For example:
   Sensor waitButton.
   Display reverse.
   Sensor waitNoButton.
   Display reverse
reverses the screen as soon as one of the mouse buttons
is pressed and keeps it in this state until the button
is released.
```

21 Model-View-Controller System

In the preceding chapters we have met several different window types in the Smalltalk system, each designed to solve a particular problem. The programmer can also build special windows for his own individual applications. We recall that a window specifies a part of the screen in which interaction takes place between the user and the application.

In this chapter we shall see how interactive applications in the Smalltalk system are constructed and how to build new ones.

Within Smalltalk all interactive applications are built to the same plan, called the MVC system. The approach is based on the following three entities:

- the object on which the work is to be done,
- an output interface that serves to present this object to the user,
- an input interface that allows interaction with the object or with its representation.

In the Smalltalk system, the object on which the work is to be done is called the **model**, the output interface is called the **view**, and the input interface is called the **controller**.

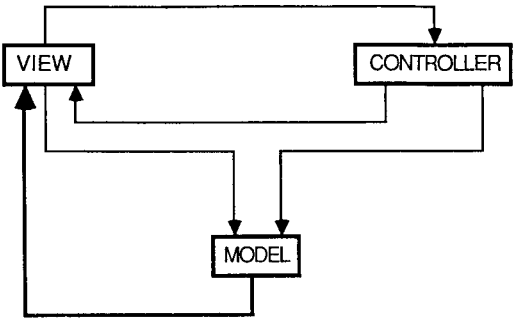
Whenever the model is changed, its representation must be updated. For this updating to be automatic, a dependency must be established between the view and the model that it represents. The concepts of dependence were examined in chapter 7 on the class Object.

In addition to the dependency of the view on the model, the view and the controller must be aware of one another and aware of the model on which they are to work.

The basic classes that allow views and controllers to be defined are the classes **View** and **Controller**. These two classes are subclasses of the class Object. The relationships between a model, its view and its controller are shown in the following figure.

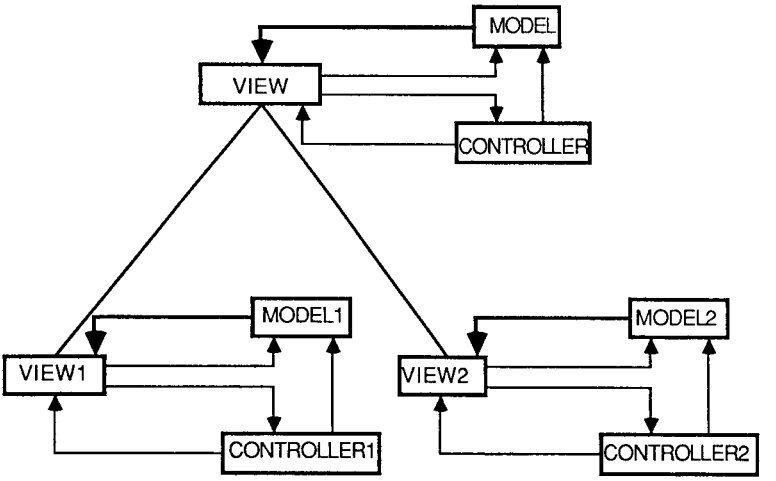
21.1 The class View

The instances of the class View are windows. Two of the instance variables of this class are model and con-



troller, which refer respectively to the model and the controller assigned to the view.

When several objects are to be represented in the same view, the view can be divided into several subviews. Thus, a hierarchy of views is defined, since a subview is a completely separate view. To implement this hierarchy each view has a list of subviews and one superview. The following figure shows the relationships between models, views and controllers in the case where a view is made up of several subviews.



21.1.1 The class WindowingTransformation

The coordinates of a view may be defined in several coordinate systems:

- in its own coordinate system,
- in the coordinate system of its superview,
- in the coordinate system of the screen.

In order to move from one coordinate system to another, simple geometric transformations are used.

The class WindowingTransformation is a subclass of the class Object whose instances serve to represent the

geometric transformations consisting of a translation and a scaling factor. The two instance variables of this class are scale and translation. These geometric transformations will be used to change coordinate system in a view.

The messages that allow new geometric transformations to be created are:

identity

which returns a transformation without translation and with an enlarging factor of 1,

scale: anEnlargingFactor translation: aTranslation

which returns an initialised geometric translation with the arguments of the method,

window: aWindow viewport: aViewport

which returns a transformation to move from the rectangle aWindow to the rectangle aViewport. This method is the one that constructs the transformation for passing the coordinates of a view to the coordinates of its superview.

The messages that can be sent to the instances of WindowingTransformation are:

applyTo: aPoint

which returns the point obtained by applying the geometric transformation to the point aPoint,

applyInverseTo: aPoint

which returns the point obtained by applying the inverse geometric transformation to the point aPoint,

compose: aGeometricTransformation

which returns a new instance of WindowingTransformation obtained by composing the receiver and the argument aGeometricTransformation.

21.1.2 Instance variables of the class View

The instance variables of the class View are:

model

which is the object represented by the view,

controller

which is the controller that allows interaction with the view or with the model,

superView

which is the superview of the view,

subViews

which is an instance of OrderedCollection containing the subviews of the view,

transformation

which is an instance of WindowTransformation allowing transfer from the coordinate system belonging to the view (given by window) to the coordinate system belonging to the superview,

viewport

which is a rectangle giving the coordinates of the view

expressed in the coordinate system of the superview,

window

which is a rectangle giving the coordinate system of the view,

displayTransformation

which is an instance of WindowTransformation allowing transfer from the coordinate system belonging to the view (given by window) to the coordinate system of the screen,

borderWidth

which may be a number, a point or a rectangle, giving the width of the border that encloses the view,

borderColour

which is a motif giving the colour of the border,

insetDisplayBox

which is a rectangle expressed in the screen coordinates representing the interior rectangle of the view (the view without its border),

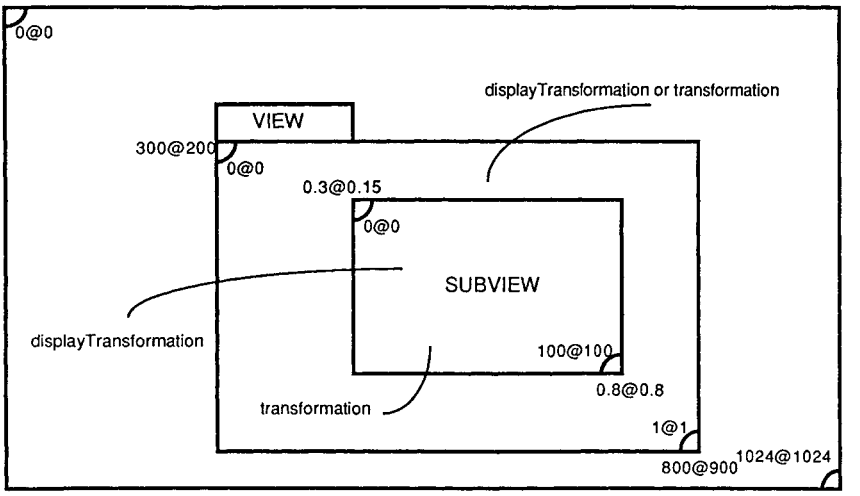
insideColour

which is a motif giving the background colour of the interior of the view,

boundingBox

which is the minimum rectangle expressed in the view coordinates containing the view and its subviews.

The following figure recapitulates the different coordinate systems in which the position of a point can be expressed and the transformations that allow transfer from one system to another.



The coordinates belonging to a view are indicated inside the view and the coordinates of that view within its superview are shown outside.

21.1.3 Messages understood by instances of View

The class View, which is the basis of the entire windowing system, supports a large number of messages. These messages are divided into several categories which are examined in the following sections.

Manipulation of views and subviews

addSubView: aView
adds the argument aView to the list of subviews of the receiver,

addSubView: aView align: aPoint with: anOtherPoint
adds the argument aView to the list of subviews of the receiver, by fixing the viewport such that the point aPoint, expressed in the coordinate system of aView, is located at the point anOtherPoint, expressed in the receiver's coordinate system,

addSubView: aView below: aPoint
adds the argument aView to the list of subviews of the receiver, by fixing the viewport such that the origin of the view aView is located at the point aPoint, expressed in the receiver coordinate system,

firstSubView
returns the first subview of the receiver,

lastSubView
returns the last subview of the receiver,

removeFromSuperView
removes the receiver from the list of subviews of its superview, if it exists,

removeSubView: aView
removes the argument aView from the list of subviews of the receiver,

resetSubViews
removes all the subviews from the list of subviews of the receiver,

superView
returns the superview of the receiver,

superView: aView
replaces the superview of the receiver by the view given as argument,

topView
returns the main view of the hierarchy of views containing the receiver,

isTopView
returns true if the receiver has no superview,

release
before removing a view, one must also remove the dependence of this view in relation to the model, together with all the dependencies of its subviews; this is the function of the message release.

Access to coordinate systems
displayBox
returns the rectangle containing the receiver,
insetDisplayBox
returns the rectangle interior to the receiver,
window
returns the coordinate system of the receiver,
window: aRectangle
replaces the coordinate system of the receiver by that given as argument,
window: aRectangle viewport: anOtherRectangle
replaces the coordinate system of the receiver and the coordinates of the receiver within its superview by the coordinates given as argument,
viewport
returns the coordinates of the receiver expressed in the coordinate system of its superview,
boundingBox
returns the minimum rectangle containing the receiver and its subviews,
containsPoint: aPoint
returns true if the point given as argument is within the receiver.

Access to the controller
controller
returns the controller assigned to the receiver,
controller: aController
replaces the controller of the receiver by the controller given as argument,
defaultController
returns a new controller which is a default instance of the class.
defaultControllerClass
returns the class of the controller.

Access to the model
model
returns the object represented by the receiver,
model: anObject
replaces the object represented by the receiver by the object given as argument,
model: anObject controller: aController
replaces the object represented by the receiver and the assigned controller by the objects given as argument.

Display
borderColour: aMotif
replaces the motif that provides the border colour of the receiver by the motif given as argument,
borderWidth: aWidth

replaces the width of the receiver border by the width given as argument,

clear: aMotif
fills the area of screen occupied by the receiver with the motif given as argument,

clearInside
fills the interior of the receiver with the motif given by the instance variable insideColour,

clearInside: aMotif
fills the interior of the receiver with the motif given as argument,

display
displays the receiver border, the receiver itself, and its subviews,

displayBorder
displays the receiver border,

flash
makes that part of the screen occupied by the receiver flash,

highlight
inverts that part of the screen occupied by the receiver,

insideColour
returns the motif that defines the background colour of the receiver,

insideColour: aMotif
replaces the background colour of the receiver by the motif given as argument.

The class View has several subclasses which allow different types of object to be represented (text, forms, list, etc). Interaction varies, according to the object represented; it is therefore necessary to assign a specific controller to each view. All its controllers are defined in subclasses of the class Controller.

21.2 The class Controller

The class Controller is a subclass of the class Object. Defined in this class are the basic operations for controlling interactions and locating the active controller.

21.2.1 Instance variables of Controller

This class has three instance variables:

model
which is the object to be interacted with,

view
which is the view representing the model,

sensor
which is a reference to the global variable Sensor, the only instance of the class InputSensor; use of such a reference saves time in locating this object.

21.2.2 Messages understood by instances of Controller

The messages supported by the class Controller are:

controlActivity
finds the subview of the view assigned to the receiver which requires control and activates the controller assigned to this subview,

controlInitialize
this message is sent to the receiver when the latter has just been activated, thus allowing certain parameters to be set,

controlTerminate
this message is sent to the receiver just before the latter is deactivated, thus allowing certain parameters to be restored,

controlLoop
loops for as long as the view assigned to the receiver contains the cursor, in order to discover which of the subviews wishes to take control (perhaps none),

startUp
this message is sent to a controller to activate it; the method sends to the controller the messages control-Initialize, controlLoop and controlTerminate.

isControlActive
returns true if the view is to retain control,

model
returns the object with which the receiver is interacting,

model: anObject
replaces the object with which the receiver interacts by the object given as argument,

view
returns the view assigned to the receiver,

view: aView
replaces the view assigned to the receiver with the view given as argument,

release
removes the references of the receiver to the view and the model.

To activate a controller, the message startUp must be sent to it. The startUp method is written as follows:

```
startUp
        self controlInitialize.
        self controlLoop.
        self controlTerminate.
```

The methods controlInitialize and controlTerminate are redefined in the subclasses of Controller. For example, for the class ScrollController, which controls scrolling in a text window, these methods cause the scrollBar to appear on the left of the view when the controller is first activated and disappear at the end.

The controlLoop method is written as follows:

```
controlLoop
    [self isControlActive] whileTrue:
        [Processor yield.
         self controlActivity]
```

When the cursor moves from one view to another, it is necessary to be able to change the active controller in order to interact with the view on which the cursor is located. There is within the Smalltalk system a class called **ControlManager** whose task is to control the activation of the different controllers.

21.3 The class ControlManager

The class ControlManager is a subclass of the class Object. The instance of the class ControlManager which controls the controllers at any given moment is referenced by the global variable called **ScheduledControllers.**

Its instance variables are:

scheduledControllers

which references the list of controllers that may be activated; this list further contains an instance of **ScreenController** which controls the screen apart from any window,

activeController

which references the active controller,

activeControllerProcess

which references the process connected to the active controller; a new process is executed every time that the active controller changes,

screenController

which references an instance of the class ScreenController that controls those parts of the screen not occupied by a window.

When there is no longer an active controller, the instance of ControlManager referenced by ScheduledControllers decides to which window it should hand over control.

If there is no window that wishes to take control, it hands control to the screen as soon as the user presses the yellow button on the mouse.

Each project contains a collection of windows and is directed by an instance of the class ControlManager that belongs to it. When a project is changed, the controlManager of this project is still referenced by the global variable ScheduledControllers.

When the controller of a window is activated, it will decide itself whether it is to retain control or hand it to one of the subviews of the window.

21.4 The class MouseMenuController

The class MouseMenuController is a subclass of the class

Controller. The instances of this class handle mouse events. The instance variables of this class are:

redButtonMenu

which is a PopUpMenu that appears whenever the user presses the red button,

redButtonMessages

which is an array of symbols corresponding to the messages that will be sent to the controller if one of the menu options assigned to the red button is selected,

yellowButtonMenu

which is a PopUpMenu that appears whenever the user presses the yellow button,

yellowButtonMessages

which is an array of symbols corresponding to the messages that will be sent to the controller if one of the yellow button menu options is selected,

blueButtonMenu

which is a PopUpMenu that appears whenever the user presses the blue button,

blueButtonMessages

which is array of symbols corresponding to the messages that will be sent to the controller if one of the blue button menu options is selected.

If no menu is to be assigned to a mouse button, the instance variable containing the menu corresponding to this button will be set to nil.

The messages understood by the instances of the class MouseMenuController are:

redButtonMenu: aMenu

redButtonMessages: anArrayOfMessages

allows initialisation of the red button menu and list of messages,

yellowButtonMenu: aMenu

yellowButttonMessages: anArrayOfMessages

allows initialisation of the yellow button menu and list of messages,

blueButtonMenu: aMenu

blueButtonMessages: anArrayOfMessages

allows initialisation of the blue button meru and list of messages,

redButtonActivity

this message is sent to the receiver when the red button is pressed,

yellowButtonActivity

this message is sent to the receiver when the yellow button is pressed,

blueButtonActivity

this message is sent to the receiver when the blue button is pressed,

For the preceding three messages, if there is a menu assigned to the button, then the menu is displayed for

as long as the button is held down. When it is released, if one of the options was selected, the corresponding message is sent to the receiver.

Almost all the controllers are instances of MouseMenuController or of its subclasses, because almost all of them use the mouse.

The two important subclasses of this class are:

ScrollController

whose instances allow control of the scrollbar that appears to the left of windows that require scrolling,

StandardSystemController

whose instances allow screen windows to be manipulated (movement, changing size, closing, etc.).

21.5 The classes StandardSystemView and StandardSystemController

These two classes implement the basic mechanisms that allow windows to be defined and controlled in the most general sense.

A window in the Smalltalk system is made up of a rectangle defining a part of the screen, surmounted by another rectangle containing the name of the window.

We have already seen how a window may contain subwindows which may themselves contain other windows. The window located at the top of the hierarchy, again called topView, is generally an instance of the class StandardSystemView. Only these windows may be deactivated or reactivated by the ControlManager.

The class StandardSystemView is a subclass of the class View. The additional messages supported by this class are:

label

returns the name of the receiver,

label: aString

replaces the name of the receiver by the string given as argument,

labelDisplayBox

returns the rectangle containing the name,

reverseLabel

inverts the name of the receiver,

expand

changes the size of the receiver interactively,

minimumSize

returns the minimum size of the receiver that is used when beginning to interact with it,

minimumSize: aSize

fixes the minimum size of the receiver,

maximumSize

returns the maximum size of the receiver that is used when beginning to interact with it,

maximumSize: aSize

fixes the maximum size of the receiver.

The class StandardSystemController is a subclass of the class MouseMenuController whose instances generally control the windows that are instances of the class StandardSystemView. It has two class variables, namely:

ScheduledBlueButtonMenu

which is the PopUpMenu that is assigned by default to the blue button for all instances of this class,

ScheduledBlueButtonMessages

which is an array of messages sent to the receiver according to the command selected from the menu assigned to the blue button.

It has an instance variable called **status** which provides the status of the controller. The different possible states of a window are:

open

which is the status of the controller when interaction with the designated window is begun,

closed

which is the status of the controller when the designated window is closed (it is no longer visible on the screen),

active

which is the status of the controller when the designated window is active (that is, when the user can interact with this window),

inactive

which is the status of the controller when the designated window is no longer active.

To open a window, the message open must be sent to its controller.

21.6 Text-scrolling windows

When a section of text is too long to appear in a window in its entirety, it is only possible to display part of it, while allowing access to the remainder using a scrollbar.

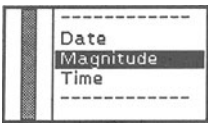
The control mechanisms for the scrollbar are implemented in the class **ScrollController** which is a subclass of the class MouseMenuController.

When a window has a scrollbar, the latter is only visible when the cursor is located within the window. Before making the scrollbar appear, the ScrollController saves the area of the screen that will be covered by the scrollbar in an instance variable called savedArea. When the ScrollController makes the scrollbar disappear, it redisplays the form contained in savedArea.

In the Smalltalk system the scrollbars are used in two types of window:

- text-editing windows

- windows for selecting an element from a list (see figure below).



The two subclasses of the class ScrollController that allow management of these two types of view are the classes **ParagraphEditor** and **ListController.**

21.6.1 Text-editing windows

We have seen that, in order to represent sections of text on the screen, we may use instances of the class DisplayText. To present a text in a window, it also has to be formatted; that is, there has to be automatic control for moving to a new line when a line is filled or a word will not fit on the current line. The objects that allow formatted text to be produced are instances of the class **Paragraph.**

21.6.1.1 Paragraphs

The class Paragraph is a subclass of DisplayText, whose instances responds to certain additional messages.

A paragraph may be created from a section of text or from a string of characters by sending them the message asParagraph, for example:

´this is a string of characters´ asParagraph

returns a paragraph that represents the receiver.

As the text of a paragraph is formatted, there are several ways of putting this text in shape, and the messages that allow choice of page layout are:

centred

which allows the text to be centred on a line,

justified

which allows the text to be aligned on the left and right margins,

leftFlush

which allows the text to be aligned on the left margin; this is the default option,

rightFlush

which allows the text to be aligned on the right margin.

When the size of a text window changes (for example, with the option frame via the blue button), the text formatting of the text in this window must also be changed. There is a message that allows a section of text to be reformatted, namely:

recomposeIn: aRectangle clippingBox: anOtherRectangle

which recalculates the page layout for the paragraph, so

that the lines will fit into the limits of the argument aRectangle and the display is restricted to the argument anOtherRectangle.

The controllers that allow a paragraph to be edited are instances of the class ParagraphEditor, which is a subclass of ScrollController. These controllers respond to a significant number of messages which the user has no need to access; these are therefore not described in this chapter.

In a text-editing window, the trio model-view-controller is made up of:

- a section of text or a string of characters for the model,
- an instance of View or of one of its subclasses, which displays the paragraph created from the model, for the view,
- an instance of ParagraphEditor or of one of its subclasses for the controller.

21.6.1.2 StringHolder StringHolderView StringHolderController

These three classes make up a trio of model-view-controller that allow the text of the model to be edited in a text-editing window.

The class **StringHolder** is a subclass of the class Object, whose instance variables are:

contents

which is the string of characters that is edited,

isLocked

which is a boolean that indicates whether the text of the view is the same as that of contents.

The class **StringHolderView** is a subclass of View. One of its instance variables, called displayContents, references the paragraph that represents the formatted text of the model. The instances of StringHolderView are not main views (main views are generally instances of StandardSystemView), but simply subviews.

The class **StringHolderController** is a subclass of the class ParagraphEditor which allows the paragraph assigned to the model to be edited.

21.6.2 Lists

Lists are views which contain one or more lines of text and from which it is possible to select a line. For example, lists may be found in the four windows at the top of a browser; these allow access to the categories, classes, protocols and methods.

The MVC trio assigned to the control of a list consists of:

- an object that may in certain cases be considered as

a list,
- a view that is an instance of a subclass of **ListView,**
- a controller that is an instance of a subclass of **ListController.**

The subclasses of ListView and ListController that allow any lists to be manipulated are **SelectionInList-View** and **SelectionInListController.**

The object which represents the model may be any object that responds to the messages that cause the system to:
- return the selected element
- change the selected element
- return the list of all elements
- return a menu assigned to the centre button which allows action on the elements in the list.

The name of the messages that will cause the above functions to occur will be given to the view at the time that it is created. Views of this type are rather special, because their model may belong to any class; they are said to be ´pluggable´ views.

To create a view of this type, a message consisting of the following various selectors must be sent to the class SelectionInListView:

```
on: anObject
printItems: aBoolean
oneItem: anOtherBoolean
aspect: aspectMessage
change: changeMessage
list: listMessage
menu: menuMessage
initialSelection: selectionMessage
```

which returns an instance of SelectionInListView whose model is the argument anObject.

The argument aBoolean determines if the text of the view must be obtained by sending the message printString to all the elements in the list or if these elements are already strings of characters.

The argument anOtherBoolean determines if the list contains one and only one element.

The argument aspectMessage is the message that will be sent to the model to access the element selected from the list.

The argument changeMessage is a message to an argument that will be sent to the model to change the selected element.

The argument listMessage is the message that will be sent to the model to access the list of elements.

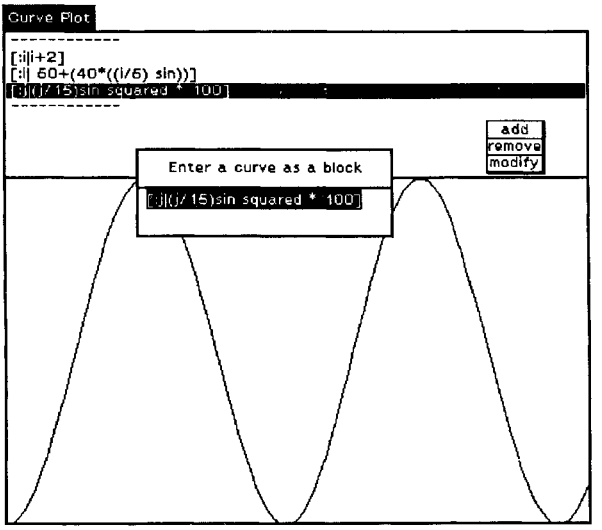
The argument menuMessage is the message that will be sent to the model to have a menu assigned to the centre button.

The argument selectionMessage is the message that

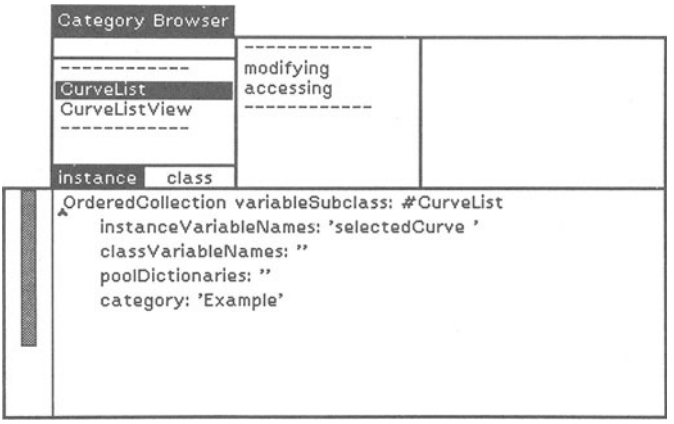
will be sent to the model to access the element selected at the start.

21.6 Example

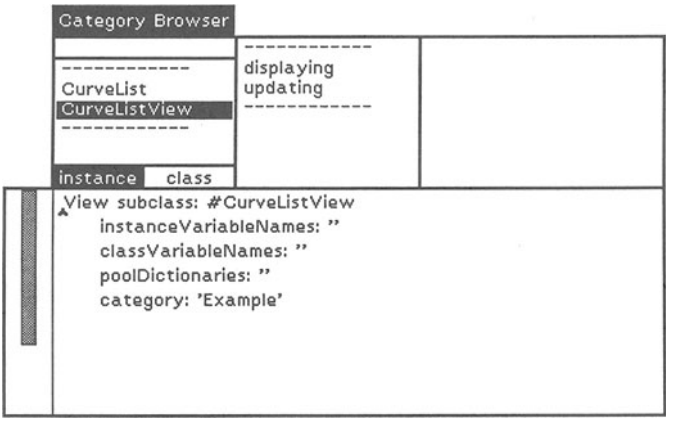
The following review example shows how to construct an application that will allow a list of curves to be manipulated in one window, while the selected curve is represented in a second window, as shown below.



The equation of a curve may be represented by a block with one argument. The orientation of the axes in the view is that normally adopted in mathematics. The scale goes from 0 to 100 along the abscissa and the ordinate. To implement this application, only two classes are required, as shown below.

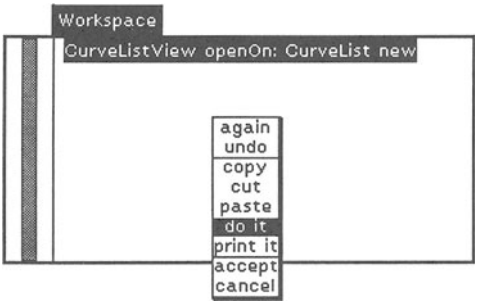


The class Curvelist is a subclass of OrderedCollection whose elements will be the strings of characters that provide the equations of the curves.

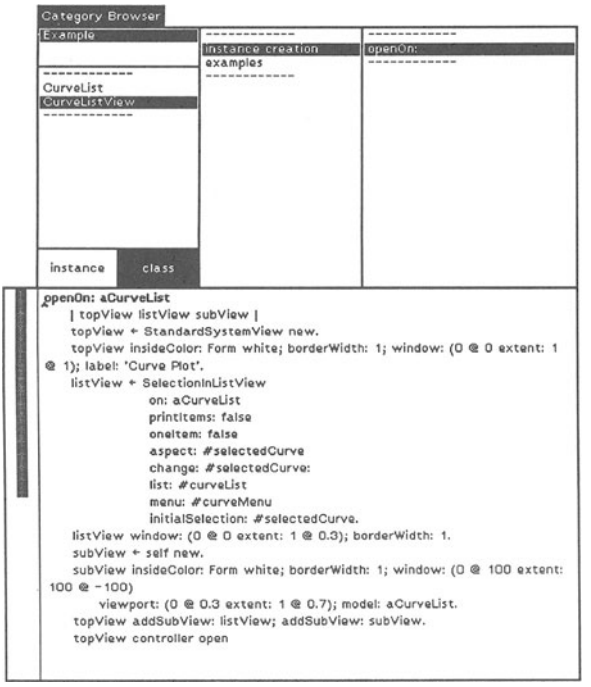


The class CurveListView is a subclass of View which simply serves to redefine the method displayView that allows a curve to be displayed.

To open a curves window it is simply a question of evaluating the following expression.



The first method to be written is the method for creating a window. The text of this method appears in the window below.



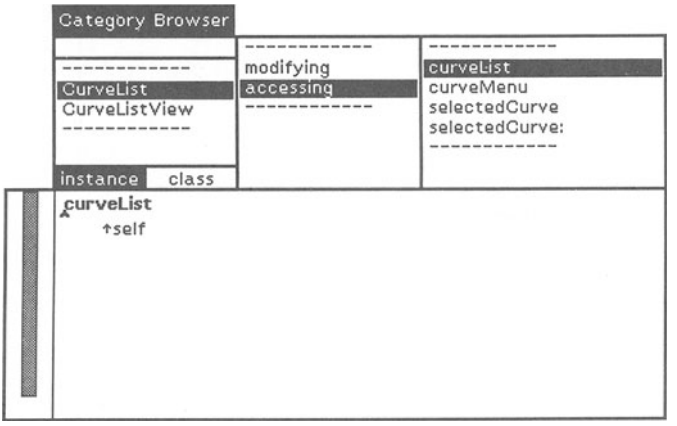
The window is made up of three views:

- a main view which is an instance of StandardSystemView; the coordinates of this view (given by window:) range from 0@0 to 1@1,
- a subview (an instance of SelectionInListView) which allows the list of curves to be represented and manipulated; the coordinates of this view in its superview are 0@0 and 1@0.3,
- a subview (an instance of CurveListView) which is the view in which the selected curve is drawn; the coordinates of this view in its superview are 0@0.3 and 1@1. The coordinates belonging to this view range from 0@100 to 100@0, which represent a conventional (mathematical) coordinate system (the axis of the abscissas from left to right, and the axis of the ordinates from bottom to top).

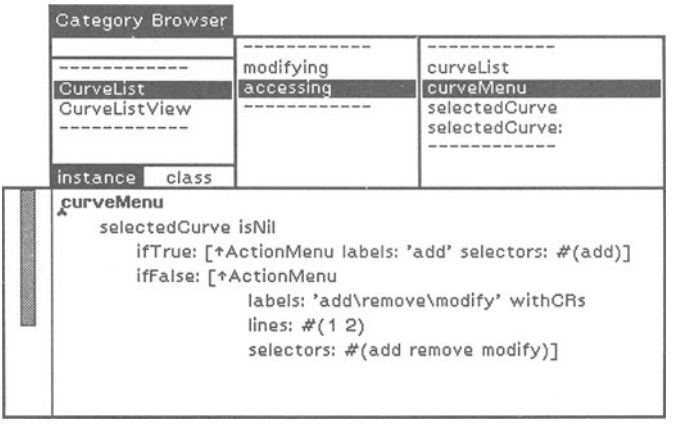
To create the instance of SelectionInListView, it was necessary to give the messages that will be sent to the model in order to access the list of curves; now, the methods that correspond to these methods must be defined. These methods will be defined within the model, namely ListCurve.

The method curveList allows access to the list of elements of CurveList; here, this list directly consists

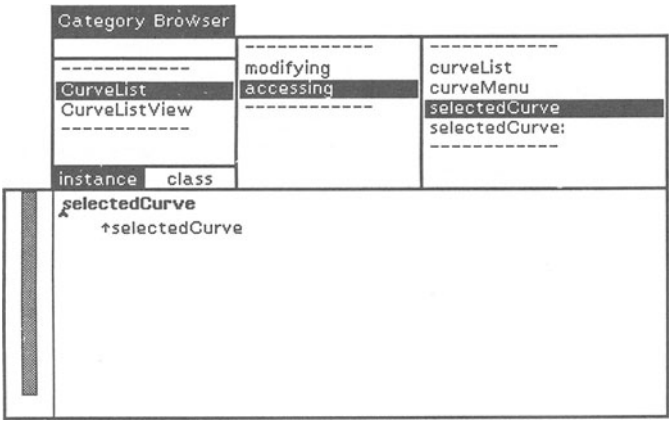
of the receiver, as shown in the figure below.



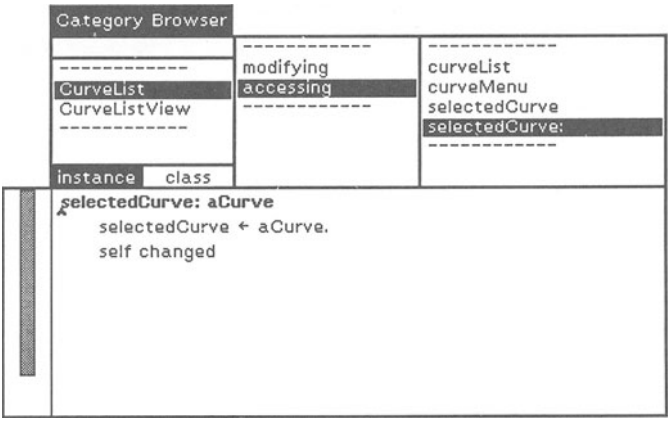
The method curveMenu returns a menu (instance of the class ActionMenu) which is a subclass of PopUpMenu including an additional instance variable which references the list of messages assigned to the menu options. If no line is selected, the menu returned only includes the option to add a curve to the list. If a curve is selected from the list, the menu returned includes three options, allowing a curve to be added, removed, or the selected curve to be altered.



The method selectedCurve returns the variable called selectedCurve (see below).



The method which allows a different curve to be selected is described below. Note that sending the message changed to the receiver has the effect of updating the view; in fact, it should be recalled that there is a dependence between the model and its view, and therefore any alteration of the model is signalled to the view.



There are two protocols in the class CurveList: the first, which contains the access methods, has already been described; the second contains the methods for manipulating the list.

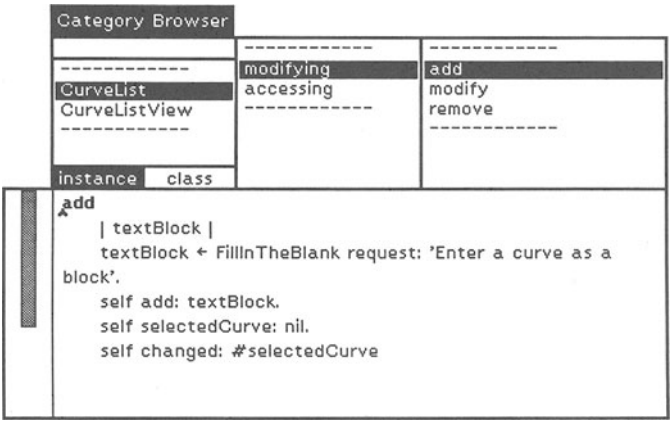
To add a curve, its curve must be input in the form of a block; this block must have an argument which, during evaluation, will take the different values of the abscissa.

The class **FillInTheBlank** allows text in a window to

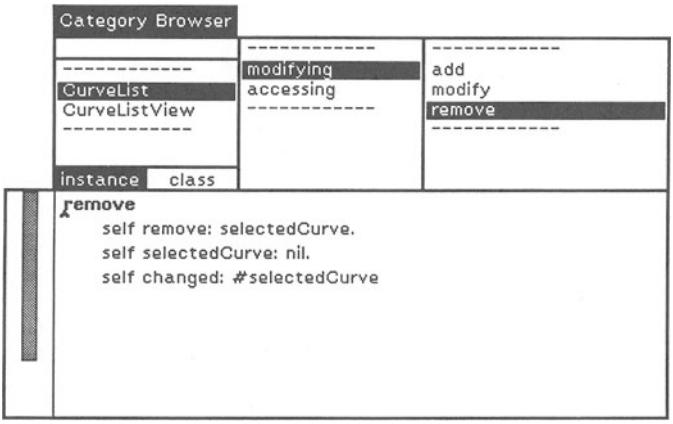
be captured; the window closes automatically when the capture is completed by using the accept option on the middle button or <return>. The expression

```
self changed: #selectedCurve
```

redisplays the list.

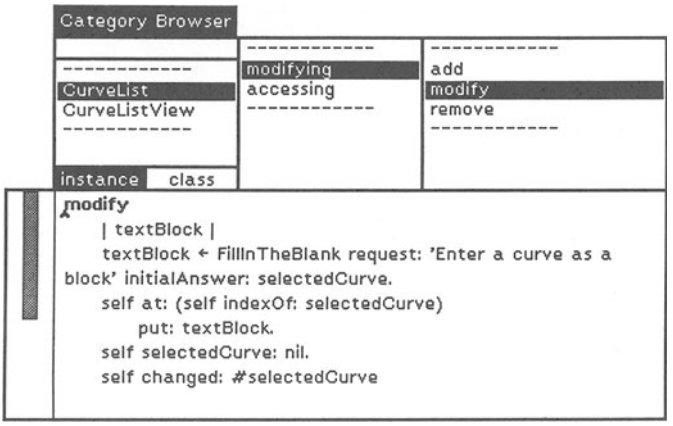


The following method allows a curve to be removed from the list.

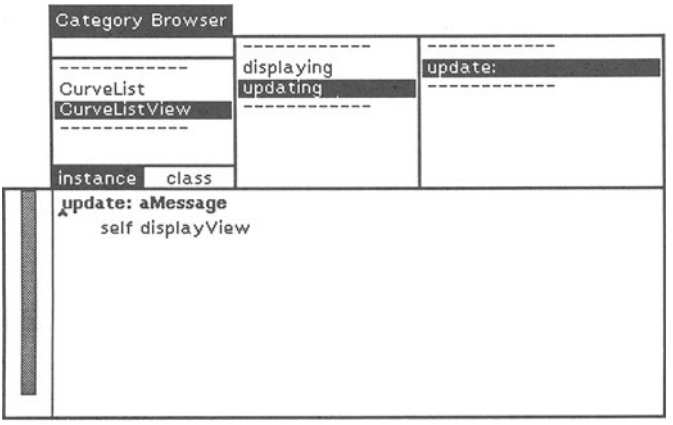


To alter a curve, a window needs to be opened on the text of the selected curve; then, once the text has been altered, the selected curve is replaced by the new text.

All the methods of CurveList are written; all that remains is to write the instance methods of CurveListView. The first of these methods is very simple, since it involves redefining the message update:. Remember that this message is sent to all the dependents of an object when the latter receives the message changed:. Here, any alteration of the model



(that is, the list of curves), must be result in a redisplay of the view.

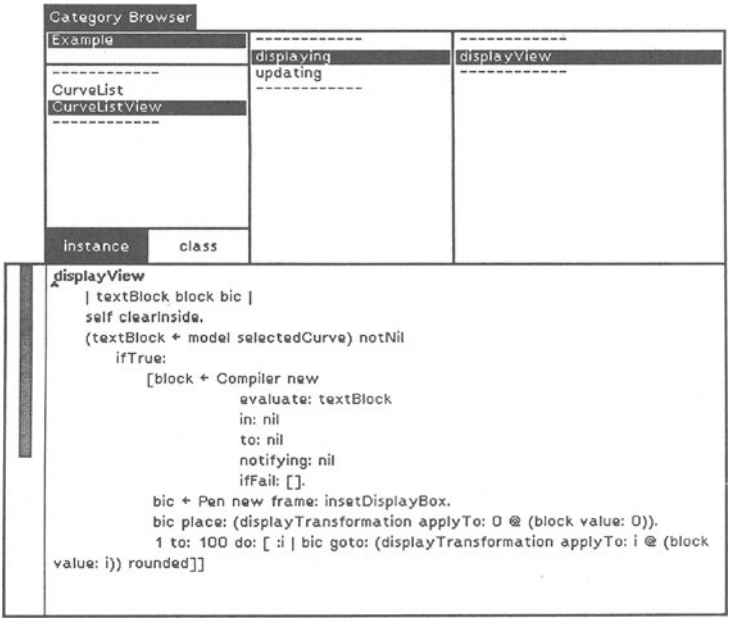


The method that displays the interior of a view is called displayView. Here, this method must be redefined in order to display the curve within the view.

The method begins by erasing the interior of the view, since if there is a selected curve it draws it. The first task to be done is to transform the string of characters that represents the curve into a block; to do this, the text of the curve must be evaluated.

The pen can then be placed on the first point and be made to draw straight lines between all the following points. The transformation called displayTransformation must be applied to all the points in order to transfer the coordinates of the view to the coordinates of the screen.

This example illustrates how it is possible to construct graphics applications quickly, using but a few classes and methods. It will be noted that some methods



serve to redefine existing methods, which is a feature often found in Smalltalk, because an application is often defined by the differences it has in comparison with an existing application (differential programming).

This example is also typical of what might be an application prototype.

The constraints imposed on such prototypes are as follows:

- shorter realisation time
- ease of development
- minimum operation; that is, there is no need to wait for the complete realisation of the prototype before running those parts already defined.

The MVC system is fundamental to Smalltalk, because it provides a methodology for breaking down graphics and interactive applications. Given that there are more than 30 subclasses of the class View, as well as more than 30 subclasses of the class Controller, it has only been possible to discuss the main classes in this chapter.

Index